PRINCIPLES OF COMPUTER SYSTEM DESIGN



Jerome H. Saltzer & M. Frans Kaashoek
Massachusetts Institute of Technology

LibreTexts™

Massachusetts Institute of Technology

Principles of Computer System Design

Jerome H. Saltzer & M. Frans Kaashoek

This text is disseminated via the Open Education Resource (OER) LibreTexts Project (https://LibreTexts.org) and like the thousands of other texts available within this powerful platform, it is freely available for reading, printing, and "consuming."

The LibreTexts mission is to bring together students, faculty, and scholars in a collaborative effort to provide and accessible, and comprehensive platform that empowers our community to develop, curate, adapt, and adopt openly licensed resources and technologies; through these efforts we can reduce the financial burden born from traditional educational resource costs, ensuring education is more accessible for students and communities worldwide.

Most, but not all, pages in the library have licenses that may allow individuals to make changes, save, and print this book. Carefully consult the applicable license(s) before pursuing such effects. Instructors can adopt existing LibreTexts texts or Remix them to quickly build course-specific resources to meet the needs of their students. Unlike traditional textbooks, LibreTexts' web based origins allow powerful integration of advanced features and new technologies to support learning.



LibreTexts is the adaptable, user-friendly non-profit open education resource platform that educators trust for creating, customizing, and sharing accessible, interactive textbooks, adaptive homework, and ancillary materials. We collaborate with individuals and organizations to champion open education initiatives, support institutional publishing programs, drive curriculum development projects, and more.

The LibreTexts libraries are Powered by NICE CXone Expert and was supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

Have questions or comments? For information about adoptions or adaptions contact info@LibreTexts.org or visit our main website at https://LibreTexts.org.

This text was compiled on 06/05/2025

Acknowledgments

This textbook began as a set of notes for the advanced undergraduate course Engineering of Computer Systems (6.033, originally 6.233), offered by the Department of Electrical Engineering and Computer Science of the Massachusetts Institute of Technology starting in 1968. The text has benefited from some four decades of comments and suggestions by many faculty members, visitors, recitation instructors, teaching assistants, and students. Over 5,000 students have used (and suffered through) draft versions, and observations of their learning experiences (as well as frequent confusion caused by the text) have informed the writing. We are grateful for those many contributions. In addition, certain aspects deserve specific acknowledgment.

1. Naming (Section 2.2 and Chapter 3 [not online])

The concept and organization of the materials on naming grew out of extensive discussions with Michael D. Schroeder. The naming model (and part of our development) follows closely the one developed by D. Austin Henderson in his Ph.D. thesis. Stephen A. Ward suggested some useful generalizations of the naming model, and Roger Needham suggested several concepts in response to an earlier version of this material. That earlier version, including in-depth examples of the naming model applied to addressing architectures and file systems, and an historical bibliography, was published as Chapter 3 in Rudolf Bayer et al., editors, *Operating Systems: An Advanced Course*, *Lecture Notes in Computer Science 60*, pages 99–208. Springer-Verlag, 1978, reprinted 1984. Additional ideas have been contributed by many others, including Ion Stoica, Karen Sollins, Daniel Jackson, Butler Lampson, David Karger, and Hari Balakrishnan.

2. Enforced Modularity and Virtualization (Chapters 4 and 5 [not online])

Chapter 4 was heavily influenced by lectures on the same topic by David L. Tennenhouse. Both chapters have been improved by substantial feedback from Hari Balakrishnan, Russ Cox, Michael Ernst, Eddie Kohler, Chris Laas, Barbara H. Liskov, Nancy Lynch, Samuel Madden, Robert T. Morris, Max Poletto, Martin Rinard, Susan Ruff, Gerald Jay Sussman, Julie Sussman, and Michael Walfish.

3. Networks (Chapter 7 [online])

Conversations with David D. Clark and David L. Tennenhouse were instrumental in laying out the organization of this chapter, and lectures by Clark were the basis for part of the presentation. Robert H. Halstead Jr. wrote an early draft set of notes about networking, and some of his ideas have also been borrowed. Hari Balakrishnan provided many suggestions and corrections and helped sort out muddled explanations, and Julie Sussman and Susan Ruff pointed out many opportunities to improve the presentation. The material on congestion control was developed with the help of extensive discussions with Hari Balakrishnan and Robert T. Morris, and is based in part on ideas from Raj Jain.

4. Fault Tolerance (Chapter 8 [online])

Most of the concepts and examples in this chapter were originally articulated by Claude Shannon, Edward F. Moore, David Huffman, Edward J. McCluskey, Butler W. Lampson, Daniel P. Siewiorek, and Jim N. Gray.

5. Transactions and Consistency (Chapters 9 [online] and 10 [online])

The material of the transactions and consistency chapters has been developed over the course of four decades with aid and ideas from many sources. The concept of version histories is due to Jack Dennis, and the particular form of all-or-nothing and before-or-after atomicity with version histories developed here is due to David P. Reed. Jim N. Gray not only came up with many of the ideas described in these two chapters, he also provided extensive comments. (That doesn't imply endorsement—he disagreed strongly about the importance of some of the ideas!) Other helpful comments and suggestions were made by Hari Balakrishnan, Andrew Herbert, Butler W. Lampson, Barbara H. Liskov, Samuel R. Madden, Larry Rudolph, Gerald Jay Sussman, and Julie Sussman.

6. Computer Security (Chapter 11 [online])

Sections 11.1 and 11.6 draw heavily from the paper "The Protection of Information in Computer Systems" by Jerome H. Saltzer and Michael D. Schroeder, Proceedings of the IEEE 63, 9 (September, 1975), pages 1278–1308. Ronald Rivest, David Mazières, and Robert T. Morris made significant contributions to material presented throughout the chapter. Brad Chen, Michael Ernst, Kevin Fu, Charles Leiserson, Susan Ruff, and Seth Teller made numerous suggestions for improving the text.

7. Suggested Outside Readings

Ideas for suggested readings have come from many sources. Particular thanks must go to Michael D. Schroeder, who uncovered several of the classic systems papers in places outside computer science where nobody else would have thought to look, Edward D. Lazowska, who provided an extensive reading list used at the University of Washington, and Butler W. Lampson, who provided a thoughtful review of the list.

8. Exercises and Problem Sets

The exercises at the end of each chapter and the problem sets at the end of the book have been collected, suggested, tried, debugged, and revised by many different faculty members, instructors, teaching assistants, and undergraduate students over a period of 40 years in the process of constructing quizzes and examinations while teaching the material of the text.

Certain of the longer exercises and most of the problem sets, which are based on lead-in stories and include several related questions, represent a substantial effort by a single individual. For those problem sets not developed by one of the authors, a credit line appears in a footnote on the first page of the problem set. Following each problem or problem set is an identifier of the form "*1978–3–14*". This identifier reports the year, examination number, and problem number of the examination in which some version of that problem first appeared.

TABLE OF CONTENTS

Acknowledgments

Computer System Design Principles

Licensing

1: The Network as a System and as a System Component

- 1.1: Overview
- 1.2: Interesting Properties of Networks
- 1.3: Getting Organized - Layers
- 1.4: The Link Layer
- 1.5: The Network Layer
- 1.6: The End-to-End Layer
- 1.7: A Network System Design Issue - Congestion Control
- 1.8: Wrapping Up Networks
- 1.9: Case Study - Mapping the Internet to the Ethernet
- 1.10: War Stories - Surprises in Protocol Design
- 1.11: Exercises

2: Fault Tolerance - Reliable Systems from Unreliable Components

- 2.1: Overview
- 2.2: Faults, Failures, and Fault-Tolerant Design
- 2.3: Measures of Reliability and Failure Tolerance
- 2.4: Tolerating Active Faults
- 2.5: Systematically Applying Redundancy
- 2.6: Applying Redundancy to Software and Data
- 2.7: Wrapping Up Reliability
- 2.8: Application - A Fault Tolerance Model for CMOS RAM
- 2.9: War Stories - Fault-Tolerant Systems That Failed
- 2.10: Exercises

3: Atomicity - All-or-nothing and Before-or-after

- 3.1: Overview
- 3.2: Atomicity
- 3.3: All-Or-Nothing Atomicity I - Concepts
- 3.4: All-Or-Nothing Atomicity II - Pragmatics
- 3.5: Before-Or-After Atomicity I - Concepts
- 3.6: Before-Or-After Atomicity II - Pragmatics
- 3.7: Atomicity Across Layers and Multiple Sites
- 3.8: A More Complete Model of Disk Failure (Advanced Topic)
- 3.9: Case Studies - Machine Language Atomicity
- 3.10: Exercises

4: Consistency

- 4.1: Overview
- 4.2: Constraints and Interface Consistency

- 4.3: Cache Coherence
- 4.4: Durable Storage Revisited - Geographically Separated Replicas
- 4.5: Reconciliation
- 4.6: Perspectives
- 4.7: Exercises

5: Information Security

- 5.1: Overview
- 5.2: Introduction to Secure Systems
- 5.3: Authenticating Principals
- 5.4: Authenticating Messages
- 5.5: Message Confidentiality
- 5.6: Security Protocols
- 5.7: Authorization-Controlled Sharing
- 5.8: Reasoning About Authentication (Advanced Topic)
- 5.9: Cryptography as a Building Block (Advanced Topic)
- 5.10: Summary
- 5.11: Case Study - Transport Layer Security (TLS) for the Web
- 5.12: War Stories - Security System Breaches
- 5.13: Exercises

6: Ancillary Materials

- 6.1: Powerpoint Presentations
- 6.2: Problem Sets
  - 6.2.1: Problem Sets 1-10
  - 6.2.2: Problem Sets 11-20
  - 6.2.3: Problem Sets 21-30
  - 6.2.4: Problem Sets 31-32
- 6.3: Suggestions for Further Reading

Index

Glossary

Detailed Licensing

Computer System Design Principles

Throughout the text, the description of a design principle presents its name in a **bold-faced** display, and each place that the principle is used highlights it in *underlined italics*.

Design principles applicable to many areas of computer systems:

- **Adopt sweeping simplifications**
  - So you can see what you are doing.
- **Avoid excessive generality**
  - If it is good for everything, it is good for nothing.
- **Avoid rarely used components**
  - Deterioration and corruption accumulate unnoticed—until the next use.
- **Be explicit**
  - Get all of the assumptions out on the table.
- **Decouple modules with indirection**
  - Indirection supports replaceability.
- **Design for iteration**
  - You won't get it right the first time, so make it easy to change.
- **End-to-end argument**
  - The application knows best.
- **Escalating complexity principle**
  - Adding a feature increases complexity out of proportion.
- **Incommensurate scaling rule**
  - Changing a parameter by a factor of ten requires a new design.
- **Keep digging principle.**
  - Complex systems fail for complex reasons.
- **Law of diminishing returns**
  - The more one improves some measure of goodness, the more effort the next improvement will require.
- **Open design principle**
  - Let anyone comment on the design; you need all the help you can get.
- **Principle of least astonishment**
  - People are part of the system. Choose interfaces that match the user's experience, expectations, and mental models.
- **Robustness principle**
  - Be tolerant of inputs, strict on outputs.
- **Safety margin principle**
  - Keep track of the distance to the edge of the cliff or you may fall over the edge.
- **Unyielding foundations rule**
  - It is easier to change a module than to change the modularity.

Design principles applicable to specific areas of computer systems:

- ***Atomicity:*** **Golden rule of atomicity**
  - Never modify the only copy!
- ***Coordination:*** **One-writer principle**
  - If each variable has only one writer, coordination is simpler.
- ***Durability:*** **The durability mantra**

- Have multiple copies, widely separated and independently administered.

- ***Security:* Minimize secrets**
  - Because they probably won't remain secret for long.
- ***Security:* Complete mediation**
  - Check every operation for authenticity, integrity, and authorization.
- ***Security:* Fail-safe defaults**
  - Most users won't change them, so set defaults to do something safe.
- ***Security:* Least privilege principle**
  - Don't store lunch in the safe with the jewels.
- ***Security:* Economy of mechanism**
  - The less there is, the more likely you will get it right.
- ***Security:* Minimize common mechanism**
  - Shared mechanisms provide unwanted communication paths.

Design Hints (useful but not as compelling as design principles):

- Exploit brute force
- Instead of reducing latency, hide it
- Optimize for the common case
- Separate mechanism from policy

Licensing

*A detailed breakdown of this resource's licensing can be found in **Back Matter/Detailed Licensing**.*

CHAPTER OVERVIEW

1: The Network as a System and as a System Component

1.1: Overview
1.2: Interesting Properties of Networks
1.3: Getting Organized - Layers
1.4: The Link Layer
1.5: The Network Layer
1.6: The End-to-End Layer
1.7: A Network System Design Issue - Congestion Control
1.8: Wrapping Up Networks
1.9: Case Study - Mapping the Internet to the Ethernet
1.10: War Stories - Surprises in Protocol Design
1.11: Exercises

This page titled 1: The Network as a System and as a System Component is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

1.1: Overview

Almost every computer system includes one or more communication links, and these communication links are usually organized to form a **network**, which can be loosely defined as a communication system that interconnects several entities. The basic abstraction remains `SEND(message)` and `RECEIVE (message)` , so we can view a network as an elaboration of a communication link. Networks have several interesting properties— interface style, interface timing, latency, failure modes, and parameter ranges —that require careful design attention. Although many of these properties appear in latent form in other system components, they become important or even dominate when the design includes communication.

Our study of networks begins, in Section 1.2, by identifying and investigating the interesting properties just mentioned, as well as methods of coping with those properties. Section 1.3 describes a three-layer reference model for a data communication network that is based on a best-effort contract, and Sections 1.4, 1.5, and 1.6 then explore more carefully a number of implementation issues and techniques for each of the three layers. Finally, Section 1.7 examines the problem of controlling network congestion.

A data communication network is an interesting example of a system itself. Most network designs make extensive use of layering as a modularization technique. Networks also provide in-depth examples of the issues involved in naming objects, in achieving fault tolerance, and in protecting information. (This chapter mentions fault tolerance and protection only in passing. Later chapters will return to these topics in proper depth.)

In addition to layering, this chapter identifies several techniques that have wide applicability both within computer networks and elsewhere in networked computer systems—framing, multiplexing, exponential backoff, best-effort contracts, latency masking, error control, and the end-to-end argument. A glance at the glossary will show that the chapter defines a large number of concepts. A particular network design is not likely to require them all, and in some contexts some of the ideas would be overkill. The engineering of a network as a system component requires trade-offs and careful judgment.

It is easy to be diverted into an in-depth study of networks because they are a fascinating topic in their own right. However, we will limit our exploration to their uses as system components and as a case study of system issues. If this treatment sparks a deeper interest in the topic, the Suggestions for Further Reading in Back Matter for this book include several good books and papers that provide wide-ranging treatments of all aspects of networks.

This page titled 1.1: Overview is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

1.2: Interesting Properties of Networks

The design of communication networks is dominated by three intertwined considerations: (1) a trio of fundamental physical properties, (2) the mechanics of sharing, and (3) a remarkably wide range of parameter values. The first dominating consideration is the trio of fundamental physical properties:

1. *The speed of light is finite.* Using the most direct route, and accounting for the velocity of propagation in real-world communication media, it takes about 20 milliseconds to transmit a signal across the 2,600 miles from Boston to Los Angeles. This time is known as the propagation delay, and there is no way to avoid it without moving the two cities closer together. If the signal travels via a geostationary satellite perched 22,400 miles above the equator and at a longitude halfway between those two cities, the propagation delay jumps to 244 milliseconds, a latency large enough that a human, not just a computer, will notice. However, communication between two computers in the same room may have a propagation delay of only 10 nanoseconds. That shorter latency makes some things easier to do, but the important implication is that network systems may have to accommodate a range of delay that spans seven orders of magnitude.
2. *Communication environments are hostile.* Computers are usually constructed of incredibly reliable components, and they are usually operated in relatively benign environments. But communication is carried out using wires, glass fibers, or radio signals that must traverse far more hostile environments ranging from under the floor to deep in the ocean. These environments endanger communication. Threats range from a burst of noise that wipes out individual bits to careless backhoe operators who sever cables that can require days to repair.
3. *Communication media have limited bandwidth.* Every transmission medium has a maximum rate at which one can transmit distinct signals. This maximum rate is determined by its physical properties, such as the distance between transmitter and receiver and the attenuation characteristics of the medium. Signals can be multilevel, not just binary, so the *data rate* can be greater than the signaling rate. However, noise limits the ability of a receiver to distinguish one signal level from another. The combination of limited signaling rate, finite signal power, and the existence of noise limits the rate at which data can be sent over a communication link.* Different network links may thus have radically different data rates, ranging from a few kilobits per second over a long-distance telephone line to several tens of gigabits per second over an optical fiber. Available data rate thus represents a second network parameter that may range over seven orders of magnitude.

* The formula that relates signaling rate, signal power, noise level, and maximum data rate, known as **Shannon's capacity theorem**, appears in Sidebar 2 in Section 1.4.

The second dominating consideration of communications networks is that they are nearly always shared. Sharing arises for two distinct reasons.

1. *Any-to-any connection.* Any communication system that connects more than two things intrinsically involves an element of sharing. If you have three computers, you usually discover quickly that there are times when you want to communicate between any pair. You can start by building a separate communication path between each pair, but this approach runs out of steam quickly because the number of paths required grows with the square of the number of communicating entities. Even in a small network, a shared communication system is usually much more practical—it is more economical and it is easier to manage. When the number of entities that need to communicate begins to grow, as suggested in Figure 1.2.1, there is little choice. A closely related observation is that networks may connect three entities or 300 million entities. The number of connected entities is thus a third network parameter with a wide range, in this case covering eight orders of magnitude.
2. *Sharing of communication costs.* Some parts of a communication system follow the same technological trends as do processors, memory, and disk: things made of silicon chips seem to fall in price every year. Other parts, such as digging up streets to lay wire or fiber, launching a satellite, or bidding to displace an existing radiobased service, are not getting any cheaper. Worse, when communication links leave a building, they require right-of-way, which usually subjects them to some form of regulation. Regulation operates on a majestic time scale, with procedures that involve courts and attorneys, legislative action, long-term policies, political pressures, and expediency. These procedures can eventually produce useful results, but on time scales measured in decades, whereas technological change makes new things feasible every year. This incommensurate rate of change means that communication costs rarely fall as fast as technology would permit, so sharing of those costs between otherwise independent users persists even in situations where the technology might allow them to avoid it.

The third dominating consideration of network design is the wide range of parameter values. We have already seen that propagation times, data rates, and the number of communicating computers can each vary by seven or more orders of magnitude. There is a

fourth such wide-ranging parameter: a single computer may at different times present a network with widely differing loads, ranging from transmitting a file at 30 megabytes per second to interactive typing at a rate of one byte per second.

These three considerations, unyielding physical limits, sharing of facilities, and existence of four different parameters that can each range over seven or more orders of magnitude, intrude on every level of network design, and even carefully thought-out modularity cannot completely mask them. As a result, systems that use networks as a component must take them into account.

1.2.1: Isochronous and Asynchronous Multiplexing

Sharing has significant consequences. Consider the simplified (and gradually becoming obsolescent) telephone network of Figure 1.2.1, which allows telephones in Boston to talk with telephones in Los Angeles. There are three shared components in this picture: a switch in Boston, a switch in Los Angeles, and an electrical circuit acting as a communication link between the two switches. The communication link is **multiplexed**, which means simply that it is used for several different communications at the same time. Let's focus on the multiplexed link. Suppose that there is an earthquake in Los Angeles, and many people in Boston simultaneously try to call their relatives in Los Angeles to find out what happened. The multiplexed link has a limited capacity, and at some point the next caller will be told the "network is busy." (In the U.S. telephone network this event is usually signaled with "fast busy," a series of beeps repeated at twice the speed of a usual busy signal.)

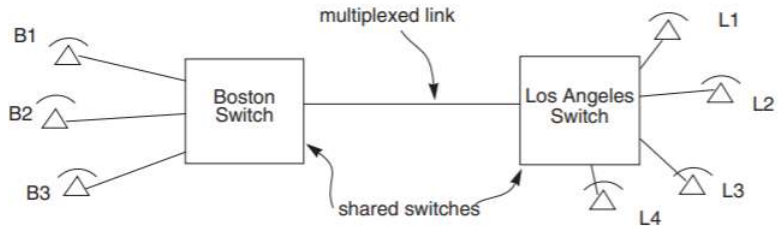


Figure 1.2.1: A simple telephone network.

This "network busy" phenomenon strikes rather abruptly because the telephone system traditionally uses a line multiplexing technique known as **isochronous** (from Greek roots meaning "equally timed") communication. Suppose that the telephones are all digital, operating at 64 kilobits per second, and the multiplexed link runs at 45 megabits per second. If we look for the bits that represent the conversation between B_2 and L_3, we will find them on the wire as shown in Figure 1.2.2: At regular intervals we will find 8-bit blocks (called **frames**) carrying data from B_2 to L_3. To maintain the required data rate of 64 kilobits per second, another B_2-to-L_3 frame comes by every 5,624 bit times or 125 microseconds, producing a rate of 8,000 frames per second. In between each pair of B_2-to-L_3 frames there is room for 702 other frames, which may be carrying bits belonging to other telephone conversations. A 45 megabits/second link can thus carry up to 703 simultaneous conversations, but if a 704th person tries to initiate a call, that person will receive the “network busy” signal. Such a capacity-limiting scheme is sometimes called hard-edged, meaning in this case that it offers no resistance to the first 703 calls, but it absolutely refuses to accept the 704th one.

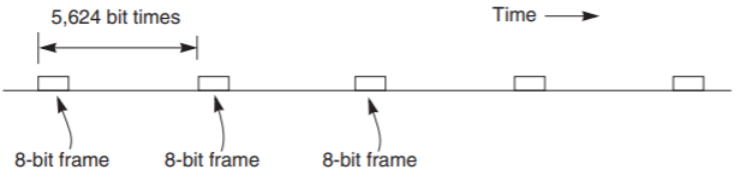


Figure 1.2.2: Data flow on an isochronous multiplexed link.

This scheme of dividing up the data into equal-size frames and transmitting the frames at equal intervals—known in communications literature as **time-division multiplexing** (TDM)—is especially suited to telephony because, from the point of view of any one telephone conversation, it provides a constant rate of data flow and the delay from one end to the other is the same for every frame.

One prerequisite to using isochronous communication is that there must be some prior arrangement between the sending switch and the receiving switch: an agreement that this periodic series of frames should be sent along to L_3. This agreement is an example of a **connection** and it requires some previous communication between the two switches to set up the connection, storage for

remembered state at both ends of the link, and some method to discard (**tear down**) that remembered state when the conversation between B_2 and L_3 is complete.

Data communication networks usually use a strategy different from telephony for multiplexing shared links. The starting point for this different strategy is to examine the data rate and latency requirements when one computer sends data to another. Usually, computer-related activities send data on an irregular basis—in bursts called **messages**—as compared with the continuous **stream** of bits that flows out of a simple digital telephone. Bursty traffic is particularly ill-suited to fixed size and spacing of isochronous frames. During those times when B_2 has nothing to send to L_3, the frames allocated to that connection go unused. Yet when B_2 does have something to send it may be larger than one frame in size, in which case the message may take a long time to send because of the rigidly fixed spacing between frames. Even if intervening frames belonging to other connections are unfilled, they can't be used by the connection from B_2 to L_3. When communicating data between two computers, a system designer is usually willing to forgo the guarantee of uniform data rate and uniform latency if in return an entire message can get through more quickly. Data communication networks achieve this trade-off by using what is called asynchronous (from Greek roots meaning "untimed") multiplexing. For example, in Figure 1.2.3, a network connects several personal computers and a service. In the middle of the network is a 45 megabits/second multiplexed link, shared by many network users. But, unlike the telephone example, this link is multiplexed asynchronously.

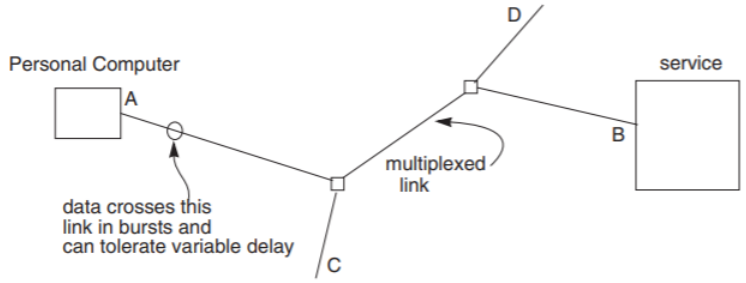


Figure 1.2.3: A simple data communication network.

On an asynchronous link, a frame can be of any convenient length, and can be carried at any time that the link is not being used for another frame. Thus in the time sequence shown in Figure 1.2.4 we see two frames, the first going to B and the second going to D. Since the receiver can no longer figure out where the message in the frame is destined by simply counting bits, each frame must include a few extra bits that provide guidance about where to deliver it. A variable-length frame, together with its guidance information, is called a **packet**. The guidance information can take any of several forms. A common form is to provide the **destination address** of the message: the name of the place to which the message should be delivered. In addition to delivery guidance information, asynchronous data transmission requires some way of figuring out where each frame starts and ends, a process known as framing. In contrast, both addressing and **framing** with isochronous communication are done implicitly, by watching the clock.

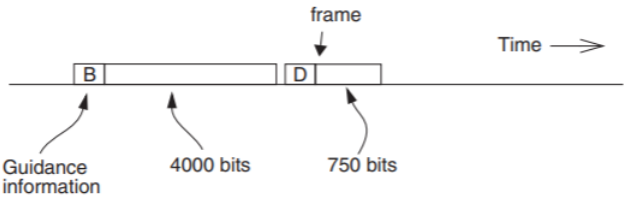


Figure 1.2.4: Data flow on an asynchronous multiplexed link.

Since a packet carries its own destination guidance, there is no need for any prior agreement between the ends of the multiplexed link. Asynchronous communication thus offers the possibility of **connectionless** transmission, in which the switches do not need to maintain state about particular end-user communications.*

An additional complication arises because most links place a limit on the maximum size of a frame. When a message is larger than this maximum size, it is necessary for the sender to break it up into **segments**, each of which the network carries in a separate packet, and include enough information with each segment to allow the original message to be **reassembled** at the other end.

Asynchronous transmission can also be used for continuous streams of data, such as that from a digital telephone, by breaking the stream up into segments. Doing so does create a problem: the segments may not arrive at the other end at a uniform rate or with a uniform delay. On the other hand, if the variations in rate and delay are small enough, or the application can tolerate occasional missing segments of data, the method is still effective. In the case of telephony, the technique is called "packet voice" and it is gradually replacing many parts of the traditional isochronous voice network.

---

* Network experts make a subtle distinction among different kinds of packets by using the word **datagram** to describe a packet that carries all of the state information (for example, its destination address) needed to guide the packet through a network of packet forwarders that do not themselves maintain any state about particular end-to-end connections.

1.2.2: Packet Forwarding; Delay

Asynchronous communication links are usually organized in a communication structure known as a **packet forwarding network**. In this organization, a number of slightly specialized computers known as **packet switches** (in contrast with the **circuit switches** of Figure 1.2.1) are placed at convenient locations and interconnected with asynchronous links. Asynchronous links may also connect customers of the network to **network attachment points**, as in Figure 1.2.5. This figure shows two attachment points, named A and B, and it is evident that a packet going from A to B may follow any of several different paths, called **routes**, through the network. Choosing a particular path for a packet is known as **routing**. The upper right packet switch has three numbered links connecting it to three other packet switches. The packet coming in on its link #1, which originated at the workstation at attachment point A and is destined for the service at attachment point B, contains the address of its destination. By studying this address, the packet switch will be able to figure out that it should send the packet on its way via its link #3. Choosing an outgoing link is known as **forwarding**, and is usually done by table lookup. The construction of the forwarding tables is one of several methods of routing, so packet switches are also called **forwarders** or **routers**. The resulting organization resembles that of the postal service.

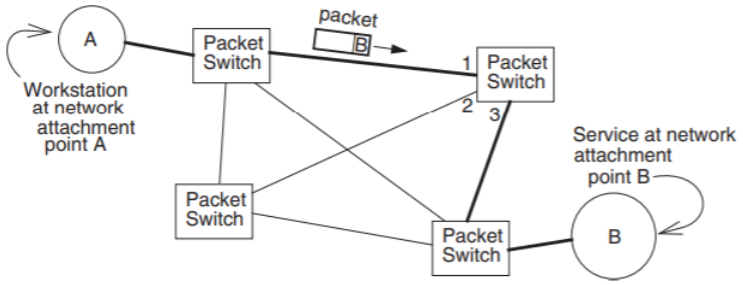


Figure 1.2.5: A packet forwarding network.

A forwarding network imposes a delay (known as its **transit time**) in sending something from A to B. There are four contributions to transit time, several of which may be different from one packet to the next.

1. **Propagation delay**. The time required for the signal to travel across a link is determined by the speed of light in the transmission medium connecting the packet switches and the physical distance the signals travel. Although it does vary slightly with temperature, from the point of view of a network designer propagation delay for any given link can be considered constant. (Propagation delay also applies to the isochronous network.)
2. **Transmission delay**. Since the frame that carries the packet may be long or short, the time required to send the frame at one switch—and receive it at the next switch—depends on the data rate of the link and the length of the frame. This time is known as transmission delay. Although some packet switches are clever enough to begin sending a packet out before completely receiving it (a trick known as **cut-through**), error recovery is simpler if the switch does not forward a packet until the entire packet is present and has passed some validity checks. Each time the packet is transmitted over another link, there is another transmission delay. A packet going from A to B via the dark links in Figure 1.2.5 will thus be subject to four transmission delays: one when A sends it to the first packet switch, one at each forwarding step, and finally one to transmit it to B.
3. **Processing delay**. Each packet switch will have to examine the guidance information in the packet to decide to which outgoing link to send it. The time required to figure this out, together with any other work performed on the packet, such as calculating a checksum (see Sidebar 1.2.1) to allow error detection or copying it to an output buffer that is somewhere else in memory, is

known as processing delay. This delay typically has one part that is relatively constant from one packet to the next and a second part that is proportional to the length of the packet.

Sidebar 1.2.1

Error detection, checksums, and witnesses

A **checksum** on a block of data is a stylized kind of error-detection code in which redundant error-detecting information, rather than being encoded into the data itself (as Chapter 2 will explain), is placed in a separate field. A typical simple checksum algorithm breaks the data block up into k-bit chunks and performs an exclusive OR on the chunks to produce a k-bit result. (When $k = 1$, this procedure is called a **parity check**.) That simple k-bit checksum would catch any one-bit error, but it would miss some two-bit errors, and it would not detect that two chunks of the block have been interchanged. Much more sophisticated checksum algorithms have been devised that can detect multiple-bit errors or that are good at detecting particular kinds of expected errors. As will be seen in Chapter 5, by using cryptographic techniques it is possible to construct a high-quality checksum with the property that it can detect *all* changes—even changes that have been intentionally introduced by a malefactor—with near-certainty. Such a checksum is called a **witness** or **fingerprint** and is useful for ensuring long-term integrity of stored data. The trade-off is that more elaborate checksums usually require more time to calculate and thus add to processing delay. For that reason, communication systems typically use the simplest checksum algorithm that has a reasonable chance of detecting the expected errors.

4. **Queuing delay**. When the packet from A to B arrives at the upper right packet switch, link #3 may already be transmitting another packet, perhaps one that arrived from link #2, and there may also be other packets queued up waiting to use link #3. If so, the packet switch will hold the arriving packet in a queue in memory until it has finished transmitting the earlier packets. The duration of this delay depends on the amount of other traffic passing through that packet switch, so it can be quite variable.

Queuing delay can sometimes be estimated with queuing theory (not discussed in this text). If packets arrive according to a random, memoryless process and have randomly distributed service times (technically, a Poisson distribution in which for this case the service time is the transmission delay of the outgoing link), the average queuing delay, measured in units of the packet service time and including the service time of this packet, will be $1 / (1 - \rho)$. Here ρ is the utilization of the outgoing line, which can range from 0 to 1. When we plot this result in Figure 1.2.6 we notice a typical system phenomenon: delay rises rapidly as the line utilization approaches 100%. This plot tells us that the asynchronous system has introduced a trade-off: if we wish to limit the average queuing delay, for example to the amount labeled in the figure "maximum tolerable delay," it will be necessary to leave unused, on average, some of the capacity of each link; in the example this maximum utilization is labeled ρ_{max}. Alternatively, if we allow the utilization to approach 100%, delays will grow without bound. The asynchronous system seems to have replaced the abrupt appearance of the busy signal of the isochronous network with a gradual trade-off: as the system becomes busier, the delays increase. However, as we will see in Section 1.2.3 below, the replacement is actually more subtle than that.

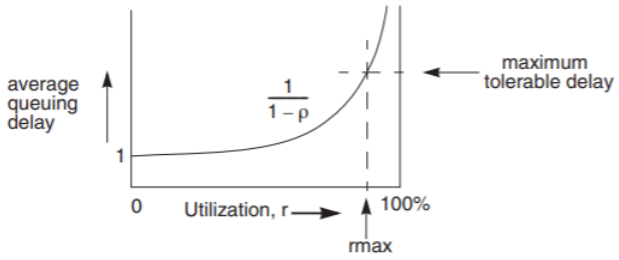


Figure 1.2.6: Queuing delay as a function of utilization.

The formula and accompanying graph tell us only the average delay. If we try to load up a link so that its utilization is ρ_{max}, the actual delay will exceed our tolerance threshold about as often as it is below that threshold. If we are serious about keeping the maximum delay almost always below a given value, we must prepare for occasional worse peaks by holding utilization below the level of ρ_{max} suggested by the figure. If packets do not obey memoryless arrival statistics (for example, they arrive in long convoys, and all are the same, maximum size), the model no longer applies, and we need a better understanding of the arrival

process before we can say anything about delays. This same utilization versus delay trade-off also applies to non-network components of a computer system that have queues: for example, scheduling the processor or reading and writing a magnetic disk.

We have talked about queuing theory as if it might be useful in predicting the behavior of a network. It is not. In practice, network systems put a bound on link queuing delays by limiting the size of queues and by exerting control on arrivals. These mechanisms allow individual links to achieve high utilization levels, while shifting delays to other places in the network. The next section explains how, and it also explains just what happened to the isochronous network's hard-edged busy signal. Later, in Section 1.7 of this chapter, we will see how the delays can be shifted all the way back to the entry point of the network.

1.2.3: Buffer Overflow and Discarded Packets

Continuing for a moment to apply queuing theory, queuing has an implication: buffer space is needed to hold the queue of packets waiting for transmission. How large a buffer should the designer allocate? Under the memoryless arrival interval assumption, the average number of packets awaiting transmission (including the one currently being transmitted) is $1/(1-\rho)$. As with queuing delay, that number is only the average—queuing theory tells us that the variance of the queue length is also $1/(1-\rho)$. For a ρ of 0.8 the average queue length and the variance are both 5, so if one wishes to allow enough buffers to handle peaks that are, say, three standard deviations above the average, one must be prepared to buffer not only the 5 packets predicted as the average but also $(3 \times \sqrt{5} \simeq 7)$ more, a total of 12 packets. Worse, in many real networks packets don't actually arrive independently at random; they come in buffer-bursting batches.

At this point, we can imagine three quite different strategies for choosing a buffer size:

1. *Plan for the worst case*. Examine the network traffic carefully, figure out what the worst-case traffic situation will be, and allocate enough buffers to handle it.
2. *Plan for the usual case and fight back.* Based on a calculation such as the one above, choose a buffer size that will work most of the time, and if the buffers fill up send messages back through the network asking someone to stop sending.
3. *Plan for the usual case and discard overflow.* Again, choose a buffer size that will work most of the time, and ruthlessly discard packets when the buffers are full.

Let's explore these three possibilities in turn.

Buffer memory is usually low in cost, so planning for the worst case seems like an attractive idea, but it is actually much harder than it sounds. For one thing, in a large network, it may be impossible to figure out what the worst case is—there just isn't enough information available about what can happen. Even if one can estimate the worst case, the estimate may not be useful. Consider, for example, the Hypothetical Bank of Canada, which has 21,000 tellers scattered across the country. The branch at Moose Jaw, Saskatchewan, has one teller and usually is the target of only three transactions a day. Although it has never happened, and almost certainly never will, the worst case is that every one of the 20,999 other tellers simultaneously posts a withdrawal against a Moose Jaw account. Thus, a worst-case design would require that there be enough buffers in the packet switch leading to Moose Jaw to handle 20,999 simultaneous messages. The problem with worst-case analysis is that the worst case can be many orders of magnitude larger than the average case, as well as extremely unlikely. Moreover, even if one decided to buy that large a buffer, the resulting queue to process all the transactions would be so long that many of the other tellers would give up in disgust and abort their transactions, so the large buffer wouldn't really help.

This observation makes it sound attractive to choose a buffer size based on typical, rather than worst-case, loads. But then there is always going to be a chance that traffic will exceed the average for long enough to run out of buffer space. This situation is called **congestion**. What to do then?

One idea is to push back. If buffer space begins to run low, send a message back along an incoming link saying "please don't send any more until you hear from me." This message (called a **quench** request) may go to the packet switch at the other end of that link, or it may go all the way back to the original source that introduced the data into the network. Either way, pushing back is also harder than it sounds. If a packet switch is experiencing congestion, there is a good chance that the adjacent switch is also congested (if it is not already congested, it soon will be if it is told to stop sending data over the link to this switch), and sending an extra message is adding to the congestion. Worse, a set of packet switches configured in a cycle like that of Figure 1.2.5 can easily end up in a form of deadlock (called gridlock when it happens to automobile traffic), with all buffers filled and each switch waiting for the next switch to say that it is OK to start sending again.

One way to avoid deadlock among the packet switches is to send the quench request all the way back to the source. This method is hard too, for at least three reasons. First, it may not be clear which source should be the destination of the quench. In our Moose Jaw example, there are 21,000 different sources, no one of which is, by itself, the cause of (nor capable of doing much about) the problem. Second, such a request may not have any effect because the source you choose to quench is no longer sending anyway. Again in our example, by the time the packet switch on the way to Moose Jaw detects the overload, all of the 21,000 tellers may have already sent their transaction requests, so asking them not to send anything else would accomplish nothing. Third, assuming that the quench message is itself forwarded back through the packet-switched network, it may run into congestion and be subject to queuing delays. The busier the network, the longer it will take to exert control. We are proposing to create a feedback system with delay and should expect to see oscillations. Even if all the data is coming from one source, by the time the quench gets back and the source acts on it, the packets already in the pipeline may exceed the buffer capacity. Controlling congestion by quenching either the adjacent switch or the source is used in various special situations, but as a general technique it is currently an unsolved problem.

The remaining possibility is what most packet networks actually do in the face of congestion: when the buffers fill up, they start throwing packets away. This seems like a somewhat startling thing for a communication system to do because it will disrupt the communication, and eventually each discarded packet will have to be sent again, so the effort to send the packet this far will have been wasted. Nevertheless, this is an action that every packet switching network that is not configured for the worst case must be prepared to take.

Overflowing buffers and discarded packets lead to two remarkable consequences. First, the sender of a packet can interpret the lack of its acknowledgment as a sign that the network is congested, and can in turn reduce the rate at which it introduces new packets into the network. This idea, called **automatic rate adaptation**, is explored in depth in Section 1.7 of this chapter. The combination of discarded packets and automatic rate adaptation in turn produce the second consequence: simple theoretical models of network behavior based on standard queuing theory do not apply when a service may serve some requests and may discard others. Modeling of networks that have rate adaptation requires a much deeper understanding of the specific algorithms used not just by the network but also by network applications.

In the final analysis, the asynchronous network replaces the hard-edged blocking of the isochronous network with a variable transmission rate that depends on the instantaneous network load. Which scheme for dealing with overload (asynchronous or isochronous) is preferable depends on the application. For some applications it may be better to be told at the outset of a communications attempt to come back later, rather than to be allowed to start work only to encounter such variations in available capacity that it is hard to do anything useful. In other applications it may be more helpful to have some work done, slowly or at variable rates, rather than none at all.

The possibility that a network may actually discard packets to cope with congestion leads to a useful distinction between two kinds of forwarding networks. So far, we have been discussing what is usually described as a **best-effort** network, which, if it cannot dispatch a packet soon after receipt, may discard it. The alternative design is the **guaranteed-delivery** network (sometimes called a **store-and-forward** network, although that term is often applied to all forwarding networks), which takes heroic measures to avoid ever discarding payload data. Guaranteed-delivery networks usually are designed to work with complete messages rather than packets. Typically, a guaranteed-delivery network uses non-volatile storage such as a magnetic disk for buffering, so that it can handle large peaks of message load and can be confident that messages will not be lost even if there is a power failure or the forwarding computer crashes. Also, a guaranteed-delivery network usually, when faced with the prospect of being completely unable to deliver a message (perhaps because the intended recipient has vanished), explicitly returns the message to its originator along with an explanation of why delivery failed. Finally, in keeping with the spirit of not losing a message, a guaranteed-delivery switch usually tracks individual messages carefully to make sure that none are lost or damaged during transmission, for example by a burst of noise. A switch of a best-effort network can be quite a bit simpler than a switch of a guaranteed-delivery network. Since the best-effort network may casually discard packets anyway, it does not need to make any special provisions for retransmitting damaged packets, for preserving packets in transit when the switch crashes and restarts, or for worrying about the case when the link to a destination node suddenly stops accepting data.

The best-effort network is said to provide a **best-effort contract** to its customers (this contract is defined more carefully in Section 1.2.7 below), rather than a guarantee of delivery. Of course, in the real world there are no absolute guarantees—the real distinction between the two designs is that there is intended to be a significant difference in the probability of undetected loss. When we examine network layering in Section 1.3 of this chapter, it will become apparent that these differences can be characterized another way: guaranteed-delivery networks are usually implemented in a higher network layer and best-effort networks in a lower network layer.

In these terms, the U.S. Postal Service operates a guaranteed delivery system for first-class mail, but a best-effort system for third-class (junk) mail, because postal regulations allow it to discard third-class mail that is misaddressed or when congestion gets out of hand. The Internet is organized as a best-effort system, but the Internet mechanisms for handling email are designed as a guaranteed delivery system. The Western Union company has always prided itself on operating a true guaranteed-delivery system, to the extent that when it decommissions an office it normally disassembles the site completely in a search for misplaced telegrams. There is a (possibly apocryphal) tale that such a disassembly once discovered a 75-year-old telegram that had fallen behind a water pipe. The company promptly delivered it to the astonished heirs of the original addressee.

1.2.4: Duplicate Packets and Duplicate Suppression

As it turns out, discarded packets are not as much of a problem to the higher-level application as one might expect because when a client sends a request to a service, it is always possible that the service is not available, or the service crashed just after receiving the request. So unanswered requests are actually a routine occurrence, and many network protocols include some kind of timer expiration and resend mechanism to recover from such failures. The timing diagram of Figure 1.2.7 illustrates the situation, showing a first packet carrying a request, followed by a packet going the other way carrying the response to the first request. A has set a timer, indicated by a vertical line, but the arrival of response 1 before the expiration of the timer causes A to switch off the timer, indicated by the small *x*. The packet carrying the second request is lost in transit (as indicated by the large *X*), perhaps having been damaged or discarded by an overloaded forwarder; the timer expires; and A resends request 2 in the packet labeled request 2'.

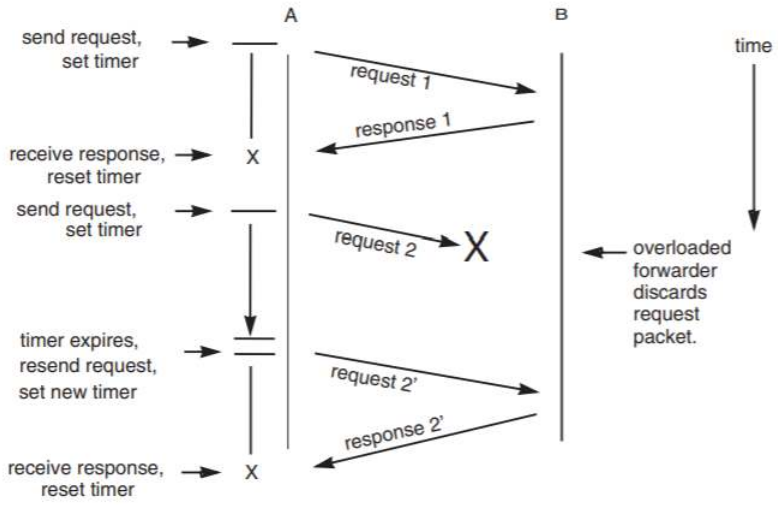


Figure 1.2.7: Lost packet recovery.

When a congested forwarder discards a packet, there are two important consequences. First, the client doesn't receive a response as quickly as originally hoped because a timer expiration period has been added to the overall response time. This extra delay can have a significant impact on performance. Second, users of the network must be prepared for *duplicate* requests and responses. The reason lies in the recovery mechanism just described. Suppose a network packet switch gets overloaded and must discard a response packet, as in Figure 1.2.8. Client A can't tell the difference between this case and the case of Figure 1.2.7, so it resends its request. The service sees this resent request as a duplicate. Suppose B does not realize this is a duplicate, does what is requested, and sends back a response. Client A receives the response and assumes that everything is OK. That may be a correct assumption, or it may not, depending on whether or not the first arrival of request 3 changed B's state. If B is a spelling checker, it will probably give the same response to both copies of the request. But if B is a bank and the request is to transfer funds, doing the request twice would be a mistake. So detecting duplicates may or may not be important, depending on the particular application.

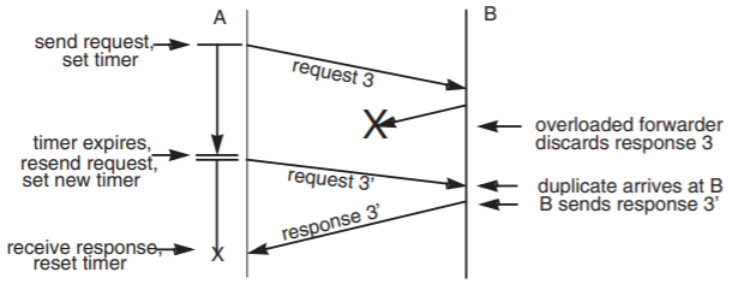


Figure 1.2.8: Lost packet recovery leading to duplicate request.

For another example, if for some reason the network delays pile up and exceed the resend timer expiration period, the client may resend a request even though the original response is still in transit. Since B can't tell any difference between this case and the previous one, it responds in the same way, by doing what is requested. But now A receives a duplicate response, as in Figure 1.2.9. Again, this duplicate may or may not matter to A, but at a minimum A must take steps not to be confused by the arrival of a duplicate response.

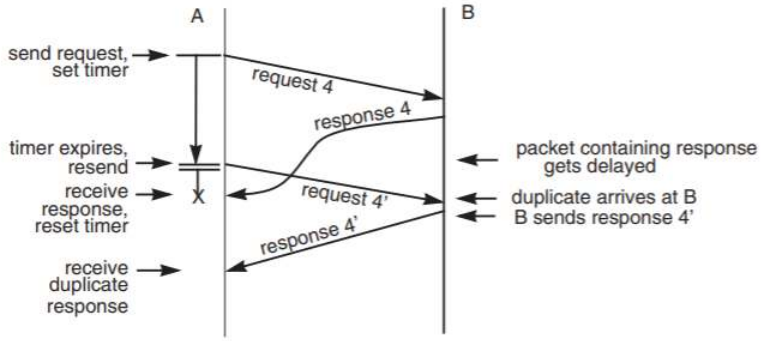


Figure 1.2.9: Network delay combined with recovery leading to duplicate response.

What if the arrival of a request from A causes B to change state, as in the bank transfer example? If so, it is usually important to detect and suppress duplicates generated by the lost packet recovery mechanism. The general procedure to suppress duplicates has two components. The first component is hinted at by the request and response numbers used in the illustrations: each request includes a **nonce**, which is a unique identifier that will never be reused by A when sending requests to B. The illustration uses monotonically increasing serial numbers as nonces, but any unique identifier will do. The second duplicate suppression component is that B must maintain a list of nonces on which it has taken action or is still working, and whenever a request arrives B should look through this list to see whether or not this apparently new request is actually a duplicate of one previously received. If it is a duplicate, B must not perform the action requested. On the other hand, B should not simply ignore the request, either, because the reason for the duplicate may be that A never received B's response. So, B needs some way of reconstructing and resending that previous response. The simplest way of doing this is usually for B to add to its list of previously handled nonces a copy of the corresponding responses so that it can easily resend them. Thus in Figure 1.2.9, the last action of B should be replaced with "*B resends response 4*".

In some network designs, A may even receive duplicate responses to a single, unrepeated request. The reason is that a forwarding link deep inside the network may be using a timer expiration and resend protocol similar to the one above. For this reason, most protocols that are concerned about duplicate suppression include a copy of the nonce in the response, and the originator, A, maintains a list of nonces used in its outstanding requests. When a response comes back, A can check for the nonce in the list and delete that list entry or, if there is no list entry, assume it is a duplicate of a previously received response and ignore it.

The procedure we have just described allows A to keep its list of nonces short, but B might have to maintain an ever-growing list of nonces and responses to be certain that it never accidentally processes a request twice. A related problem concerns what happens if either participant crashes and restarts, losing its volatile memory, which is probably where it is keeping its list of nonces. Refinements to cope with these problems will be explored in detail when we revisit the topic of duplicate suppression in Section 1.6.3 of this chapter.

Ensuring suppression of duplicates is a significant complication. Therefore, if possible it is wise to design the service and its protocol in such a way that suppression is not required. Recall that the reason that duplicate suppression became important was that a request changed the state of the service. It is often possible to design a service interface so that it is idempotent, which for a network request means that repeating the same request or sequence of requests several times has the same effect as doing it just once. This design approach is explored in depth in the discussion of atomicity and error recovery in Chapter 3.

1.2.5: Damaged Packets and Broken Links

At the beginning of the chapter we noted that noise is one of the fundamental considerations that dominates the design of data communication. Data can be damaged during transmission, during transit through a switch, or in the memory of a forwarding node. Noise, transmission errors, and techniques for detecting and correcting errors are fascinating topics in their own right, explored in some depth in Chapter 2. As a general rule it is possible to sub-contract this area to a specialist in the theory of error detection and correction, with one requirement in the contract: when we receive data, we want to know whether or not it is correct. That is, we require that a reliable error detection mechanism be part of any underlying data transmission system. Section 1.4 of this chapter expands a bit on this error detection requirement.

Once we have contracted for data transmission with an error detection mechanism in which we have confidence, intermediate packet switches can then handle noise-damaged packets by simply discarding them. This approach changes the noise problem into one for which there is already a recovery procedure. Put another way, this approach transforms data loss into performance degradation.

Finally, because transmission links traverse hostile environments and must be considered fragile, a packet network usually has multiple interconnection paths, as in Figure 1.2.5. Links can go down while transmitting a frame; they may stay down briefly, e.g. because of a power interruption, or for long periods of time while waiting for someone to dig up a street or launch a replacement satellite. Flexibility in routing is an important property of a network of any size. We will return to the implications of broken links in the discussion of the network layer, in Section 1.5 of this chapter.

1.2.6: Reordered Delivery

When a packet-forwarding network has an interconnection topology like that of Figure 1.2.5, in which there is more than one path that a packet can follow from A to B, there is a possibility that a series of packets departing from A in sequential order may arrive at B in a different order. Some networks take special precautions to avoid this possibility by forcing all packets between the same two points to take the same path or by delaying delivery at the destination until all earlier packets have arrived. Both of these techniques introduce additional delay, and there are applications for which reducing delay is more important than receiving the segments of a message in the order in which they were transmitted.

Recalling that a message may have been divided into segments, the possibility of reordered delivery means that reassembly of the original message requires close attention. We have here a model of communication much like when a friend is touring on holiday by car, stopping each night in a different motel, and sending a motel postcard with an account of the day's adventures. Whenever a day's story doesn't fit on one card, your friend uses two or three postcards, as necessary. The Post Office may deliver these cards to you in almost any order, and something on the postcard—probably the date—will be needed to enable you to read them in the proper order. Even when two cards are mailed at the same time from the same motel (as indicated by the motel photograph on the front) the Post Office may deliver them to you on different days, so there must be further information on the postcard to allow you to realize that sender broke the original message into segments and you may need to wait for the next delivery before starting to read.

1.2.7: Summary of Interesting Properties and the Best-Effort Contract

Most of the ideas introduced in this section can be captured in just two illustrations. Figure 1.2.10 summarizes the differences in application characteristics and in response to overload between isochronous and asynchronous multiplexing.

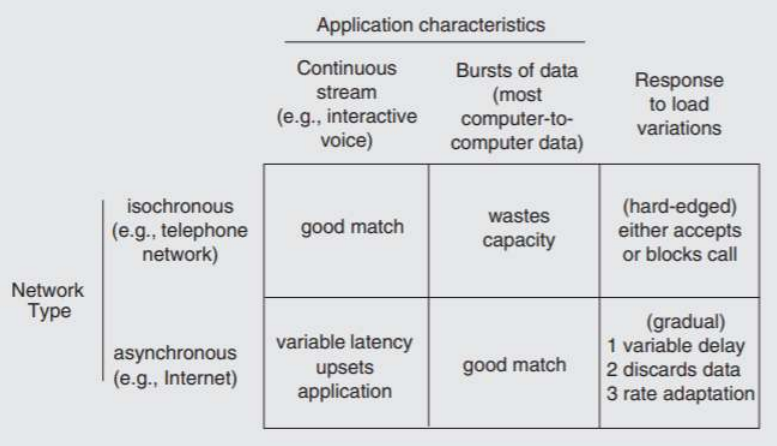


Figure 1.2.10: Isochronous versus asynchronous multiplexing.

Similarly, Figure 1.2.11 briefly summarizes the interesting (the term "challenging" may also come to mind) properties of computer networks that we have encountered. The "best-effort contract" of the caption means that when a network accepts a segment, it offers the expectation that it will usually deliver the segment to its destination, but it does not guarantee success, and the client of the network is expected to be sophisticated enough to take in stride the possibility that segments may be lost, duplicated, variably delayed, or delivered out of order.

Figure 1.2.11:

1. Networks encounter a vast range of:
   - Data rates
   - Propagation, transmission, queuing, and processing delays
   - Loads
   - Numbers of users
2. Networks traverse hostile environments
   - Noise damages data
   - Links stop working
3. Best-effort networks have
   - Variable delays
   - Variable transmission rates
   - Discarded packets
   - Duplicate packets
   - Maximum packet length
   - Reordered delivery

This page titled 1.2: Interesting Properties of Networks is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

1.3: Getting Organized - Layers

To deal with the interesting properties of networks that we identified in Section 1.2, it is necessary to get organized. The primary organizing tool for networks is an example of the design principle *adopt sweeping simplifications*. All networks use the divide-and-conquer technique known as **layering of protocols**. But before we come to layers, we must establish what a protocol is.

Suppose we are examining the set of programs used by a defense contractor who is retooling for a new business, video games. In the main program we find the procedure call

```
FIRE (#_of_missiles, target, action_if_defended)
```

and elsewhere we find the corresponding procedure, which begins

```
procedure FIRE (nmissiles, where, reaction)
```

These constructs are interpreted at two levels. First, the system matches the name `FIRE` in the main program with the program that exports a procedure of the same name, and it arranges to transfer control from the main program to that procedure. The procedure, in turn, matches the arguments of the calling program, position by position, with its own parameters. Thus, in this example, the second argument `target` of the calling program is matched with the second parameter `where` of the called procedure. Beyond this mechanical matching, there is an implicit agreement between the programmer of the main program and the programmer of the procedure that this second argument is to be interpreted as the location that the missiles are intended to hit.

This set of agreements on how to interpret both the order and the meaning of the arguments stands as a kind of contract between the two programs. In programming languages, such contracts are called “specifications”; in networks, such contracts are called **protocols**. More generally, a protocol goes beyond just the interpretation of the arguments; it encompasses everything that either of the two parties can depend on about how the other will act or react. For example, in a client/service system, a request/response protocol might specify that the service send an immediate acknowledgment when it gets a request, so that the client knows that the service is there, and send the eventual response as a third message.

Let us suppose that our defense contractor wishes to further convert the software from a single-user game to a multiuser game, using a client/service organization. The main program will run as a client and the `FIRE` program will now run in a multiclient, game-coordinating service. To simplify the conversion, the contractor has chosen to use the remote procedure call (RPC) protocol illustrated in Figure 1.3.1. A stub procedure that runs in the client machine exports the name `FIRE` so that when the main program calls `FIRE` , control actually passes to the stub with that name. The stub collects the arguments, marshals them into a request message, and sends them over the network to the game-coordinating service. At the service, a corresponding stub waits for such a request to arrive, unmarshals the arguments in the request message, and uses them to perform a call to the real `FIRE` procedure. When `FIRE` completes its operation and returns, the service stub marshals any output value into a response message and sends it to the client. The client stub waits for this response message, and when it arrives, it unmarshals the return value in the response message and returns it as its own value to the main program. The procedure call protocol has been honored and the main program continues as if the procedure named `FIRE` had executed locally.

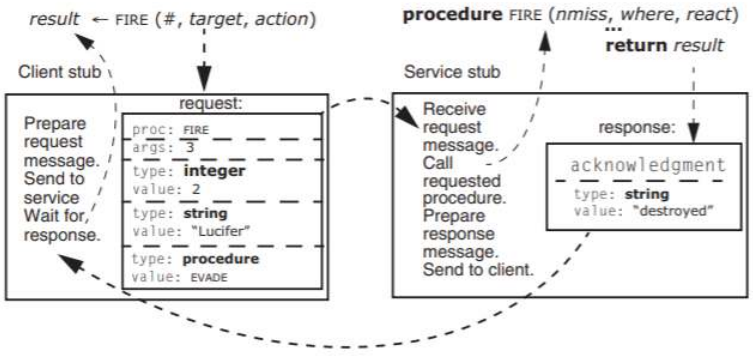


Figure 1.3.1: A remote procedure call.

Figure 1.3.1 also illustrates a second, somewhat different, protocol between the client stub and the service stub, as compared with the protocol between the main program and the procedure it calls. Between the two stubs the request message spells out the name of the procedure to be called, the number of arguments, and the types of each argument. The details of the protocol between the

RPC stubs need have little in common with the corresponding details of the protocol between the original main program and the procedure it calls.

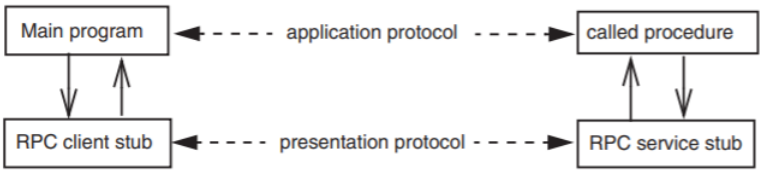


Figure 1.3.2: Two protocol layers.

1.3.1: Layers

In that example, the independence of the `MAIN` -to- `FIRE` procedure call protocol from the RPC stub-to-stub protocol is characteristic of a layered design. We can make those layers explicit by redrawing our picture as in Figure 1.3.2. The contract between the main program and the procedure it calls is called the **application protocol**. The contract between the client-side and service-side RPC stubs protocol is known as a **presentation protocol** because it translates data formats and semantics to and from locally preferred forms.

The request message must get from the client RPC stub to the service RPC stub. To communicate, the client stub calls some network procedure, using an elaboration of the `SEND` abstraction:

```
SEND_MESSAGE (request_message, service_name)
```

specifying in a second argument the identity of the service that should receive this request message. The service stub invokes a similar procedure that provides the `RECEIVE` abstraction to pick up the message. These two procedures represent a third layer, which provides a **transport protocol**, and we can extend our layered protocol picture as in Figure 1.3.3.

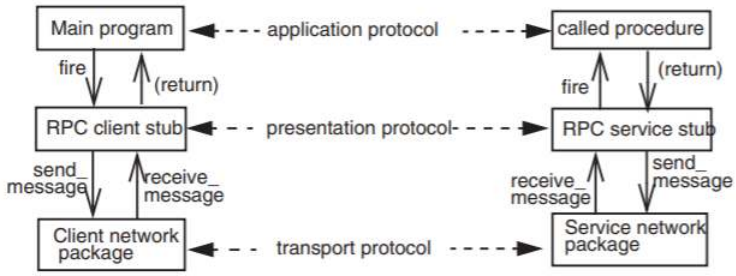


Figure 1.3.3: Three protocol layers.

This figure makes apparent an important property of layering as used in network designs: every module has not two, but *three* interfaces. In the usual layered organization, a module has just two interfaces, an interface to the layer above, which hides a second interface to the layer below. But as used in a network, layering involves a third interface. Consider, for example, the RPC client stub in the figure. As expected, it provides an interface that the main program can use, and it uses an interface of the client network package below. But the whole point of the RPC client stub is to construct a request message that convinces its correspondent stub at the service to do something. The presentation protocol thus represents a third interface of the presentation layer module. The presentation module thus hides both the lower layer interface and the presentation protocol from the layer above. This observation is a general one—each layer in a network implementation provides an interface to the layer above, and it hides the interface to the layer below as well as the protocol interface to the correspondent with which it communicates.

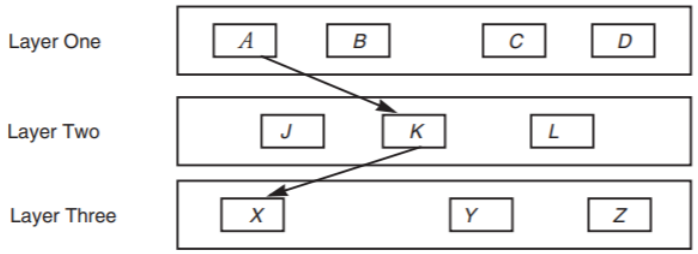


Figure 1.3.4: A layered system.

Layered design has proven to be especially effective, and it is used in some form in virtually every network implementation. The primary idea of layers is that each layer hides the operation of the layer below from the layer above, and instead provides its own interpretation of all the important features of the lower layer. Every module is assigned to some layer, and interconnections are restricted to go between modules in adjacent layers. Thus in the three-layer system of Figure 1.3.4, module A may call any of the modules J, K, or L, but A doesn't even know of the existence of X, Y, and Z. The figure shows A using module K. Module K, in turn, may call any of X, Y, or Z.

Different network designs, of course, will have different layering strategies. The particular layers we have discussed are only an illustration—as we investigate the design of the transport protocol of Figure 1.3.3 in more detail, we will find it useful to impose further layers, using a three-layer reference model that provides quite a bit of insight into how networks are organized. Our choice strongly resembles the layering that is used in the design of the Internet. The three layers we choose divide the problem of implementing a network as follows (from the bottom up):

- The **link layer**: moving data directly from one point to another.
- The **network layer**: forwarding data through intermediate points to move it to the place it is wanted.
- The **end-to-end layer**: everything else required to provide a comfortable application interface.

The application itself can be thought of as a fourth, highest layer, not part of the network. On the other hand, some applications intertwine themselves so thoroughly with the end-to-end layer that it is hard to make a distinction.

The terms **frame**, **packet**, **segment**, **message**, and **stream** that were introduced in Section 1.2 can now be identified with these layers. Each is the unit of transmission of one of the protocol layers. Working from the top down, an application starts by asking the end-to-end layer to transmit a **message** or a **stream** of data to a correspondent. The end-to-end layer splits long messages and streams into **segments**, it copes with lost or duplicated segments, it places arriving segments in proper order, it enforces specific communication semantics, it performs presentation transformations, and it calls on the network layer to transmit each segment. The network layer accepts segments from the end-to-end layer, constructs **packets**, and transmits those packets across the network, choosing which links to follow to move a given packet from its origin to its destination. The link layer accepts packets from the network layer, and constructs and transmits **frames** across a single link between two forwarders or between a forwarder and a customer of the network.

Some network designs attempt to impose a strict layering among various parts of what we call the end-to-end layer, but it is often such a hodgepodge of function that no single layering can describe it in a useful way. On the other hand, the network and link layers are encountered frequently enough in data communication networks that one can almost consider them universal.

With this high-level model in mind, we next sketch the basic contracts for each of the three layers and show how they relate to one another. Later, we examine in much more depth how each of the three layers is actually implemented.

1.3.2: The Link Layer

At the bottom of a packet-switched network there must be some underlying communication mechanism that connects one packet switch with another or a packet switch to a customer of the network. The link layer is responsible for managing this low-level communication. The goal of the link layer is to move the bits of the packet across one (usually, but not necessarily, physical) link, hiding the particular mechanics of data transmission that are involved.

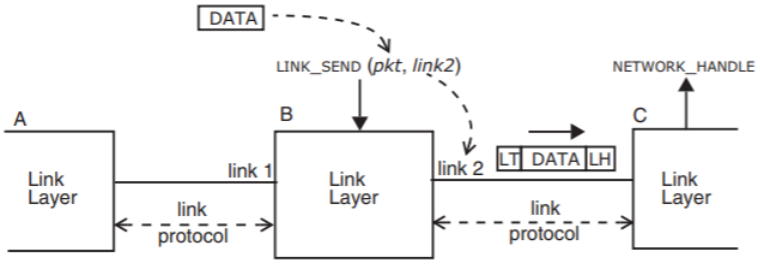


Figure 1.3.5: A link layer in a packet switch that has two physical links.

A typical, somewhat simplified interface to the link layer looks something like this:

```
LINK_SEND (data_buffer, link_identifier)
```

where *data_buffer* names a place in memory that contains a packet of information ready to be transmitted, and *link_identifier* names, in a local address space, one of possibly several links to use. Figure 1.3.5 illustrates the link layer in packet switch B, which has links to two other packet switches, A and C. The call to the link layer identifies a packet buffer named `pkt` and specifies that the link layer should place the packet in a frame suitable for transmission over `link2`, the link to packet switch C. Switches B and C both have implementations of the link layer, a program that knows the particular protocol used to send and receive frames on this link. The link layer may use a different protocol when sending a frame to switch A using link number 1. Nevertheless, the link layer typically presents a uniform interface (`LINK_SEND`) to higher layers. Packet switch B and packet switch C may use different labels for the link that connects them. If packet switch C has four links, the frame may arrive on what C considers to be its link number 3. The link identifier is thus a name whose scope is limited to one packet switch.

The data that actually appears on the physical wire is usually somewhat different from the data that appeared in the packet buffer at the interface to the link layer. The link layer is responsible for taking into account any special properties of the underlying physical channel, so it may, for example, encode the data in a way that is less fragile in the local noise environment, it may fragment the data because the link protocol requires shorter frames, and it may repeatedly resend the data until the other end of the link acknowledges that it has received it.

These channel-specific measures generally require that the link layer add information to the data provided by the network layer. In a layered communication system, the data passed from an upper layer to a lower layer for transmission is known as the **payload**. When a lower layer adds to the front of the payload some data intended only for the use of the corresponding lower layer at the other end, the addition is called a **header**, and when the lower layer adds something to the end, the addition is called a **trailer**. In Figure 1.3.5, the link layer has added a link layer header `LH` (perhaps indicating which network layer program to deliver the packet to) and a link layer trailer `LT` (perhaps containing a checksum for error detection). The combination of the header, payload, and trailer becomes the link-layer frame. The receiving link layer module will, after establishing that the frame has been correctly received, remove the link layer header and trailer before passing the payload to the network layer.

The particular method of waiting for a frame, packet, or message to arrive and transferring payload data and control from a lower layer to an upper layer depends on the available thread coordination procedures. Throughout this chapter, rather than having an upper layer call down to a lower-layer procedure named `RECEIVE`, we use **upcalls**, which means that when data arrives, the lower layer makes a procedure call up to an entry point in the higher layer. Thus in Figure 1.3.5, the link layer calls a procedure named `NETWORK_HANDLE` in the layer above.

1.3.3: The Network Layer

A segment enters a forwarding network at one of its **network attachment points** (the **source**), accompanied by instructions to deliver it to another network attachment point (the **destination**). To reach the destination it will probably have to traverse several links. Providing a systematic naming scheme for network attachment points, determining which links to traverse, creating a packet that contains the segment, and forwarding the packet along the intended path are the jobs of the network layer. The interface to the network layer, again somewhat simplified, resembles that of the link layer:

```
NETWORK_SEND (segment_buffer, network_identifier, destination)
```

The NETWORK_SEND procedure transmits the segment found in *segment_buffer* (the payload, from the point of view of the network layer), using the network named in *network_identifier* (a single computer may participate in more than one network), to *destination* (the address within that network that names the network attachment point to which the segment should be delivered).

The network layer, upon receiving this call, creates a network-layer header, labeled NH in Figure 1.3.6, and/or trailer, labeled NT , to accompany the segment as it traverses the network named " IP ", and it assembles these components into a packet. The key item of information in the network-layer header is the address of the destination, for use by the next packet switch in the forwarding chain.

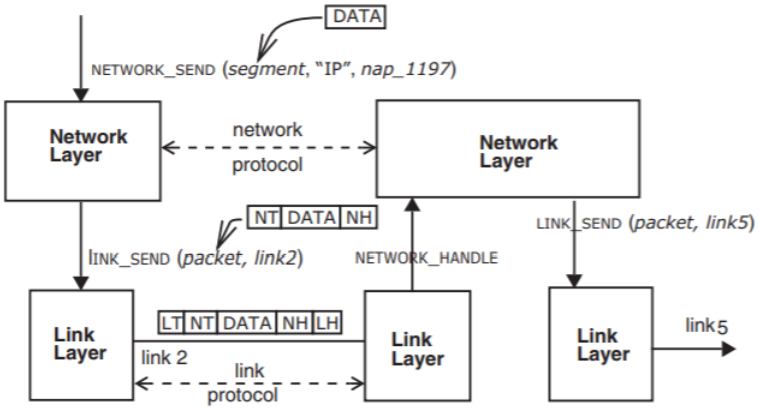


Figure 1.3.6: Relation between the network layer and the link layer.

Next, the network layer consults its tables to choose the most appropriate link over which to send this packet with the goal of getting it closer to its destination. Finally, the network layer calls the link layer asking it to send the packet over the chosen link. When the frame containing the packet arrives at the other end of the link, the receiving link layer strips off the link layer header and trailer (LH and LT in the figure) and hands the packet to its network layer by an upcall to NETWORK_HANDLE . This network layer module examines the network layer header and trailer to determine the intended destination of the packet. It consults its own tables to decide on which outgoing link to forward the packet, and it calls the link layer to send the packet on its way. The network layer of each packet switch along the way repeats this procedure, until the packet traverses the link to its destination. The network layer at the end of that link recognizes that the packet is now at its destination, it extracts the data segment from the packet, and passes that segment to the end-to-end layer, with another upcall.

1.3.4: The End-to-End Layer

We can now put the whole picture together. The network and link layers together provide a best-effort network, which has the "interesting" properties that were listed in Figure 1.2.11. These properties may be problematic to an application, and the function of the end-to-end layer is to create a less “interesting” and thus easier to use interface for the application. For example, Figure 1.3.7 shows the remote procedure call of Figure 1.3.1 from a different perspective. Here the RPC protocol is viewed as an end-to-end layer of a complete network implementation. As with the lower layers, the end-to-end layer has added a header and a trailer to the data that the application gave it, and inspecting the bits on the wire we now see three distinct headers and trailers, corresponding to the three layers of the network implementation.

The RPC implementation in the end-to-end layer provides several distinct end-toend services, each intended to hide some aspect of the underlying network from its application:

- **Presentation services.** Translating data formats and emulating the semantics of a procedure call. For this purpose the end-to-end header might contain, for example, a count of the number of arguments in the procedure call.
- **Transport services.** Dividing streams and messages into segments and dealing with lost, duplicated, and out-of-order segments. For this purpose, the end-to-end header might contain serial numbers of the segments.
- **Session services.** Negotiating a search, handshake, and binding sequence to locate and prepare to use a service that knows how to perform the requested procedure. For this purpose, the end-to-end header might contain a unique identifier that tells the service which client application is making this call.

Depending on the requirements of the application, different end-to-end layer implementations may provide all, some, or none of these services, and the end-to-end header and trailer may contain various different bits of information.

There is one other important property of this layering that becomes evident in examining Figure 1.3.7. Each layer considers the payload transmitted by the layer above to be information that it is not expected, or even permitted, to interpret. Thus the end-to-end layer constructs a segment with an end-to-end header and trailer that it hands to the network layer, with the expectation that the network layer will not look inside or perform any actions that require interpretation of the segment. The network layer, in turn, adds a network-layer header and trailer and hands the resulting packet to the link layer, again with the expectation that the link layer will consider this packet to be an opaque string of bits, a payload to be carried in a link-layer frame. Violation of this rule would lead to interdependence across layers and consequent loss of modularity of the system.

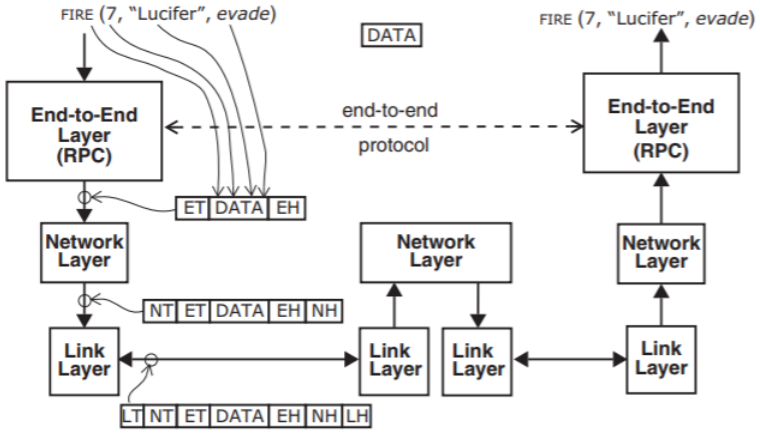


Figure 1.3.7: Three network layers in action.The arguments of the procedure call become the payload of the end-to-end segment. The network layer forwards the packet across two links on the way from the client to the service.The frame on the wire contains the headers and trailers of three layers.

1.3.5: Additional Layers and the End-to-End Argument

To this point, we have suggested that a three-layer reference model is both necessary and sufficient to provide insight into how networks operate. Standard textbooks on network design and implementation mention a reference model from the International Organization for Standardization, known as "Open Systems Interconnect", or OSI. The OSI reference model has not three, but seven layers. What is the difference?

There are several differences. Some are trivial; for example, the OSI reference model divides the link layer into a strategy layer (known as the "data link layer") and a physical layer, recognizing that many different kinds of physical links can be managed with a small number of management strategies. There is a much more significant difference between our reference model and the OSI reference model in the upper layers. The OSI reference model systematically divides our end-to-end layer into four distinct layers. Three of these layers directly correspond, in the RPC example, to the layers of Figure 1.3.3: an application layer, a presentation layer, and a transport layer. In addition, just above the transport layer, the ISO model inserts a layer that provides the session services mentioned just above.

We have avoided this approach for the simple reason that different applications have radically different requirements for transport, session, and presentation services—even to the extent that the order in which they should be applied may be different. This situation makes it difficult to propose any single layering, since a layering implies an ordering.

For example, an application that consists of sending a file to a printer would find most useful a transport service that guarantees to deliver to the printer a stream of bytes in the same order in which they were sent, with none missing and none duplicated. But a file transfer application might not care in what order different blocks of the file are delivered, so long as they all eventually arrive at the destination. A digital telephone application would like to see a stream of bits representing successive samples of the sound waveform delivered in proper order, but here and there a few samples can be missing without interfering with the intelligibility of the conversation. This rather wide range of application requirements suggests that any implementation decisions that a lower layer makes (for example, to wait for out-of-order segments to arrive so that data can be delivered in the correct order to the next higher

layer) may be counterproductive for at least some applications. Instead, it is likely to be more effective to provide a library of service modules that can be selected and organized by the programmer of a specific application. Thus, our end-to-end layer is an unstructured library of service modules, of which the RPC protocol is an example.

This argument against additional layers is an example of a design principle known as

> **The End-to-End Argument**
>
> The application knows best.

In this case, the basic thrust of the end-to-end argument is that the application knows best what its real communication requirements are, and for a lower network layer to try to implement any feature other than transporting the data risks implementing something that isn't quite what the application needed. Moreover, if it isn't exactly what is needed, the application will probably have to re-implement that function on its own. The end-to-end argument can thus be paraphrased as: *don't bury it in a lower layer, let the end points deal with it because they know best what they need.*

A simple example of this phenomenon is file transfer. To transfer a file carefully, the appropriate method is to calculate a checksum from the contents of the file as it is stored in the file system of the originating site. Then, after the file has been transferred and written to the new file system, the receiving site should read the file back out of its file system, recalculate the checksum anew, and compare it with the original checksum. If the two checksums are the same, the file transfer application has quite a bit of confidence that the new site has a correct copy; if they are different, something went wrong and recovery is needed.

Given this end-to-end approach to checking the accuracy of the file transfer, one can question whether or not there is any value in, for example, having the link layer protocol add a frame checksum to the link layer trailer. This link layer checksum takes time to calculate, it adds to the data to be sent, and it verifies the correctness of the data only while it is being transmitted across that link. Despite this protection, the data may still be damaged while it is being passed through the network layer, or while it is buffered by the receiving part of the file transfer application, or while it is being written to the disk. Because of those threats, the careful file transfer application cannot avoid calculating its end-to-end checksum, despite the protection provided by the link layer checksum.

This is not to say that the link layer checksum is worthless. If the link layer provides a checksum, that layer will discover data transmission errors at a time when they can be easily corrected by resending just one frame. Absent this link-layer checksum, a transmission error will not be discovered until the end-to-end layer verifies its checksum, by which point it may be necessary to redo the entire file transfer. So there may be a significant performance gain in having this feature in a lower-level layer. The interesting observation is that a lower-layer checksum does *not* eliminate the need for the application layer to implement the function, and it is thus *not* required for application correctness. It is just a performance enhancement.

The end-to-end argument can be applied to a variety of system design issues in addition to network design. It does not provide an absolute decision technique, but rather a useful argument that should be weighed against other arguments in deciding where to place function.

1.3.6: Mapped and Recursive Applications of the Layered Model

When one begins decomposing a particular existing network into link, network, and end-to-end layers, it sometimes becomes apparent that some of the layers of the network are themselves composed of what are obviously link, network, or end-to-end layers. These compositions come in two forms: mapped and recursive.

Mapped composition occurs when a network layer is built directly on another network layer by mapping higher-layer network addresses to lower-layer network addresses. A typical application for mapping arises when a better or more popular network technology comes along, yet it is desirable to keep running applications that are designed for the old network. For example, Apple designed a network called Appletalk that was used for many years, and then later mapped the Appletalk network layer to the Ethernet, which, as described in Section 1.9, has a network and link layer of its own but uses a somewhat different scheme for its network layer addresses.

Another application for mapped composition is to interconnect several independently designed network layers, a scheme called **internetworking**. Probably the best example of internetworking is the Internet itself (described in Sidebar 1.3.1), which links together many different network layers by mapping them all to a universal network layer that uses a protocol known as Internet protocol (IP). Section 1.9 explains how the network layer addresses of the Ethernet are mapped to and from the IP addresses of the

Internet using what is known as an Address Resolution Protocol. The Internet also maps the internal network addresses of many other networks—wireless networks, satellite networks, cable TV networks, etc.—into IP addresses.

Sidebar 1.3.1

The Internet

The Internet provides examples of nearly every concept in this chapter. Much of the Internet is a network layer that is mapped onto some other network layer such as a satellite network, a wireless network, or an Ethernet. Internet protocol (IP) is the primary network layer protocol, but it is not the only network layer protocol used in the Internet. There is a network layer protocol for managing the Internet, known as ICMP. There are also several different network layer routing protocols, some providing routing within small parts of the Internet, others providing routing between major regions. But every point that can be reached via the Internet implements IP.

The link layer of the Internet includes all of the link layers of the networks that the Internet maps onto and it also includes many separate, specialized links: a wire, a dial-up telephone line, a dedicated line provided by the telephone company, a microwave link, a digital subscriber line (DSL), a free-space optical link, etc. Almost anything that carries bits has been used somewhere as a link in the Internet.

The end-to-end protocols used on the Internet are many and varied. The primary transport protocols are TCP, UDP, and RTP, described briefly in Section 1.6.1. Built on these transport protocols are hundreds of application protocols. A short list of some of the most widely used application protocols would include file transfer (FTP), the World Wide Web (HTTP), mail dispatch and pickup (SMTP and POP), text messaging (IRC), telephone (VoIP), and file exchange (Gnutella, bittorrent, etc.).

The current chapter presents a general model of networks, rather than a description of the Internet. To learn more about the Internet, see the books and papers listed in the section for Chapter 1 in the Suggestions for Further Reading.

Recursive composition occurs when a network layer rests on a link layer that itself is a complete three-layer network. Recursive composition is not a general property of layers, but rather it is a specific property of layered communication systems: The send/receive semantics of an end-to-end connection through a network can be designed to be have the same semantics as a single link, so such an end-to-end connection can be used as a link in a higher-level network. That property facilitates recursive composition, as well as the implementation of various interesting and useful network structures. Here are some examples of recursive composition:

- A dial-up telephone line is often used as a link to an attachment point of the Internet. This dial-up line goes through a telephone network that has its own link, network, and end-to-end layers.
- An **overlay network** is a network layer structure that uses as links the end-to-end layer of an existing network. Gnutella (see problem set 20) is an example of an overlay network that uses the end-to-end layer of the Internet for its links.
- With the advance of "voice over IP" (VoIP), the traditional voice telephone network is gradually converting to become an overlay on the Internet.
- A **tunnel** is a structure that uses the end-to-end layer of an existing network as a link between a local network-layer attachment point and a distant one to make it appear that the attachment is at the distant point. Tunnels, combined with the encryption techniques described in Chapter 5, are used to implement what is commonly called a "virtual private network" (VPN).

Recursive composition need not be limited to two levels. Figure 1.3.8 illustrates the case of Gnutella overlaying the Internet, with a dial-up telephone connection being used as the Internet link layer.

The primary concern when one is dealing with a link layer that is actually an end-to-end connection through another network is that discussion can become confusing unless one is careful to identify which level of decomposition is under discussion. Fortunately our terminology helps keep track of the distinctions among the various layers of a network, so it is worth briefly reviewing that terminology. At the interface between the application and the end-to-end layer, data is identified as a **stream** or **message**. The end-to-end layer divides the stream or message up into a series of segments and hands them to the network layer for delivery. The network layer encapsulates each segment in a packet which it forwards through the network with the help of the link layer. The link layer transmits the packet in a frame. If the link layer is itself a network, then this frame is a message as viewed by the underlying network.

This discussion of layered network organization has been both general and abstract. In the next three sections we investigate in more depth the usual functions and some typical implementation techniques of each of the three layers of our reference model. However, as the introduction pointed out, what follows is not a comprehensive treatment of networking. Instead it identifies many of the major issues and for each issue exhibits one or two examples of how that issue is typically handled in a real network design. For readers who have a goal of becoming network engineers, and who therefore would like to learn the whole remarkable range of implementation strategies that have been used in networks, the Suggestions for Further Reading list several comprehensive books on the subject.

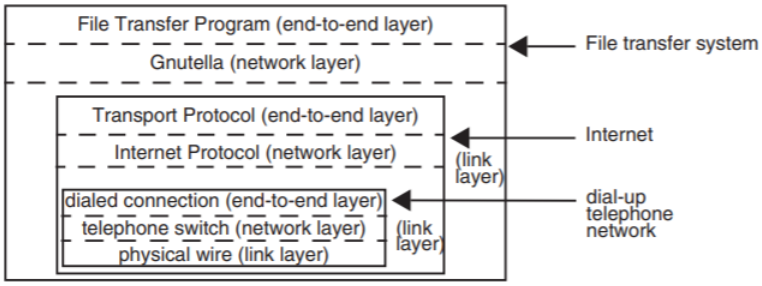


Figure 1.3.8: A typical recursive network composition.The overlay network Gnutella uses for its link layer an end-to-end transport protocol of the Internet. The Internet uses for one of its links an end-to-end transport protocol of the dial-up telephone system.

This page titled 1.3: Getting Organized - Layers is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

1.4: The Link Layer

The link layer is the bottom-most of the three layers of our reference model. The link layer is responsible for moving data directly from one physical location to another. It thus gets involved in several distinct issues: physical transmission, framing bits and bit sequences, detecting transmission errors, multiplexing the link, and providing a useful interface to the network layer above.

1.4.1: Transmitting Digital Data in an Analog World

The purpose of the link layer is to move bits from one place to another. If we are talking about moving a bit from one register to another on the same chip, the mechanism is fairly simple: run a wire that connects the output of the first register to the input of the next. Wait until the first register's output has settled and the signal has propagated to the input of the second; the next clock tick reads the data into the second register. If all of the voltages are within their specified tolerances, the clock ticks are separated enough in time to allow for the propagation, and there is no electrical interference, then that is all there is to it.

Maintaining those three assumptions is relatively easy within a single chip, and even between chips on the same printed circuit board. However, as we begin to consider sending bits between boards, across the room, or across the country, these assumptions become less and less plausible, and they must be replaced with explicit measures to ensure that data is transmitted accurately. In particular, when the sender and receiver are in separate systems, providing a correctly timed clock signal becomes a challenge.

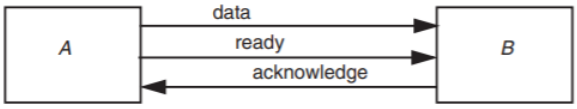


Figure 1.4.1: A simple protocol for data acknowledge communication.

A simple method for getting data from one module to another module that does not share the same clock is with a three-wire (plus common ground) **ready/acknowledge** protocol, as shown in Figure 1.4.1. Module A, when it has a bit ready to send, places the bit on the `data` line, and then changes the steady-state value on the `ready` line. When B sees the `ready` line change, it acquires the value of the bit on the `data` line, and then changes the `acknowledge` line to tell A that the bit has been safely received. The reason that the `ready` and `acknowledge` lines are needed is that, in the absence of any other synchronizing scheme, B needs to know when it is appropriate to look at the `data` line, and A needs to know when it is safe to stop holding the bit value on the `data` line. The signals on the `ready` and `acknowledge` lines frame the bit.

If the propagation time from A to B is Δt, then this protocol would allow A to send one bit to B every $2\Delta t$ plus the time required for A to set up its output and for B to acquire its input, so the maximum data rate would be a little less than $1/(2\Delta t)$. Over short distances, one can replace the single `data` line with N parallel `data` lines, all of which are framed by the same pair of `ready/acknowledge` lines, and thereby increase the data rate to $N/(2\Delta t)$. Many backplane bus designs, as well as peripheral attachment systems such as SCSI and personal computer printer interfaces, use this technique, known as **parallel transmission**, along with some variant of a ready/acknowledge protocol to achieve a higher data rate.

However, as the distance between A and B grows, Δt also grows and the maximum data rate declines in proportion, so the ready/acknowledge technique rapidly breaks down. The usual requirement is to send data at higher rates over longer distances with fewer wires, and this requirement leads to employment of a different system called **serial transmission**. The idea is to send a stream of bits down a single transmission line, without waiting for any response from the receiver and with the expectation that the receiver will somehow recover those bits at the other end with no additional signaling. Thus the output at the transmitting end of the link looks like Figure 1.4.2.

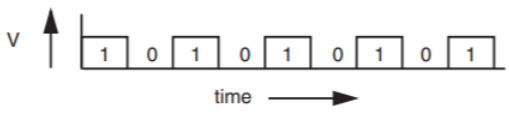


Figure 1.4.2: Outgoing serial transmission.

Unfortunately, because the underlying transmission line is analog, the farther these bits travel down the line, the more attenuation, noise, and line-charging effects they suffer. By the time they arrive at the receiver they will be little more than pulses with exponential leading and trailing edges, as suggested by Figure 1.4.3. The receiving module, B, now has a significant problem in understanding this transmission: because it does not have a copy of the clock that A used to create the bits, it does not know exactly when to sample the incoming line.



Figure 1.4.3: Bit shape deterioration with distance.

A typical solution involves having the two ends agree on an approximate data rate, so that the receiver can run a voltage-controlled oscillator (VCO) at about that same data rate. The output of the VCO is multiplied by the voltage of the incoming signal and the product suitably filtered and sent back to adjust the VCO. If this circuit is designed correctly, it will lock the VCO to both the frequency and phase of the arriving signal. (This device is commonly known as a **phase-locked loop**.) The VCO, once locked, then becomes a clock source that a receiver can use to sample the incoming data.

One complication is that with certain patterns of data (for example, a long string of zeros) there may be no transitions in the data stream, in which case the phase-locked loop will not be able to synchronize. To deal with this problem, the transmitter usually encodes the data in a way that ensures that no matter what pattern of bits is sent, there will be some transitions on the transmission line. A frequently used method is called **phase encoding**, in which there is at least one level transition associated with every data bit. A common phase encoding is the Manchester code, in which the transmitter encodes each bit as two bits: a zero is encoded as a zero followed by a one, while a one is encoded as a one followed by a zero. This encoding guarantees that there is a level transition in the center of every transmitted bit, thus supplying the receiver with plenty of clocking information. It has the disadvantage that the maximum data rate of the communication channel is effectively cut in half, but the resulting simplicity of both the transmitter and the receiver is often worth this price. Other, more elaborate, encoding schemes can ensure that there is at least one transition for every few data bits. These schemes don't reduce the maximum data rate as much, but they complicate encoding, decoding, and synchronization.

The usual goal for the design space of a physical communication link is to achieve the highest possible data rate for the encoding method being used. That highest possible data rate will occur exactly at the point where the arriving data signal is just on the ragged edge of being correctly decodable, and any noise on the line will show up in the form of clock jitter or signals that just miss expected thresholds, either of which will lead to decoding errors.

Sidebar 1.4.1

Framing phase-encoded bits

The astute reader may have spotted a puzzling gap in the brief description of the Manchester code: while it is intended as a way of framing bits as they appear on the transmission line, it is also necessary to frame the data bits themselves, in order to know whether a data bit is encoded as the bits $(n, n+1)$ or the bits $(n+1, n+2)$. A typical approach is to combine code bit framing with data bit framing (and even provide some help in higher-level framing) by specifying that every transmission must begin with a standard pattern, such as some minimum number of coded one-bits followed by a coded zero. The series of consecutive ones gives the Phase-Locked Loop something to synchronize on, and at the same time provides examples of the positions of known data bits. The zero frames the end of the framing sequence.

The data rate of a digital link is conventionally measured in bits per second. Since digital data is ultimately carried using an analog channel, the question arises of what might be the maximum digital carrying capacity of a specified analog channel. A perfect analog channel would have an infinite capacity for digital data because one could both set and measure a transmitted signal level with infinite precision, and then change that setting infinitely often. In the real world, noise limits the precision with which a receiver can measure the signal value, and physical limitations of the analog channel such as chromatic dispersion (in an optical fiber), charging capacitance (in a copper wire), or spectrum availability (in a wireless signal) put a ceiling on the rate at which a receiver can detect a change in value of a signal. These physical limitations are summed up in a single measure known as the

bandwidth of the analog channel. To be more precise, the number of different signal values that a receiver can distinguish is proportional to the logarithm of the ratio of the signal power to the noise power, and the maximum rate at which a receiver can distinguish changes in the signal value is proportional to the analog bandwidth.

These two parameters (signal-to-noise ratio and analog bandwidth) allow one to calculate a theoretical maximum possible channel capacity (that is, data transmission rate) using Shannon's capacity theorem (see Sidebar 1.4.2).* Although this formula adopts a particular definition of bandwidth, assumes a particular randomness for the noise, and says nothing about the delay that might be encountered if one tries to operate near the channel capacity, it turns out to be surprisingly useful for estimating capacities in the real world.

Sidebar 1.4.2

Shannon's capacity theorem

$$C \leq W \cdot \log_2\left(1 + \frac{S}{NW}\right), \text{ where:} \tag{1.4.1}$$

C = channel capacity, in bits per second
W = channel bandwidth, in hertz
S = maximum allowable signal power, as seen by the receiver
N = noise power per unit of bandwidth

Since some methods of digital transmission come much closer to Shannon's theoretical capacity than others, it is customary to use as a measure of goodness of a digital transmission system the number of bits per second that the system can transmit per hertz of bandwidth. Setting $W = 1$, the capacity theorem says that the maximum bits per second per hertz is $\log_2(1 + S/N)$. An elementary signaling system in a low-noise environment can easily achieve 1 bit per second per hertz. On the other hand, for a 28 kilobits per second modem to operate over the 2.4 kilohertz telephone network, it must transmit about 12 bits per second per hertz. The capacity theorem says that the logarithm must be at least 12, so the signal-to-noise ratio must be at least 2^{12}, or using a more traditional analog measure, 36 decibels, which is just about typical for the signal-to-noise ratio of a properly working telephone connection. The copper-pair link between a telephone handset and the telephone office does not go through any switching equipment, so it actually has a bandwidth closer to 100 kilohertz and a much better signal-to-noise ratio than the telephone system as a whole; these combine to make possible "digital subscriber line" (DSL) modems that operate at 1.5 megabits/second—and even up to 50 megabits/second over short distances—using a physical link that was originally designed to carry just voice.

One other parameter is often mentioned in characterizing a digital transmission system: the **bit error rate**, abbreviated BER and measured as a ratio to the transmission rate. For a transmission system to be useful, the bit error rate must be quite low; it is typically reported with numbers such as one error in 10^6, 10^7, or 10^8 transmitted bits. Even the best of those rates is not good enough for digital systems; higher levels of the system must be prepared to detect and compensate for errors.

---

* The derivation of this theorem is beyond the scope of this textbook. The capacity theorem was originally proposed by Claude E. Shannon in the paper "A mathematical theory of communication," *Bell System Technical Journal 27* (1948), pages 379–423 and 623–656. Most modern texts on information theory explore it in depth.

1.4.2: Framing Frames

The previous section explained how to obtain a stream of neatly framed bits, but because the job of the link layer is to deliver **frames** across the link, it must also be able to figure out where in this stream of bits each frame begins and ends. Framing frames is a distinct, and quite independent, requirement from framing bits, and it is one of the reasons that some network models divide the link layer into two layers—a lower layer that manages physical aspects of sending and receiving individual bits and an upper layer that implements the strategy of transporting entire frames.

There are many ways to frame frames. One simple method is to choose some pattern of bits—for example, seven one-bits in a row—as a frame-separator mark. The sender simply inserts this mark into the bit stream at the end of each frame. Whenever this pattern appears in the received data, the receiver takes it to mark the end of the previous frame, and assumes that any bits that

follow belong to the next frame. This scheme works nicely, as long as the payload data stream never contains the chosen pattern of bits.

Rather than explaining to the higher layers of the network that they cannot transmit certain bit patterns, the link layer implements a technique known as **bit stuffing**. The transmitting end of the link layer, in addition to inserting the frame-separator mark between frames, examines the data stream itself, and if it discovers six ones in a row it stuffs an extra bit into the stream, a zero. The receiver, in turn, watches the incoming bit stream for long strings of ones. When it sees six one-bits in a row it examines the next bit to decide what to do. If the seventh bit is a zero, the receiver discards the zero bit, thus reversing the stuffing done by the sender. If the seventh bit is a one, the receiver takes the seven ones as the frame separator. Figure 1.4.4 shows a simple pseudocode implementation of the procedure to send a frame with bit stuffing, and Figure 1.4.5 shows the corresponding procedure on the receiving side of the link. (For simplicity, the illustrated receive procedure ignores two important considerations. First, the receiver uses only one frame buffer. A better implementation would have multiple buffers to allow it to receive the next frame while processing the current one. Second, the same thread that acquires a bit also runs the network level protocol by calling `LINK_RECEIVE` . A better implementation would probably `NOTIFY` a separate thread that would then call the higher-level protocol, and this thread could continue processing more incoming bits.)

```
procedure FRAME_TO_BIT (frame_data, length)
    ones_in_a_row = 0
    for i from 1 to length do                    // First send frame contents
        SEND_BIT (frame_data[i]);
        if frame_data[i] = 1 then
            ones_in_a_row ← ones_in_a_row + 1;
            if ones_in_a_row = 6 then
                SEND_BIT (0);                    // Stuff a zero so that data doesn't
                ones_in_a_row ← 0;               // look like a framing marker
        else
            ones_in_a_row ← 0;
    for i from 1 to 7 do                         // Now send framing marker.
        SEND_BIT (1)
```

Figure 1.4.4: Sending a frame with bit stuffing.

```
procedure BIT_TO_FRAME (rcvd_bit)
    ones_in_a_row integer initially 0
    if ones_in_a_row < 6 then
        bits_in_frame ← bits_in_frame + 1
        frame_data[bits_in_frame] ← rcvd_bit
        if rcvd_bit = 1 then ones_in_a_row ← ones_in_a_row + 1
        else ones_in_a_row ← 0
    else                          // This may be a seventh one-bit in a row, check it out.
        if rcvd_bit = 0 then
            ones_in_a_row ← 0                    // Stuffed bit, don't use it.
        else                                     // This is the end-of-frame marker
            LINK_RECEIVE (frame_data, (bits_in_frame - 6), link_id)
            bits_in_frame ← 0
            ones_in_a_row ← 0
```

Figure 1.4.5: Receiving a frame with bit stuffing.

Bit stuffing is one of many ways to frame frames. There is little need to explore all the possible alternatives because frame framing is easily specified and subcontracted to the implementer of the link layer—the entire link layer, along with bit framing, is often done in the hardware—so we now move on to other issues.

1.4.3: Error Handling

An important issue is what the receiving side of the link layer should do about bits that arrive with doubtful values. Since the usual design pushes the data rate of a transmission link up until the receiver can barely tell the ones from the zeros, even a small amount of extra noise can cause errors in the received bit stream.

The first and perhaps most important line of defense in dealing with transmission errors is to require that the design of the link be good at *detecting* such errors when they occur. The usual method is to encode the data with an **error detection code**, which entails adding a small amount of redundancy. A simple form of such a code is to have the transmitter calculate a checksum and place the checksum at the end of each frame. As soon as the receiver has acquired a complete frame, it recalculates the checksum and compares its result with the copy that came with the frame. By carefully designing the checksum algorithm and making the number

of bits in the checksum large enough, one can make the probability of not detecting an error as low as desired. The more interesting issue is what to do when an error is detected. There are three alternatives:

1. Have the sender encode the transmission using an **error correction code**, which is a code that has enough redundancy to allow the receiver to identify the particular bits that have errors and correct them. This technique is widely used in situations where the noise behavior of the transmission channel is well understood and the redundancy can be targeted to correct the most likely errors. For example, compact disks are recorded with a burst error-correction code designed to cope particularly well with dust and scratches. Error correction is one of the topics of Chapter 2.
2. Ask the sender to retransmit the frame that contained an error. This alternative requires that the sender hold the frame in a buffer until the receiver has had a chance to recalculate and compare its checksum. The sender needs to know when it is safe to reuse this buffer for another frame. In most such designs the receiver explicitly acknowledges the correct (or incorrect) receipt of every frame. If the propagation time from sender to receiver is long compared with the time required to send a single frame, there may be several frames in flight, and acknowledgments (especially the ones that ask for retransmission) are disruptive. On a high-performance link, an explicit acknowledgment system can be surprisingly complex.
3. Let the receiver discard the frame. This alternative is a reasonable choice in light of our previous observation (see Section 1.2.3) that congestion in higher network levels must be handled by discarding packets anyway. Whatever higher-level protocol is used to deal with those discarded packets will also take care of any frames that are discarded because they contained errors.

Real-world designs often involve blending these techniques, for example by having the sender apply a simple error-correction code that catches and repairs the most common errors and that reliably detects and reports any more complex irreparable errors, and then by having the receiver discard the frames that the error-correction code could not repair.

1.4.4: The Link Layer Interface: Link Protocols and Multiplexing

The link layer, in addition to sending bits and frames at one end and receiving them at the other end, also has interfaces to the network layer above, as illustrated in Figure 1.3.5 in Section 1.3.2. As described so far, the interface consists of an ordinary procedure call (to `LINK_SEND`) that the network layer uses to tell the link layer to send a packet, and an upcall (to `NETWORK_HANDLE`) from the link layer to the network layer at the other end to alert the network layer that a packet arrived.

To be practical, this interface between the network layer and the link layer needs to be expanded slightly to incorporate two additional features not previously mentioned: multiple lower-layer protocols, and higher-layer protocol multiplexing. To support these two functions we add two arguments to `LINK_SEND`, named *`link_protocol`* and *`network_protocol`*:

```
LINK_SEND (data_buffer, link_identifier, link_protocol, network_protocol)
```

Over any given link, it is sometimes appropriate to use different protocols at different times. For example, a wireless link may occasionally encounter a high noise level and need to switch from the usual link protocol to a "robustness" link protocol that employs a more expensive form of error detection with repeated retry, but runs more slowly. At other times it may want to try out a new, experimental link protocol. The third argument to `LINK_SEND`, namely *`link_protocol`*, tells `LINK_SEND` which link protocol to use for this data, and its addition leads to the protocol layering illustrated in Figure 1.4.6.

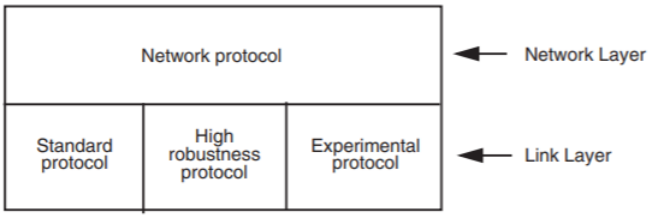


Figure 1.4.6: Layer composition with multiple link protocols.

The second feature of the interface to the link layer is more involved: the interface should support protocol multiplexing. Multiplexing allows several different network layer protocols to use the same link. For example, Internet Protocol, Appletalk Protocol, and Address Resolution Protocol (we will talk about some of these protocols later in this chapter) might all be using the same link. Several steps are required. First, the network layer protocol on the sending side needs to specify which protocol handler should be invoked on the receiving side, so one more argument, *`network_protocol`*, is needed in the interface to `LINK_SEND`.

Second, the value of `network_protocol` needs to be transmitted to the receiving side, for example by adding it to the link-level packet header. Finally, the link layer on the receiving side needs to examine this new header field to decide to which of the various network layer implementations it should deliver the packet. Our protocol layering organization is now as illustrated in Figure 1.4.7. This figure demonstrates the real power of the layered organization: any of the four network layer protocols in the figure may use any of the three link layer protocols.

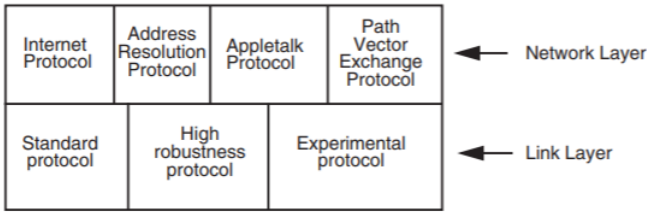


Figure 1.4.7: Layer composition with multiple link protocols *and* link layer multiplexing to support multiple network layer protocols.

With the addition of multiple link protocols and link multiplexing, we can summarize the discussion of the link layer in the form of pseudocode for the procedures `LINK_SEND` and `LINK_RECEIVE` , together with a structure describing the frame that passes between them, as in Figure 1.4.8. In procedure `LINK_SEND` , the procedure variable `sendproc` is selected from an array of link layer protocols; the value found in that array might be, for example, a version of the procedure `PACKET_TO_BIT` of Figure 1.4.5 that has been extended with a third argument that identifies which link to use. The procedures `CHECKSUM` and `LENGTH` are programs we assume are found in the library. Procedure `LINK_RECEIVE` might be called, for example, by procedure `BIT_TO_FRAME` of Figure 1.4.5. The procedure `LINK_RECEIVE` verifies the checksum, and then

extracts `net_data` and `net_protocol` from the frame and passes them to the procedure that calls the network handler together with the identifier of the link over which the packet arrived.

```
structure frame
    structure checked_contents
        bit_string net_protocol                    // multiplexing parameter
        bit_string payload                         // payload data
    bit_string checksum

procedure LINK_SEND (data_buffer, link_identifier, link_protocol, network_protocol)
    frame instance outgoing_frame
    outgoing_frame.checked_contents.payload ← data_buffer
    outgoing_frame.checked_contents.net_protocol ← data_buffer.network_protocol
    frame_length ← LENGTH (data_buffer) + header_length
    outgoing_frame.checksum ← CHECKSUM (frame.checked_contents, frame_length)
    sendproc ← link_protocol[that_link.protocol]              // Select link protocol.
    sendproc (outgoing_frame, frame_length, link_identifier)  // Send frame.

procedure LINK_RECEIVE (received_frame, length, link_id)
    frame instance received_frame
    if CHECKSUM (received_frame.checked_contents, length) =
                                              received_frame.checksum
        then                                  // Pass good packets up to next layer.
        good_frame_count ← good_frame_count + 1;
        GIVE_TO_NETWORK_HANDLER (received_frame.checked_contents.payload,
                        received_frame.checked_contents.net_protocol);
    else bad_frame_count ← bad_frame_count + 1    // Just count damaged frame.

// Each network layer protocol handler must call SET_HANDLER before the first packet
// for that protocol arrives...

procedure SET_HANDLER (handler_procedure, handler_protocol)
    net_handler[handler_protocol] ← handler_procedure

procedure GIVE_TO_NETWORK_HANDLER (received_packet, network_protocol)
    handler ← net_handler[network_protocol]
    if (handler ≠ NULL) call handler(received_packet, network_protocol)
    else unexpected_protocol_count ← unexpected_protocol_count + 1
```

Figure 1.4.8: The `LINK_SEND` and `LINK_RECEIVE` procedures, together with the structure of the frame transmitted over the link and a dispatching procedure for the network layer.

These procedures also illustrate an important property of layering that was discussed in Section 1.3.4. The link layer handles its argument `data_buffer` as an unstructured string of bits. When we examine the network layer in the next section of the chapter, we will see that `data_buffer` contains a network-layer packet, which has its own internal structure. The point is that as we pass from an upper layer to a lower layer, the content and structure of the payload data is not supposed to be any concern of the lower layer.

As an aside, the division we have chosen for our sample implementation of a link layer, with one program doing framing and another program verifying checksums, corresponds to the OSI reference model division of the link layer into physical and strategy layers, as was mentioned in Section 1.3.5.

Since the link is now multiplexed among several network-layer protocols, when a frame arrives, the link layer must dispatch the packet contained in that frame to the proper network layer protocol handler. Figure 1.4.8 shows a

handler dispatcher named `GIVE_TO_NETWORK_HANDLER` . Each of several different network-layer protocol-implementing programs specifies the protocol it knows how to handle, through arguments in a call to `SET_HANDLER` . Control then passes to a particular network-layer handler only on arrival of a frame containing a packet of the protocol it specified. With some additional effort (not illustrated—the reader can explore this idea as an exercise), one could also make this dispatcher multithreaded, so that as it passes a packet up to the network layer a new thread takes over and the link layer thread returns to work on the next arriving frame.

With or without threads, the *`network_protocol`* field of a frame indicates to whom in the network layer the packet contained in the frame should be delivered. From a more general point of view, we are multiplexing the lower-layer protocol among several higher-layer protocols. This notion of multiplexing, together with an identification field to support it, generally appears in every protocol layer, and in every layer-to-layer interface, of a network architecture.

An interesting challenge is that the multiplexing field of a layer names the protocols of the next higher layer, so some method is needed to assign those names. Since higher-layer protocols are likely to be defined and implemented by different organizations, the usual solution is to hand the name conflict avoidance problem to some national or international standard-setting body. For example, the names of the protocols of the Internet are assigned by an outfit called ICANN, which stands for the Internet Corporation for Assigned Names and Numbers.

1.4.5: Link Properties

Some final details complete our tour of the link layer. First, links come in several flavors, for which there is some standard terminology:

A **point-to-point** link directly connects exactly two communicating entities. A **simplex** link has a transmitter at one end and a receiver at the other; two-way communication requires installing two such links, one going in each direction. A **duplex** link has both a transmitter and a receiver at each end, allowing the same link to be used in both directions. A **half-duplex** link is a duplex link in which transmission can take place in only one direction at a time, whereas a **full-duplex** link allows transmission in both directions at the same time over the same physical medium.

A **broadcast** link is a shared transmission medium in which there can be several transmitters and several receivers. Anything sent by any transmitter can be received by many—perhaps all—receivers. Depending on the physical design details, a broadcast link may limit use to one transmitter at a time, or it may allow several distinct transmissions to be in progress at the same time over the same physical medium. This design choice is analogous to the distinction between half duplex and full duplex but there is no standard terminology for it. The link layers of the standard Ethernet and the popular wireless system known as Wi-Fi are one-transmitter-at-a-time broadcast links. The link layer of a CDMA Personal Communication System (such as ANSI–J–STD–008, which is used by cellular providers Verizon and Sprint PCS) is a broadcast link that permits many transmitters to operate simultaneously.

Finally, most link layers impose a maximum frame size, known as the **maximum transmission unit (MTU)**. The reasons for limiting the size of a frame are several:

1. The MTU puts an upper bound on link commitment time, which is the length of time that a link will be tied up once it begins to transmit the frame. This consideration is more important for slow links than for fast ones.
2. For a given bit error rate, the longer a frame is, the greater the chance of an uncorrectable error in that frame. Since the frame is usually also the unit of error control, an uncorrectable error generally means loss of the entire frame, so as the frame length increases not only does the probability of loss increase, but the cost of the loss increases because the entire frame will probably have to be retransmitted. The MTU puts a ceiling on both of these costs.
3. If congestion leads a forwarder to discard a packet, the MTU limits the amount of transmission capacity required to retransmit the packet.

4. There may be mechanical limits on the maximum length of a frame. A hardware interface may have a small buffer or a short counter register tracking the number of bits in the frame. Similar limits sometimes are imposed by software that was originally designed for another application or to comply with some interoperability standard.

Whatever the reason for the MTU, when an application needs to send a message that does not fit in a maximum-sized frame, it becomes the job of some end-to-end protocol to divide the message into segments for transmission and to reassemble the segments into the complete message at the other end. The way in which the end-to-end protocol discovers the value of the MTU is complicated—it needs to know not just the MTU of the link it is about to use, but the smallest MTU that the segment will encounter on the path through the network to its destination. For this purpose, it needs some help from the network layer, which is our next topic.

This page titled 1.4: The Link Layer is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

1.5: The Network Layer

The network layer is the middle layer of our three-layer reference model. The network layer moves a packet across a series of links. While conceptually quite simple, the challenges in implementation of this layer are probably the most difficult in network design because there is usually a requirement that a single design span a wide range of performance, traffic load, and number of attachment points. In this section we develop a simple model of the network layer and explore some of the challenges.

1.5.1: Addressing Interface

The conceptual model of a network is a cloud bristling with **network attachment points** identified by numbers known as **network addresses**, as in Figure 1.5.1 below. A segment enters the network at one attachment point, known as the **source**. The network layer wraps the segment in a packet and carries the packet across the network to another attachment point, known as the **destination**, where it unwraps the original segment and delivers it.

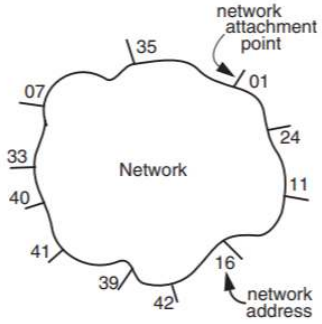


Figure 1.5.1: The network layer.

The model in the figure is misleading in one important way: it suggests that delivery of a segment is accomplished by sending it over one final, physical link. A network attachment point is actually a virtual concept rather than a physical concept. Every network participant, whether a packet forwarder or a client computer system, contains an implementation of the network layer, and when a packet finally reaches the network layer of its destination, rather than forwarding it further, the network layer unwraps the segment contained in the packet and passes that segment to the end-to-end layer inside the system that contains the network attachment point. In addition, a single system may have several network attachment points, each with its own address, all of which result in delivery to the same end-to-end layer; such a system is said to be **multihomed**. Even packet forwarders need network attachment points with their own addresses, so that a network manager can send them instructions about their configuration and maintenance.

Since a network has many attachment points, the the end-to-end layer must specify to the network layer not only a data segment to transmit but also its intended destination. Further, there may be several available networks and protocols, and several end-to-end protocol handlers, so the interface from the end-to-end layer to the network layer is parallel to the one between the network layer and the link layer:

```
NETWORK_SEND (segment_buffer, destination, network_protocol, end_layer_protocol)
```

The argument `network_protocol` allows the end-to-end layer to select a network and protocol with which to send the current segment, and the argument `end_layer_protocol` allows for multiplexing, this time of the network layer by the end-to-end layer. The value of `end_layer_protocol` identifies to the network layer the destination to which end-to-end protocol handler the segment should be delivered.

The network layer also has a link-layer interface, across which it receives packets. Following the upcall style of the link layer of Section 1.4 this interface would be

```
NETWORK_HANDLE (packet, network_protocol)
```

and this procedure would be the `handler_procedure` argument of a call to `SET_HANDLER` in Figure 1.4.8. Thus whenever the link layer has a packet to deliver to the network layer, it does so by calling `NETWORK_HANDLE` .

The pseudocode of Figure 1.5.2 describes a model network layer in detail, starting with the structure of a packet, and followed by implementations of the procedures `NETWORK_HANDLE` and `NETWORK_SEND` . `NETWORK_SEND` creates a packet, starting with the segment provided by the end-to-end layer and adding a network-layer header, which here comprises three fields: `source` , `destination` , and `end_layer_protocol` . It fills in the `destination` and `end_layer_protocol` fields from the corresponding arguments, and it fills in the source field with the address of its own network attachment point. Figure 1.5.3 shows this latest addition to the overhead of a packet.

```
    structure packet
        bit_string source
        bit_string destination
        bit_string end_protocol
        bit_string payload

1   procedure NETWORK_SEND (segment_buffer, destination,
2                                   network_protocol, end_protocol)
3       packet instance outgoing_packet
4       outgoing_packet.payload ← segment_buffer
5       outgoing_packet.end_protocol ← end_protocol
6       outgoing_packet.source ← MY_NETWORK_ADDRESS
7       outgoing_packet.destination ← destination
8       NETWORK_HANDLE (outgoing_packet, net_protocol)

9   procedure NETWORK_HANDLE (net_packet, net_protocol)
10      packet instance net_packet
11      if net_packet.destination ≠ MY_NETWORK_ADDRESS then
12          next_hop ← LOOKUP (net_packet.destination, forwarding_table)
13          LINK_SEND (net_packet, next_hop, link_protocol, net_protocol)
14      else
15          GIVE_TO_END_LAYER (net_packet.payload,
16                  net_packet.end_protocol, net_packet.source)
```

Figure 1.5.2: Model implementation of a network layer. The procedure NETWORK_SEND originates packets, while NETWORK_HANDLE receives packets and either forwards them or passes them to the local end-to-end layer.

Procedure NETWORK_HANDLE may do one of two rather different things with a packet, distinguished by the test on line 11. If the packet is not at its destination, NETWORK_HANDLE looks up the packet's destination in *forwarding_table* to determine the best link on which to forward it, and then it calls the link layer to send the packet on its way. On the other hand, if the received packet is at its destination, the network layer passes its payload up to the end-to-end layer rather than sending the packet out over another link. As in the case of the interface between the link layer and the network layer, the interface to the end-to-end layer is another upcall that is intended to go through a handler dispatcher similar to that of the link layer dispatcher of Figure 1.4.8. Because in a network, any network attachment point can send a packet to any other, the last argument of GIVE_TO_END_LAYER , which is the source of the packet, is a piece of information that the end-layer recipient generally finds useful in deciding how to handle the packet.

One might wonder what led to naming the procedure NETWORK_HANDLE rather than NETWORK_RECEIVE . The insight in choosing that name is that forwarding a packet is always done in exactly the same way, whether the packet comes from the layer above or from the layer below. Thus, when we consider the steps to be taken by NETWORK_SEND , the straightforward implementation is simply to place the data in a packet, add a network layer header, and hand the packet to NETWORK_HANDLE . As an extra feature, this architecture allows a source to send a packet to itself without creating a special case.

Just as the link layer used the *net_protocol* field to decide which of several possible network handlers to give the packet to, NETWORK_SEND can use the *net_protocol* argument for the same purpose. That is, rather than calling NETWORK_HANDLE directly, it could call the procedure GIVE_TO_NETWORK_HANDLER of Figure 1.4.8.

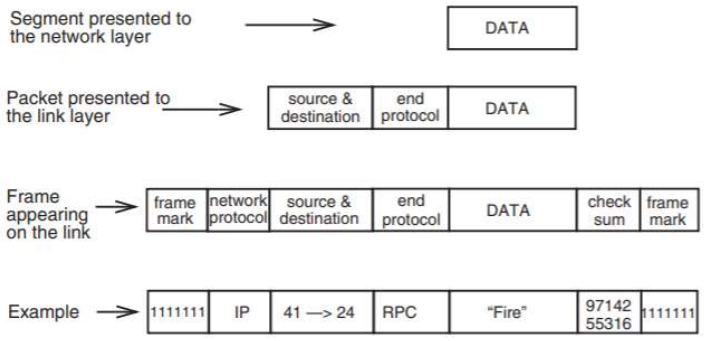


Figure 1.5.3: A typical accumulation of network layer and link layer headers and trailers. The additional information added at each layer can come from control information passed from the higher layer as arguments (for example, the end protocol type and the destination are arguments in the call to the network layer). In other cases they are added by the lower layer (for example, the link layer adds the frame marks and checksum).

1.5.2: Managing the Forwarding Table: Routing

The primary challenge in a packet forwarding network is to set up and manage the forwarding tables, which generally must be different for each network-layer participant. Constructing these tables requires first figuring out appropriate paths (sometimes called **routes**) to follow from each source to each destination, so the exercise is variously known as **path-finding** or **routing**. In a small network, one might set these tables up by hand. As the scale of a network grows, this approach becomes impractical, for several reasons:

1. The amount of calculation required to determine the best paths grows combinatorially with the number of nodes in the network.
2. Whenever a link is added or removed, the forwarding tables must be recalculated. As a network grows in size, the frequency of links being added and removed will probably grow in proportion, so the combinatorially growing routing calculation will have to be performed more and more frequently.
3. Whenever a link fails or is repaired, the forwarding tables must be recalculated. For a given link failure rate, the number of such failures will be proportional to the number of links, so for a second reason the combinatorially growing routing calculation will have to be performed an increasing number of times.
4. There are usually several possible paths available, and if traffic suddenly causes the originally planned path to become congested, it would be nice if the forwarding tables could automatically adapt to the new situation.

All four of these reasons encourage the development of automatic routing algorithms. If reasons 1 and 2 are the only concerns, one can leave the resulting forwarding tables in place for an indefinite period, a technique known as **static routing**. The on-the-fly recalculation called for by reasons 3 and 4 is known as **adaptive routing**, and because this feature is vitally important in many networks, routing algorithms that allow for easy update when things change are almost always used. A packet forwarder that also participates in a routing algorithm is usually called a **router**. An adaptive routing algorithm requires exchange of current reachability information. Typically, the routers exchange this information using a network-layer routing protocol transmitted over the network itself.

To see how adaptive routing algorithms might work, consider the modest-sized network of Figure 1.5.4. To minimize confusion in interpreting this figure, each network address is lettered, rather than numbered, while each link is assigned two one-digit link identifiers, one from the point of view of each of the stations it connects. In this figure, routers are rectangular while workstations and services are round, but all have network addresses and all have network layer implementations.

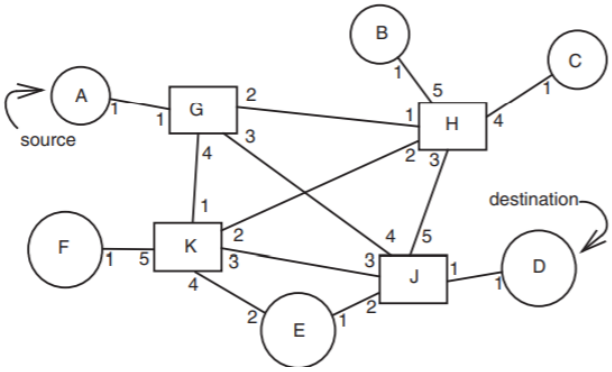


Figure 1.5.4: Routing example.

Suppose now that the source A sends a packet addressed to destination D. Since A has only one outbound link, its forwarding table is short and simple:

Table 1.5.1: Forwarding table for source A

destination	link
A	end-layer
all other	1

so the packet departs from A by way of link 1, going to router G for its next stop. However, the forwarding table at G must be considerably more complicated. It might contain, for example, the following values:

Table 1.5.2: Forwarding table for router G

destination	link
A	1
B	2

destination	link
C	2
D	3
E	4
F	4
G	end-layer
H	2
J	3
K	4

This is not the only possible forwarding table for G. Since there are several possible paths to most destinations, there are several possible values for some of the table entries. In addition, it is essential that the forwarding tables in the other routers be coordinated with this forwarding table. If they are not, when router G sends a packet destined for E to router K, router K might send it back to G, and the packet could loop forever.

The interesting question is how to construct a consistent, efficient set of forwarding tables. Many algorithms that sound promising have been proposed and tried; few work well. One that works moderately well for small networks is known as **path vector exchange**. Each participant maintains, in addition to its forwarding table, a **path vector**, each element of which is a complete path to some destination. Initially, the only path it knows about is the zero-length path to itself, but as the algorithm proceeds it gradually learns about other paths. Eventually its path vector accumulates paths to every point in the network. After each step of the algorithm it can construct a new forwarding table from its new path vector, so the forwarding table gradually becomes more and more complete. The algorithm involves two steps that every participant repeats over and over, path **advertising** and **path selection**.

Table 1.5.3: Initial state of path vector for G. < > is an empty path.

to	path
G	< >

To illustrate the algorithm, suppose participant G starts with a path vector that contains just one item, an entry for itself, as in Table 1.5.3. In the **advertising** step, each participant sends its own network address and a copy of its path vector down every attached link to its immediate neighbors, specifying the network-layer protocol `PATH_EXCHANGE` . The routing algorithm of G would thus receive from its four neighbors the four path vectors of Table 1.5.4. This advertisement allows G to discover the names, which are in this case network addresses, of each of its neighbors.

Table 1.5.4: Path vectors received by G in the first round.

From A, via link 1		From H, via link 2		From J, via link 3		From K, via link 4	
to	path	to	path	to	path	to	path
A	< >	H	< >	J	< >	K	< >

G now performs the **path selection** step by merging the information received from its neighbors with that already in its own previous path vector. To do this merge, G takes each received path, prepends the network address of the neighbor that supplied it, and then decides whether or not to use this path in its own path vector. Since on the first round in our example all of the information from neighbors gives paths to previously unknown destinations, G adds all of them to its path vector, as in Table 1.5.5. G can also now construct a forwarding table for use by `NET_HANDLE` that allows `NET_HANDLE` to forward packets to destinations A, H, J, and K as well as to the end-to-end layer of G itself. In a similar way, each of the other participants has also constructed a better path vector and forwarding table.

Table 1.5.5: First-round path vector and forwarding table for G.

path vector	forwarding table

to	path
A	<A>
G	<G>
H	<H>
J	<J>
K	<K>

to	link
A	1
G	end-layer
H	2
J	3
K	4

Now, each participant advertises its new path vector. This time, G receives the four path vectors of Table 1.5.6, which contain information about several participants of which G was previously unaware. Following the same procedure again, G prepends to each element of each received path vector the identity of the router that provided it, and then considers whether or not to use this path in its own path vector. For previously unknown destinations, the answer is yes. For previously known destinations, G compares the paths that its neighbors have provided with the path it already had in its table to see if the neighbor has a better path.

Table 1.5.6: Path vectors received by G in the second round.

From A, via link 1		From H, via link 2:		From J, via link 3:		From K, via link 4:	
to	path	to	path	to	path	to	path
A	< >	B	<B>	D	<D>	E	<E>
G	<G>	C	<C>	E	<E>	F	<F>
		G	<G>	G	<G>	G	<G>
		H	<H>	H	<H>	H	<H>
		J	<J>	J	<J>	J	<J>
		K	<K>	K	<K>	K	<K>

This comparison raises the question of what metric to use for "better". One simple answer is to count the number of hops. More elaborate schemes might evaluate the data rate of each link along the way or even try to keep track of the load on each link of the path by measuring and reporting queue lengths. Assuming G is simply counting hops, G looks at the path that A has offered to reach G, namely

```
to G: <A, G>
```

and notices that G's own path vector already contains a zero-length path to G, so it ignores A's offering. A second reason to ignore this offering is that its own name, G, is in the path, which means that this path would involve a loop. To ensure loop-free forwarding, the algorithm always ignores any offered path that includes this router's own name. When it is finished with the second round of path selection, G will have constructed the second-round path vector and forwarding table of Table 1.5.7.

Table 1.5.7: Second-round path vector and forwarding table for G.

path vector	forwarding table

to	path
A	<A>
B	<H, B>
C	<H, C>
D	<J, D>
E	<J, E>
F	<K, F>
G	< >
H	<H>
J	<J>
K	<K>

to	link
A	1
B	2
C	2
D	3
E	3
F	4
G	end-layer
H	2
J	3
K	4

On the next round G will begin receiving longer paths. For example it will learn that H offers the path

```
to D: <H, J, D>
```

Since this path is longer than the one that G already has in its own path vector for D, G will ignore the offer. If the participants continue to alternate advertising and path selection steps, this algorithm ensures that eventually every participant will have in its own path vector the best (in this case, shortest) path to every other participant and there will be no loops.

If static routing would suffice, the path vector construction procedure described above could stop once everyone's tables had stabilized. But a nice feature of this algorithm is that it is easily extended to provide adaptive routing. One method of extension would be, on learning of a change in topology, to redo the entire procedure, starting again with path vectors containing just the path to the local end layer. A more efficient approach is to use the existing path vectors as a first approximation. The one or two participants who, for example, discover that a link is no longer working simply adjust their own path vectors to stop using that link and then advertise their new path vectors to the neighbors they can still reach. Once we realize that readvertising is a way to adjust to topology change, it is apparent that the straightforward way to achieve adaptive routing is simply to have every router occasionally repeat the path vector exchange algorithm.

If someone adds a new link to the network, on the next iteration of the exchange algorithm, the routers at each end of the new link will discover it and propagate the discovery throughout the network. On the other hand, if a link goes down, an additional step is needed to ensure that paths that traversed that link are discarded: each router discards any paths that a neighbor stops advertising. When a link goes down, the routers on each end of that link stop receiving advertisements; as soon as they notice this lack they discard all paths that went through that link. Those paths will be missing from their own next advertisements, which will cause any neighbors using those paths to discard them in turn; in this way the fact of a down link retraces each path that contains the link, thereby propagating through the network to every router that had a path that traversed the link. A model implementation of all of the parts of this path vector algorithm appears in Figure 1.5.5.

```
// Maintain routing and forwarding tables.

vector associative array        // vector[d_addr] contains path to destination d_addr
neighbor_vector instance of vector  // A path vector received from some neighbor
my_vector  instance of vector   // My current path vector.
addr associative array          // addr[j] is the address of the network attachment
                                // point at the other end of link j.
                                // my_addr is address of my network attachment point.
                                // A path is a parsable list of addresses, e.g. {a,b,c,d}

procedure main()                        // Initialize, then start advertising.
    SET_TYPE_HANDLER (HANDLE_ADVERTISEMENT, exchange_protocol)
    clear my_vector;                    // Listen for advertisements
    do occasionally                     // and advertise my paths
        for each j in link_ids do       // to all of my neighbors.
            status ← SEND_PATH_VECTOR (j, my_addr, my_vector, exch_protocol)
            if status ≠ 0 then          // If the link was down,
                clear new_vector        // forget about any paths
                FLUSH_AND_REBUILD (j)   // that start with that link.

procedure HANDLE_ADVERTISEMENT (advt, link_id)   // Called when an advt arrives.
    addr[link_id] ← GET_SOURCE (advt)            // Extract neighbor's address
    neighbor_vector ← GET_PATH_VECTOR (advt)     //    and path vector.
    for each neighbor_vector.d_addr do           // Look for better paths.
        new_path ←{addr[link_id], neighbor_vector[d_addr]}     // Build potential path.
        if my_addr is not in new_path then       // Skip it if I'm in it.
            if my_vector[d_addr] = NULL) then    // Is it a new destination?
                my_vector[d_addr] ← new_path     // Yes, add this one.
            else                                 // Not new; if better, use it.
                my_vector[d_addr] ← SELECT_PATH (new_path, my_vector[d_addr])
    FLUSH_AND_REBUILD (link_id)

procedure SELECT_PATH (new, old)        // Decide if new path is better than old one.
    if first_hop(new) = first_hop(old) then return new  // Update any path we were
                                                        // already using.
    else if length(new) ≥ length(old) then return old   // We know a shorter path, keep
    else return new                                     // OK, the new one looks better.

procedure FLUSH_AND_REBUILD (link_id)   // Flush out stale paths from this neighbor.
    for each d_addr in my_vector
        if first_hop(my_vector[d_addr]) = addr[link_id] and new_vector[d_addr] = NULL
            then
            delete my_vector[d_addr]    // Delete paths that are no longer advertised.
    REBUILD_FORWARDING_TABLE (my_vector, addr)      // Pass info to forwarder.
```

Figure 1.5.5: Model implementation of a path vector exchange routing algorithm. These procedures run in every participating router. They assume that the link layer discards damaged packets. If an advertisement is lost, it is of little consequence because the next advertisement will replace it The procedure `REBUILD_FORWARDING_TABLE` is not shown; it simply constructs a new forwarding table for use by this router, using the latest path vector information.

When designing a routing algorithm, there are a number of questions that one should ask. Does the algorithm converge? (Because it selects the shortest path this algorithm will converge, assuming that the topology remains constant.) How rapidly does it converge? (If the shortest path from a router to some participant is N steps, then this algorithm will insert that shortest path in that router's table after N advertising/path-selection exchanges.) Does it respond equally well to link deletions? (No, it can take longer to convince all participants of deletions. On the other hand, there are other algorithms—such as distance vector, which passes around just the lengths of paths rather than the paths themselves—that are much worse.) Is it safe to send traffic before the algorithm converges? (If a link has gone down, some packets may loop for a while until everyone agrees on the new forwarding tables. This problem is serious, but in the next paragraph we will see how to fix it by discarding packets that have been forwarded too many times.) How many destinations can it reasonably handle? (The Border Gateway Protocol, which uses a path vector algorithm similar to the one described above, has been used in the Internet to exchange information concerning 100,000 or so routes.)

The possibility of temporary loops in the forwarding tables or more general routing table inconsistencies, buggy routing algorithms, or misconfigurations can be dealt with by a network layer mechanism known as the **hop limit**. The idea is to add a field to the network-layer header containing a hop limit counter. The originator of the packet initializes the hop limit. Each router that handles the packet decrements the hop limit by one as the packet goes by. If a router finds that the resulting value is zero, it discards the packet. The hop limit is thus a safety net that ensures that no packet continues bouncing around the network forever.

There are some obvious refinements that can be made to the path vector algorithm. For example, since nodes such as A, B, C, D, and F are connected by only one link to the rest of the network, they can skip the path selection step and just assume that all destinations are reachable via their one link—but when they first join the network they must do an advertising step, to ensure that the rest of the network knows how to reach them (and it would be wise to occasionally repeat the advertising step, to make sure that link failures and router restarts don't cause them to be forgotten). A service node such as E, which has two links to the network but is not intended to be used for transit traffic, may decide never to advertise anything more than the path to itself. Because each participant can independently decide which paths it advertises, path vector exchange is sometimes used to implement restrictive routing policies. For example, a country might decide that packets that both originate and terminate domestically should not be allowed to transit another country, even if that country advertises a shorter path.

The exchange of data among routers is just another example of a network layer protocol. Since the link layer already provides network layer protocol multiplexing, no extra effort is needed to add a routing protocol to the layered system. Further, there is nothing preventing different

groups of routers from choosing to use different routing protocols among themselves. In the Internet, there are many different routing protocols simultaneously in use, and it is common for a single router to use different routing protocols over different links.

1.5.3: Hierarchical Address Assignment and Hierarchical Routing

The system for identifying attachment points of a network as described so far is workable, but does not scale up well to large numbers of attachment points. There are two immediate problems:

1. Every attachment point must have a unique address. If there are just ten attachment points, all located in the same room, coming up with a unique identifier for an eleventh is not difficult. But if there are several hundred million attachment points in locations around the world, as in the Internet, it is hard to maintain a complete and accurate list of addresses already assigned.
2. The path vector grows in size with the number of attachment points. Again, for routers to exchange a path vector with ten entries is not a problem; a path vector with 100 million entries could be a hassle.

The usual way to tackle these two problems is to introduce hierarchy: invent some scheme by which network addresses have a hierarchical structure that we can take advantage of, both for decentralizing address assignments and for reducing the size of forwarding tables and path vectors.

For example, consider again the abstract network of Figure 1.5.1, in which we arbitrarily assigned two-digit numbers as network addresses. Suppose we instead adopt a more structured network address consisting, say, of two parts, which we might call "region" and "station". Thus in Figure 1.5.4 we might assign to A the network address "11,75" where 11 is a region identifier and 75 is a station identifier.

By itself, this change merely complicates things. However, if we also adopt a policy that regions must correspond to the set of network attachment points served by an identifiable group of closely-connected routers, we have a lever that we can use to reduce the size of forwarding tables and path vectors. Whenever a router for region 11 gets ready to advertise its path vector to a router that serves region 12, it can condense all of the paths for the region 11 network destinations it knows about into a single path, and simply advertise that it knows how to forward things to any region 11 network destination. The routers that serve region 11 must, of course, still maintain complete path vectors for every region 11 station, and exchange those vectors among themselves, but these vectors are now proportional in size to the number of attachment points in region 11, rather than to the number of attachment points in the whole network.

When a network uses hierarchical addresses, the operation of forwarding involves the same steps as before, but the table lookup process is slightly more complicated: The forwarder must first extract the region component of the destination address and look that up in its forwarding table. This lookup has two possible outcomes: either the forwarding table contains an entry showing a link over which to send the packet to that region, or the forwarding table contains an entry saying that this forwarder is already in the destination region, in which case it is necessary to extract the station identifier from the destination address and look that up in a distinct part of the forwarding table. In most implementations, the structure of the forwarding table reflects the hierarchical structure of network addresses. Figure 1.5.6 illustrates the use of a forwarding table for hierarchical addresses that is constructed of two sections.

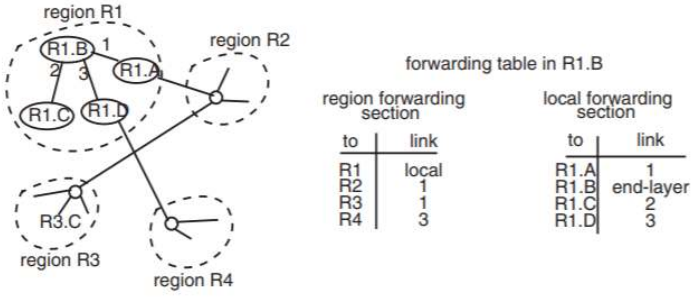


Figure 1.5.6: Example of a forwarding table with regional addressing in network node R1.B. The forwarder first looks up the region identifier in the region forwarding section of the table. If the target address is R3.C, the region identifier is R3, so the table tells it that it should forward the packet on link 1. If the target address is R1.C, which is in its own region R1, the region forwarding table tells it that R1 is the local region, so it then looks up R1.C in the local forwarding section of the table. There may be hundreds of network attachment points in region R3, but just one entry is needed in the forwarding table at node R1.B.

Hierarchical addresses also offer an opportunity to grapple with the problem of assigning unique addresses in a large network because the station part of a network address needs to be unique only within its region. A central authority can assign region identifiers, while different local authorities can assign the station identifiers within each region, without consulting other regional authorities. For this decentralization to work, the boundaries of each local administrative authority must coincide with the boundaries of the regions served by the packet forwarders. While this seems like a simple thing to arrange, it can actually be problematic. One easy way to define regions of closely connected packet

forwarders is to do it geographically. However, administrative authority is often not organized on a strictly geographic basis. So there may be a significant tension between the needs of address assignment and the needs of packet forwarding.

Hierarchical network addresses are not a panacea—in addition to complexity, they introduce at least two new problems. With the non-hierarchical scheme, the geographical location of a network attachment point did not matter, so a portable computer could, for example, connect to the network in either Boston or San Francisco, announce its network address, and after the routers have exchanged path vectors a few times, expect to communicate with its peers. But with hierarchical routing, this feature stops working. When a portable computer attaches to the network in a different region, it cannot simply advertise the same network address that it had in its old region. It will instead have to first acquire a network address within the region to which it is attaching. In addition, unless some provision has been made at the old address for forwarding, other stations in the network that remember the old network address will find that they receive no-answer responses when they try to contact this station, even though it is again attached to the network.

The second complication is that paths may no longer be the shortest possible because the path vector algorithm is working with less detailed information. If there are two different routers in region 5 that have paths leading to region 7, the algorithm will choose the path to the nearest of those two routers, even though the other router may be much closer to the actual destination inside region 7.

We have used in this example a network address with two hierarchical levels, but the same principle can be extended to as many levels as are needed to manage the network. In fact, any region can do hierarchical addressing within just the part of the address space that it controls, so the number of hierarchical levels can be different in different places. The public Internet uses just two hierarchical addressing levels, but some large subnetworks of the Internet implement the second level internally as a two-level hierarchy. Similarly, North American telephone providers have created a four-level hierarchy for telephone numbers: country code, area code, exchange, and line number, for exactly the same reasons: to reduce the size of the tables used in routing calls, and to allow local administration of line numbers. Other countries agree on the country codes but internally may have a different number of hierarchical levels.

1.5.4: Reporting Network Layer Errors

The network layer can encounter trouble when trying to forward a packet, so it needs a way of reporting that trouble. The network layer is in a uniquely awkward position when this happens because the usual reporting method (return a status value to the higher-layer program that asked for this operation) may not be available. An intermediate router receives a packet from a link layer below, and it is expected to forward that packet via another link layer. Even if there is a higher layer in the router, that layer probably has no interest in this packet. Instead, the entity that needs to hear about the problem is more likely to be the upper layer program that originated the packet, and that program may be located several hops away in another computer. Even the network layer at the destination address may need to report something to the original sender such as the lack of an upper-layer handler for the end-to-end type that the sender specified.

The obvious thing to do is send a message to the entity that needs to know about the problem. The usual method is that the network layer of the router creates a new packet on the spot and sends it back to the source address shown in the problem packet. The message in this new packet reports details of the problem using some standard error reporting protocol. With this design, the original higher-layer sender of a packet is expected to listen not only for replies but also for messages of the error reporting protocol. Here are some typical error reports:

- The buffers of the router were full, so the packet had to be discarded.
- The buffers of the router are getting full—please stop sending so many packets.
- The region identifier part of the target address does not exist.
- The station identifier part of the target address does not exist.
- The end type identifier was not recognized.
- The packet is larger than the maximum transmission unit of the next link.
- The packet hop limit has been exceeded.

In addition, a copy of the header of the doomed packet goes into a data field of the error message, so that the recipient can match it with an outstanding `SEND` request.

One might suggest that a router send an error report when discarding a packet that is received with a wrong checksum. This idea is not as good as it sounds because a damaged packet may have garbled header information, in which case the error message might be sent to a wrong —or even nonexistent—place. Once a packet has been identified as containing unknown damage, it is not a good idea to take any action that depends on its contents.

A network-layer error reporting protocol is a bit unusual. An error message originates in the network layer, but is delivered to the end-to-end layer. Since it crosses layers, it can be seen as violating (in a minor way) the usual separation of layers: we have a network layer program preparing an end-to-end header and inserting end-to-end data; a strict layer doctrine would insist that the network layer not touch anything but network layer headers.

An error reporting protocol is usually specified to be a best-effort protocol, rather than one that takes heroic efforts to get the message through. There are two reasons why this design decision makes sense. First, as will be seen in Section 1.6, implementing a more reliable protocol adds a fair amount of machinery: timers, keeping copies of messages in case they need to be retransmitted, and watching for receipt acknowledgments. The network layer is not usually equipped to do any of these functions, and not implementing them minimizes the violation of layer separation. Second, error messages can be thought of as hints that allow the originator of a packet to more quickly discover a problem. If an error message gets lost, the originator should, one way or another, eventually discover the problem in some other way, perhaps after timing out, resending the original packet, and getting an error message on the retry.

A good example of the best-effort nature of an error reporting protocol is that it is common to not send an error message about every discarded packet; if congestion is causing the discard rate to climb, that is exactly the wrong time to increase the network load by sending many “I discarded your packet” notices. But sending a few such notices can help alert sources who are flooding the network that they need to back off—this topic is explored in more depth in Section 1.7.

The basic idea of an error reporting protocol can be used for other communications to and from the network layer of any participant in the network. For example, the Internet has a protocol named **internet control message protocol** (ICMP) that includes an echo request message (also known as a "ping," from an analogy with submarine active sonar systems). If an end node sends an echo request to any network participant, whether a packet forwarder or another end node, the network layer in that participant is expected to respond by immediately sending the data of the message back to the sender in an echo reply message. Echo request/reply messages are widely used to determine whether or not a participant is actually up and running. They are also sometimes used to assess network congestion by measuring the time until the reply comes back.

Another useful network error report is "hop limit exceeded". Recall from Section 1.5.2 that to provide a safety net against the possibility of forwarding loops, a packet may contain a hop limit field, which a router decrements in each packet that it forwards. If a router finds that the hop limit field contains zero, it discards the packet and it also sends back a message containing the error report. The "hop limit exceeded" error message provides feedback to the originator, for example it may have chosen a hop limit that is too small for the network configuration. The "hop limit exceeded" error message can also be used in an interesting way to help locate network problems: send a test message (usually called a probe) to some distant destination address, but with the hop limit set to 1. This probe will cause the first router that sees it to send back a “hop limit exceeded” message whose source address identifies that first router. Repeat the experiment, sending probes with hop limits set to 2, 3,…, etc. Each response will reveal the network address of the next router along the current path between the source and the destination. In addition, the time required for the response to return gives a rough indication of the network load between the source and that router. In this way one can trace the current path through the network to the destination address, and identify points of congestion.

Another way to use an error reporting protocol is for the end-to-end layer to send a series of probes to learn the smallest maximum transmission unit (MTU) that lies on the current path between it and another network attachment point. It first sends a packet of the largest size the application has in mind. If this probe results in an "MTU exceeded" error response, it halves the packet size and tries again. A continued binary search will quickly home in on the smallest MTU along the path. This procedure is known as **MTU discovery**.

1.5.5: Network Address Translation (An Idea That Almost Works)

From a naming point of view, the Internet provides a layered naming environment with two contexts for its network attachment points, known as "Internet addresses". An Internet address has two components, a network number and a host number. Most network numbers are global names, but a few, such as network 10, are designated for use in private networks. These network numbers can be used either completely privately, or in conjunction with the public Internet. Completely private use involves setting up an independent private network, and assigning host addresses using the network number 10. Routers within this network advertise and forward just as in the public Internet. Routers on the public Internet follow the convention that they do not accept routes to network 10, so if this private network is also directly attached to the public Internet, there is no confusion. Assuming that the private network accepts routes to globally named networks, a host inside the private network could send a message to a host on the public Internet, but a host on the public Internet cannot send a response back because of the routing convention. Thus any number of private networks can each independently assign numbers using network number 10—but hosts on different private networks cannot talk to one another and hosts on the public Internet cannot talk to them.

Network Address Translation (NAT) is a scheme to bridge this gap. The idea is that a specialized translating router (known informally as a "NAT box") stands at the border between a private network and the public Internet. When a host inside the private network wishes to communicate with a service on the public Internet, it first makes a request to the translating router. The translator sets up a binding between that host’s private address and a temporarily assigned public address, which the translator advertises to the public Internet. The private host then launches a packet that has a destination address in the public Internet, and its own private network source address. As this packet passes through the translating router, the translator modifies the source address by replacing it with the temporarily assigned public address. It then sends the packet on its way into the public Internet. When a response from the service on the public Internet comes back to the translating router, the translator extracts the destination address from the response, looks it up in its table of temporarily assigned public addresses, finds

the internal address to which it corresponds, modifies the destination address in the packet, and sends the packet on its way on the internal network, where it finds its way to the private host that initiated the communication.

The scheme works, after a fashion, but it has a number of limitations. The most severe limitation is that some end-to-end network protocols place Internet addresses in fields buried in their payloads; there is nothing restricting Internet addresses to packet source and destination fields of the network layer header. For example, some protocols between two parties start by mentioning the Internet address of a third party, such as a bank, that must also participate in the protocol. If the Internet address of the third party is on the public Internet, there may be no problem, but if it is an address on the private network, the translator needs to translate it as it goes by. The trouble is that translation requires that the translator peer into the payload data of the packet and understand the format of the higher-layer protocol. The result is that NAT works only for those protocols that the translator is programmed to understand. Some protocols may present great difficulties. For example, if a secure protocol uses key-driven cryptographic transformations for either privacy or authentication, the NAT gateway would need to have a copy of the keys, but giving it the keys may defeat the purpose of the secure protocol. (This concern will become clearer after reading Chapter 5.

A second problem is that all of the packets between the public Internet and the private network must pass through the translating router, since it is the only place that knows how to do the address translation. The translator thus introduces both a potential bottleneck and a potential single point of failure, and NAT becomes a constraint on routing policy.

A third problem arises if two such organizations later merge. Each organization will have assigned addresses in network 10, but since their assignments were not coordinated, some addresses will probably have been assigned in both organizations, and all of the colliding addresses must be discovered and changed.

Although originally devised as a scheme to interconnect private networks to the public Internet, NAT has become popular as a technique to beef up security of computer systems that have insecure operating system or network implementations. In this application, the NAT translator inspects every packet coming from the public Internet and refuses to pass along any whose origin seems suspicious or that try to invoke services that are not intended for public use. The scheme does not in itself provide much security, but in conjunction with other security mechanisms described in Chapter 5, it can help create what that chapter describes as "defense in depth".

This page titled 1.5: The Network Layer is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

1.6: The End-to-End Layer

The network layer provides a useful but not completely dependable best-effort communication environment that will deliver data segments to any destination, but with no guarantees about delay, order of arrival, certainty of arrival, accuracy of content, or even of delivery to the right place. This environment is too hostile for most applications, and the job of the end-to-end layer is to create a more comfortable communication environment that has the features of performance, reliability, and certainty that an application needs. The complication is that different applications can have quite different communication needs, so no single end-to-end design is likely to suffice. At the same time, applications tend to fall in classes all of whose members have somewhat similar requirements. For each such class it is usually possible to design a broadly useful protocol, known as a **transport protocol**, for use by all the members of the class.

1.6.1: Transport Protocols and Protocol Multiplexing

A transport protocol operates between two attachment points of a network, with the goal of moving either messages or a stream of data between those points while providing a particular set of specified assurances. It is convenient to distinguish the two attachment points by referring to the application program that initiates action as the client and the application program that responds as the service. At the same time, data may flow either from client to service, from service to client, or both, so we will need to refer to the sending and receiving sides for each message or stream. Transport protocols almost always include multiplexing, to tell the receiving side to which application it should deliver the message or direct the stream. Because the mechanics of application multiplexing can be more intricate than in lower layers, we first describe a transport protocol interface that omits multiplexing, and then add multiplexing to the interface.

In contrast with the network layer, where an important feature is a uniform application programming interface, the interface to an end-to-end transport protocol varies with the particular end-to-end semantics that the protocol provides. Thus a simple **message-sending protocol** that is intended to be used by only one application might have a first-version interface such as:

```
v.1    SEND_MESSAGE (destination, message)
```

in which, in addition to supplying the content of the message, the sender specifies in `destination` the network attachment point to which the message should be delivered. The sender of a message needs to know both the message format that the recipient expects and the destination address. There are several methods of discovering destination addresses, any of which might be used.

The prospective receiver must provide an interface by which the transport protocol delivers the message to the application. Just as in the link and network layers, receiving a message can't happen until the message arrives, so receiving involves waiting and the corresponding receive-side interface depends on the system mechanisms that are available for waiting and for thread or event coordination. For illustration, we again use an upcall: when a message arrives, the message transport protocol delivers it by calling an application-provided procedure entry point:

```
v.1    DELIVER_MESSAGE (message)
```

This first version of an upcall interface omits not only multiplexing but another important requirement: When sending a message, the sender usually expects a reply. While a programmer may be able to ask someone down the hall the appropriate destination address to use for some service, it is usually the case that a service has many clients. Thus the service needs to know where each message came from so that it can send a reply. A message transport protocol usually provides this information, for example by including a second argument in the upcall interface:

```
v.2    DELIVER_MESSAGE (source, message)
```

In this second (but not quite final) version of the upcall, the transport protocol sets the value of `source` to the address from which this message originated. The transport protocol obtains the value of `source` as an argument of an upcall from the network layer.

Since the reason for designing a message transport protocol is that it is expected to be useful to several applications, the interface needs additional information to allow the protocol to know which messages belong to which application. End-to-end layer multiplexing is generally a bit more complicated than that of lower layers because not only can there be multiple applications, there can be multiple **instances** of the same application using the same transport protocol. Rather than assigning a single multiplexing identifier to an application, each instance of an application receives a distinct multiplexing identifier, usually known as a **port**. In a client/service situation, most application services advertise one of these identifiers, called that application's **well-known port**. Thus the second (and again not final) version of the send interface is

```
v.2     SEND_MESSAGE (destination, service_port, message)
```

where *service_port* identifies the well-known port of the application service to which the sender wants to have the message delivered. At the receiving side each application that expects to receive messages needs to tell the message transport protocol what port it expects clients to use, and it must also tell the protocol what program to call to deliver messages. The application can provide both pieces of information invoking the transport protocol procedure

```
LISTEN_FOR_MESSAGES (service_port, message_handler)
```

which alerts the transport protocol implementation that whenever a message arrives at

this destination carrying the port identifier *service_port* , the protocol should deliver it by calling the procedure named in the second argument (that is, the procedure *message_handler*). LISTEN_FOR_MESSAGES enters its two arguments in a transport layer table for future reference. Later, when the transport protocol receives a message and is ready to deliver it, it invokes a dispatcher similar to that of Figure 1.4.8, from Section 1.4.4. The dispatcher looks in the table for the service port that came with the message, identifies the associated *message_handler* procedure, and calls it, giving as arguments the *source* and the *message* .

One might expect that the service might send replies back to the client using the same application port number, but since one service might have several clients at the same network attachment point, each client instance will typically choose a distinct port number for its own replies, and the service needs to know to which port to send the reply. So the SEND interface must be extended one final time to allow the sender to specify a port number to use for reply:

```
v.3     SEND_MESSAGE (destination, service_port, reply_port, message)
```

where *reply_port* is the identifier that the service can use to send a message back to this particular client. When the service does send its reply message, it may similarly specify a *reply_port* that is different from its well-known port if it expects that same client to send further, related messages. The *reply_port* arguments in the two directions thus allow a series of messages between a client and a service to be associated with one another.

Having added the port number to SEND_MESSAGE , we must communicate that port number to the recipient by adding an argument to the upcall by the message transport protocol when it delivers a message to the recipient:

```
v.3     DELIVER_MESSAGE (source, reply_port, message)
```

This third and final version of DELIVER_MESSAGE is the handler that the application designated when it called LISTEN_FOR_MESSAGES . The three arguments tell the handler (1) who sent the message (*source*), (2) the port on which that sender said it will listen for a possible reply (*reply_port*) and (3) the content of the message itself (*message*).

The interface set { LISTEN_FOR_MESSAGE, SEND_MESSAGE, DELIVER_MESSAGE } is specialized to end-to-end transport of discrete messages. Sidebar 1.6.1 illustrates two other, somewhat different end-to-end transport protocol interfaces: one for a request/response protocol and the second for streams. Each different transport protocol can be thought of as a prepackaged set of improvements on the best-effort contract of the network layer.

Sidebar 1.6.1

Other end-to-end transport protocol interfaces

Since there are many different combinations of services that an end-to-end transport protocol might provide, there are equally many transport protocol interfaces. Here are two more examples:

1. A **request/response protocol** sends a request message and waits for a response to that message before returning to the application. Since an interface that waits for a response ensures that there can be only one such call per thread outstanding, neither an explicit multiplexing parameter nor an upcall are necessary. A typical client interface to a request/response transport protocol is

```
response ← SEND_REQUEST (service_identifier, request)
```

where *service_identifier* is a name used by the transport protocol to locate the service destination and service port. It then sends a message, waits for a matching response, and delivers the result. The corresponding application programming interface at the service side of a request/response protocol may be equally simple or it can be quite complex, depending on the performance requirements.

2. A **reliable message stream protocol** sends several messages to the same destination with the intent that they be delivered reliably and in the order in which they were sent. There are many ways of defining a stream protocol interface. In the following example, an application client begins by creating a stream:

```
client_stream_id ← OPEN_STREAM (destination, service_port, reply_port)
```

followed by several invocations of:

```
WRITE_STREAM (client_stream_id, message)
```

and finally ends with:

```
CLOSE_STREAM (client_stream_id)
```

The service-side programming interface allows for several streams to be coming in to an application at the same time. The application starts by calling a `LISTEN_FOR_STREAMS` procedure to post a listener on the service port, just as with the message interface. When a client opens a new stream, the service's network layer, upon receiving the open request, upcalls to the stream listener that the application posted:

```
OPEN_STREAM_REQUEST (source, reply_port)
```

Upon receiving such an upcall, `OPEN_STREAM_REQUEST` assigns a stream identifier for use within the service and invokes a transport layer dispatcher with

```
ACCEPT_STREAM (service_stream_id, next_message_handler)
```

The arrival of each message on the stream then leads the dispatcher to perform an upcall to the program identified in the variable `next_message_handler` :

```
HANDLE_NEXT_MESSAGE (stream_id, message);
```

With this design, a `message` value of `NULL` might signal that the client has closed the stream.

Here are three examples of transport protocols used widely in the Internet, and the assurances they provide:

1. **User datagram protocol (UDP)**. This protocol adds ports for multiple applications and a checksum for data integrity to the network-layer packet. Although UDP is used directly for some simple request/reply applications such as asking for the time of day or looking up the network address of a service, its primary use is as a component of other message transport protocols, to provide end-to-end multiplexing and data integrity. [For details, see Internet standard STD0006 or Internet request for comments RFC–768.]
2. **Transmission control protocol (TCP).** Provides a stream of bytes with the assurances that data is delivered in the order it was originally sent, nothing is missing, nothing is duplicated, and that the data has a modest (but not terribly high) probability of integrity. There is also provision for flow control, which means that the sender takes care not to overrun the ability of the receiver to accept data, and TCP cooperates with the network layer to avoid congestion. This protocol is used for applications such as interactive typing that require a telephone-like connection in which the order of delivery of data is important. (It is also used in many bulk transfer applications that do not require delivery order, but that do want to take advantage of its data integrity, flow control, and congestion avoidance assurances.) [For details, see Internet standard STD0007 or Internet request for comments RFC–793.]
3. **Real-time transport protocol (RTP)**. Built on UDP (but with checksums switched off), RTP provides a stream of time-stamped packets with no other integrity guarantee. This kind of protocol is useful for applications such as streaming video or voice, where order and stream timing are important, but an occasional lost packet is not a catastrophe, so out-of-order packets can be discarded, and packets with bits in error may still contain useful data. [For details, see Internet request for comments RFC–1889.]

There have, over the years, been several other transport protocols designed for use with the Internet, but they have not found enough application to be widely implemented. There are also several end-to-end protocols that provide services in addition to message transport, such as file transfer, file access, remote procedure call, and remote system management, and that are built using UDP or TCP as their underlying transport mechanism. These protocols are usually classified as **presentation protocols** because the primary additional service they provide is translating data formats between different computer platforms. This collection of protocols illustrates that the end-to-end layer is itself sometimes layered and sometimes not, depending on the requirements of the application.

Finally, end-to-end protocols can be **multipoint**, which means they involve more than two players. For example, to complete a purchase transaction, there may be a buyer, a seller, and one or more banks, each of which needs various end-to-end assurances about agreement, order of delivery, and data integrity.

In the next several sections, we explore techniques for providing various kinds of end-to-end assurances. Any of these techniques may be applied in the design of a message transport protocol, a presentation protocol, or by the application itself.

1.6.2: Assurance of At-Least-Once Delivery; the Role of Timers

A property of a best-effort network is that it may lose packets, so a goal of many end-to-end transport protocols is to eliminate the resulting uncertainty about delivery. A persistent sender is a protocol participant that tries to ensure that at least one copy of each data segment is delivered, by sending it repeatedly until it receives an acknowledgment. The usual implementation of a persistent sender is to add to the application data a header containing a nonce and to set a timer that the designer estimates will expire in a little more than one network round-trip time; this is the sum of the network transit time for the outbound segment, the time the receiver spends absorbing the segment and preparing an acknowledgment, and the network transit time for the acknowledgment. Having set the timer, the sender passes the segment to the network layer for delivery, taking care to keep a copy. The receiving side of the protocol strips off the end-to-end header, passes the application data along to the application, and in addition sends back an acknowledgment that contains the nonce. When the acknowledgment gets back to the sender, the sender uses the nonce to identify which previously-sent segment is being acknowledged. It then turns off the corresponding timer and discards its copy of that segment. If the timer expires before the acknowledgment returns, the sender restarts the timer and resends the segment, repeating this sequence indefinitely, until it receives an acknowledgment. For its part, the receiver sends back an acknowledgment every time it receives a segment, thereby extending the persistence in the reverse direction, thus covering the possibility that the best-effort network has lost one or more acknowledgments.

A protocol that includes a persistent sender does its best to provide an assurance of at-least-once delivery: the nonce, timer, retry, and acknowledgment together act to ensure that the data segment will eventually get through. As long as there is a non-zero probability of a message getting through, this protocol will eventually succeed. On the other hand, the probability may actually be zero, either for an indefinite time—perhaps the network is partitioned or the destination is not currently listening, or permanently—perhaps the destination is on a ship that has sunk. Because of the possibility that there will not be an acknowledgment forthcoming soon, or perhaps ever, a practical sender is not infinitely persistent. The sender limits the number of retries, and if the number exceeds the limit, the sender returns error status to the application that asked to send the message. The application must interpret this error status with some understanding of network communications. The lack of an acknowledgment means that one of two—significantly different—events has occurred:

1. The data segment was not delivered.
2. The data segment was delivered, but the acknowledgment never returned.

The good news is that the application is now aware that there is a problem. The bad news is that there is no way to determine which of the two problems occurred. This dilemma is intrinsic to communication systems, and the appropriate response depends on the particular application. Some applications will respond to this dilemma by making a note to later ask the other side whether or not it got the message; other applications may just ignore the problem. Chapter 4 investigates this issue further.

In summary, the at-least-once delivery protocol does not provide the absolute assurance that its name implies; it instead provides the assurance that if it is possible to get through, the message will get through, and if it is not possible to confirm delivery, the application will know about it.

The at-least-once delivery protocol provides no assurance about duplicates—it actually tends to generate duplicates. Furthermore, the assurance of delivery is weaker than what it appears to be on the surface: the data may have been corrupted along the way, or it may have been delivered to the wrong destination—and acknowledged—by mistake. Assurances on any of those points require additional techniques. Finally, the at-least-once delivery protocol ensures only that the message was delivered, not that the application actually acted on it—the receiving system may have been so overloaded that it ignored the message or it may have crashed an instant after acknowledging the message. When examining end-to-end assurances, it is important to identify the end points. In this case, the receiving end point is the place in the protocol code that sends the acknowledgment of message receipt.

This protocol requires the sender to choose a value for the retry timer at the time it sends a packet. One possibility would be to choose in advance a timer value to be used for every packet—**a fixed timer**. But using a timer value fixed in advance is

problematic because there is no good way to make that choice. To detect a lost packet by noticing that no acknowledgment has returned, the appropriate timer interval would be the expected network round-trip time plus some allowance for unusual queuing delays. But even the expected round-trip time between two given points can vary by quite a bit when routes change. In fact, one can argue that since the path to be followed and the amount of queuing to be tolerated is up to the network layer, and the individual transit times of links are properties of the link layer, for the end-to-end layer to choose a fixed value for the timer interval would violate the layering abstraction—it would require that the end-to-end layer know something about the internal implementation of the link and network layers.

Even if we are willing to ignore the abstraction concern, the end-to-end transport protocol designer has a dilemma in choosing a fixed timer interval. If the designer chooses too short an interval, there is a risk that the protocol will resend packets unnecessarily, which wastes network capacity as well as resources at both the sending and receiving ends. But if the designer sets the timer too long, then genuinely lost packets will take a long time to discover, so recovery will be delayed and overall performance will decline. Worse, setting a fixed value for a timer will not only force the designer to choose between these two evils, it will also embed in the system a lurking surprise that may emerge long in the future when someone else changes the system, for example to use a faster network connection. Going over old code to understand the rationale for setting the timers and choosing new values for them is a dismal activity that one would prefer to avoid by better design.

An **adaptive timer** is one whose setting dynamically adjusts to currently observed conditions. A common implementation scheme is to observe the round-trip times for each data segment and its corresponding response and calculate an exponentially weighted moving average of those measurements (Sidebar 1.6.2 explains the method). The protocol then sets its timers to, say, 150% of that estimate, with the intent that minor variations in queuing delay should rarely cause the timer to expire. Keeping an estimate of the round-trip time turns out to be useful for other purposes, too. An example appears in the discussion of flow control in Section 1.6.6 below.

Sidebar 1.6.2

Exponentially weighted moving averages

One way of keeping a running average A of a series of measurements M_i is to calculate an **exponentially weighted moving average**, defined as

$$A = \left(M_0 + M_1 \times \alpha + M_2 \times \alpha^2 + M_3 \times \alpha^3 + \ldots\right) \times (1 - \alpha) \tag{1.6.1}$$

where $\alpha < 1$ and the subscript indicates the age of the measurement, with the most recent being M_0. The end multiplier $(1 - \alpha)$ normalizes the result. This scheme has two advantages over a simple average. First, it gives more weight to recent measurements. The multiplier, α, is known as the **decay factor**. A smaller value for the decay factor means that older measurements lose weight more rapidly as succeeding measurements are added into the average. The second advantage is that it can be easily calculated as new measurements become available using the recurrence relation:

$$A_{new} \leftarrow (\alpha \times A_{old} + (1 - \alpha) \times M_{new}) \tag{1.6.2}$$

where M_{new} is the latest measurement. In a high-performance environment where measurements arrive frequently and calculation time must be minimized, one can instead calculate

$$\frac{A_{new}}{(1-\alpha)} \leftarrow \left(\alpha \times \frac{A_{old}}{(1-\alpha)} + M_{new}\right) \tag{1.6.3}$$

which requires only one multiplication and one addition. Furthermore, if $(1 - \alpha)$ is chosen to be a fractional power of two (e.g., $1/8$) the multiplication can be done with one register shift and one addition. Calculated this way, the result is too large by the constant factor $1/(1 - \alpha)$, but it may be possible to take a constant factor into account at the time the average is used. In both computer systems and networks there are many situations in which it is useful to know the average value of an endless series of observations. Exponentially weighted moving averages are probably the most frequently used method.

A refinement for an adaptive timer is to assume that duplicate acknowledgments mean that the timer setting is too small, and immediately increase it. (Since a too-small timer setting would expire before the first acknowledgment returns, causing the sender to resend the original data segment, which would trigger the duplicate acknowledgment.) It is usually a good idea to make any increase a big one, for example by doubling the value previously used to set the timer. Repeatedly increasing a timer setting by

multiplying its previous value by a constant on each retry (thus succeeding timer values might be, say, 1, 2, 4, 8, 16, … seconds) is known as **exponential backoff**, a technique that we will see again in other, quite different system applications. Doubling the value, rather than multiplying by, say, ten, is a good choice because it gets within a factor of two of the "right" value quickly without overshooting too much.

Adaptive techniques are not a panacea: the protocol must still select a timer value for the first data segment, and it can be a challenge to choose a value for the decay factor (in the sidebar, the constant α) that both keeps the estimate stable and also quickly responds to changes in network conditions. The advantage of an adaptive timer comes from being able to amortize the cost of an uninformed choice on that first data segment over the ensuing several segments.

A different method for minimizing use of fixed timers is for the receiving side of a stream of data segments to infer, from the arrival of later data segments, the loss of earlier ones and to request their retransmission by sending a **negative acknowledgment (NAK)**. A NAK is simply a message that lists missing items. Since data segments may be delivered out of order, the recipient needs some way of knowing which segment is missing. For example, the sender might assign sequential numbers as nonces, so arrival of segments #13 and #14 without having previously received segment #12 might cause the recipient to send a NAK requesting retransmission of segment #12. To distinguish transmission delays from lost segments, the recipient must decide how long to wait before sending a NAK, but that decision can be made by counting later-arriving segments rather than by measuring a time interval.

Since the recipient reports lost packets, the sender does not need to be persistent, so it does not need to use a timer at all—that is, until it sends the last segment of a stream. Because the recipient can't depend on later segment arrivals to discover that the last segment has been lost, that discovery still requires the help of a timer. With NAKs, the persistent-sender strategy with a timer is needed only once per stream, so the penalty for choosing a timer setting that is too long (or too short) is just one excessive delay (or one risk of an unnecessary duplicate transmission) on the last segment of the stream. Compared with using an adaptive timer on every segment of the stream, this is probably an improvement.

The appropriate conclusion about timers is that fixed timers are a terrible mechanism to include in an end-to-end protocol (or indeed anywhere—this conclusion applies to many applications of timers in systems). Adaptive timers work better, but add complexity and require careful thought to make them stable. Avoidance and minimization of timers are the better strategies, but it is usually impossible to completely eliminate them. Where timers must be used they should be designed with care and the designer should clearly document them as potential trouble spots.

1.6.3: Assurance of At-Most-Once Delivery: Duplicate Suppression

At-least-once delivery assurance was accomplished by remembering state at the sending side of the transport protocol: a copy of the data segment, its nonce, and a flag indicating that an acknowledgment is still needed. But a side effect of at-least-once delivery is that it tends to generate duplicates. To ensure **at-most-once** delivery, it is necessary to suppress these duplicates, as well as any other duplicates created elsewhere within the network, perhaps by a persistent sender in some link-layer protocol.

The mechanism of suppressing duplicates is a mirror image of the mechanism of at-least-once delivery: add state at the receiving side. We saw a preview of this mechanism in Section 1.2—the receiving side maintains a table of previously-seen nonces. Whenever a data segment arrives, the transport layer implementation checks the nonce of the incoming segment against the list of previously-seen nonces. If this nonce is new, it adds the nonce to the list, delivers the data segment to the application, and sends an acknowledgment back to the sender. If the nonce is already in its list, it discards the data segment, but it resends the acknowledgment, in case the sender did not receive the previous one. If, in addition, the application has already sent a response to the original request, the transport protocol also resends that response.

The main problem with this technique is that the list of nonces maintained at the receiving side of the transport protocol may grow indefinitely, taking up space and, whenever a data segment arrives, taking time to search. Because they may have to be kept indefinitely, these nonces are described colorfully as **tombstones**. A challenge in designing a duplicate-suppression technique is to avoid accumulating an unlimited number of tombstones.

One possibility is for the sending side to use monotonically increasing sequence numbers for nonces, and include as an additional field in the end-to-end header of every data segment the highest sequence number for which it has received an acknowledgment. The receiving side can then discard that nonce and any others from that sender that are smaller, but it must continue to hold a nonce for the most recently-received data segment. This technique reduces the magnitude of the problem, but it leaves a dawning

realization that it may never be possible to discard the *last* nonce, which threatens to become a genuine tombstone, one per sender. Two pragmatic responses to the tombstone problem are:

1. Move the problem somewhere else. For example, change the port number on which the protocol accepts new requests. The protocol should never reuse the old port number (the old port number becomes the tombstone), but if the port number space is large then it doesn't matter.
2. Accept the possibility of making a mistake, but make its probability vanishingly small. If the sending side of the transport protocol always gives up and stops resending requests after, say, five retries, then the receiving side can safely discard nonces that are older than five network round-trip times plus some allowance for unusually large delays. This approach requires keeping track of the age of each nonce in the table, and it has some chance of failing if a packet that the network delayed a long time finally shows up. A simple defense against this form of failure is to wait a long time before discarding a tombstone.

Another form of the same problem concerns what to do when the computer at the receiving side crashes and restarts, losing its volatile memory. If the receiving side stores the list of previously handled nonces in volatile memory, following a crash it will not be able to recognize duplicates of packets that it handled before the crash. But if it stores that list in a non-volatile storage device such as a hard disk, it will have to do one write to that storage device for every message received. Writes to non-volatile media tend to be slow, so this approach may introduce a significant performance loss. To solve the problem without giving up performance, techniques parallel to the last two above are typically employed. For example, one can use a new port number each time the system restarts. This technique requires remembering which port number was last used, but that number can be stored on a disk without hurting performance because it changes only once per restart. Or, if we know that the sending side of the transport protocol always gives up after some number of retries, whenever the receiving side restarts, it can simply ignore all packets until that number of round-trip times has passed since restarting. Either procedure may force the sending side to report delivery failure to its application, but that may be better than taking the risk of accepting duplicate data.

When techniques for at-least-once delivery (the persistent sender) and at-most-once delivery (duplicate detection) are combined, they produce an assurance that is called **exactly-once** delivery. Despite its name, and even if the sender is prepared to be infinitely persistent, exactly-once delivery is not a guarantee that the message will eventually be delivered. Instead, it ensures that if the message is delivered, it will be delivered only once, and if delivery fails, the sender will learn, by lack of acknowledgment despite repeated requests, that delivery probably failed. However, even if no acknowledgment returns, there is a still a possibility that the message was delivered. Section 3.7.2 introduces a protocol known as **two-phase commit** that can reduce the uncertainty by adding a persistent sender of the acknowledgement. Unfortunately, there is no way to completely eliminate the uncertainty.

1.6.4: Division into Segments and Reassembly of Long Messages

Recall that the requirements of the application determine the length of a message, but the network sets a maximum transmission unit, arising from limits on the length of a frame at the link layer. One of the jobs of the end-to-end transport protocol is to bridge this difference. Division of messages that are too long to fit in a single packet is relatively straightforward. Each resulting data segment must contain, in its end-to-end header, an identifier to show to which message this segment belongs and a segment number indicating where in the message the segment fits (e.g., "message 914, segment 3 of 7"). The message identifier and segment number together can also serve as the nonce used to ensure at-least-once and at-most-once delivery.

Reassembly is slightly more complicated because segments of the same message may arrive at the receiving side in any order, and may be mingled with segments from other messages. The reassembly process typically consists of allocating a buffer large enough to hold the entire message, placing the segments in the proper position within that buffer as they arrive, and keeping a checklist of which segments have not yet arrived. Once the message has been completely reassembled, the receiving side of the transport protocol can deliver the message to the application and discard the checklist. Message division and reassembly is a special case of stream division and reassembly, the topic of Section 1.6.7, below.

1.6.5: Assurance of Data Integrity

Data integrity is the assurance that when a message is delivered, its contents are the same as when they left the sender. Adding data integrity to a protocol with a persistent sender creates a **reliable delivery** protocol. Two additions are required, one at the sending side and one at the receiving side. The sending side of the protocol adds a field to the end-to-end header or trailer containing a checksum of the contents of the application message. The receiving side recalculates the checksum from the received

version of the reassembled message and compares it with the checksum that came with the message. Only if the two checksums match does the transport protocol deliver the reassembled message to the application and send an acknowledgment. If the checksums do not match the receiver discards the message and waits for the sending side to resend it. (One might suggest immediately sending a NAK, to alert the sending side to resend the data identified with that nonce, rather than waiting for timers to expire. This idea has the hazard that the source address that accompanies the data may have been corrupted along with the data. For this reason, sending a NAK on a checksum error isn't usually done in end-to-end protocols. However, as was described in Section 1.4.3, requesting retransmission as soon as an error is detected is useful at the link layer, where the other end of a point-to-point link is the only possible source.)

It might seem redundant for the transport protocol to provide a checksum, given that link layer protocols often also provide checksums. The reason the transport protocol might do so is an end-to-end argument: the link layer checksums protect the data only while it is in transit on the link. During the time the data is in the memory of a forwarding node, while being divided into multiple segments, being reassembled at the receiving end, or while being copied to the destination application buffer, it is still vulnerable to undetected accidents. An end-to-end transport checksum can help defend against those threats. On the other hand, reapplying the end-to-end argument suggests that an even better place for this checksum would be in the application program. But in the real world, many applications assume that a transport-protocol checksum covers enough of the threats to integrity that they don't bother to apply their own checksum. Transport protocol checksums cater to this assumption.

As with all checksums, the assurance is not absolute. Its quality depends on the number of bits in the checksum, the structure of the checksum algorithm, and the nature of the likely errors. In addition, there remains a threat that someone has maliciously modified both the data and its checksum to match while en route; this threat is explored briefly in Section 1.6.9 below, and in more depth in Chapter 5.

A related integrity concern is that a packet might be misdelivered, perhaps because its address field has been corrupted. Worse, the unintended recipient may even acknowledge receipt of the segment in the packet, leading the sender to believe that it was correctly delivered. The transport protocol can guard against this possibility by, on the sending side, including a copy of the destination address in the end-to-end segment header, and, on the receiving side, verifying that the address is the recipient's own before delivering the packet to the application and sending an acknowledgment back.

1.6.6: End-to-End Performance: Overlapping and Flow Control

End-to-end transport of a multisegment message raises some questions of strategy for the transport protocol, including an interesting trade-off between complexity and performance. The simplest method of sending a multisegment message is to send one segment, wait for the receiving side to acknowledge that segment, then send the second segment, and so on. This protocol, known as lock-step, is illustrated in Figure 1.6.1. An important virtue of the lock-step protocol is that it is easy to see how to apply each of the previous end-to-end assurance techniques to one segment at a time. The downside is that transmitting a message that occupies N segments will take N network round-trip times. If the network transit time is large, both ends may spend most of their time waiting.

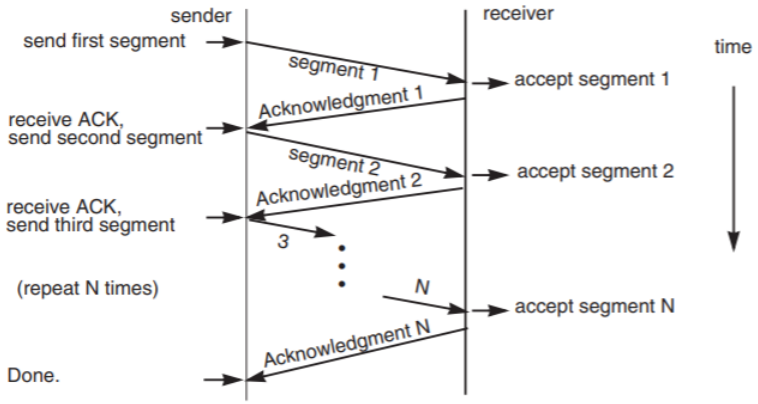


Figure 1.6.1: Lock-step transmission of multiple segments.

1.6.6.1: Overlapping Transmissions

To avoid the wait times, we can employ a pipelining technique: As soon as the first segment has been sent, immediately send the second one, then the third one, and so on, without waiting for acknowledgments. This technique allows both close spacing of transmissions and overlapping of transmissions with their corresponding acknowledgments. If nothing goes wrong, the technique leads to a timing diagram such as that of Figure 1.6.2. When the pipeline is completely filled, there may be several segments "in the net" traveling in both directions down transmission lines or sitting in the buffers of intermediate packet forwarders.

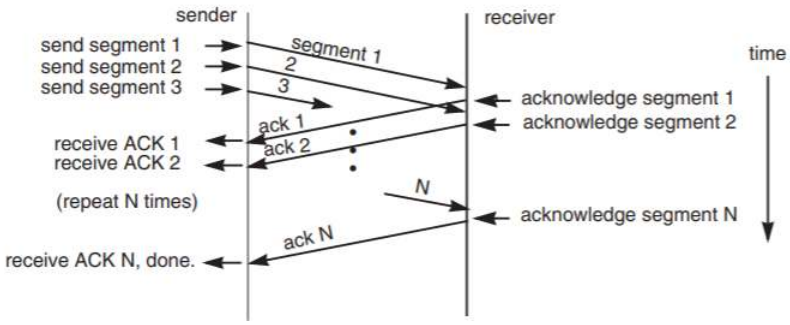


Figure 1.6.2: Overlapped transmission of multiple segments.

Figure 1.6.1 shows a small time interval between the sending of segment 1 and the sending of segment 2. This interval accounts for the time to generate and transmit the next segment. It also shows a small time interval at the receiving side that accounts for the time required for the recipient to accept the segment and prepare the acknowledgment. Depending on the details of the protocol, it may also include the time the receiver spends acting on the segment (see Sidebar 1.6.3). With this approach, the total time to send N segments has dropped to N packet transmission times plus one round-trip time for the last segment and its acknowledgment—if nothing goes wrong. Unfortunately, several things can go wrong, and taking care of them can add quite a bit of complexity to the picture.

Sidebar 1.6.3

What does an acknowledgment really mean?

An end-to-end acknowledgment is a widely used technique for the receiving side to tell the sending side something of importance, but since there are usually several different things going on in the end-to-end layer, there can also be several different purposes for acknowledgments. Some possibilities include

- it is OK to stop the timer associated with the acknowledged data segment
- it is OK to release the buffer holding a copy of the acknowledged segment
- it is OK to send another segment
- the acknowledged segment has been accepted for consideration
- the work requested in the acknowledged segment has been completed.

In some protocols, a single acknowledgment serves several of those purposes, while in other protocols a different form of acknowledgment may be used for each one; there are endless combinations. As a result, whenever the word acknowledgment is used in the discussion of a protocol, it is a good idea to establish exactly what the acknowledgment really means. This understanding is especially important if one is trying to estimate round-trip times by measuring the time for an acknowledgment to return; in some protocols such a measurement would include time spent doing processing in the receiving application, while in other cases it would not.

If there really are five different kinds of acknowledgments, there is a concern that for every outgoing packet there might be five different packets returning with acknowledgments. In practice this is rarely the case because acknowledgments can be implemented as data items in the end-to-end header of any packet that happens to be going in the reverse direction. A single packet may thus carry any number of different kinds of acknowledgments and acknowledgments for a range of received packets, in addition to application data that may be flowing in the reverse direction. The technique of placing one or more acknowledgments in the header of the next packet that happens to be going in the reverse direction is known as **piggybacking**.

1.6.6.2: Bottlenecks, Flow Control, and Fixed Windows

A second set of issues has to do with the relative speeds of the sender in generating segments, the entry point to the network in accepting them, any bottleneck inside the network in transmitting them, and the receiver in consuming them. The timing diagram and analysis above assumed that the bottleneck was at the sending side, either in the rate at which the sender generates segments or the rate that at which the first network link can transmit them.

A more interesting case is when the sender generates data, and the network transmits it, faster than the receiver can accept it, perhaps because the receiver has a slow processor and eventually runs out of buffer space to hold not-yet-processed data. When this is a possibility, the transport protocol needs to include some method of controlling the rate at which the sender generates data. This mechanism is called flow control. The basic concept involved is that the sender starts by asking the receiver how much data the receiver can handle. The response from the receiver, which may be measured in bits, bytes, or segments, is known as a **window**. The sender asks permission to send, and the receiver responds by quoting a window size, as illustrated in Figure 1.6.3. The sender then sends that much data and waits until it receives permission to send more. Any intermediate acknowledgments from the receiver allow the sender to stop the associated timer and release the send buffer, but they cannot be used as permission to send more data because the receiver is only acknowledging data arrival, not data consumption. Once the receiver has actually consumed the data in its buffers, it sends permission for another window's worth of data. One complication is that the implementation must guard against both missing permission messages that could leave the sender with a zero-sized window and also duplicated permission messages that could increase the window size more than the receiver intends: messages carrying window-granting permission require exactly-once delivery.

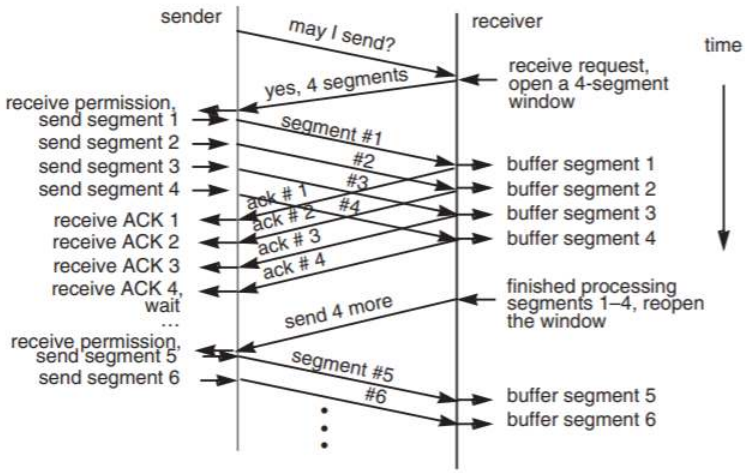


Figure 1.6.3: Flow control with a fixed window.

The window provided by the scheme of Figure 1.6.3 is called a fixed window. The lock-step protocol described earlier is a flow control scheme with a window that is one data segment in size. With any window scheme, one network round-trip time elapses between the receiver's sending of a window-opening message and the arrival of the first data that takes advantage of the new window. Unless we are careful, this time will be pure delay experienced by both parties. A clever receiver could anticipate this delay, and send the window-opening message one round-trip time before it expects to be ready for more data. This form of prediction is still using a fixed window, but it keeps data flowing more smoothly. Unfortunately, it requires knowing the network round-trip time which, as the discussion of timers explained, is a hard thing to estimate. Chapter 1 Exercises 13 and 16 explore the **bang-bang protocol** and **pacing**, two more variants on the fixed window idea.

1.6.6.3: Sliding Windows and Self-Pacing

An even more clever scheme is the following: as soon as it has freed up a segment buffer, the receiver could immediately send permission for a window that is one segment larger (either by sending a separate message or, if there happens to be an ACK ready to go, piggy-backing on that ACK). The sender keeps track of how much window space is left, and increases that number whenever additional permission arrives. When a window can have space added to it on the fly it is called a **sliding window**. The advantage of a sliding window is that it can automatically keep the pipeline filled, without need to guess when it is safe to send permission-granting messages.

The sliding window appears to eliminate the need to know the network round-trip time, but this appearance is an illusion. The real challenge in flow control design is to develop a single flow control algorithm that works well under all conditions, whether the bottleneck is the sender's rate of generating data, the network transmission capacity, or the rate at which the receiver can accept data. When the receiver is the bottleneck, the goal is to ensure that the receiver never waits. Similarly, when the sender is the bottleneck, the goal is to ensure that the sender never waits. When the network is the bottleneck, the goal is to keep the network moving data at its maximum rate. The question is what window size will achieve these goals.

The answer, no matter where the bottleneck is located, is determined by the bottleneck data rate and the round-trip time of the network. If we multiply these two quantities, the product tells us the amount of buffering, and thus the minimum window size, needed to ensure a continuous flow of data. That is,

$$\text{window size} \geq \text{round-trip time} \times \text{bottleneck data rate} \tag{1.6.4}$$

To see why, imagine for a moment that we are operating with a sliding window one segment in size. As we saw before, this window size creates a lock-step protocol with one segment delivered each round-trip time, so the realized data rate will be the window size divided by the round-trip time. Now imagine operating with a window of two segments. The network will then deliver two segments each round-trip time. The realized data rate is still the window size divided by the round-trip time, but the window size is twice as large. Now, continue to try larger window sizes until the realized data rate just equals the bottleneck data rate. At that point the window size divided by the round-trip time still tells us the realized data rate, so we have equality in the formula above. Any window size less than this will produce a realized data rate less than the bottleneck. The window size can be larger than this minimum, but since the realized data rate cannot exceed the bottleneck, there is no advantage. There is actually a disadvantage to a larger window size: if something goes wrong that requires draining the pipeline, it will take longer to do so. Further, a larger window puts a larger load on the network, and thereby contributes to congestion and discarded packets in the network routers.

The most interesting feature of a sliding window whose size satisfies the inequality is that, although the sender does not know the bottleneck data rate, it is sending at exactly that rate. Once the sender fills a sliding window, it cannot send the next data element until the acknowledgment of the oldest data element in the window returns. At the same time, the receiver cannot generate acknowledgments any faster than the network can deliver data elements. Because of these two considerations, the rate at which the window slides adjusts itself automatically to be equal to the bottleneck data rate, a property known as **self-pacing**. Self-pacing provides the needed mechanism to adjust the sender's data rate to exactly equal the data rate that the connection can sustain.

Let us consider what the window-size formula means in practice. Suppose a client computer in Boston that can absorb data at 500 kilobytes per second wants to download a file from a service in San Francisco that can send at a rate of 1 megabyte per second, and the network is not a bottleneck. The round-trip time for the Internet over this distance is about 70 milliseconds*, so the minimum window size would be

$$70 \text{ milliseconds} \times 500 \text{ kilobytes/second} = 35 \text{ kilobytes}$$

and if each segment carries 512 bytes, there could be as many as 70 such segments en route at once. If, instead, the two computers were in the same building, with a 1 millisecond round-trip time separating them, the minimum window size would be 500 bytes. Over this short distance a lock-step protocol would work equally well.

So, despite the effort to choose the appropriate window size, we still need an estimate of the round-trip time of the network, with all the hazards of making an accurate estimate. The protocol may be able to use the same round-trip time estimate that it used in setting its timers, but there is a catch. To keep from unnecessarily retransmitting packets that are just delayed in transit, an estimate that is used in timer setting should err by being too large. But if a too-large round-trip time estimate is used in window setting, the resulting excessive window size will simply increase the length of packet forwarding queues within the network; those longer queues will increase the transit time, in turn leading the sender to think it needs a still larger window. To avoid this positive feedback, a round-trip time estimator that is to be used for window size adjustment needs to err on the side of being too small, and be designed not to react too quickly to an apparent increase in round-trip time—exactly the opposite of the desiderata for an estimate used for setting timers.

Once the window size has been established, there is still a question of how big to make the buffer at the receiving side of the transport protocol. The simplest way to ensure that there is always space available for arriving data is to allocate a buffer that is at least as large as the window size.

---

* Measurements of round-trip time from Boston to San Francisco over the Internet in 2005 typically show a minimum of about 70 milliseconds. A typical route might take a packet via New York, Cleveland, Indianapolis, Kansas City, Denver, and Sacramento, a

distance of 11,400 kilometers, and through 15 packet forwarders in each direction. The propagation delay over that distance, assuming a velocity of propagation in optical fiber of 66% of the speed of light, would be about 57 milliseconds. Thus the 30 packet forwarders apparently introduce about another 13 milliseconds of processing and transmission delay, roughly 430 microseconds per forwarder.

1.6.6.4: Recovery of Lost Data Segments with Windows

While the sliding window may have addressed the performance problem, it has complicated the problem of recovering lost data segments. The sender can still maintain a checklist of expected acknowledgments, but the question is when to take action on this list. One strategy is to associate with each data segment in the list a timestamp indicating when that segment was sent. When the clock indicates that more than one round-trip time has passed, it is time for a resend. Or, assuming that the sender is numbering the segments for reassembly, the receiver might send a NAK when it notices that several segments with higher numbers have arrived. Either approach raises a question of how resent segments should count against the available window. There are two cases: either the original segment never made it to the receiver, or the receiver got it but the acknowledgment was lost. In the first case, the sender has already counted the lost segment, so there is no reason to count its replacement again. In the second case, presumably the receiver will immediately discard the duplicate segment. Since it will not occupy the recipient's attention or buffers for long, there is no need to include it in the window accounting. So in both cases the answer is the same: do not count a resent segment against the available window. (This conclusion is fortunate because the sender can't tell the difference between the two cases.)

We should also consider what might go wrong if a window-increase permission message is lost. The receiver will eventually notice that no data is forthcoming, and may suspect the loss. But simply resending permission to send more data carries the risk that the original permission message has simply been delayed and may still be delivered, in which case the sender may conclude that it can send twice as much data as the receiver intended. For this reason, sending a window-increasing message as an incremental value is fragile. Even resending the current permitted window size can lead to confusion if window-opening messages happen to be delivered out of order. A more robust approach is for the receiver to always send the cumulative total of all permissions granted since transmission of this message or stream began. (A cumulative total may grow large, but a field size of 64 bits can handle window sizes of 10^{30} transmissions units, which probably is sufficient for most applications.) This approach makes it easy to discover and ignore an out-of-order total because a cumulative total should never decrease. Sending a cumulative total also simplifies the sender's algorithm, which now merely maintains the cumulative total of all permissions it has used since the transmission began. The difference between the total used so far and the largest received total of permissions granted is a self-correcting, robust measure of the current window size.

Sending of a message that contains the cumulative permission count can be repeated any number of times without affecting the correctness of the result. Thus a persistent sender (in this case, the receiver of the data is the persistent sender of the permission message) is sufficient to ensure exactly-once delivery of a permission increase. With this design, the sender’s permission receiver is an example of an idempotent service interface, as suggested in the last paragraph of Section 1.2.4.

There is yet one more rate-matching problem: the blizzard of packets arising from a newly-opened flow control window may encounter or even aggravate congestion somewhere within the network, resulting in packets being dropped. Avoiding this situation requires some cooperation between the end-to-end protocol and the network forwarders, so we defer its discussion to Section 1.7.

1.6.7: Assurance of Stream Order, and Closing of Connections

A **stream transport protocol** transports a related series of elements, which may be bits, bytes, segments, or messages, from one point to another with the assurance that they will be delivered to the recipient in the order in which the sender dispatched them. A stream protocol usually—but not always—provides additional assurances, such as no missing elements, no duplicate elements, and data integrity. Because a telephone circuit has some of these same properties, a stream protocol is sometimes said to create a **virtual circuit**.

The simple-minded way to deliver things in order is to use the lock-step transmission protocol described in Section 1.6.3, in which the sending side does not send the next element until the receiving side acknowledges that the previous one has arrived safely. But applications often choose stream protocols to send large quantities of data, and the round-trip delays associated with a lock-step transmission protocol are enough of a problem that stream protocols nearly always employ some form of overlapped transmission. When overlapped transmission is added, the several elements that are simultaneously enroute can arrive at the receiving side out of order. Two quite different events can lead to elements arriving out of order: different packets may follow different paths that have

different transit times, or a packet may be discarded if it traverses a congested part of the network or is damaged by noise. A discarded packet will have to be retransmitted, so its replacement will almost certainly arrive much later than its adjacent companions.

The transport protocol can ensure that the data elements are delivered in the proper order by adding to the transport-layer header a serial number that indicates the position in the stream where the element or elements in the current data segment belong. At the receiving side, the protocol delivers elements to the application and sends acknowledgments back to the sender as long as they arrive in order. When elements arrive out of order, the protocol can follow one of two strategies:

1. Acknowledge only when the element that arrives is the next element expected or a duplicate of a previously received element. Discard any others. This strategy is simple, but it forces a capacity-wasting retransmission of elements that arrive before their predecessors.
2. Acknowledge every element as it arrives, and hold in buffers any elements that arrive before their predecessors. When the predecessors finally arrive, the protocol can then deliver the elements to the application in order and release the buffers. This technique is more efficient in its use of network resources, but it requires some care to avoid using up a large number of buffers while waiting for an earlier element that was in a packet that was discarded or damaged.

The two strategies can be combined by acknowledging an early-arriving element only if there is a buffer available to hold it, and discarding any others. This approach raises the question of how much buffer space to allocate. One simple answer is to provide at least enough buffer space to hold all of the elements that would be expected to arrive during the time it takes to sort out an out-of-order condition. This question is closely related to the one explored earlier of how many buffers to provide to go with a given size of sliding window. A requirement of delivery in order is one of the reasons why it is useful to make a clear distinction between acknowledging receipt of data and opening a window that allows the sending of more data.

It may be possible to speed up the resending of lost packets by taking advantage of the additional information implied by arrival of numbered stream elements. If stream elements have been arriving quite regularly, but one element of the stream is missing, rather than waiting for the sender to time out and resend, the receiver can send an explicit negative acknowledgment (NAK) for the missing element. If the usual reason for an element to appear to be missing is that it has been lost, sending NAKs can produce a useful performance enhancement. On the other hand, if the usual reason is that the missing element has merely suffered a bit of extra delay along the way, then sending NAKs may lead to unnecessary retransmissions, which waste network capacity and can degrade performance. The decision whether or not to use this technique depends on the specific current conditions of the network. One might try to devise an algorithm that figures out what is going on (e.g., if NAKs are causing duplicates, stop sending NAKs) but it may not be worth the added complexity.

As the interface described in Section 1.6.1 above suggests, using a stream transport protocol involves a call to open the stream, a series of calls to write to or read from the stream, and a call to close the stream. Opening a stream involves creating a record at each end of the connection. This record keeps track of which elements have been sent, which have been received, and which have been acknowledged. Closing a stream involves two additional considerations. First and simplest, after the receiving side of the transport protocol delivers the last element of the stream to the receiving application, it then needs to report an end-of-stream indication to that application. Second, both ends of the connection need to agree that the network has delivered the last element and the stream should be closed. This agreement requires some care to reach.

A simple protocol that ensures agreement is the following: Suppose that Alice has opened a stream to Bob, and has now decided that the stream is no longer needed. She begins persistently sending a close request to Bob, specifying the stream identifier. Bob, upon receiving a close request, checks to see if he agrees that the stream is no longer needed. If he does agree, he begins persistently sending a close acknowledgment, again specifying the stream identifier. Alice, upon receiving the close acknowledgment, can turn off her persistent sender and discard her record of the stream, confident that Bob has received all elements of the stream and will not be making any requests for retransmissions. In addition, she sends Bob a single "all done" message, containing the stream identifier. If she receives a duplicate of the close acknowledgment, her record of the stream will already be discarded, but it doesn't matter; she can assume that this is a duplicate close acknowledgment from some previously closed stream and, from the information in the close acknowledgment, she can fabricate an "all done" message and send it to Bob. When Bob receives the "all done" message he can turn off his persistent sender and, confident that Alice agrees that there is no further use for the stream, discard his copy of the record of the stream. Alice and Bob can, in the future, safely discard any late duplicates that mention a stream for which they have no record. (The tombstone problem still exists for the stream itself. It would be a good idea for Bob to delay deletion of his record until there is no chance that a long-delayed duplicate of Alice's original request to open the stream will arrive.)

1.6.8: Assurance of Jitter Control

Some applications, such as delivering sound or video to a person listening or watching on the spot, are known as **real-time**. For real-time applications, reliability, in the sense of never delivering an incorrect bit of data, is often less important than timely delivery. High reliability can actually be counter-productive if the transport protocol achieves it by requesting retransmission of a damaged data element, and then holds up delivery of the remainder of the stream until the corrected data arrives. What the application wants is continuous delivery of data, even if the data is not completely perfect. For example, if a few bits are wrong in one frame of a movie (note that this video use of the term “frame” has a meaning similar but not identical to the "frame" used in data communications), it probably won’t be noticed. In fact, if one video frame is completely lost in transit, the application program can probably get away with repeating the previous video frame while waiting for the following one to be delivered. The most important assurance that an end-to-end stream protocol can provide to a real-time application is that delivery of successive data elements be on a regular schedule. For example, a standard North American television set consumes one video frame every 33.37 milliseconds and the next video frame must be presented on that schedule.

Transmission across a forwarding network can produce varying transit times from one data segment to the next. In real-time applications, this variability in delivery time is known as **jitter**, and the requirement is to control the amount of jitter. The basic strategy is for the receiving side of the transport protocol to delay *all* arriving segments to make it look as though they had encountered the worst allowable amount of delay. One can, in principle, estimate an appropriate amount of extra buffering for the delayed segments as follows (assume for the television example that there is one video frame in each segment):

1. Measure the distribution of segment delivery delays between sending and receiving points and plot that distribution in a chart showing delay time versus frequency of that delay.
2. Choose an acceptable frequency of delivery failure. For a television application, one might decide that 1 out of 100 video frames won’t be missed.
3. From the distribution, determine a delay time large enough to ensure that 99 out of 100 segments will be delivered in less than that delay time. Call this delay D_{long}.
4. From the distribution determine the shortest delay time that is observed in practice. Call this value D_{long}.
5. Now, provide enough buffering to delay every arriving segment so that it appears to have arrived with delay D_{long}. The largest number of segments that would need to be buffered is

$$\text{Number of segment buffers} = \frac{D_{long} - D_{short}}{D_{headway}} \tag{1.6.5}$$

where $D_{headway}$ is the average time between arriving segments. With this much buffering, we would expect that about one out of every 100 segments will arrive too late; when that occurs, the transport protocol simply reports “missing data” to the application and discards that segment if it finally does arrive.

In practice, there is no easy way to measure one-way segment delivery delay, so a common strategy is simply to set the buffer size by trial and error.

Although the goal of this technique is to keep the rate of missing video frames below the level of human perceptibility, you can sometimes see the technique fail when watching a television program that has been transmitted by satellite or via the Internet. Occasionally there may be a freeze-frame that persists long enough that you can see it, but that doesn't seem to be one that the director intended. This event probably indicates that the transmission path was disrupted for a longer time than the available buffers were prepared to handle.

1.6.9: Assurance of Authenticity and Privacy

Most of the assurance-providing techniques described above are intended to operate in a benign environment, in which the designer assumes that errors can occur but that the errors are not maliciously constructed to frustrate the intended assurances. In many real-world environments, the situation is worse than that: one must defend against the threat of someone hostile intercepting and maliciously modifying packets, or of some end-to-end layer participants violating a protocol with malicious intent.

To counter these threats, the end-to-end layer can apply two kinds of key-based mathematical transformations to the data:

1. **sign** and **verify**, to establish the authenticity of the source and the integrity of the contents of a message, and
2. **encrypt** and **decrypt**, to maintain the privacy of the contents of a message.

These two techniques can, if applied properly, be effective, but they require great care in design and implementation. Without such care, they may not work, but because they were applied the user may believe that they do, and thus have a false sense of security. A false assurance can be worse than no assurance at all. The issues involved in providing security assurances are a whole subject in themselves, and they apply to many system components in addition to networks, so we defer them to Chapter 5, which provides an in-depth discussion of protecting information in computer systems.

With this examination of end-to-end topics, we have worked our way through the highest layer that we identify as part of the network. The next section of this chapter, on congestion control, is a step sideways to explore a topic that requires cooperation of more than one layer.

This page titled 1.6: The End-to-End Layer is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

1.7: A Network System Design Issue - Congestion Control

1.7.1: Managing Shared Resources

In the context of a network, the resource that is shared is a set of communication links and the supporting packet forwarders. The geographically and administratively distributed nature of those components and their users adds delay and complication to resource management. In Section 1.2.2 of this chapter we saw how queues manage the problem that packets may arrive at a packet switch at a time when the outgoing link is already busy transmitting another packet, and Figure 1.2.6 showed the way that queues grow with increased utilization of the link. This same phenomenon applies to processor scheduling and supermarket checkout lines: any time there is a shared resource, and the demand for that resource comes from several statistically independent sources, there will be fluctuations in the arrival of load, and thus in the length of the queue and the time spent waiting for service. Whenever the offered load (in the case of a packet switch, that is the rate at which packets arrive and need to be forwarded) is greater than the capacity (the rate at which the switch can forward packets) of a resource for some duration, the resource is overloaded for that time period.

When sources are statistically independent of one another, occasional overload is inevitable but its significance depends critically on how long it lasts. If the duration is comparable to the **service time**, which is the typical time for the resource to handle one customer (in a supermarket), one thread (in a processor manager), or one packet (in a packet forwarder), then a queue is simply an orderly way to delay some requests for service until a later time when the offered load drops below the capacity of the resource. Put another way, a queue handles short bursts of too much demand by time-averaging with adjacent periods when there is excess capacity.

If, on the other hand, overload persists for a time significantly longer than the service time, there begins to develop a risk that the system will fail to meet some specification such as maximum delay or acceptable response time. When this occurs, the resource is said to be **congested**. Congestion is not a precisely defined concept. The duration of overload that is required to classify a resource as congested is a matter of judgment, and different systems (and observers) will use different thresholds.

Congestion may be temporary, in which case clever resource management schemes may be able to rescue the situation, or it may be chronic, meaning that the demand for service continually exceeds the capacity of the resource. If the congestion is chronic, the length of the queue will grow without bound until something breaks: the space allocated for the queue may be exceeded, the system may fail completely, or customers may go elsewhere in disgust.

The stability of the offered load is another factor in the frequency and duration of congestion. When the load on a resource is aggregated from a large number of statistically independent small sources, averaging can reduce the frequency and duration of load peaks. On the other hand, if the load comes from a small number of large sources (even if the sources are independent), the probability that they all demand service at about the same time can be high enough that congestion can be frequent or long-lasting.

A counter-intuitive concern of shared resource management is that competition for a resource sometimes leads to wasting of that resource. For example, in a grocery store, customers who are tired of waiting in the checkout line may just walk out of the store, leaving filled shopping carts behind. Someone has to put the goods from the abandoned carts back on the shelves. Suppose that one or two of the checkout clerks leave their registers to take care of the accumulating abandoned carts. The rate of sales being rung up drops while they are away from their registers, so the queues at the remaining registers grow longer, causing more people to abandon their carts, and more clerks will have to turn their attention to restocking. Eventually, the clerks will be doing nothing but restocking and the number of sales rung up will drop to zero. This regenerative overload phenomenon is called **congestion collapse**. Figure 1.7.1 plots the useful work getting done as the offered load increases, for three different cases of resource limitation and waste, including one that illustrates collapse. Congestion collapse is dangerous because it can be self-sustaining. Once temporary congestion induces a collapse, even if the offered load drops back to a level that the resource could handle, the already-induced waste rate can continue to exceed the capacity of the resource, causing it to continue to waste the resource and thus remain congested indefinitely.

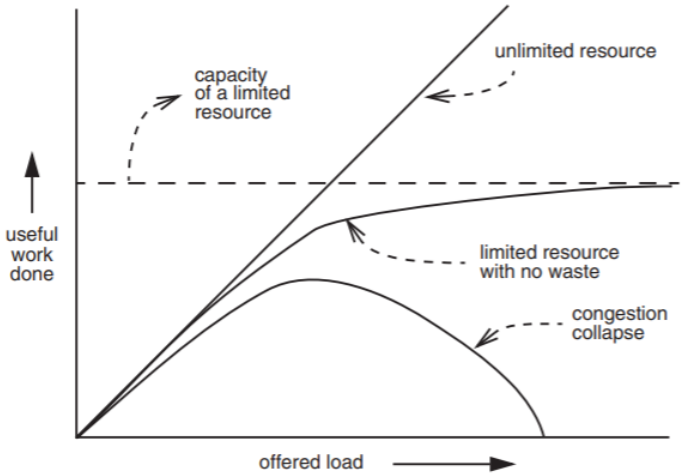


Figure 1.7.1: Offered load versus useful work done. The more work offered to an ideal unlimited resource, the more work gets done, as indicated by the 45-degree unlimited resource line. Real resources are limited, but in the case with no waste, useful work asymptotically approaches the capacity of the resource. On the other hand, if overloading the resource also wastes it, useful work can decline when offered load increases, as shown by the congestion collapse line.

When developing or evaluating a resource management scheme, it is important to keep in mind that you can't squeeze blood out of a turnip: if a resource is congested, either temporarily or chronically, delays in receiving service are inevitable. The best a management scheme can do is redistribute the total amount of delay among waiting customers. The primary goal of resource management is usually quite simple: to avoid congestion collapse. Occasionally other goals, such as enforcing a policy about who gets delayed, are suggested, but these goals are often hard to define and harder to achieve. (Doling out delays is a tricky business; overall satisfaction may be higher if a resource serves a few customers well and completely discourages the remainder, rather than leaving all equally disappointed.)

There are two general approaches to managing congestion. Either:

- *increase the capacity of the resource*, or
- *reduce the offered load.*

In both cases the goal is to move quickly to a state in which the load is less than the capacity of the resource. When measures are taken to reduce offered load, it is useful to separately identify the **intended load**, which would have been offered in the absence of control. Of course, in reducing offered load, the amount by which it is reduced doesn't really go away, it is just deferred to a later time. Reducing offered load acts by averaging periods of overload with periods of excess capacity, just like queuing, but with involvement of the source of the load, and typically over a longer period of time.

To increase capacity or to reduce offered load it is necessary to provide feedback to one or more **control points**. A control point is an entity that determines, in the first case, the amount of resource that is available and, in the second, the load being offered. A congestion control system is thus a feedback system, and delay in the feedback path can lead to oscillations in load and in useful work done.

For example, in a supermarket, a common strategy is for the store manager to watch the queues at the checkout lines; whenever there are more than two or three customers in any line the manager calls for staff elsewhere in the store to drop what they are doing and temporarily take stations as checkout clerks, thereby increasing capacity. In contrast, when you call a customer support telephone line you may hear an automatic response message that says something such as, "Your call is important to us. It will be approximately 21 minutes till we are able to answer it." That message will probably lead some callers to hang up and try again at a different time, thereby decreasing (actually deferring) the offered load. In both the supermarket and the telephone customer service system, it is easy to create oscillations. By the time the fourth supermarket clerk stops stacking dog biscuits and gets to the front of the store, the lines may have vanished, and if too many callers decide to hang up, the customer service representatives may find there is no one left to talk to.

In the commercial world, the choice between these strategies is a complex trade-off involving economics, physical limitations, reputation, and customer satisfaction. The same thing is true inside a computer system or network.

1.7.2: Resource Management in Networks

In a computer network, the shared resources are the communication links and the processing and buffering capacity of the packet forwarders. There are several things that make this resource management problem more difficult than, say, scheduling a processor among competing threads.

1. *There is more than one resource*. Even a small number of resources can be used up in an alarmingly large number of different ways, and the mechanisms needed to keep track of the situation can rapidly escalate in complexity. In addition, there can be dynamic interactions among different resources—as one nears capacity it may push back on another, which may push back on yet another, which may push back on the first one. These interactions can create either deadlock or livelock, depending on the details.
2. *It is easy to induce congestion collapse*. The usually beneficial independence of the layers of a packet forwarding network contributes to the ease of inducing congestion collapse. As queues for a particular communication link grow, delays grow. When queuing delays become too long, the timers of higher layer protocols begin to expire and trigger retransmissions of the delayed packets. The retransmitted packets join the long queues but, since they are duplicates that will eventually be discarded, they just waste capacity of the link. Designers sometimes suggest that an answer to congestion is to buy more or bigger buffers. As memory gets cheaper, this idea is tempting, but it doesn't work. To see why, suppose memory is so cheap that a packet forwarder can be equipped with an infinite number of packet buffers. That many buffers can absorb an unlimited amount of overload, but as more buffers are used, the queuing delay grows. At some point the queuing delay exceeds the time-outs of the end-to-end protocols and the end-to-end protocols begin retransmitting packets. The offered load is now larger, perhaps twice as large as it would have been in the absence of congestion, so the queues grow even longer. After a while the retransmissions cause the queues to become long enough that end-to-end protocols retransmit yet again, and packets begin to appear in the queue three times, and then four times, etc. Once this phenomenon begins, it is self-sustaining until the real traffic drops to less than half (or 1/3 or 1/4, depending on how bad things got) of the capacity of the resource. The conclusion is that the infinite buffers did not solve the problem, they made it worse. Instead, it may be better to discard old packets than to let them use up scarce transmission capacity.
3. *There are limited options to expand capacity*. In a network there may not be many options to raise capacity to deal with temporary overload. Capacity is generally determined by physical facilities: optical fibers, coaxial cables, wireless spectrum availability, and transceiver technology. Each of these things can be augmented, but not quickly enough to deal with temporary congestion. If the network is mesh-connected, one might consider sending some of the queued packets via an alternate path. That can be a good response, but doing it on a fast enough timescale to overcome temporary congestion requires knowing the instantaneous state of queues throughout the network. Strategies to do that have been tried; they are complex and haven't worked well. It is usually the case that the only realistic strategy is to reduce demand.
4. *The options to reduce load are awkward*. The alternative to increasing capacity is to reduce the offered load. Unfortunately, the control point for the offered load is distant and probably administered independently of the congested packet forwarder. As a result, there are at least three problems:
    - The feedback path to a distant control point may be long. By the time the feedback signal gets there the sender may have stopped sending (but all the previously sent packets are still on their way to join the queue) or the congestion may have disappeared and the sender no longer needs to hold back. Worse, if we use the network to send the signal, the delay will be variable, and any congestion on the path back may mean that the signal gets lost. The feedback system must be robust to deal with all these eventualities.
    - The control point (in this case, an end-to-end protocol or application) must be capable of reducing its offered load. Some end-to-end protocols can do this quite easily, but others may not be able to. For example, a stream protocol that is being used to send files can probably reduce its average data rate on short notice. On the other hand, a real-time video transmission protocol may have a commitment to deliver a certain number of bits every second. A single-packet request/response protocol will have no control at all over the way it loads the network; control must be exerted by the application, which means there must be some way of asking the application to cooperate—if it can.
    - The control point must be willing to cooperate. If the congestion is discovered by the network layer of a packet forwarder, but the control point is in the end-to-end layer of a leaf node, there is a good chance these two entities are under the responsibility of different administrations. In that case, obtaining cooperation can be problematic; the administration of the control point may be more interested in keeping its offered load equal to its intended load in the hope of capturing more of the capacity in the face of competition.

These problems make it hard to see how to apply a central planning approach such as the one that worked in the grocery store. Decentralized schemes seem more promising. Many mechanisms have been devised to try to manage network congestion. Sections 1.7.3 and 1.7.4 describe the design considerations surrounding one set of decentralized mechanisms, similar to the ones that are currently used in the public Internet. These mechanisms are not especially well understood, but they not only seem to work, they have allowed the Internet to operate over an astonishing range of capacity. In fact, the Internet is probably the best existing counterexample of the *incommensurate scaling rule*. Recall that the rule suggests that a system needs to be redesigned whenever any important parameter changes by a factor of ten. The Internet has increased in scale from a few hundred attachment points to a few hundred million attachment points with only modest adjustments to its underlying design.

1.7.3: Cross-layer Cooperation: Feedback

If the designer can arrange for cross-layer cooperation, then one way to attack congestion would be for the packet forwarder that notices congestion to provide feedback to one or more end-to-end layer sources, and for the end-to-end source to respond by reducing its offered load.

Several mechanisms have been suggested for providing feedback. One of the first ideas that was tried is for the congested packet forwarder to send a control message, called a **source quench**, to one or more of the source addresses that seems to be filling the queue. Unfortunately, preparing a control message distracts the packet forwarder at a time when it least needs extra distractions. Moreover, transmitting the control packet adds load to an already-overloaded network. Since the control protocol is best-effort the chance that the control message will itself be discarded increases as the network load increases, so when the network most needs congestion control the control messages are most likely to be lost.

A second feedback idea is for a packet forwarder that is experiencing congestion to set a flag on each forwarded packet. When the packet arrives at its destination, the end-to-end transport protocol is expected to notice the congestion flag and in the next packet that it sends back it should include a "slow down!" request to alert the other end about the congestion. This technique has the advantage that no extra packets are needed. Instead, all communication is piggybacked on packets that were going to be sent anyway. But the feedback path is even more hazardous than with a source quench—not only does the signal have to first reach the destination, the next response packet of the end-to-end protocol may not go out immediately.

Both of these feedback ideas would require that the feedback originate at the packet forwarding layer of the network. But it is also possible for congestion to be discovered in the link layer, especially when a link is, recursively, another network. For these reasons, Internet designers converged on a third method of communicating feedback about congestion: a congested packet forwarder just discards a packet. This method does not require interpretation of packet contents and can be implemented simply in any component in any layer that notices congestion. The hope is that the source of that packet will eventually notice a lack of response (or perhaps receive a NAK). This scheme is not a panacea because the end-to-end layer has to assume that every packet loss is caused by congestion, and the speed with which the end-to-end layer responds depends on its timer settings. But it is simple and reliable.

This scheme leaves a question about which packet to discard. The choice is not obvious; one might prefer to identify the sources that are contributing most to the congestion and signal them, but a congested packet forwarder has better things to do than extensive analysis of its queues. The simplest method, known as **tail drop**, is to limit the size of the queue; any packet that arrives when the queue is full gets discarded. A better technique (**random drop**) may be to choose a victim from the queue at random. This approach has the virtue that the sources that are contributing most to the congestion are the most likely to be receive the feedback. One can even make a plausible argument to discard the packet at the **front** of the queue, on the basis that of all the packets in the queue, the one at the front has been in the network the longest, and thus is the one whose associated timer is most likely to have already expired.

Another refinement (**early drop**) is to begin dropping packets before the queue is completely full, in the hope of alerting the source sooner. The goal of early drop is to start reducing the offered load as soon as the possibility of congestion is detected, rather than waiting until congestion is confirmed, so it can be viewed as a strategy of avoidance rather than of recovery. Random drop and early drop are combined in a scheme known as RED, for **random early detection**.

1.7.4: Cross-layer Cooperation: Control

Suppose that the end-to-end protocol implementation learns of a lost packet. What then? One possibility is that it just drives forward, retransmitting the lost packet and continuing to send more data as rapidly as its application supplies it. The end-to-end protocol implementation is in control, and there is nothing compelling it to cooperate. Indeed, it may discover that by sending packets at the greatest rate it can sustain, it will push more data through the congested packet forwarder than it would otherwise. The problem, of course, is that if this is the standard mode of operation of every client, congestion will set in and all clients of the network will suffer, as predicted by the tragedy of the commons (see Sidebar 1.7.1).

Sidebar 1.7.1

The tragedy of the commons

> "Picture a pasture open to all…As a rational being, each herdsman seeks to maximize his gain…he asks, 'What is the utility to me of adding one more animal to my herd?' This utility has one negative and one positive component…Since the herdsman receives all the proceeds from the sale of the additional animal, the positive utility is nearly +1. Since, however, the effects of overgrazing are shared by all the herdsmen, the negative utility for any particular decision-making herdsman is only a fraction of −1.
>
> "Adding together the component partial utilities, the rational herdsman concludes that the only sensible course for him to pursue is to add another animal to his herd. And another…. But this is the conclusion reached by each and every rational herdsman sharing a commons. Therein is the tragedy. Each man is locked into a system that compels him to increase his herd without limit—in a world that is limited…Freedom in a commons brings ruin to all."
>
> — Garrett Hardin, *Science 162*, 3859 (December 13, 1968), pages 1243–1248. Extensions of "the tragedy of the commons". *Science 280*, 5364 (May 1, 1998), pages 682–683.

There are at least two things that the end-to-end protocol can do to cooperate. The first is to be careful about its use of timers, and the second is to pace the rate at which it sends data, a technique known as **automatic rate adaptation**. Both these things require having an estimate of the round-trip time between the two ends of the protocol.

The usual way of detecting a lost packet in a best-effort network is to set a timer to expire after a little more than one round-trip time, and assume that if an acknowledgment has not been received by then the packet is lost. In Section 1.6.2 we introduced timers as a way of ensuring at-least-once delivery via a best-effort network, expecting that lost packets had encountered mishaps such as misrouting, damage in transmission, or an overflowing packet buffer. With congestion management in operation, the dominant reason for timer expiration is probably that either a queue in the network has grown too long or a packet forwarder has intentionally discarded the packet. The designer needs to take this additional consideration into account when choosing a value for a retransmit timer.

As described in Section 1.6.6, a protocol can develop an estimate of the round trip time by directly measuring it for the first packet exchange and then continuing to update that estimate as additional packets flow back and forth. Then, if congestion develops, queuing delays will increase the observed round-trip times for individual packets, and those observations will increase the round-trip estimate used for setting future retransmit timers. In addition, when a timer does expire, the algorithm for timer setting should use exponential backoff for successive retransmissions of the same packet (exponential backoff was described in Section 1.6.2). It does not matter whether the reason for expiration is that the packet was delayed in a growing queue or it was discarded as part of congestion control. Either way, exponential backoff immediately reduces the retransmission rate, which helps ease the congestion problem. Exponential backoff has been demonstrated to be quite effective as a way to avoid contributing to congestion collapse. Once acknowledgments begin to confirm that packets are actually getting through, the sender can again allow timer settings to be controlled by the round-trip time estimate.

The second cooperation strategy involves managing the flow control window. Recall from the discussion of flow control in Section 1.6.6 that to keep the flow of data moving as rapidly as possible without overrunning the receiving application, the flow control window and the receiver's buffer should both be at least as large as the bottleneck data rate multiplied by the round trip time. Anything larger than that will work equally well for end-to-end flow control. Unfortunately, when the bottleneck is a congested link inside the network, a larger than necessary window will simply result in more packets piling up in the queue for that link. The additional cooperation strategy, then, is to ensure that the flow control window is no larger than necessary. Even if the receiver has buffers large enough to justify a larger flow control window, the sender should restrain itself and set the flow control window to the

smallest size that keeps the connection running at the data rate that the bottleneck permits. In other words, the sender should force equality in Equation 1.6.4.

Relatively early in the history of the Internet, it was realized (and verified in the field) that congestion collapse was not only a possibility, but that some of the original Internet protocols had unexpectedly strong congestion-inducing properties. Since then, almost all implementations of TCP, the most widely used end-to-end Internet transport protocol, have been significantly modified to reduce the risk, as described in Sidebar 1.7.2.

While having a widely-deployed, cooperative strategy for controlling congestion reduces both congestion and the chance of congestion collapse, there is one unfortunate consequence: Since every client that cooperates may be offering a load that is less than its intended load, there is no longer any way to estimate the size of that intended load. Intermediate packet forwarders know that if they are regularly discarding some packets, they need more capacity, but they have no clue how *much* more capacity they really need.

Sidebar 1.7.2

Retrofitting TCP

The Transmission Control Protocol (TCP), probably the most widely used end-to-end transport protocol of the Internet, was designed in 1974. At that time, previous experience was limited to lock-step protocols on networks with no more than a few hundred nodes. As a result, avoiding congestion collapse was not in its list of requirements. About a decade later, when the Internet first began to expand rapidly, this omission was noticed, and a particular collapse-inducing feature of its design drew attention.

The only form of acknowledgment in the original TCP was "I have received all the bytes up to X." There was no way for a receiver to say, for example, "I am missing bytes Y through Z." In consequence, when a timer expired because some packet or its acknowledgment was lost, as soon as the sender retransmitted that packet the timer of the next packet expired, causing its retransmission. This process would repeat until the next acknowledgment finally returned, a full round trip (and full flow control window) later. On long-haul routes, where flow control windows might be fairly large, if an overloaded packet forwarder responded to congestion by discarding a few packets (each perhaps from a different TCP connection), each discarded packet would trigger retransmission of a window full of packets and the ensuing blizzard of retransmitted packets could immediately induce congestion collapse. In addition, an insufficiently adaptive time-out scheme ensured that the problem would occur frequently.

By the time this effect was recognized, TCP was widely deployed, so changes to the protocol were severely constrained. The designers found a way to change the implementation without changing the data formats. The goal was to allow new and old implementations to interoperate, so new implementations could gradually replace the old. The new implementation works by having the sender tinker with the size of the flow control window (Warning: this explanation is somewhat oversimplified!):

1. **Slow start**. When starting a new connection, send just one packet, and wait for its acknowledgment. Then, for each acknowledged packet, add one to the window size and send two packets. The result is that in each round trip time, the number of packets that the sender dispatches doubles. This doubling procedure continues until one of three things happens: (1) the sender reaches the window size suggested by the receiver, in which case the network is not the bottleneck, and the sender maintains the window at that size; (2) all the available data has been dispatched; or (3) the sender detects that a packet it sent has been discarded, as described in step 2.
2. **Duplicate acknowledgment**: The receiving TCP implementation is modified very slightly: whenever it receives an out-of-order packet, it sends back a duplicate of its latest acknowledgment. The idea is that a duplicate acknowledgment can be interpreted by the sender as a negative acknowledgment for the next unacknowledged packet.
3. **Equilibrium**: Upon duplicate acknowledgment, the sender retransmits just the first unacknowledged packet and also drops its window size to some fixed fraction (for example, 1/2) of its previous size. From then on it operates in an equilibrium mode in which it continues to watch for duplicate acknowledgments but it also probes gently to see if more capacity might be available. The equilibrium mode has two components:
    - **Additive increase**: Whenever all of the packets in a round trip time are successfully acknowledged, the sender increases the size of the window by one.

- **Multiplicative** decrease: Whenever a duplicate acknowledgment arrives, the sender decreases the size of the window by the fixed fraction.

4. **Restart**: If the sender's retransmission timer expires, self-pacing based on ACKs has been disrupted, perhaps because something in the network has radically changed. So the sender waits a short time to allow things to settle down, and then goes back to slow start, to allow assessment of the new condition of the network.

By interpreting a duplicate acknowledgment as a negative acknowledgment for a single packet, TCP eliminates the massive retransmission blizzard, and by reinitiating slow start on each timer expiration, it avoids contributing to congestion collapse.

The figure below illustrates the evolution of the TCP window size with time in the case where the bottleneck is inside the network. TCP begins with one packet and slow start, until it detects the first packet loss. The sender immediately reduces the window size by half and then begins gradually increasing it by one for each round trip time until detecting another lost packet. This sawtooth behavior may continue indefinitely, unless the retransmission timer expires. The sender pauses and then enters another slow start phase, this time switching to additive increase as soon as it reaches the window size it would have used previously, which is half the window size that was in effect before it encountered the latest round of congestion.

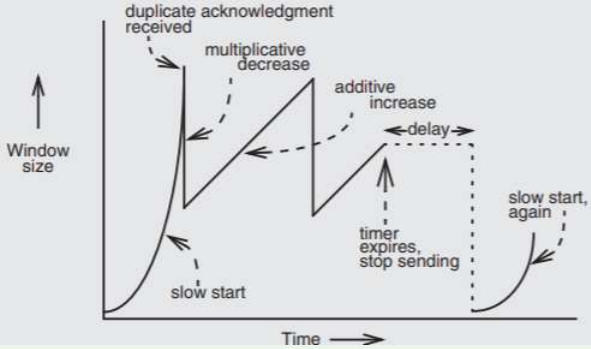


Figure 1.7.2 Window size versus time for TCP when a bottleneck is inside the network. After a slow start, a sawtooth pattern of multiplicative decrease and additive increase, prompted by the receipt of duplicate acknowledgements, continues until the timer expires.

This cooperative scheme has not been systematically analyzed, but it seems to work in practice, even though not all of the traffic on the Internet uses TCP as its end-to-end transport protocol. The long and variable feedback delays that inevitably accompany lost packet detection by the use of duplicate acknowledgments induce oscillations (as evidenced by the sawteeth) but the additive increase—multiplicative decrease algorithms strongly damp those oscillations.

Exercise 1.11.12 compares slow start with "fast start", another scheme for establishing an initial estimate of the window size. There have been dozens (perhaps hundreds) of other proposals for fixing both real and imaginary, problems in TCP. The interested reader should consult Topic 1.4: Practical Aspects in the Suggestions for Further Reading.

1.7.5: Other Ways of Controlling Congestion in Networks

Overprovisioning: Configure each link of the network to have 125% (or 150% or 200%) as much capacity as the offered load at the busiest minute (or five minutes or hour) of the day. This technique works best on interior links of a large network, where no individual client represents more than a tiny fraction of the load. When that is the case, the average load offered by the large number of statistically independent sources is relatively stable and predictable. Internet backbone providers generally use overprovisioning to avoid congestion. The problems with this technique are:

- Odd events can disrupt statistical independence. An earthquake in California or a hurricane in Florida typically clogs up all the telephone trunks leading to and from the affected state, even if the trunks themselves haven't been damaged. Everyone tries to place a call at once.
- Overprovisioning on one link typically just moves the congestion to a different link. So every link in a network must be overprovisioned, and the amount of overprovisioning has to be greater on links that are shared by fewer customers because

statistical averaging is not as effective in limiting the duration of load peaks.

- At the edge of the network, statistical averaging across customers stops working completely. The link to an individual customer may become congested if the customer's Web service is featured in *Newsweek*—a phenomenon known as a "flash crowd". Permanently increasing the capacity of that link to handle what is probably a temporary but large overload may not make economic sense.
- Adaptive behavior of users can interfere with the plan. In Los Angeles, the opening of a new freeway initially provides additional traffic capacity, but new traffic soon appears and absorbs the new capacity, as people realize that they can conveniently live in places that are farther from where they work. Because of this effect, it does not appear to be physically possible to use overprovisioning as a strategy in the freeway system—the load always increases to match (or exceed) the capacity. Anecdotally, similar effects seem to occur in the Internet, although they have not yet been documented.

Over the life of the Internet there have been major changes in both telecommunications regulation and fiber optic technology that between them have transformed the Internet's central core from capacity-scarce to capacity-rich. As a result, the locations at which congestion occurs have moved as rapidly as techniques to deal with it have been invented. But so far congestion hasn't gone away.

Pricing: Another approach to congestion control is to rearrange the rules so that the interest of an individual client coincides with the interest of the network community and let the invisible hand take over, as explained in Sidebar 1.7.3. Since network resources are just another commodity, it should be possible to use pricing as a congestion control mechanism. The idea is that, if demand for a resource temporarily exceeds its capacity, clients will bid up the price. The increased price will cause some clients to defer their use of the resource until a time when it is cheaper, thereby reducing offered load; it will also induce additional suppliers to provide more capacity.

Sidebar 1.7.3

The Invisible Hand

Economics 101: In a free market, buyers have the option of buying a good or walking away, and sellers similarly have the option of offering a good or leaving the market. The higher the price, the more sellers will be attracted to the profit opportunity, and they will collectively thus make additional quantities of the good available. At the same time, the higher the price, the more buyers will balk, and collectively they will reduce their demand for the good. These two effects act to create an equilibrium in which the supply of the good exactly matches the demand for the good. Every buyer is satisfied with the price paid and every seller with the price received. When the market is allowed to set the price, surpluses and shortages are systematically driven out by this equilibrium-seeking mechanism.

"Every individual necessarily labors to render the annual revenue of the society as great as he can. He generally indeed neither intends to promote the public interest, nor knows how much he is promoting it. He intends only his own gain, and he is in this, as in many other cases, led by an invisible hand to promote an end which was no part of his intention. By pursuing his own interest he frequently promotes that of the society more effectually than when he really intends to promote it." *

* Adam Smith (1723–1790). *The Wealth of Nations 4*, Chapter 2. (1776)

There is a challenge in trying to make pricing mechanisms work on the short timescales associated with network congestion; in addition there is a countervailing need for predictability of costs in the short term that may make the idea unworkable. However, as a long-term strategy, pricing can be quite an effective mechanism to match the supply of network resources with demand. Even in the long term, the invisible hand generally requires that there be minimal barriers to entry by alternate suppliers; this is a hard condition to maintain when installing new communication links involves digging up streets, erecting microwave towers or launching satellites.

Congestion control in networks is by no means a solved problem—it is an active research area. This discussion has just touched the highlights, and there are many more design considerations and ideas that must be assimilated before one can claim to understand this topic.

1.7.6: Delay Revisited

Section 1.2.2 of this chapter identified four sources of delay in networks: propagation delay, processing delay, transmission delay, and queuing delay. Congestion control and flow control both might seem to add a fifth source of delay, in which the sender waits for permission from the receiver to launch a message into the network. In fact this delay is not of a new kind, it is actually an example of a transmission delay arising in a different protocol layer. At the time when we identified the four kinds of delay, we had not yet discussed protocol layers, so this subtlety did not appear.

Each protocol layer of a network can impose any or all of the four kinds of delay. For example, what Section 1.2.2 identified as processing delay is actually composed of processing delay in the link layer (e.g., time spent bit-stuffing and calculating checksums), processing delay in the network layer (e.g., time spent looking up addresses in forwarding tables), and processing delay in the end-to-end layer (e.g., time spent compressing data, dividing a long message into segments and later reassembling it, and encrypting or decrypting message contents).

Similarly, transmission delay can also arise in each layer. At the link layer, transmission delay is measured from when the first bit of a frame enters a link until the last bit of that same frame enters the link. The length of the frame and the data rate of the link together determine its magnitude. The network layer does not usually impose any additional transmission delays of its own, but in choosing a route (and thus the number of hops) it helps determine the number of link-layer transmission delays. The end-to-end layer imposes an additional transmission delay whenever the pacing effect of either congestion control or flow control causes it to wait for permission to send. The data rate of the bottleneck in the end-to-end path, the round-trip time, and the size of the flow-control window together determine the magnitude of the end-to-end transmission delay. The end-to-end layer may also delay delivering a message to its client when waiting for an out-of-order segment of that message to arrive, and it may delay delivery in order to reduce jitter. These delivery delays are another component of end-to-end transmission delay.

Any layer that imposes either processing or transmission delays can also cause queuing delays for subsequent packets. The transmission delays of the link layer can thus create queues, where packets wait for the link to become available. The network layer can impose queuing delays if several packets arrive at a router during the time it spends figuring out how to forward a packet. Finally, the end-to-end layer can also queue up packets waiting for flow control or congestion control permission to enter the network.

Propagation delay might seem to be unique to the link layer, but a careful accounting will reveal small propagation delays contributed by the network and end-to-end layers as messages are moved around inside a router or end-node computer. Because the distances involved in a network link are usually several orders of magnitude larger than those inside a computer, the propagation delays of the network and end-to-end layers can usually be ignored.

This page titled 1.7: A Network System Design Issue - Congestion Control is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

1.8: Wrapping Up Networks

This chapter has introduced a lot of concepts and techniques for designing and dealing with data communication networks. A natural question arises: "Is all of this stuff really needed?"

The answer, of course, is "It depends." It obviously depends on the application, which may not require all of the features that the various network layers provide. It also depends on several lower-layer aspects.

For example, if at the link layer the entire network consists of just a single point-to-point link, there is no need for a network layer at all. There may still be a requirement to multiplex the link, but multiplexing does not require any of the routing function of a network layer because everything that goes in one end of the link is destined for whatever is attached at the other end. In addition, there is probably no need for some of the transport services of the end-to-end layer because frames, segments, streams, or messages come out of the link in the same order they went in. A short link is sometimes quite reliable, in which case the end-to-end layer may not need to provide a duplicate-generating resend mechanism and in turn can omit duplicate suppression. What remains in the end-to-end function is session services (such as authenticating the identity of the user and encrypting the communication for privacy) and presentation services (marshaling application data into a form that can be transmitted as a message or a stream.)

Similarly, if at the link layer the entire network consists of just a single broadcast link, a network layer is needed, but it is vestigial: it consists of just enough intelligence at each receiver to discard packets addressed to different targets. For example, a backplane bus described is a reliable broadcast network with an end-to-end layer that provides only presentation services. For another example, an Ethernet, which is less reliable, needs a healthier set of end-to-end services because it exhibits greater variations in delay. On the other hand, packet loss is still rare enough that it may be possible to ignore it, and reordered packet delivery is not a problem.

As with all aspects of computer system design, good judgment and careful consideration of trade-offs are required for a design that works well and also is economical.

This summary completes our conceptual material about networks. In the remaining sections of this chapter are a case study of a popular network design, the Ethernet, and a collection of network-related war stories.

This page titled 1.8: Wrapping Up Networks is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

1.9: Case Study - Mapping the Internet to the Ethernet

This case study begins with a brief description of Ethernet using the terminology and network model of this chapter. It then explores the issues involved in routing that are raised when one maps a packet-forwarding network such as the Internet to an Ethernet.

1.9.1: A Brief Overview of Ethernet

Ethernet is the generic name for a family of local area networks based on broadcast over a shared wire or fiber link on which all participants can hear one another's transmissions. Ethernet uses a listen-before-sending rule (known as "carrier sense") to control access and it uses a listen-while-sending rule to minimize wasted transmission time if two stations happen to start transmitting at the same time, an error known as a **collision**. This protocol is named **Carrier Sense Multiple Access with Collision Detection**, and abbreviated CSMA/CD. Ethernet was demonstrated in 1974 and documented in a 1976 paper by Metcalfe and Boggs [Suggestions for Further Reading 1.1.3]. Since that time several successively higher-speed versions have evolved. Originally designed as a half duplex system, a full duplex, point-to-point specification that relaxes length restrictions was a later development. The primary forms of Ethernet that one encounters either in the literature or in the field are the following:

- **Experimental Ethernet**, a long obsolete 3 megabit per second network that was used only in laboratory settings. The 1976 paper describes this version.
- **Standard Ethernet**, a 10 megabit per second version.
- **Fast Ethernet**, a 100 megabit per second version.
- **Gigabit Ethernet**, which operates at the eponymous speed.

Standard, fast, and gigabit Ethernet all share the same basic protocol design and format. The format of an Ethernet frame (with some subfield details omitted) is:

Table 1.9.1: Basic format of an Ethernet frame

leader	destination	source	type	data	checksum
64 bits	48 bits	48 bits	16 bits	368 to 12,000 bits	32 bits

The `leader` field contains a standard bit pattern that frames the payload and also provides an opportunity for the receiver's phase-locked loop to synchronize. The `destination` and `source` fields identify specific stations on the Ethernet. The type field is used for protocol multiplexing in some applications and to contain the length of the data field in others. (The format diagram does not show that each frame is followed by 96 bit times of silence, which allows finding the end of the frame when the length field is absent.)

The maximum extent of a half duplex Ethernet is determined by its propagation time; the controlling requirement is that the maximum two-way propagation time between the two most distant stations on the network be less than the 576 bit times required to transmit the shortest allowable packet. This restriction guarantees that if a collision occurs, both colliding parties are certain to detect it. When a sending station does detect a collision, it waits a random time before trying again; when there are repeated collisions it uses exponential backoff to increase the interval from which it randomly chooses the time to wait. In a full duplex, point-to-point Ethernet there are no collisions, and the maximum length of the link is determined by the physical medium.

There are many fascinating aspects of Ethernet design and implementation ranging from debates about its probabilistic character to issues of electrical grounding; we omit all of them here. For more information, a good place to start is with the paper by Metcalfe and Boggs. The Ethernet is completely specified in a series of IEEE standards numbered 802.3, and it is described in great detail in most books devoted to networking.

1.9.2: Broadcast Aspects of Ethernet

Section 1.4.5 of this chapter mentioned Ethernet as an example of a network that uses a broadcast link. As illustrated in Figure 1.9.1, the Ethernet link layer is quite simple: every frame is delivered to every station. At its network layer, each Ethernet station has a 48 bit address, which to avoid confusion with other addresses we will call a station identifier. (To help reduce ambiguity in the examples that follow, station identifiers will be the only two-digit numbers.)

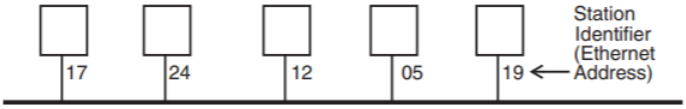


Figure 1.9.1: An Ethernet.

The network layer of Ethernet is quite simple. On the sending side, `ETHERNET_SEND` does nothing but pass the call along to the link layer. On the receiving side, the network handler procedure of the Ethernet network layer is straightforward:

```
procedure ETHERNET_HANDLE (net_packet, length)
    destination ← net_packet.target_id
    if destination = my_station_id then
                    GIVE_TO_END_LAYER  (net_packet.data,  net_packet.end_protocol,
net_packet.source_id)
    else
        ignore packet
```

There are two differences between this network layer handler and the network layer handler of a packet-forwarding network:

- Because the underlying physical link is a broadcast link, it is up to the network layer of the station to figure out that it should ignore packets not addressed specifically to it.
- Because every packet is delivered to every Ethernet station, there is no need to do any forwarding.

Most Ethernet implementations actually place `ETHERNET_HANDLE` completely in hardware. One consequence is that the hardware of each station must know its own station identifier, so it can ignore packets addressed to other stations. This identifier is wired in at manufacturing time, but most implementations also provide a programmable identifier register that overrides the wired-in identifier.

Since the link layer of Ethernet is a broadcast link, it offers a convenient additional opportunity for the network layer to create a **broadcast network**. For this purpose, Ethernet reserves one station identifier as a **broadcast address**, and the network handler procedure acquires one additional test:

```
procedure ETHERNET_HANDLE (net_packet, length)
    destination ← net_packet.target_id
    if destination = my_station_id or destination = BROADCAST_ID then
                    GIVE_TO_END_LAYER  (net_packet.data,  net_packet.end_protocol,
net_packet.source_id)
    else
        ignore packet
```

The Ethernet broadcast feature is seductive. It has led people to propose also adding broadcast features to packet-forwarding networks. It is possible to develop broadcast algorithms for a forwarding network, but it is a much trickier business. Even in Ethernet, broadcast must be used judiciously. Reliable transport protocols that require every receiving station to send back an acknowledgment lead to a problematic flood of acknowledgment packets. In addition, broadcast mechanisms are too easily triggered by mistake. For example, if a request is accidentally sent with its source address set to the broadcast address, the response

will be broadcast to all network attachment points. The worst case is a broadcast sent from the broadcast address, which can lead to a flood of broadcasts. Such mechanisms make a good target for malicious attack on a network, so it is usually thought to be preferable not to implement them at all.

1.9.3: Layer Mapping: Attaching Ethernet to a Forwarding Network

Suppose we have several workstations and perhaps a few servers in one building, all connected using an Ethernet, and we would like to attach this Ethernet to the packet-forwarding network illustrated in Figure 1.5.4 from Section 1.5.2, by making the Ethernet a sixth link on router K in that figure. This connection produces the configuration of Figure 1.9.2.

There are three kinds of network-related labels in the figure. First, each link is numbered with a local single-digit link identifier (in *italics*), as viewed from within the station that attaches that link. Second, as in Figure 1.9.1, each Ethernet attachment point has a two-digit Ethernet station identifier. Finally, each station has a one-letter name, just as in the packet-forwarding network in the figure from Section 1.5.2. With this configuration, workstation L sends a remote procedure call to server N by sending one or more packets to station 18 of the Ethernet attached to it as link number 1.

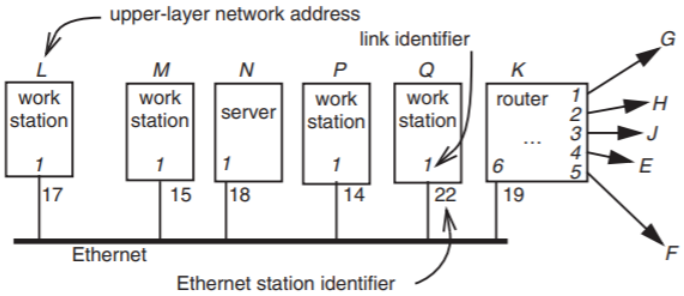


Figure 1.9.2: Connecting an Ethernet to a packet forwarding network.

Workstation L might also want to send a request to the computer connected to the destination E, which requires that L actually send the request packet to router K at Ethernet station 19 for forwarding to destination E. The complication is that E may be at address 15 of the packet-forwarding network, while workstation M is at station 15 of the Ethernet. Since Ethernet station identifiers may be wired into the hardware interface, we probably can't set them to suit our needs, and it might be a major hassle to go around changing addresses on the original packet-forwarding network. The bottom line here is that we can't simply use Ethernet station identifiers as the network addresses in our packet-forwarding network. But this conclusion seems to leave station L with no way of expressing the idea that it wants to send a packet to address E.

We were able to express this idea in words because in the two figures we assigned a unique letter identifier to every station. What our design needs is a more universal concept of network—a cloud that encompasses every station in both the Ethernet and the packet-forwarding network and assigns each station a unique network address. Recall that the letter identifiers originally stood for addresses in the packet-forwarding network; they may even be hierarchical identifiers. We can simply extend that concept and assign identifiers from that same numbering plan to each Ethernet station, in addition to the wired-in Ethernet station identifiers.

What we are doing here is mapping the letter identifiers of the packet-forwarding network to the station identifiers of the Ethernet. Since the Ethernet is itself decomposable into a network layer and a link layer, we can describe this situation, as was suggested in Section 1.3.6, as a mapping composition—an upper-level network layer is being mapped to lower-level network layer. The upper network layer is a simplified version of the Internet, so we will label it with the name "internet," using a lower case initial letter as a reminder that it is simplified. Our internet provides us with a language in which workstation L can express the idea that it wants to send an RPC request to server E, which is located somewhere beyond the router:

```
NETWORK_SEND (data, length, RPC, INTERNET, E)
```

where E is the internet address of the server, and the fourth argument selects our internet forwarding protocol from among the various available network protocols. With this scheme, station A also uses the same network address E to send a request to that server. In other words, this internet provides a universal name space.

Our new, expanded, internet network layer must now map its addresses into the Ethernet station identifiers required by the Ethernet network layer. For example, when workstation L sends a remote procedure call to server N by

`NETWORK_SEND (data, length, RPC, INTERNET, N)`

the internet network layer must turn this into the Ethernet network-layer call

`NETWORK_SEND (data, length, RPC, ENET, 18)`

in which we have named the Ethernet network-layer protocol `ENET` .

For this purpose, L must maintain a table like Table 1.9.2, in which each internet address maps to an Ethernet station identifier. This table maps, for example, address N to `ENET` , station 18, as required for the `NETWORK_SEND` call above. Since our internet is a forwarding network, our table also indicates that for address E the thing to do is send the packet on `ENET` to station 19, in the hope that it (a router in our diagram) will be sufficiently well connected to pass the packet along to its destination. This table is just another example of a forwarding table like the ones in Section 1.5 of this chapter.

Table 1.9.2: Forwarding table to connect upper- and lower-layer addresses

internet address	Ethernet/station
M	enet/15
N	enet/18
P	enet/14
Q	enet/22
K	enet/19
E	enet/19

1.9.4: The Address Resolution Protocol

The forwarding table could simply be filled in by hand, by a network administrator who, every time a new station is added to an Ethernet, visits every station already on that Ethernet and adds an entry to its forwarding table. But the charm of manual network management quickly wears thin as the network grows in number of stations, and a more automatic procedure is usually implemented.

An elegant scheme, known as the **address resolution protocol (ARP)**, takes advantage of the broadcast feature of Ethernet to dynamically fill in the forwarding table as it is needed. Suppose we start with an empty forwarding table and that an application calls the internet `NETWORK_SEND` interface in L, asking that a packet be sent to internet address M. The internet network layer in L looks in its local forwarding table, and finding nothing there that helps, it asks the Ethernet network layer to send a query such as the following:

`NETWORK_SEND ("where is M?", 11, ARP, ENET, BROADCAST)`

where 10 is the number of bytes in the query, `ARP` is the network-layer protocol we are using, rather than `INTERNET` , and `BROADCAST` is the station identifier that is reserved for broadcast on this Ethernet.

Since this query uses the broadcast address, it will be received by the Ethernet network layer of every station on the attached Ethernet. Each station notices the `ARP` protocol type and passes it to its `ARP` handler in the upper network layer. Each `ARP` handler checks the query, and if it discovers its own internet address in the inquiry, sends a response:

`NETWORK_SEND ("M is at station 15", 18, ARP, ENET, BROADCAST)`

At most, one station—the one whose internet address is named by the `ARP` request—will respond. All the others will ignore the `ARP` request. When the `ARP` response arrives at station 17, that station's Ethernet network layer will pass it up to the `ARP` handler in its upper network layer, which will immediately add an entry relating address M to station 15 to its forwarding table, shown below. The internet network handler of station 17 can now proceed with its originally requested send operation.

Table 1.9.3: First entry of the forwarding table

Internet address	Ethernet/station
M	enet/15

Suppose now that station L tries to send a packet to server E, which is on the internet but not directly attached to the Ethernet. In that case, server E does not hear the Ethernet broadcast, but the router at station 19 does, and it sends a suitable ARP response instead. The forwarding table then has a second entry as shown below. Station L can now send the packet to the router, which presumably knows how to forward the packet to its intended destination.

Table 1.9.4: Building up the forwarding table

Internet address	Ethernet/station
M	enet/15
E	enet/19

One more step is required—the server at E will not be able to reply to station L unless L is in its own forwarding table. This step is easy to arrange: whenever router K hears, via ARP , of the existence of a station on its attached Ethernet, it simply adds that internet address to the list of addresses that it advertises, and whatever routing protocol it is using will propagate that information throughout the internet. If hierarchical addresses are in use, the region designer might assign a region number to be used exclusively for all the stations on one Ethernet, to simplify routing.

Mappings from Ethernet station identifiers to the addresses of the higher network level are thus dynamically built up, and eventually station L will have the full table shown in Table 1.9.2. Typical systems deployed in the field have developed and refined this basic set of dynamic mapping ideas in many directions: The forwarding table is usually managed as a cache, with entries that time out or can be explicitly updated, to allow stations to change their station identifiers; the ARP response may also be noted by stations that didn't send the original ARP request for their own future reference; a newly-attached station may, without being asked, broadcast what appears to be an ARP response simply to make itself known to existing stations (advertising); and there is even a reverse version of the ARP protocol that can be used by a station to ask if anyone knows its own higher-level network address, or to ask that a higher-level address be assigned to it. These refinements are not important to our case study, but many of them are essential to smooth network management.

This page titled 1.9: Case Study - Mapping the Internet to the Ethernet is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

1.10: War Stories - Surprises in Protocol Design

1.10.1: Fixed Timers Lead to Congestion Collapse in NFS

A classic example of congestion collapse appeared in early releases of the Sun Network File System (NFS). The NFS server implemented at-least-once semantics with an idempotent stateless interface. The NFS client was programmed to be persistent. If it did not receive a response after some fixed number of seconds, it would resend its request, repeating forever if necessary. The server simply ran a first-in, first-out queue, so if several NFS clients happened to make requests of the server at about the same time, the server would handle the requests one at a time in the order that they arrived. These apparently plausible arrangements on the parts of the client and the server, respectively, set the stage for the problem.

As the number of clients increased, the length of the queue increased accordingly. With enough clients, the queue would grow long enough that some requests would time out before the server got to them. Those clients, upon timing out, would repeat their requests. In due course, the server would handle the original request of a client that had timed out, send a response, and that client would go away happy. But that client's duplicate request was still in the server's queue. The stateless NFS server had no way to tell that it had already handled the duplicate request, so when it got to the duplicate it would go ahead and handle it again, taking the same time as before, and sending an unneeded response. The client ignored this response, but the time spent by the server handling the duplicate request was wasted, and the waste occurred at a time when the server could least afford it—it was already so heavily loaded that at least one client had timed out.

Once the server began wasting time handling duplicate requests, the queue grew still longer, causing more clients to time out, leading to more duplicate requests. The observed effect was that a steady increase of load would result in a steady increase of satisfied requests, up to the point that the server was near full capacity. If the load ever exceeded the capacity, even for a short time, every request from then on would time out and be duplicated, resulting in a doubling of the load on the server. That wasn't the end —with a doubled load, clients would begin to time out a second time, send their requests yet again, thus tripling the load. From there, things would continue to deteriorate, with no way to recover.

From the NFS server’s point of view, it was just doing what its clients were asking, but from the point of view of the clients the useful throughput had dropped to zero. The solution to this problem was for the clients to switch to an exponential backoff algorithm in their choice of timer setting: each time a client timed out it would double the size of its timer setting for the next repetition of the request.

Lesson:

Fixed timers are always a source of trouble, sometimes catastrophic trouble.

1.10.2: Autonet Broadcast Storms

Autonet, an experimental local area network designed at the Digital Equipment Corporation Systems Research Center, handled broadcast in an elegant way. The network was organized as a tree. When a node sent a packet to the broadcast address, the network first routed the packet up to the root of the tree. The root turned the packet around and sent it down every path of the tree. Nodes accepted only packets going downward, so this procedure ensured that a broadcast packet would reach every connected node, but no more than once. But every once in a while, the network collapsed with a storm of repeated broadcast packets. Analysis of the software revealed no possible source of the problem. It took a hardware expert to figure it out.

The physical layer of the Autonet consisted of point-to-point coaxial cables. An interesting property of an unterminated coaxial cable is that it will almost perfectly reflect any signal sent down the cable. The reflection is known as an “echo”. Echoes are one of the causes of ghosts in analog cable television systems.

In the case of the Autonet, the network card in each node properly terminated the cable, eliminating echoes. But if someone disconnected a computer from the network, and left the cable dangling, that cable would echo everything back to its source.

Suppose someone disconnects a cable, and someone else in the network sends a packet to the broadcast address. The network routes the packet up to the root of the tree, the root turns the packet around and sends it down the tree. When the packet hits the end of the unterminated cable, it reflects and returns to the other end of the cable looking like a new upward bound packet with the

broadcast address. The node at that end dutifully forwards the packet toward the root node, which, upon receipt turns it around and sends it again. And again, and again, as fast as the network can carry the packet.

Lessons:

Emergent properties often arise from the interaction of apparently unrelated system features operating at different system layers: in this case, link-layer reflections and network-layer broadcasts.

1.10.3: Emergent Phase Synchronization of Periodic Protocols

Some network protocols involve periodic polling. Examples include picking up mail, checking for chat buddies, and sending "are-you-there?" inquiries for reassurance that a coworker hasn't crashed. For a specific example, a workstation might send a broadcast packet every five minutes to announce that it is still available for conversations. If there are dozens of such workstations on the same local area network, the designer would prefer that they not all broadcast simultaneously. One might assume that, even if they all broadcast with the same period, if they start at random their broadcasts would be out of phase and it would take a special effort to synchronize their phases and keep them that way. Unfortunately, it is common to discover that they have somehow synchronized themselves and are all trying to broadcast at the same time.

How can this be? Suppose, for example, that each one of a group of workstations sends a broadcast and then sets a timer for a fixed interval. When the timer expires, it sends another broadcast and, after sending, it again sets the timer. During the time that it is sending the broadcast message, the timer is not running. If a second workstation happens to send a broadcast during that time, both workstations take a network interrupt, each accepts the other station's broadcast, and makes a note of it, as might be expected. But the time required to handle the incoming broadcast interrupts slightly delays the start of the next timing cycle for both of the workstations, whereas broadcasts that arrive while a workstation's timer is running don't affect the timer. Although the delay is small, it does shift the timing of these workstation's broadcasts relative to all of the other workstations. The next time this workstation's timer expires, it will again be interrupted by the other workstation, since they are both using the same timer value, and both of their timing cycles will again be slightly lengthened. The two workstations have formed a phase-locked group, and will remain that way indefinitely.

More important, the two workstations that were accidentally synchronized are now polling with a period that is slightly larger than all the other workstations. As a result, their broadcasts now precess relative to the others, and eventually will overlap the time of broadcast of a third workstation. That workstation will then join the phase-locked group, increasing the rate of precession, and things continue from there. The problem is that the system design unintentionally includes an emergent phase-locked loop, similar to the one described in Section 1.4.1.

The generic mechanism is that the supposed "fixed" interval does not count the running time of the periodic program, and that for some reason that running time is different when two or more participants happen to run concurrently. In a network, it is quite common to find that unsynchronized activities with identical timing periods become synchronized.

Lesson:

Fixed timers have many evils. Don't assume that unsynchronized periodic activities will stay that way.

1.10.4: Wisconsin Time Server Meltdown

NETGEAR®, a manufacturer of Ethernet and wireless equipment, added a feature to four of its low-cost wireless routers intended for home use: a log of packets that traverse the router. To be useful in debugging, the designers realized that the log needed to timestamp each log entry, but adding timestamps required that the router know the current date and time. Since the router would be attached to the Internet, the designers added a few lines of code that invoked a simple network time service protocol known as SNTP. Since SNTP requires that the client invoke a specific time service, there remained a name discovery problem. They solved it by configuring the firmware code with the Internet address of a network time service. Specifically, they inserted the address 128.105.39.11, the network address of one of the time servers operated by the University of Wisconsin. The designers surrounded this code with a persistent sender that would retry the protocol once per second until it received a response. Upon receiving a

response, it refreshed the clock with another invocation of SNTP, using the same persistent sender, on a schedule ranging from once per minute to once per day, depending on the firmware version.

On May 14, 2003, at about 8:00 a.m. local time, the network staff at the University of Wisconsin noticed an abrupt increase in the rate of inbound Internet traffic at their connection to the Internet—the rate jumped from 20,000 packets per second to 60,000 packets per second. All of the extra traffic seemed to be SNTP packets targeting one of their time servers, and specifying the same UDP response port, port 23457. To prevent disruption to university network access, the staff installed a temporary filter at their border routers that discarded all incoming SNTP request packets that specified a response port of 23457. They also tried invoking an SNTP protocol access control feature in which the service can send a response saying, in effect, “go away”, but it had no effect on the incoming packet flood.

Over the course of the next few weeks, SNTP packets continued to arrive at an increasing rate, soon reaching around 270,000 packets per second, and consuming about 150 megabits per second of Internet connection capacity. Analysis of the traffic showed that the source addresses seemed to be legitimate and that any single source was sending a packet about once per second. A modest amount of sleuthing identified the NETGEAR routers as the source of the packets and the firmware as containing the target address and response port numbers. Deeper analysis established that the immediate difficulty was congestion collapse. NETGEAR had sold over 700,000 routers containing this code world-wide. As the number in operation increased, the load on the Wisconsin time service grew gradually until one day the response latency of the server exceeded one second. At that point, the NETGEAR router that made that request timed out and retried, thereby increasing its load on the time service, which increased the time service response latency for future requesters. After a few such events, essentially all of the NETGEAR routers would start to time out, thereby multiplying the load they presented by a factor of 60 or more, which ensured that the server latency would continue to exceed their one second timer.

How Wisconsin and NETGEAR solved this problem, and at whose expense, is a whole separate tale. *

Lessons:

1. Fixed timers were once again found at the scene of an accident.
2. Configuring a fixed Internet address, which is overloaded with routing information, is a bad idea. In this case, the wired-in address made it difficult to repair the problem by rerouting requests to a different time service, such as one provided by NETGEAR. The address should have been a variable, preferably one that could be **hidden with indirection** (decouple modules with indirection).
3. There is a reason for features such as the “go away” response in SNTP; it is risky for a client to implement only part of a protocol.

* For that story, see http://pages.cs.wisc.edu/~plonka/netgear-sntp/. This incident is also described in David Mills, Judah Levine, Richard Schmidt and David Plonka. “Coping with overload on the Network Time Protocol public servers.” *Proceedings of the Precision Time and Time Interval (PTTI) Applications and Planning Meeting* (Washington DC, December 2004), pages 5-16.

This page titled 1.10: War Stories - Surprises in Protocol Design is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

1.11: Exercises

Exercise 1.11.1

1996–1–1e

Chapter 1 discussed four general methods for coping with complexity: modularity, abstraction, hierarchy, and layering. Which of those four methods does a protocol stack use as its primary organizing scheme?

Exercise 1.11.2

1999–2–03

The *end-to-end argument*

A. is a guideline for placing functions in a computer system;

B. is a rule for placing functions in a computer system;

C. is a debate about where to place functions in a computer system;

D. is a debate about anonymity in computer networks.

Exercise 1.11.3

1998–2–01

Of the following, the best example of an *end-to-end argument* is:

A. If you laid all the Web hackers in the world end to end, they would reach from Cambridge to CERN.

B. Every byte going into the write end of a UNIX pipe eventually emerges from the pipe's read end.

C. Even if a chain manufacturer tests each link before assembly, he'd better test the completed chain.

D. Per-packet checksums must be augmented by a parity bit for each byte.

E. All important network communication functions should be moved to the application layer.

Exercise 1.11.4

1995–1–5a

Give two scenarios in the form of timing diagrams showing how a duplicate request might end up at a service.

Exercise 1.11.5

1987–1–2a,b

After sending a frame, a certain piece of network software waits one second for an acknowledgment before retransmitting the frame. After each retransmission, it cuts delay in half, so after the first retransmission the wait is $1/2$ second, after the second retransmission the wait is $1/4$ second, etc. If it has reduced the delay to $1/1024$ second without receiving an acknowledgment, the software gives up and reports to its caller that it was not able to deliver the frame.

a) Is this a good way to manage retransmission delays for Ethernet? Why or why not?

b) Is this a good way to manage retransmission delays for a receive-and-forward network? Why or why not?

Exercise 1.11.6

1995–1–1f

Variable delay is an intrinsic problem of isochronous networks. True or false?

Exercise 1.11.7

1987–1–3a,b

Host A is sending frames to host B over a noisy communication link. The median transit time over the communication link is 100 milliseconds. The probability of a frame being damaged en route in either direction across the communication link is α, and B can reliably detect the damage. When B gets a damaged frame it simply discards it. To ensure that frames arrive safely, B sends an acknowledgment back to A for every frame received intact.

a) How long should A wait for a frame to be acknowledged before retransmitting it?

b) What is the average number of times that A will have to send each frame?

Exercise 1.11.8

2000–2–02

Consider the protocol reference model of this chapter with the link, network, and end-to-end layers. Which of the following is a behavior of the reference model?

A. An end-to-end layer at an end host tells its network layer which network layer protocol to use to reach a destination.

B. The network layer at a router maintains a separate queue of packets for each end-to-end protocol.

C. The network layer at an end host looks at the end-to-end type field in the network header to decide which end-to-end layer protocol handler to invoke.

D. The link layer retransmits packets based on the end-to-end type of the packets: if the end-to-end protocol is reliable, then a link-layer retransmission occurs when a loss is detected at the link layer, otherwise not.

Exercise 1.11.9

1997–1–1e

Congestion is said to occur in a receive-and-forward network when

A. Communication stalls because of cycles in the flow-control dependencies.

B. The throughput demanded of a network link exceeds its capacity.

C. The volume of e-mail received by each user exceeds the rate at which users can read e-mail.

D. The load presented to a network link persistently exceeds its capacity.

E. The amount of space required to store routing tables at each node becomes burdensome.

Exercise 1.11.10

1994–2–6

Alice has arranged to send a stream of data to Bob using the following protocol:

- Each message segment has a block number attached to it; block numbers are consecutive starting with 1.
- Whenever Bob receives a segment of data with the number N, he sends back an acknowledgment saying "OK to send block $N+1$".
- Whenever Alice receives an "OK to send block K," she sends block K.

Alice initiates the protocol by sending a block numbered 1; she terminates the protocol by ignoring any "OK to send block K" for which K is larger than the number on the last block she wants to send. The network has been observed to never lose message segments, so Bob and Alice have made no provision for timer expirations and retries. They have also made no provision for deduplication. Unfortunately, the network systematically delivers every segment twice. Alice starts the protocol, planning to send a three-block stream. How many "OK to send block 4" responses does she ignore at the end?

Exercise 1.11.11

1980–3–3

A and B agree to use a simple window protocol for flow control for data going from A to B: When the connection is first established, B tells A how many message segments B can accept, and as B consumes the segments it occasionally sends a message to A saying "you can send M more". In operation, B notices that occasionally A sends more segments than it was supposed to. Explain.

Exercise 1.11.12

1995–2–2a...c

Assume a client and a service are directly connected by a private, 800,000 bytes per second link. Also assume that the client and the service produce and consume message segments at the same rate. Using acknowledgments, the client measures the round-trip between itself and the service to be 10 milliseconds.

a) If the client is sending message segments that require 1000-byte frames, what is the smallest window size that allows the client to achieve 800,000 bytes per second throughput?

b) One scheme for establishing the window size is similar to the **slow start** congestion control mechanism. The idea is that the client starts with a window size of one. For every segment received, the service responds with an acknowledgment telling the client to double the window size. The client does so until it realizes that there is no point in increasing it further. For the same parameters as in part (a), how long would it take for the client to realize it has reached the maximum throughput?

c) Another scheme for establishing the window size is called **fast start**. In (an oversimplified version of) fast start, the client simply starts sending segments as fast as it can, and watches to see when the first acknowledgment returns. At that point, it counts the number of outstanding segments in the pipeline, and sets the window size to that number. Again using the same parameters as in part (a), how long will it take for the client to know it has achieved the maximum throughput?

Exercise 1.11.13

1988–2–2a...c

A satellite in stationary orbit has a two-way data channel that can send frames containing up to 1000 data bytes in a millisecond. Frames are received without error after 249 milliseconds of propagation delay. A transmitter T frequently has a data file that takes 1000 of these maximal-length frames to send to a receiver R. T and R start using lock-step flow control. R allocates a buffer which can hold one message segment. As soon as the buffered segment is used and the buffer is available to hold new data, R sends an acknowledgment of the same length. T sends the next segment as soon as it sees the acknowledgment for the last one.

a) What is the minimum time required to send the file?

b) T and R decide that lock-step is too slow, so they change to a **bang-bang protocol**. A bang-bang protocol means that R sends explicit messages to T saying "go ahead" or "pause". The idea is that R will allocate a receive buffer of some size B, send a goahead message when it is ready to receive data. T then sends data segments as fast as the channel can absorb them. R sends a pause message at just the right time so that its buffer will not overflow even if R stops consuming message segments. Suppose that R sends a go-ahead, and as soon as it sees the first data arrive it sends a pause. What is the minimum buffer size B_{min} that it needs?)

c) What now is the minimum time required to send the file?

Exercise 1.11.14

2000–2–09

Some end-to-end protocols include a destination field in the end-to-end header. Why?

A. So the protocol can check that the network layer routed the packet containing the message segment correctly.

B. Because an *end-to-end argument* tells us that routing should be performed at the endto-end layer.

C. Because the network layer uses the end-to-end header to route the packet.

D. Because the end-to-end layer at the sender needs it to decide which network protocol to use.

Exercise 1.11.15

1994–2–5

One value of hierarchical naming of network attachment points is that it allows a reduction in the size of routing tables used by packet forwarders. Do the packet forwarders themselves have to be organized hierarchically to take advantage of this space reduction?

Exercise 1.11.16

1982–3–4

The System Network Architecture (SNA) protocol family developed by IBM uses a flow control mechanism called pacing. With **pacing**, a sender may transmit a fixed number of message segments, and then must pause. When the receiver has accepted all of these segments, it can return a **pacing response** to the sender, which can then send another burst of message segments.

Suppose that this scheme is being used over a satellite link, with a delay from earth station to earth station of 250 milliseconds. The frame size on the link is 1000 bits, four segments are sent before pausing for a pacing response, and the satellite channel has a data rate of one megabit per second.

a) The timing diagram below illustrates the frame carrying the first segment. Fill in the diagram to show the next six frames exchanged in the pacing system. Assume no frames are lost, delays are uniform, and sender and receiver have no internal delays (for example, the first bit of the second frame may immediately follow the last bit of the first).

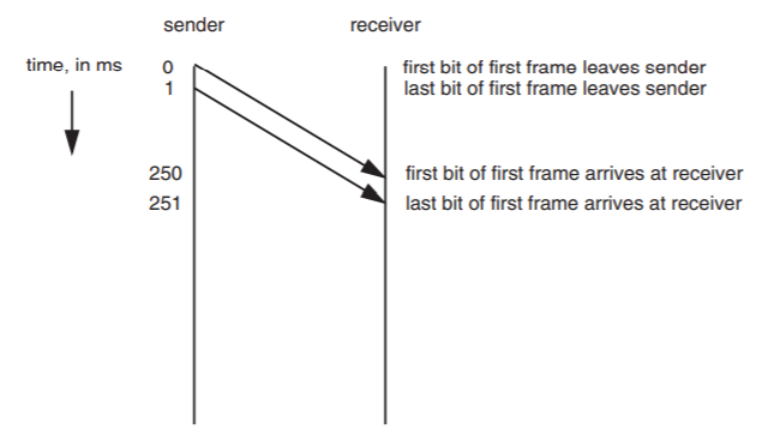


Figure 1.11.1: Timing diagram for the pacing system described in Exercise 1.11.16*a*

b) What is the maximum fraction of the available satellite capacity that can be used by this pacing scheme?

c) We would like to increase the utilization of the channel to 50% but we can't increase the frame size. How many message segments would have to be sent between pacing responses to achieve this capacity?

Exercise 1.11.17

2001–2–01, 2002–2–02, and 2004–2–2

Which are true statements about network address translators as described in Section 1.5.5?

A. NATs break the universal addressing scheme of the Internet.

B. NATs break the layering abstraction of the network model of Chapter 7.

C. NATs increase the consumption of Internet addresses.

D. NATs address the problem that the Internet has a shortage of Internet addresses.

E. NATs constrain the design of new end-to-end protocols.

F. When a NAT translates the Internet address of a packet, it must also modify the Ethernet checksum, to ensure that the packet is not discarded by the next router that handles it. The client application might be sending its Internet address in the TCP payload to the server.

G. When a packet from the public Internet arrives at a NAT box for delivery to a host behind the NAT, the NAT must examine the payload and translate any Internet addresses found therein.

H. Clients behind a NAT cannot communicate with servers that are behind the same NAT because the NAT does not know how to forward those packets.

Exercise 1.11.18

1994–1–2

Some network protocols deal with both big-endian and little-endian clients by providing two different network ports. Big-endian clients send requests and data to one port, while little-endian clients send requests and data to the other. The service may, of course, be implemented on either a big-endian or a little-endian machine. This approach is unusual—most Internet protocols call for just one network port, and require that all data be presented at that port in "network standard form", which is little- endian. Explain the advantage of the two port structure as compared with the usual structure.

Exercise 1.11.19

1994–1–3b

True or False? Ethernet cannot scale to large sizes because a centralized mechanism is used to control network contention.

Exercise 1.11.20

1999–2–01, 2000–1–04

Ethernet

A. uses luminiferous ether to carry packets.

B. uses Manchester encoding to frame bits.

C. uses exponential back-off to resolve repeated conflicts between multiple senders.

D. uses retransmissions to avoid congestion.

E. delegates arbitration of conflicting transmissions to each station.

F. always guarantees the delivery of packets.

G. can support an unbounded number of computers.

H. has limited physical range.

Exercise 1.11.21

1998-2-02

Ethernet cards have unique addresses built into them. What role do these unique addresses play in the Internet?

A. None. They are there for Macintosh compatibility only.

B. A portion of the Ethernet address is used as the Internet address of the computer using the card.

C. They provide routing information for packets destined to non-local subnets.

D. They are used as private keys in the Security Layer of the ISO protocol.

E. They provide addressing within each subnet for an Internet address resolution protocol.

F. They provide secure identification for warranty service.

Exercise 1.11.22

2004–1–3

If eight stations on an Ethernet all want to transmit one packet, which of the following statements is true?

A. It is guaranteed that all transmissions will succeed.

B. With high probability all stations will eventually end up being able to transmit their data successfully.

C. Some of the transmissions may eventually succeed, but it is likely some may not.

D. It is likely that none of the transmissions will eventually succeed.

Exercise 1.11.23

1996–2–1a

Ben Bitdiddle has been thinking about remote procedure call. He remembers that one of the problems with RPC is the difficulty of passing pointers: since pointers are really just addresses, if the service dereferences a client pointer, it'll get some value from *its* address space, rather than the intended value in the client's address space. Ben decides to redesign his RPC system to always pass, in the place of a bare pointer, a structure consisting of the original pointer plus a context reference. Louis Reasoner, excited by Ben's insight, decides to change *all* end-to-end protocols along the same lines. Argue for or against Louis's decision.

Exercise 1.11.24

1996–2–2

*Alyssa's mobiles.** Alyssa P. Protocol-Hacker is designing an end-to-end protocol for locating mobile hosts. A **mobile host** is a computer that plugs into the network at different places at different times, and get assigned a new network address at each place. The system she starts with assigns each host a home location, which can be found simply by looking the user up in a name service. Her end-to-end protocol will use a network that can reorder packets, but doesn't ever lose or duplicate them. Her first protocol is simple: every time a user moves, store a forwarding pointer at the previous location, pointing to the new location. This creates a chain of forwarding pointers with the permanent home location at the beginning and the mobile host at the end. Packets meant for the mobile host are sent to the home location, which forwards them along the chain until they reach the mobile host itself. (The chain is truncated when a mobile host returns to a previously visited location.)

Alyssa notices that because of the long chains of forwarding pointers, performance generally gets worse each time she moves her mobile host. Alyssa's first try at fixing the problem works like this: Each time a mobile host moves, it sends a message to its home location indicating its new location. The home location maintains a pointer to the new location. With this protocol, there are no chains at all. Places other than the home location do not maintain forwarding information.

a) When this protocol is implemented, Alyssa notices that packets regularly get lost when she moves from one location to another. Explain why or give an example.

Alyssa is disappointed with her first attempt, and decides to start over. In her new scheme, no forwarding pointers are maintained anywhere, not even at the home node. Say a packet destined for a mobile host A arrives at a node N. If N can directly communicate with A, then N sends the packet to A, and we're done. Otherwise, N broadcasts a search request for A to all the other fixed nodes in the network. If A is near a different fixed node N', then N' responds to the search request. On receiving this response, N forwards the packet for A to N'.

b) Will packets get lost with this protocol, even if A moves before the packet gets to N'? Explain.

Unfortunately, the network doesn't support broadcast efficiently, so Alyssa goes back to the keyboard and tries again. Her third protocol works like this. Each time a mobile host moves, say from N to N', a forwarding pointer is stored at N pointing to

N'. Every so often, the mobile host sends a message to its permanent home node with its current location. Then, the home node propagates a message down the forwarding chain, asking the intermediate nodes to delete their forwarding state.

c) Can Alyssa ever lose packets with this protocol? Explain. (Hint: think about the properties of the underlying network.)

d) What additional steps can the home node take to ensure that the scheme in part (c) never loses packets?

---

* Credit for developing Exercise 1.11.24goes to Anant Agarwal.

Exercise 1.11.25

1997–2–2

ByteStream, Inc. sells three data-transfer products: Send-and-wait, Blast, and Flow-control. Mike R. Kernel is deciding which product to use. The protocols work as follows:

- **Send-and-wait** sends one segment of a message and then waits for an acknowledgment before sending the next segment.
- **Flow-control** uses a sliding window of 8 segments. The sender sends until the window closes (i.e., until there are 8 unacknowledged segments). The receiver sends an acknowledgment as soon as it receives a segment. Each acknowledgment opens the sender's window with one segment.
- **Blast** uses only one acknowledgment. The sender blasts all the segments of a message to the receiver as fast as the network layer can accept them. The last segment of the blast contains a bit indicating that it is the last segment of the message. After sending all segments in a single blast, the sender waits for one acknowledgment from the receiver. The receiver sends an acknowledgment as soon as it receives the last segment.

Mike asks you to help him compute for each protocol its maximum throughput. He is planning to use a 1,000,000 bytes per second network that has a packet size of 1,000 bytes. The propagation time from the sender to the receiver is 500 microseconds. To simplify the calculation, Mike suggests making the following approximations: (1) there is no processing time at the sender and the receiver; (2) the time to send an acknowledgment is just the propagation time (number of data bytes in an ACK is zero); (3) the data segments are always 1,000 bytes; and (4) all headers are zero-length. He also assumes that the underlying communication medium is perfect (frames are not lost, frames are not duplicated, etc.) and that the receiver has unlimited buffering.

a) What is the maximum throughput for the Send-and-wait?

b) What is the maximum throughput for Flow-control?

c) What is the maximum throughput for Blast?

Mike needs to choose one of the three protocols for an application which periodically sends arbitrary-sized messages. He has a reliable network, but his application involves unpredictable computation times at both the sender and the receiver. And this time the receiver has a 20,000-byte receive buffer.

d) Which product should he choose for maximum reliable operation?

A. Send-and-wait, the others might hang.

B. Blast, which outperforms the others.

C. Flow-control, since Blast will be unreliable and Send-and-wait is slower.

D. There is no way to tell from the information given.

Exercise 1.11.26

1994–1–5

Suppose the longest packet you can transmit across the Internet can contain 480 bytes of useful data, you are using a lock-step end-to-end protocol, and you are sending data from Boston to California. You have measured the round-trip time and found that it is about 100 milliseconds.

a) If there are no lost packets, estimate the maximum data rate you can achieve.

b) Unfortunately, 1% of the packets are getting lost, so you install a resend timer set to 1000 milliseconds. Estimate the data rate you now expect to achieve.

c) On Tuesdays the phone company routes some westward-bound packets via satellite link, and we notice that 50% of the round trips now take exactly 100 extra milliseconds. What effect does this delay have on the overall data rate when the resend timer is not in use. (Assume the network does not lose any packets.)

d) Ben turns on the resend timer, but since he hadn't heard about the satellite delays he sets it to 150 milliseconds. What now is the data rate on Tuesdays? (Again, assume the network does not lose any packets.)

e) Usually, when discussing end-to-end data rate across a network, the first parameter one hears is the data rate of the slowest link in the network. Why wasn't that parameter needed to answer any of the previous parts of this question?

Exercise 1.11.27

1998–1–2a…d

Ben Bitdiddle is called in to consult for Microhard. Bill Doors, the CEO, has set up an application to control the Justice department in Washington, D.C. The client running on the TNT operating system makes RPC calls from Seattle to the server running in Washington, D.C. The server also runs on TNT (surprise!). Each RPC call instructs the Justice department on how to behave; the response acknowledges the request but contains no data (the Justice department always complies with requests from Microhard). Bill Doors, however, is unhappy with the number of requests that he can send to the Justice department. He therefore wants to improve TNT's communication facilities.

Ben observes that the Microhard application runs in a single thread and uses RPC. He also notices that the link between Seattle and Washington, D.C. is reliable. He then proposes that Microhard enhance TNT with a new communication: primitive pipe calls.

Like RPCs, pipe calls initiate remote computation on the server. Unlike RPCs, however, pipe calls return immediately to the caller and execute asynchronously on the server. TNT packs multiple pipe calls into request messages that are 1000 bytes long. TNT sends the request message to the server as soon as one of the following two conditions becomes true: 1) the message is full, or 2) the message contains at least 1 pipe call and it has been 1 second since the client last performed a pipe call. Pipe calls have no acknowledgments. Pipe calls are not synchronized with respect to RPC calls.

Ben quickly settles down to work and measures the network traffic between Seattle and Washington. Here is what he observes:

Seattle to D.C. transit time:	12.5×10^{-3} seconds
D.C. to Seattle transit time:	12.5×10^{-3} seconds
Channel bandwidth in each direction:	1.5×10^{6} bits per second
RPC or Pipe data per call:	10 bytes
Network overhead per message:	40 bytes
Size of RPC request message (per call):	50 bytes = 10 bytes data + 40 bytes overhead
Size of pipe request message:	1000 bytes (96 pipe calls per message)
Size of RPC reply message (no data):	50 bytes
Client computation time per request:	100×10^{-6} seconds
Server computation time per request:	50×10^{-6} seconds

The Microhard application is the only one sending messages on the link.

a) What is the transmission delay the client thread observes in sending an RPC request message)?

b) Assuming that only RPCs are used for remote requests, what is the maximum number of RPCs per second that will be executed by this application?

c) Assuming that all RPC calls are changed to pipe calls, what is the maximum number of pipe calls per second that will be executed by this application?

d) Assuming that every pipe call includes a serial number argument, and serial numbers increase by one with every pipe call, how could you know the last pipe call was executed?

A. Ensure that serial numbers are synchronized to the time of day clock, and wait at the client until the time of the last serial number.

B. Call an RPC both before and after the pipe call, and wait for both calls to return.

C. Call an RPC passing as an argument the serial number that was sent on the last pipe call, and design the remote procedure called to not return until a pipe call with a given serial number had been processed.

D. Stop making pipe calls for twice the maximum network delay, and reset the serial number counter to zero.

Exercise 1.11.28

1996–1–4a…d

Alyssa P. Hacker is implementing a client/service spell checker in which a network will stand between the client and the service. The client scans an ASCII file, sending each word to the service in a separate message. The service checks each word against its database of correctly spelled words and returns a one-bit answer. The client displays the list of incorrectly spelled words.

a) The client's cost for preparing a message to be sent is 1 millisecond, regardless of length. The network transit time is 10 milliseconds, and network data rate is infinite. The service can look up a word and determine whether or not it is misspelled in 100 microseconds. Since the service runs on a supercomputer, its cost for preparing a message to be sent is zero milliseconds. Both the client and service can receive messages with no overhead. How long will Alyssa's design take to spell check a 1,000 word file if she uses RPC for communication (ignore acknowledgments to requests and replies, and assume that messages are not lost or reordered)?

b) Alyssa does the same computations that you did and decides that the design is too slow. She decides to group several words into each request. If she packs 10 words in each request, how long will it take to spell check the same file?

c) Alyssa decides that grouping words still isn't fast enough, so she wants to know how long it would take if she used an asynchronous message protocol (with grouping words) instead of RPC. How long will it take to spell check the same file? (For this calculation, assume that messages are not lost or reordered.)

d) Alyssa is so pleased with the performance of this last design that she decides to use it (without grouping) for a banking system. The service maintains a set of accounts and processes requests to debit and credit accounts (i.e., modify account balances). One day Alyssa deposits $10,000 and transfers it to Ben's account immediately afterwards. The transfer fails with a reply saying she is overdrawn. But when she checks her balance afterwards, the $10,000 is there! Draw a time diagram explaining these events.

This page titled 1.11: Exercises is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

CHAPTER OVERVIEW

2: Fault Tolerance - Reliable Systems from Unreliable Components

2.1: Overview
2.2: Faults, Failures, and Fault-Tolerant Design
2.3: Measures of Reliability and Failure Tolerance
2.4: Tolerating Active Faults
2.5: Systematically Applying Redundancy
2.6: Applying Redundancy to Software and Data
2.7: Wrapping Up Reliability
2.8: Application - A Fault Tolerance Model for CMOS RAM
2.9: War Stories - Fault-Tolerant Systems That Failed
2.10: Exercises

This page titled 2: Fault Tolerance - Reliable Systems from Unreliable Components is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

2.1: Overview

Construction of reliable systems from unreliable components is one of the most important applications of modularity. There are, in principle, three basic steps to building reliable systems:

1. **Error detection:** discovering that there is an error in a data value or control signal. Error detection is accomplished with the help of **redundancy**, extra information that can verify correctness.
2. **Error containment:** limiting how far the effects of an error propagate. Error containment comes from careful application of modularity. When discussing reliability, a **module** is usually taken to be the unit that fails independently of other such units. It is also usually the unit of repair and replacement.
3. **Error masking:** ensuring correct operation despite the error. Error masking is accomplished by providing enough additional redundancy that it is possible to discover correct, or at least acceptably close, values of the erroneous data or control signal. When masking involves changing incorrect values to correct ones, it is usually called **error correction**.

Since these three steps can overlap in practice, one sometimes finds a single error-handling mechanism that merges two or even all three of the steps.

In earlier chapters, some of these ideas have already appeared in specialized forms:

- A primary purpose of enforced modularity, as provided by client/server architecture, virtual memory, and threads, is error containment.
- Network links typically use error detection to identify and discard damaged frames.
- Some end-to-end protocols time out and resend lost data segments, thus masking the loss.
- Routing algorithms find their way around links that fail, masking those failures.
- Some real-time applications fill in missing data by interpolation or repetition, thus masking loss.

and, as we will see in Chapter 5, secure systems use a technique called **defense in depth** both to contain and to mask errors in individual protection mechanisms. In this chapter we explore systematic application of these techniques to more general problems, as well as learn about both their power and their limitations.

This page titled 2.1: Overview is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

2.2: Faults, Failures, and Fault-Tolerant Design

2.2.1: Faults, Failures, and Modules

Before getting into the techniques of constructing reliable systems, let us distinguish between concepts and give them separate labels. In ordinary English discourse, the three words "fault," "failure," and "error" are used more or less interchangeably, or at least with strongly overlapping meanings. In discussing reliable systems, we assign these terms to distinct formal concepts. The distinction involves modularity. Although common English usage occasionally intrudes, the distinctions are worth maintaining in technical settings.

A **fault** is an underlying defect, imperfection, or flaw that has the potential to cause problems, whether it actually has, has not, or ever will. A weak area in the casing of a tire is an example of a fault. Even though the casing has not actually cracked yet, the fault is lurking. If the casing cracks, the tire blows out, and the car careens off a cliff, the resulting crash is a **failure**. (That definition of the term "failure" by example is too informal; we will give a more careful definition in a moment.) One fault that underlies the failure is the weak spot in the tire casing. Other faults, such as an inattentive driver and lack of a guard rail, may also contribute to the failure.

Experience suggests that faults are commonplace in computer systems. Faults come from many different sources: software, hardware, design, implementation, operations, and the environment of the system. Here are some typical examples:

- Software fault: A programming mistake, such as placing a less-than sign where there should be a less-than-or-equal sign. This fault may never have caused any trouble because the combination of events that requires the equality case to be handled correctly has not yet occurred. Or, perhaps it is the reason that the system crashes twice a day. If so, those crashes are failures.
- Hardware fault: A gate whose output is stuck at the value `ZERO` . Until something depends on the gate correctly producing the output value `ONE` , nothing goes wrong. If you publish a paper with an incorrect sum that was calculated by this gate, a failure has occurred. Furthermore, the paper now contains a fault that may lead some reader to do something that causes a failure elsewhere.
- Design fault: A miscalculation that has led to installing too little memory in a telephone switch. It may be months or years until the first time that the presented load is great enough that the switch actually begins failing to accept calls that its specification says it should be able to handle.
- Implementation fault: Installing less memory than the design called for. In this case the failure may be identical to the one in the previous example of a design fault, but the fault itself is different.
- Operations fault: The operator responsible for running the weekly payroll ran the payroll program twice last Friday. Even though the operator shredded the extra checks, this fault has probably filled the payroll database with errors such as wrong values for year-to-date tax payments.
- Environment fault: Lightning strikes a power line, causing a voltage surge. The computer is still running, but a register that was being updated at that instant now has several bits in error. Environment faults come in all sizes, from bacteria contaminating ink-jet printer cartridges to a storm surge washing an entire building out to sea.

Some of these examples suggest that a fault may either be **latent**, meaning that it isn't affecting anything right now, or **active**. When a fault is active, wrong results appear in data values or control signals. These wrong results are **errors**. If one has a formal specification for the design of a module, an error would show up as a violation of some assertion or invariant of the specification. The violation means that either the formal specification is wrong (for example, someone didn’t articulate all of the assumptions) or a module that this component depends on did not meet its own specification. Unfortunately, formal specifications are rare in practice, so discovery of errors is more likely to be somewhat *ad hoc*.

If an error is not detected and masked, the module probably does not perform to its specification. Not producing the intended result at an interface is the formal definition of a **failure**. Thus, the distinction between fault and failure is closely tied to modularity and the building of systems out of well-defined subsystems. In a system built of subsystems, the failure of a subsystem is a fault from the point of view of the larger subsystem that contains it. That fault may cause an error that leads to the failure of the larger subsystem, unless the larger subsystem anticipates the possibility of the first one failing, detects the resulting error, and masks it. Thus, if you notice that you have a flat tire, you have detected an error caused by failure of a subsystem you depend on. If you miss an appointment because of the flat tire, the person you intended to meet notices a failure of a larger subsystem. If you change to a spare tire in time to get to the appointment, you have masked the error within your subsystem. **Fault tolerance** thus consists of noticing active faults and component subsystem failures, and doing something helpful in response.

One such helpful response is **error containment**, which is another close relative of modularity and the building of systems out of subsystems. When an active fault causes an error in a subsystem, it may be difficult to confine the effects of that error to just a portion of the subsystem. On the other hand, one should expect that, as seen from outside that subsystem, the only effects will be at the specified interfaces of the subsystem. In consequence, the boundary adopted for error containment is usually the boundary of the smallest subsystem inside which the error occurred. From the point of view of the next higher-level subsystem, the subsystem with the error may contain the error in one of four ways:

1. Mask the error, so the higher-level subsystem does not realize that anything went wrong. One can think of failure as falling off a cliff. and masking as a way of providing some separation from the edge.
2. Detect and report the error at its interface, producing what is called a **fail-fast** design. Fail-fast subsystems simplify the job of detection and masking for the next higher-level subsystem. If a fail-fast module correctly reports that its output is questionable, it has actually met its specification, so it has not failed. (Fail-fast modules can still fail: for example, by not noticing their own errors.)
3. Immediately stop dead, thereby hoping to limit propagation of bad values, a technique known as **fail-stop**. Fail-stop subsystems require that the higher-level subsystem take some additional measure to discover the failure: for example, by setting a timer and responding to its expiration. A problem with fail-stop design is that it can be difficult to distinguish a stopped subsystem from one that is merely running more slowly than expected. This problem is particularly acute in asynchronous systems.
4. Do nothing, simply failing without warning. At the interface, the error may have contaminated any or all output values. (Informally called a "crash" or perhaps "fail-thud".)

Another useful distinction is that of transient versus persistent faults. A **transient** fault, also known as a **single-event upset**, is temporary, triggered by some passing external event such as lightning striking a power line or a cosmic ray passing through a chip. It is usually possible to mask an error caused by a transient fault by trying the operation again. An error that is successfully masked by retry is known as a **soft error**. A **persistent** fault continues to produce errors, no matter how many times one retries, and the corresponding errors are called **hard errors**. An **intermittent** fault is a persistent fault that is active only occasionally: for example, when the noise level is higher than usual but still within specifications. Finally, it is sometimes useful to talk about **latency**, which in reliability terminology is the time between when a fault causes an error and when the error is detected or causes the module to fail. Latency can be an important parameter because some error-detection and error-masking mechanisms depend on there being at most a small fixed number of errors—often just one—at a time. If the error latency is large, there may be time for a second error to occur before the first one is detected and masked, in which case masking of the first error may not succeed. Also, a large error latency gives time for the error to propagate and may thus complicate containment.

Using this terminology, an improperly fabricated stuck-at- `ZERO` bit in a memory chip is a persistent fault: whenever the bit should contain a `ONE` the fault is active and the value of the bit is in error; at times when the bit is supposed to contain a `ZERO` , the fault is latent. If the chip is a component of a fault-tolerant memory module, the module design probably includes an error-correction code that prevents that error from turning into a failure of the module. If a passing cosmic ray flips another bit in the same chip, a transient fault has caused that bit also to be in error, but the same error-correction code may still be able to prevent this error from turning into a module failure. On the other hand, if the error-correction code can handle only single-bit errors, the combination of the persistent and the transient fault might lead the module to produce wrong data across its interface, a failure of the module. If someone were then to test the module by storing new data in it and reading it back, the test would probably not reveal a failure because the transient fault does not affect the new data. Because simple input/output testing does not reveal successfully masked errors, a fault tolerant module design should always include some way to report that the module masked an error. If it does not, the user of the module may not realize that persistent errors are accumulating but hidden.

2.2.2: The Fault-Tolerance Design Process

One way to design a reliable system would be to build it entirely of components that are individually so reliable that their chance of failure can be neglected. This technique is known as **fault avoidance**. Unfortunately, it is hard to apply this technique to every component of a large system. In addition, the sheer number of components may defeat the strategy. If all N of the components of a system must work, the probability of any one component failing is p, and component failures are independent of one another, then the probability that the system works is $(1-p)^N$. No matter how small p may be, there is some value of N beyond which this probability becomes too small for the system to be useful.

The alternative is to apply various techniques that are known collectively by the name **fault tolerance**. The remainder of this chapter describes several such techniques that are the elements of an overall design process for building reliable systems from unreliable components. Here is an overview of the **fault-tolerance design process**:

1. Begin to develop a fault-tolerance model, as described in Section 2.4:
   - Identify every potential fault.
   - Estimate the risk of each fault, as described in Section 2.3.
   - Where the risk is too high, design methods to detect the resulting errors.
2. Apply modularity to contain the damage from the high-risk errors.
3. Design and implement procedures that can mask the detected errors, using the techniques described in Section 2.5:
   - Temporal redundancy. Retry the operation, using the same components.
   - Spatial redundancy. Have different components do the operation.
4. Update the fault-tolerance model to account for those improvements.
5. Iterate the design and the model until the probability of untolerated faults is low enough that it is acceptable.
6. Observe the system in the field:
   - Check logs of how many errors the system is successfully masking. (Always keep track of the distance to the edge of the cliff.)
   - Perform postmortems on failures and identify *all* of the reasons for each failure.
7. Use the logs of masked faults and the postmortem reports about failures to revise and improve the fault-tolerance model and reiterate the design.

The fault-tolerance design process includes some subjective steps, for example, deciding that a risk of failure is "unacceptably high" or that the "probability of an untolerated fault is low enough that it is acceptable." It is at these points that different application requirements can lead to radically different approaches to achieving reliability. A personal computer may be designed with no redundant components, the computer system for a small business is likely to make periodic backup copies of most of its data and store the backup copies at another site, and some space-flight guidance systems use five completely redundant computers designed by at least two independent vendors. The decisions required involve trade-offs between the cost of failure and the cost of implementing fault tolerance. These decisions can blend into decisions involving business models and risk management. In some cases it may be appropriate to opt for a nontechnical solution, for example, deliberately accepting an increased risk of failure and covering that risk with insurance.

The fault-tolerance design process can be described as a **safety-net** approach to system design. The safety-net approach involves application of some familiar design principles and also some not previously encountered. It starts with a new design principle: *Be explicit*; get all of the assumptions out on the table. The primary purpose of creating a fault-tolerance model is to expose and document the assumptions and articulate them explicitly. The designer needs to have these assumptions not only for the initial design, but also in order to respond to field reports of unexpected failures. Unexpected failures represent omissions or violations of the assumptions.

Assuming that you won't get it right the first time, the second design principle of the safety-net approach is the familiar *design for iteration*. It is difficult or impossible to anticipate all of the ways that things can go wrong. Moreover, when working with a fast-changing technology it can be hard to estimate probabilities of failure in components and in their organization, especially when the organization is controlled by software. For these reasons, a fault-tolerant design must include feedback about actual error rates, evaluation of that feedback, and update of the design as field experience is gained. These two principles interact: to act on the feedback requires having a fault tolerance model that is explicit about reliability assumptions.

The third design principle of the safety-net approach is the *safety margin principle*. An essential part of a fault-tolerant design is to monitor how often errors are masked. When fault-tolerant systems fail, it is usually not because they had inadequate fault tolerance, but because the number of failures grew unnoticed until the fault tolerance of the design was exceeded. The key requirement is that the system log all failures and that someone pay attention to the logs. The biggest difficulty to overcome in applying this principle is that it is hard to motivate people to expend effort checking something that seems to be working.

The fourth design principle of the safety-net approach came up in the introduction to the study of systems; it shows up here in the instruction to identify all of the causes of each failure: *keep digging*. Complex systems fail for complex reasons. When a failure of a system that is supposed to be reliable does occur, always look beyond the first, obvious cause. It is nearly always the case that there

are actually several contributing causes and that there was something about the mindset of the designer that allowed each of those causes to creep in to the design.

Finally, complexity increases the chances of mistakes, so it is an enemy of reliability. The fifth design principle embodied in the safety-net approach is to *adopt sweeping simplifications*. This principle does not show up explicitly in the description of the fault-tolerance design process, but it will appear several times as we go into more detail.

The safety-net approach is applicable not just to fault-tolerant design. Chapter 5 will show that the safety-net approach is used in an even more rigorous form in designing systems that must protect information from malicious actions.

This page titled 2.2: Faults, Failures, and Fault-Tolerant Design is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

2.3: Measures of Reliability and Failure Tolerance

2.3.1: Availability and Mean Time to Failure

A useful model of a system or a system component, from a reliability point of view, is that it operates correctly for some period of time and then it fails. The **time to failure** (TTF) is thus a measure of interest, and it is something that we would like to be able to predict. If a higher-level module does not mask the failure and the failure is persistent, the system cannot be used until it is repaired, perhaps by replacing the failed component, so we are equally interested in the **time to repair** (TTR). If we observe a system through N run–fail–repair cycles and observe in each cycle i the values of TTF_i and TTR_i, we can calculate the fraction of time it operated properly, a useful measure known as availability:

$$Availability = \frac{\text{time system was running}}{\text{time system should have been running}} = \frac{\sum_{i=1}^{N} TTF_i}{\sum_{i=1}^{N} (TTF_i + TTR_i)} \tag{2.3.1}$$

By separating the denominator of the availability expression into two sums and dividing each by N (the number of observed failures) we obtain two time averages that are frequently reported as operational statistics: the **mean time to failure** (MTTF) and the **mean time to repair** (MTTR):

$$MTTF = \frac{1}{N}\sum_{i=1}^{N} TTF_i \qquad MTTR = \frac{1}{N}\sum_{i=1}^{N} TTR_i \tag{2.3.2}$$

The sum of these two statistics is usually called the **mean time between failures** (MTBF). Thus availability can be variously described as

$$Availability = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR} = \frac{MTBF - MTTR}{MTBF} \tag{2.3.3}$$

In some situations, it is more useful to measure the fraction of time that the system is *not* working, known as its **down time**:

$$\text{Down time} = (1 - Availability) = \frac{MTTR}{MTBF} \tag{2.3.4}$$

One thing that the definition of down time makes clear is that MTTR and MTBF are, in some sense, equally important. One can reduce down time either by reducing MTTR or by increasing MTBF.

Components are often repaired by simply replacing them with new ones. When failed components are discarded rather than fixed and returned to service, it is common to use a slightly different method to measure MTTF. The method is to place a batch of N components in service in different systems (or in what is hoped to be an equivalent test environment), run them until they have all failed, and use the set of failure times as the TTF_i in Equation 2.3.2. This procedure substitutes an ensemble average for the time average. We could use this same procedure on components that are not usually discarded when they fail, in the hope of determining their MTTF more quickly, but we might obtain a different value for the MTTF. Some failure processes do have the property that the ensemble average is the same as the time average (processes with this property are called **ergodic**), but other failure processes do not. For example, the repair itself may cause wear, tear, and disruption to other parts of the system, in which case each successive system failure might on average occur sooner than did the previous one. If that is the case, an MTTF calculated from an ensemble-average measurement might be too optimistic.

As we have defined them, availability, MTTF, MTTR, and MTBF are backward-looking measures. They are used for two distinct purposes: (1) for evaluating how the system is doing (compared, for example, with predictions made when the system was designed) and (2) for predicting how the system will behave in the future. The first purpose is concrete and well defined. The second requires that one take on faith that samples from the past provide an adequate predictor of the future, which can be a risky assumption. There are other problems associated with these measures. While MTTR can usually be measured in the field, the more reliable a component or system the longer it takes to evaluate its MTTF, so that measure is often not directly available. Instead, it is common to use and measure proxies to estimate its value. The quality of the resulting estimate of availability then depends on the quality of the proxy.

A typical 3.5-inch magnetic disk comes with a reliability specification of 300,000 hours "MTTF", which is about 34 years. Since the company quoting this number has probably not been in business that long, it is apparent that whatever they are calling "MTTF" is not the same as either the time-average or the ensemble-average MTTF that we just defined. It is actually a quite different statistic, which is why we put quotes around its name. Sometimes this "MTTF" is a theoretical prediction obtained by modeling the ways that the components of the disk might be expected to fail and calculating an expected time to failure.

A more likely possibility is that the manufacturer measured this "MTTF" by running an array of disks simultaneously for a much shorter time and counting the number of failures. For example, suppose the manufacturer ran 1,000 disks for 3,000 hours (about four months) each, and during that time 10 of the disks failed. The observed failure rate of this sample is 1 failure for every 300,000 hours of operation. The next step is to invert the failure rate to obtain 300,000 hours of operation per failure and then quote this number as the "MTTF". But the relation between this sample observation of failure rate and the real MTTF is problematic. If the failure process were memoryless (meaning that the failure rate is independent of time; Section 2.3.2, below, explores this idea more thoroughly), we would have the special case in which the MTTF really is the inverse of the failure rate. A good clue that the disk failure process is not memoryless is that the disk specification may also mention an "expected operational lifetime" of only 5 years. That statistic is probably the real MTTF—though even that may be a prediction based on modeling rather than a measured ensemble average. An appropriate re-interpretation of the 34-year "MTTF" statistic is to invert it and identify the result as a short-term failure rate that applies only within the expected operational lifetime. The paragraph discussing Equation 2.3.9 in Section 2.3.2, describes a fallacy that sometimes leads to miscalculation of statistics such as the MTTF.

Magnetic disks, light bulbs, and many other components exhibit a time-varying statistical failure rate known as a **bathtub curve**, illustrated in Figure 2.3.1 and defined more carefully in Section 2.3.2, below. When components come off the production line, a certain fraction fail almost immediately because of gross manufacturing defects. Those components that survive this initial period usually run for a long time with a relatively uniform failure rate. Eventually, accumulated wear and tear cause the failure rate to increase again, often quite rapidly, producing a failure rate plot that resembles the shape of a bathtub.

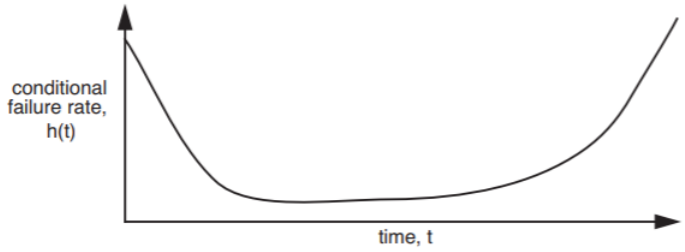


Figure 2.3.1: A bathtub curve, showing how the conditional failure rate of a component changes with time.

Several other suggestive and colorful terms describe these phenomena. Components that fail early are said to be subject to **infant mortality**, and those that fail near the end of their expected lifetimes are said to **burn out**. Manufacturers sometimes **burn in** such components by running them for a while before shipping, with the intent of identifying and discarding the ones that would otherwise fail immediately upon being placed in service. When a vendor quotes an "expected operational lifetime," it is probably the mean time to failure of those components that survive burn in, while the much larger "MTTF" number is probably the inverse of the observed failure rate at the lowest point of the bathtub. (The published numbers also sometimes depend on the outcome of a debate between the legal department and the marketing department, but that gets us into a different topic.) A chip manufacturer describes the fraction of components that survive the burn-in period as the **yield** of the production line. Component manufacturers usually exhibit a phenomenon known informally as a **learning curve**, which simply means that the first components coming out of a new production line tend to have more failures than later ones. The reason is that manufacturers *design for iteration*: upon seeing and analyzing failures in the early production batches, the production line designer figures out how to refine the manufacturing process to reduce the infant mortality rate.

One job of the system designer is to exploit the nonuniform failure rates predicted by the bathtub and learning curves. For example, a conservative designer exploits the learning curve by avoiding the latest generation of hard disks in favor of slightly older designs that have accumulated more field experience. One can usually rely on other designers who may be concerned more about cost or performance than availability to shake out the bugs in the newest generation of disks.

The 34-year "MTTF" disk drive specification may seem like public relations puffery in the face of the specification of a 5-year expected operational lifetime, but these two numbers actually are useful as a measure of the nonuniformity of the failure rate. This nonuniformity is also susceptible to exploitation, depending on the operation plan. If the operation plan puts the component in a

system such as a satellite, in which it will run until it fails, the designer would base system availability and reliability estimates on the 5-year figure. On the other hand, the designer of a ground-based storage system, mindful that the 5-year operational lifetime identifies the point where the conditional failure rate starts to climb rapidly at the far end of the bathtub curve, might include a plan to replace perfectly good hard disks before burn-out begins to dominate the failure rate—in this case, perhaps every 3 years. Since one can arrange to do scheduled replacement at convenient times—for example, when the system is down for another reason, or perhaps even without bringing the system down—the designer can minimize the effect on system availability. The manufacturer's 34-year "MTTF", which is probably the inverse of the observed failure rate at the lowest point of the bathtub curve, can then be used as an estimate of the expected rate of unplanned replacements, although experience suggests that this specification may be a bit optimistic. Scheduled replacements are an example of preventive maintenance, which is active intervention intended to increase the mean time to failure of a module or system and thus improve availability.

For some components, observed failure rates are so low that MTTF is estimated by **accelerated aging**. This technique involves making an educated guess about what the dominant underlying cause of failure will be and then amplifying that cause. For example, it is conjectured that failures in recordable Compact Disks are heat-related. A typical test scenario is to store batches of recorded CDs at various elevated temperatures for several months, periodically bringing them out to test them and count how many have failed. One then plots these failure rates versus temperature and extrapolates to estimate what the failure rate would have been at room temperature. Again making the assumption that the failure process is memoryless, that failure rate is then inverted to produce an MTTF. Published MTTFs of 100 years or more have been obtained this way. If the dominant fault mechanism turns out to be something else (such as bacteria munching on the plastic coating) or if after 50 years the failure process turns out not to be memoryless after all, an estimate from an accelerated aging study may be far wide of the mark. A designer must use such estimates with caution and understanding of the assumptions that went into them.

Availability is sometimes discussed by counting the number of nines in the numerical representation of the availability measure. Thus a system that is up and running 99.9% of the time is said to have 3-nines availability. Measuring by nines is often used in marketing because it sounds impressive. A more meaningful number is usually obtained by calculating the corresponding down time. A 3-nines system can be down nearly 1.5 minutes per day or 8 hours per year, a 5-nines system 5 minutes per year, and a 7-nines system only 3 seconds per year. Another problem with measuring by nines is that it only provides information about availability, not about MTTF. One 3-nines system may have a brief failure every day, while a different 3-nines system may have a single eight-hour outage once a year. Depending on the application, the difference between those two systems could be important. Any single measure should always be suspect.

Finally, availability can be a more fine-grained concept. Some systems are designed so that when they fail, some functions (for example, the ability to read data) remain available, while others (the ability to make changes to the data) are not. Systems that continue to provide partial service in the face of failure are called **fail-soft**, a concept defined more carefully in Section 2.4.

2.3.2: Reliability Functions

The bathtub curve expresses the conditional failure rate $h(t)$ of a module, defined to be the probability that the module fails between time t and time $t+dt$, given that the component is still working at time t. The conditional failure rate is only one of several closely related ways of describing the failure characteristics of a component, module, or system. The *reliability*, R, of a module is defined to be

$$R(t) = Pr(\text{the module has not yet failed at time } t, \text{ given that the module was operating at time } 0) \qquad (2.3.5)$$

and the unconditional failure rate $f(t)$ is defined to be

$$f(t) = Pr(\text{module fails between } t \text{ and } t+dt) \qquad (2.3.6)$$

(The bathtub curve and these two reliability functions are three ways of presenting the same information. If you are rusty on probability, a brief reminder of how they are related appears in Sidebar 2.3.1) Once $f(t)$ is at hand, one can directly calculate the MTTF:

$$MTTF = \int_0^\infty t \cdot f(t)\, dt \qquad (2.3.7)$$

One must keep in mind that this MTTF is predicted from the failure rate function $f(t)$, in contrast to the MTTF of Equation 2.3.2, which is the result of a field measurement. The two MTTFs will be the same only if the failure model embodied in $f(t)$ is accurate.

Sidebar 2.3.1

Reliability functions

The failure rate function, the reliability function, and the bathtub curve (which in probability texts is called the **conditional failure rate function**, and which in operations research texts is called the **hazard function**) are actually three mathematically related ways of describing the same information. The failure rate function, $f(t)$ as defined in Equation 2.3.6, is a probability density function, which is everywhere non-negative and whose integral over all time is 1. Integrating the failure rate function from the time the component was created (conventionally taken to be $t = 0$) to the present time yields

$$F(t) = \int_0^t f(t)\, dt$$

$F(t)$ is the cumulative probability that the component has failed by time *t*. The cumulative probability that the component has not failed is the probability that it is still operating at time t given that it was operating at time 0, which is exactly the definition of the reliability function, $R(t)$. That is,

$$R(t) = 1 - F(t)$$

The bathtub curve of Figure 2.3.1reports the conditional probability $h(t)$ that a failure occurs between t and $t + dt$, given that the component was operating at time *t*. By the definition of conditional probability, the conditional failure rate function is thus

$$h(t) = \frac{f(t)}{R(t)}$$

Some components exhibit relatively uniform failure rates, at least for the lifetime of the system of which they are a part. For these components the conditional failure rate, rather than resembling a bathtub, is a straight horizontal line, and the reliability function becomes a simple declining exponential:

$$R(t) = e^{-\left(\frac{t}{MTTF}\right)} \tag{2.3.8}$$

This reliability function is said to be **memoryless**, which simply means that the conditional failure rate is independent of how long the component has been operating. Memoryless failure processes have the nice property that the conditional failure rate is the inverse of the MTTF.

Unfortunately, as we saw in the case of the disks with the 34-year "MTTF", this property is sometimes misappropriated to quote an MTTF for a component whose conditional failure rate does change with time. This misappropriation starts with a fallacy: an assumption that the MTTF, as defined in Equation 2.3.7, can be calculated by inverting the measured failure rate. The fallacy arises because in general,

$$E(1/t) \neq 1/E(t) \tag{2.3.9}$$

That is, the expected value of the inverse is *not* equal to the inverse of the expected value, except in certain special cases. The important special case in which they *are* equal is the memoryless distribution of Equation 2.3.8. When a random process is memoryless, calculations and measurements are so much simpler that designers sometimes forget that the same simplicity does not apply everywhere.

Just as availability is sometimes expressed in an oversimplified way by counting the number of nines in its numerical representation, reliability in component manufacturing is sometimes expressed in an oversimplified way by counting standard deviations in the observed distribution of some component parameter, such as the maximum propagation time of a gate. The usual symbol for standard deviation is the Greek letter σ (sigma), and a normal distribution has a standard deviation of 1.0, so saying that a component has "4.5 σ reliability" is a shorthand way of saying that the production line controls variations in that parameter well

enough that the specified tolerance is 4.5 standard deviations away from the mean value, as illustrated in Figure 2.3.2. Suppose, for example, that a production line is manufacturing gates that are specified to have a mean propagation time of 10 nanoseconds and a maximum propagation time of 11.8 nanoseconds with 4.5 σ reliability. The difference between the mean and the maximum, 1.8 nanoseconds, is the tolerance. For that tolerance to be 4.5 σ, σ would have to be no more than 0.4 nanoseconds. To meet the specification, the production line designer would measure the actual propagation times of production line samples and, if the observed variance is greater than 0.4 ns, look for ways to reduce the variance to that level.

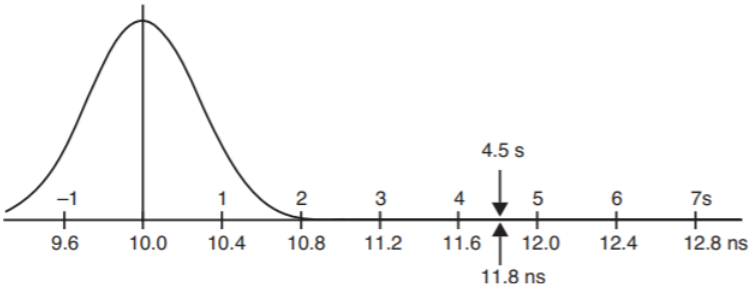


Figure 2.3.2: The normal probability density function applied to production of gates that are specified to have mean propagation time of 10 nanoseconds and maximum propagation time of 11.8 nanoseconds. The upper numbers on the horizontal axis measure the distance from the mean in units of the standard deviation, σ. The lower numbers depict the corresponding propagation times. The integral of the tail from 4.5 σ to ∞ is so small that it is not visible in this figure.

Another way of interpreting "4.5 σ reliability" is to calculate the expected fraction of components that are outside the specified tolerance. That fraction is the integral of one tail of the normal distribution from 4.5 σ to ∞, which is about 3.4×10^{-6} , so in our example no more than 3.4 out of each million gates manufactured would have delays greater than 11.8 nanoseconds. Unfortunately, this measure describes only the failure rate of the production line; it does not say anything about the failure rate of the component after it is installed in a system.

A currently popular quality control method, known as "Six Sigma", is an application of two of our design principles to the manufacturing process. The idea is to use measurement, feedback, and iteration (*design for iteration*: "you won't get it right the first time") to reduce the variance (*the robustness principle*: "be strict on outputs") of production-line manufacturing. The "Six Sigma" label is somewhat misleading because in the application of the method, the number 6 is allocated to deal with two quite different effects. The method sets a target of controlling the production line variance to the level of 4.5 σ, just as in the gate example of Figure 2.3.2. The remaining 1.5 σ is the amount that the mean output value is allowed to drift away from its original specification over the life of the production line. So even though the production line may start 6 σ away from the tolerance limit, after it has been operating for a while one may find that the failure rate has drifted upward to the same 3.4 in a million calculated for the 4.5 σ case.

In manufacturing quality control literature, these applications of the two design principles are known as **Taguchi methods**, after their popularizer, Genichi Taguchi.

2.3.3: Measuring Fault Tolerance

It is sometimes useful to have a quantitative measure of the fault tolerance of a system. One common measure, sometimes called the **failure tolerance**, is the number of failures of its components that a system can tolerate without itself failing. Although this label could be ambiguous, it is usually clear from context that a measure is being discussed. Thus a memory system that includes single-error correction (Section 2.5 describes how error correction works) has a failure tolerance of one bit.

When a failure occurs, the remaining failure tolerance of the system goes down. The remaining failure tolerance is an important thing to monitor during operation of the system because it shows how close the system as a whole is to failure. One of the most common system design mistakes is to add fault tolerance but not include any monitoring to see how much of the fault tolerance has been used up, thus ignoring the *safety margin principle*. When systems that are nominally fault-tolerant do fail, later analysis invariably discloses that there were several failures that the system successfully masked but that somehow were never reported and thus were never repaired. Eventually, the total number of failures exceeded the designed failure tolerance of the system.

Failure tolerance is actually a single number in only the simplest situations. Sometimes it is better described as a vector, or even as a matrix showing the specific combinations of different kinds of failures that the system is designed to tolerate. For example, an electric power company might say that it can tolerate the failure of up to 15% of its generating capacity, at the same time as the downing of up to two of its main transmission lines.

This page titled 2.3: Measures of Reliability and Failure Tolerance is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

2.4: Tolerating Active Faults

2.4.1: Responding to Active Faults

In dealing with active faults, the designer of a module can provide one of several responses:

- **Do nothing.** The error becomes a failure of the module, and the larger system or subsystem of which it is a component inherits the responsibilities both of discovering and of handling the problem. The designer of the larger subsystem then must choose which of these responses to provide. In a system with several layers of modules, failures may be passed up through more than one layer before being discovered and handled. As the number of do-nothing layers increases, containment generally becomes more and more difficult.
- **Be fail-fast.** The module reports at its interface that something has gone wrong. This response also turns the problem over to the designer of the next higher-level system, but in a more graceful way. Example: when an Ethernet transceiver detects a collision on a frame it is sending, it stops sending as quickly as possible, broadcasts a brief jamming signal to ensure that all network participants quickly realize that there was a collision, and it reports the collision to the next higher level, usually a hardware module of which the transceiver is a component, so that the higher level can consider resending that frame.
- **Be fail-safe.** The module transforms any value or values that are incorrect to values that are known to be acceptable, even if not right or optimal. An example is a digital traffic light controller that, when it detects a failure in its sequencer, switches to a blinking red light in all directions. Chapter 5 discusses systems that provide security. In the event of a failure in a secure system, the safest thing to do is usually to block all access. A fail-safe module designed to do that is said to be **fail-secure**.
- **Be fail-soft.** The system continues to operate correctly with respect to some predictably degraded subset of its specifications, perhaps with some features missing or with lower performance. For example, an airplane with three engines can continue to fly safely, albeit more slowly and with less maneuverability, if one engine fails. A file system that is partitioned into five parts, stored on five different small hard disks, can continue to provide access to 80% of the data when one of the disks fails, in contrast to a file system that employs a single disk five times as large.
- **Mask the error.** Any value or values that are incorrect are made right and the module meets its specification as if the error had not occurred.

We will concentrate on masking errors because the techniques used for that purpose can be applied, often in simpler form, to achieving a fail-fast, fail-safe, or fail-soft system.

As a general rule, one can design algorithms and procedures to cope only with specific, anticipated faults. Further, an algorithm or procedure can be expected to cope only with faults that are actually detected. In most cases, the only workable way to detect a fault is by noticing an incorrect value or control signal; that is, by detecting an error. Thus when trying to determine if a system design has adequate fault tolerance, it is helpful to classify errors as follows:

- A **detectable error** is one that can be detected reliably. If a detection procedure is in place and the error occurs, the system discovers it with near certainty and it becomes a **detected** error.
- A **maskable error** is one for which it is possible to devise a procedure to recover correctness. If a masking procedure is in place and the error occurs, is detected, and is masked, the error is said to be **tolerated**.
- Conversely, an **untolerated** error is one that is undetectable, undetected, unmaskable, or unmasked.

An untolerated error usually leads to a failure of the system. ("Usually," because we could get lucky and still produce a correct output, either because the error values didn't actually matter under the current conditions, or some measure intended to mask a different error incidentally masks this one, too.) This classification of errors is illustrated in Figure 2.4.1.

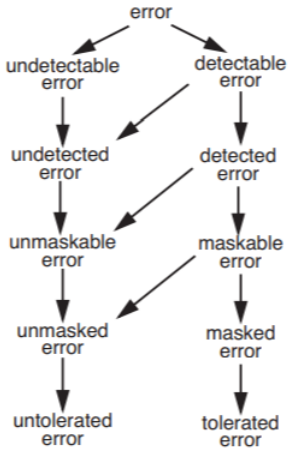


Figure 2.4.1: Classification of errors. Arrows lead from a category to mutually exclusive subcategories. For example, unmasked errors include both unmaskable errors and maskable errors that the designer decides not to mask.

A subtle consequence of the concept of a maskable error is that there must be a well-defined boundary around that part of the system state that might be in error. The masking procedure must restore all of that erroneous state to correctness, using information that has not been corrupted by the error. The real meaning of *detectable*, then, is that the error is discovered before its consequences have propagated beyond some specified boundary. The designer usually chooses this boundary to coincide with that of some module, and designs that module to be fail-fast (that is, it detects and reports its own errors). The system of which the module is a component then becomes responsible for masking the failure of the module.

2.4.2: Fault Tolerance Models

The distinctions among detectable, detected, maskable, and tolerated errors allow us to specify for a system a **fault tolerance model**, one of the components of the fault tolerance design process described in Section 2.2.2, as follows:

1. Analyze the system and categorize possible error events into those that can be reliably detected and those that cannot. At this stage, detectable or not, all errors are untolerated.
2. For each undetectable error, evaluate the probability of its occurrence. If that probability is not negligible, modify the system design in whatever way necessary to make the error reliably detectable.
3. For each detectable error, implement a detection procedure and reclassify the module in which it is detected as fail-fast.
4. For each detectable error try to devise a way of masking it. If there is a way, reclassify this error as a maskable error.
5. For each maskable error, evaluate its probability of occurrence, the cost of failure, and the cost of the masking method devised in the previous step. If the evaluation indicates it is worthwhile, implement the masking method and reclassify this error as a tolerated error.

When finished developing such a model, the designer should have a useful fault tolerance specification for the system. Some errors, which have negligible probability of occurrence or for which a masking measure would be too expensive, are identified as untolerated. When those errors occur the system fails, leaving its users to cope with the result. Other errors have specified recovery algorithms, and when those occur the system should continue to run correctly. A review of the system recovery strategy can now focus separately on two distinct questions:

- Is the designer's list of potential error events complete, and is the assessment of the probability of each error realistic?
- Is the designer's set of algorithms, procedures, and implementations that are supposed to detect and mask the anticipated errors complete and correct?

These two questions are different. The first is a question of models of the real world. It addresses an issue of experience and judgment about real-world probabilities and whether all real-world modes of failure have been discovered or some have gone unnoticed. Two different engineers, with different real-world experiences, may reasonably disagree on such judgments—they may have different models of the real world. The evaluation of modes of failure and of probabilities is a point at which a designer may easily go astray because such judgments must be based not on theory but on experience in the field, either personally acquired by the designer or learned from the experience of others. A new technology, or an old technology placed in a new environment, is

likely to create surprises. A wrong judgment can lead to wasted effort devising detection and masking algorithms that will rarely be invoked rather than the ones that are really needed. On the other hand, if the needed experience is not available, all is not lost: the iteration part of the design process is explicitly intended to provide that experience.

The second question is more abstract and also more absolutely answerable, in that an argument for correctness (unless it is hopelessly complicated) or a counterexample to that argument should be something that everyone can agree on. In system design, it is helpful to follow design procedures that distinctly separate these classes of questions. When someone questions a reliability feature, the designer can first ask, “Are you questioning the correctness of my recovery algorithm or are you questioning my model of what may fail?” and thereby properly focus the discussion or argument.

Creating a fault tolerance model also lays the groundwork for the iteration part of the fault tolerance design process. If a system in the field begins to fail more often than expected, or completely unexpected failures occur, analysis of those failures can be compared with the fault tolerance model to discover what has gone wrong. By again asking the two questions marked with bullets above, the model allows the designer to distinguish between, on the one hand, failure probability predictions being proven wrong by field experience, and on the other, inadequate or misimplemented masking procedures. With this information, the designer can work out appropriate adjustments to the model and the corresponding changes needed for the system.

Iteration and review of fault tolerance models is also important to keep them up to date in the light of technology changes. For example, the Network File System discussed in Section 1.10.1 was first deployed using a local area network, where packet loss errors are rare and may even be masked by the link layer. When later users deployed it on larger networks, where lost packets are more common, it became necessary to revise its fault tolerance model and add additional error detection in the form of end-to-end checksums. The processor time required to calculate and check those checksums caused some performance loss, which is why its designers did not originally include checksums. But loss of data integrity outweighed loss of performance and the designers reversed the trade-off.

To illustrate, an example of a fault tolerance model applied to a popular kind of memory devices, RAM, appears in Section 2.8. This fault tolerance model employs error detection and masking techniques that are described first in Section 2.5, so the reader may prefer to delay detailed study of that section until completing Section 2.5.

This page titled 2.4: Tolerating Active Faults is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

2.5: Systematically Applying Redundancy

The designer of an analog system typically masks small errors by specifying design tolerances known as **margins**, which are amounts by which the specification is better than necessary for correct operation under normal conditions. In contrast, the designer of a digital system both detects and masks errors of all kinds by adding redundancy, either in time or in space. When an error is thought to be transient, as when a packet is lost in a data communication network, one method of masking is to resend it, an example of redundancy in time. When an error is likely to be persistent, as in a failure in reading bits from the surface of a disk, the usual method of masking is with spatial redundancy, having another component provide another copy of the information or control signal. Redundancy can be applied either in cleverly small quantities or by brute force, and both techniques may be used in different parts of the same system.

2.5.1: Coding: Incremental Redundancy

The most common form of incremental redundancy, known as **forward error correction**, consists of clever coding of data values. With data that has not been encoded to tolerate errors, a change in the value of one bit may transform one legitimate data value into another legitimate data value. Encoding for errors involves choosing as the representation of legitimate data values only some of the total number of possible bit patterns, being careful that the patterns chosen for legitimate data values all have the property that to transform any one of them to any other, more than one bit must change. The smallest number of bits that must change to transform one legitimate pattern into another is known as the **Hamming distance** between those two patterns. The Hamming distance is named after Richard Hamming, who first investigated this class of codes. Thus the patterns

```
100101
000111
```

have a Hamming distance of 2 because the upper pattern can be transformed into the lower pattern by flipping the values of two bits, the first bit and the fifth bit. Data fields that have not been coded for errors might have a Hamming distance as small as 1. Codes that can detect or correct errors have a minimum Hamming distance between any two legitimate data patterns of 2 or more. The Hamming distance of a code is the minimum Hamming distance between any pair of legitimate patterns of the code. One can calculate the Hamming distance between two patterns, A and B, by counting the number of `ONES` in $A \oplus B$, where $\oplus$ is the exclusive `OR` (`XOR`) operator.

Suppose we create an encoding in which the Hamming distance between *every* pair of legitimate data patterns is 2. Then, if one bit changes accidentally, since no legitimate data item can have that pattern, we can detect that something went wrong. However, it is not possible to figure out what the original data pattern was. If the two patterns above were two members of the code and the first bit of the upper pattern were flipped from `ONE` to `ZERO` , there is no way to tell that the result, `000101` , is not the result of flipping the fifth bit of the lower pattern.

Next, suppose that we instead create an encoding in which the Hamming distance of the code is 3 or more. Here are two patterns from such a code; bits 1, 2, and 5 are different:

```
100101
010111
```

Now, a one-bit change will always transform a legitimate data pattern into an incorrect data pattern that is still at least 2 bits distant from any other legitimate pattern but only 1 bit distant from the original pattern. A decoder that receives a pattern with a one-bit error can inspect the Hamming distances between the received pattern and nearby legitimate patterns and, by choosing the nearest legitimate pattern, correct the error. If 2 bits change, this error-correction procedure will still identify a corrected data value, but it will choose the wrong one. If we expect 2-bit errors to happen often, we could choose the code patterns so that the Hamming distance is 4, in which case the code can correct 1-bit errors and detect 2-bit errors. But a 3-bit error would look just like a 1-bit error in some other code pattern, so it would decode to a wrong value. More generally, if the Hamming distance of a code is d, a little analysis reveals that one can detect $d-1$ errors and correct $\lfloor (d-1)/2 \rfloor$ errors. The reason that this form of redundancy is named "forward" error correction is that the creator of the data performs the coding before storing or transmitting it, and anyone can later decode the data without appealing to the creator. (Chapter 1 described the technique of asking the sender of a lost frame, packet, or message to retransmit it. That technique goes by the name of **backward error correction**.)

The systematic construction of forward error-detection and error-correction codes is a large field of study, which we do not intend to explore. However, two specific examples of commonly encountered codes are worth examining. The first example is a simple

parity check on a 2-bit value, in which the parity bit is the XOR of the 2 data bits. The coded pattern is 3 bits long, so there are 23 = 8 possible patterns for this 3-bit quantity, only 4 of which represent legitimate data. As illustrated in Figure 2.5.1, the 4 "correct" patterns have the property that changing any single bit transforms the word into one of the 4 illegal patterns. To transform the coded quantity into another legal pattern, at least 2 bits must change (in other words, the Hamming distance of this code is 2). The conclusion is that a simple parity check can detect any single error, but it doesn't have enough information to correct errors.

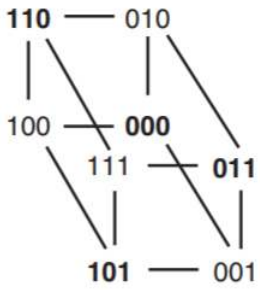


Figure 2.5.1: Patterns for a simple parity-check code. Each line connects patterns that differ in only one bit; **bold-face** patterns are the legitimate ones.

The second example, in Figure 2.5.2, shows a forward error-correction code that can correct 1-bit errors in a 4-bit data value, by encoding the 4 bits into 7-bit words. In this code, bits P_7, P_6, P_5, and P_3 carry the data, while bits P_4, P_2, and P_1 are calculated from the data bits. (This out-of-order numbering scheme creates a multidimensional binary coordinate system with a use that will be evident in a moment.) We could analyze this code to determine its Hamming distance, but we can also observe that three extra bits can carry exactly enough information to distinguish 8 cases: no error, an error in bit 1, an error in bit 2, … or an error in bit 7. Thus, it is not surprising that an error-correction code can be created. This code calculates bits P1, P2, and P4 as follows:

$$P_1 = P_7 \oplus P_5 \oplus P_3$$

$$P_2 = P_7 \oplus P_6 \oplus P_3$$

$$P_4 = P_7 \oplus P_6 \oplus P_5$$

Now, suppose that the array of bits P_1 through P_7 is sent across a network and noise causes bit P_5 to flip. If the recipient recalculates P_1, P_2, and P_4, the recalculated values of P_1 and P_4 will be different from the received bits P_1 and P_4. The recipient then writes $P_4P_2P_1$ in order, representing the troubled bits as ONE 's and untroubled bits as ZERO 's, and notices that their binary value is 101_2 = 5 , the position of the flipped bit. In this code, whenever there is a one-bit error, the troubled parity bits directly identify the bit to correct. (That was the reason for the out-of-order bit-numbering scheme, which created a 3-dimensional coordinate system for locating an erroneous bit.)

Choose P_1 so XOR of every other bit ($P_7 \oplus P_5 \oplus P_3 \oplus P_1$) is 0
Choose P_2 so XOR of every other pair ($P_7 \oplus P_6 \oplus P_3 \oplus P_2$) is 0
Choose P_4 so XOR of every other four ($P_7 \oplus P_6 \oplus P_5 \oplus P_4$) is 0

bit	P_7	P_6	P_5	P_4	P_3	P_2	P_1
	⊕		⊕		⊕		⊕
	⊕	⊕			⊕	⊕	
	⊕	⊕	⊕	⊕			

Figure 2.5.2: A single-error-correction code. In the table, the symbol ⊕ marks the bits that participate in the calculation of one of the redundant bits.The payload bits are P_7, P_6, P_5, and P_3, and the redundant bits are P_4, P_2, and P_1. The "every other" notes describe a 3-dimensional coordinate system that can locate an erroneous bit.

The use of 3 check bits for 4 data bits suggests that an error-correction code may not be efficient, but in fact the apparent inefficiency of this example is only because it is so small. Extending the same reasoning, one can, for example, provide single-error correction for 56 data bits using 7 check bits in a 63-bit code word.

In both of these examples of coding, the assumed threat to integrity is that an unidentified bit out of a group may be in error. Forward error correction can also be effective against other threats. A different threat, called erasure, is also common in digital systems. An erasure occurs when the value of a particular, identified bit of a group is unintelligible or perhaps even completely

missing. Since we know which bit is in question, the simple parity-check code, in which the parity bit is the XOR of the other bits, becomes a forward error correction code. The unavailable bit can be reconstructed simply by calculating the XOR of the unerased bits. Returning to the example of Figure 2.5.1, if we find a pattern in which the first and last bits have values 0 and 1 respectively, but the middle bit is illegible, the only possibilities are 001 and **011**. Since 001 is not a legitimate code pattern, the original pattern must have been **011**. The simple parity check allows correction of only a single erasure. If there is a threat of multiple erasures, a more complex coding scheme is needed. Suppose, for example, we have 4 bits to protect, and they are coded as in Figure 2.5.2. In that case, if as many as 3 bits are erased, the remaining 4 bits are sufficient to reconstruct the values of the 3 that are missing.

Since erasure, in the form of lost packets, is a threat in a best-effort packet network, this same scheme of forward error correction is applicable. One might, for example, send four numbered, identical-length packets of data followed by a parity packet that contains as its payload the bit-by-bit XOR of the payloads of the previous four. (That is, the first bit of the parity packet is the XOR of the first bit of each of the other four packets; the second bits are treated similarly, etc.) Although the parity packet adds 25% to the network load, as long as any four of the five packets make it through, the receiving side can reconstruct all of the payload data perfectly without having to ask for a retransmission. If the network is so unreliable that more than one packet out of five typically gets lost, then one might send seven packets, of which four contain useful data and the remaining three are calculated using the formulas of Figure 2.5.2. (Using the numbering scheme of that figure, the payload of packet 4, for example, would consist of the XOR of the payloads of packets 7, 6, and 5.) Now, if any four of the seven packets make it through, the receiving end can reconstruct the data.

Forward error correction is especially useful in broadcast protocols, where the existence of a large number of recipients, each of which may miss different frames, packets, or stream segments, makes the alternative of backward error correction by requesting retransmission unattractive. Forward error correction is also useful when controlling jitter in stream transmission because it eliminates the round-trip delay that would be required in requesting retransmission of missing stream segments. Finally, forward error correction is usually the only way to control errors when communication is one-way or round-trip delays are so long that requesting retransmission is impractical: for example, when communicating with a deep-space probe. On the other hand, using forward error correction to replace lost packets may have the side effect of interfering with congestion control techniques in which an overloaded packet forwarder tries to signal the sender to slow down by discarding an occasional packet.

Another application of forward error correction to counter erasure is in storing data on magnetic disks. The threat in this case is that an entire disk drive may fail, for example because of a disk head crash. Assuming that the failure occurs long after the data was originally written, this example illustrates one-way communication in which backward error correction (asking the original writer to write the data again) is not usually an option. One response is to use a RAID array in a configuration known as RAID 4. In this configuration, one might use an array of five disks, with four of the disks containing application data and each sector of the fifth disk containing the bit-by-bit XOR of the corresponding sectors of the first four. If any of the five disks fails, its identity will quickly be discovered because disks are usually designed to be fail-fast and report failures at their interface. After replacing the failed disk, one can restore its contents by reading the other four disks and calculating, sector by sector, the XOR of their data (see Exercise 2.10.9). To maintain this strategy, whenever anyone updates a data sector, the RAID 4 system must also update the corresponding sector of the parity disk, as shown in Figure 2.5.3. That figure makes it apparent that, in RAID 4, forward error correction has an identifiable read and write performance cost, in addition to the obvious increase in the amount of disk space used. Since loss of data can be devastating, there is considerable interest in RAID, and much ingenuity has been devoted to devising ways of minimizing the performance penalty.

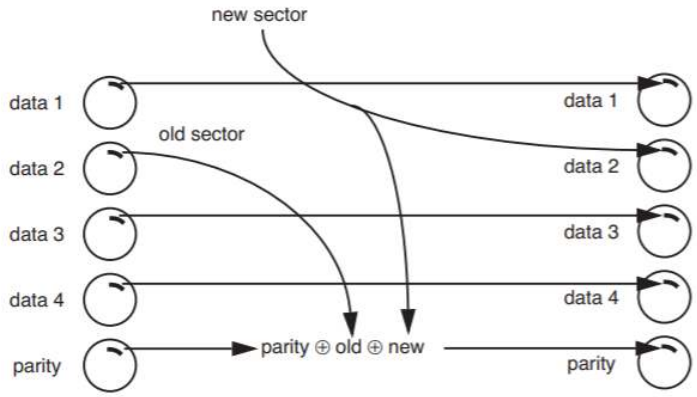


Figure 2.5.3: Update of a sector on disk 2 of a five-disk RAID 4 system. The old parity sector contains parity $\leftarrow$ data 1 $\oplus$ data 2 $\oplus$ data 3 $\oplus$ data 4 . To construct a new parity sector that includes the new data 2, one could read the corresponding sectors of data 1, data 3, and data 4 and perform three more XOR 's. But a faster way is to read just the old parity sector and the old data 2 sector and compute the new parity sector as new parity $\leftarrow$ old parity $\oplus$ old data $\oplus$ new data 2 .

Although it is an important and widely used technique, successfully applying incremental redundancy to achieve error detection and correction is harder than one might expect. The first case study of Section 2.9 provides several useful lessons on this point.

In addition, there are some situations where incremental redundancy does not seem to be applicable. For example, there have been efforts to devise error-correction codes for numerical values with the property that the coding is preserved when the values are processed by an adder or a multiplier. While it is not too hard to invent schemes that allow a limited form of error detection (for example, one can verify that residues are consistent, using analogues of casting out nines, which schoolchildren use to check their arithmetic), these efforts have not yet led to any generally applicable techniques. The only scheme that has been found to systematically protect data during arithmetic processing is massive redundancy, which is our next topic.

2.5.2: Replication: Massive Redundancy

In designing a bridge or a skyscraper, a civil engineer masks uncertainties in the strength of materials and other parameters by specifying components that are 5 or 10 times as strong as minimally required. The method is heavy-handed, but simple and effective.

The corresponding way of building a reliable system out of unreliable discrete components is to acquire multiple copies of each component. Identical multiple copies are called **replicas**, and the technique is called **replication**. There is more to it than just making copies: one must also devise a plan to arrange or interconnect the replicas so that a failure in one replica is automatically masked with the help of the ones that don't fail. For example, if one is concerned about the possibility that a diode may fail by either shorting out or creating an open circuit, one can set up a network of four diodes as in Figure 2.5.4, creating what we might call a "superdiode". This interconnection scheme, known as a **quad component**, was developed by Claude E. Shannon and Edward F. Moore in the 1950s as a way of increasing the reliability of relays in telephone systems. It can also be used with resistors and capacitors in circuits that can tolerate a modest range of component values. This particular superdiode can tolerate a single short circuit *and* a single open circuit in any two component diodes, and it can also tolerate certain other multiple failures, such as open circuits in both upper diodes plus a short circuit in one of the lower diodes. If the bridging connection of the figure is added, the superdiode can tolerate additional multiple open-circuit failures (such as one upper diode and one lower diode), but it will be less tolerant of certain short-circuit failures (such as one left diode and one right diode).

A serious problem with this superdiode is that it masks failures silently. There is no easy way to determine how much failure tolerance remains in the system.

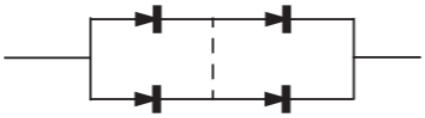


Figure 2.5.4: A quad-component superdiode. The dotted line represents an optional bridging connection, which allows the superdiode to tolerate a different set of failures, as described in the text.

2.5.3: Voting

Although there have been attempts to extend quad-component methods to digital logic, the intricacy of the required interconnections grows much too rapidly. Fortunately, there is a systematic alternative that takes advantage of the static discipline and level regeneration that are inherent properties of digital logic. In addition, it has the nice feature that it can be applied at any level of module, from a single gate on up to an entire computer. The technique is to substitute in place of a single module a set of replicas of that same module, all operating in parallel with the same inputs, and compare their outputs with a device known as a voter. This basic strategy is called ***N*-modular redundancy**, or NMR. When N has the value 3 the strategy is called **triple-modular redundancy**, abbreviated TMR. When other values are used for N the strategy is named by replacing the N of NMR

with the number, as in 5MR. The combination of N replicas of some module and the voting system is sometimes called a **supermodule**. Several different schemes exist for interconnection and voting, only a few of which we explore here.

The simplest scheme, called **fail-vote**, consists of NMR with a majority voter. One assembles N replicas of the module and a voter that consists of an N-way comparator and some counting logic. If a majority of the replicas agree on the result, the voter accepts that result and passes it along to the next system component. If any replicas disagree with the majority, the voter may in addition raise an alert, calling for repair of the replicas that were in the minority. If there is no majority, the voter signals that the supermodule has failed. In failure-tolerance terms, a triply-redundant fail-vote supermodule can mask the failure of any one replica, and it is fail-fast if any two replicas fail in different ways.

If the reliability, as was defined in Section 2.3.2, of a single replica module is R and the underlying fault mechanisms are independent, a TMR fail-vote supermodule will operate correctly if all 3 modules are working (with reliability R^3) or if 1 module has failed and the other 2 are working (with reliability $R^2(1-R)$). Since a single-module failure can happen in 3 different ways, the reliability of the supermodule is the sum

$$R_{supermodule} = R^3 + 3R^2(1-R) = 3R^2 - 2R^3 \tag{2.5.1}$$

but the supermodule is not always fail-fast. If two replicas fail in exactly the same way, the voter will accept the erroneous result and, unfortunately, call for repair of the one correctly operating replica. This outcome is not as unlikely as it sounds because several replicas that went through the same design and production process may have exactly the same set of design or manufacturing faults. This problem can arise despite the independence assumption used in calculating the probability of correct operation. That calculation assumes only that the probability that different replicas produce correct answers be independent; it assumes nothing about the probability of producing specific wrong answers. Without more information about the probability of specific errors and their correlations the only thing we can say about the probability that an incorrect result will be accepted by the voter is that it is not more than

$$1 - R_{supermodule} = (1 - 3R^2 + 2R^3) \tag{2.5.2}$$

These calculations assume that the voter is perfectly reliable. Rather than trying to create perfect voters, the obvious thing to do is replicate them, too. In fact, everything—modules, inputs, outputs, sensors, actuators, etc.—should be replicated, and the final vote should be taken by the client of the system. Thus, three-engine airplanes vote with their propellers: when one engine fails, the two that continue to operate overpower the inoperative one. On the input side, the pilot's hand presses forward on three separate throttle levers. A fully replicated TMR supermodule is shown in Figure 2.5.5. With this interconnection arrangement, any measurement or estimate of the reliability, R, of a component module should include the corresponding voter. It is actually customary (and more logical) to consider a voter to be a component of the next module in the chain rather than, as the diagram suggests, the previous module. This fully replicated design is sometimes described as **recursive**.

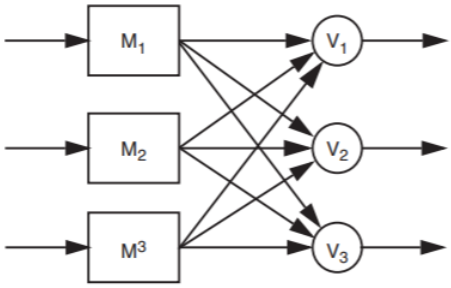


Figure 2.5.5: Triple-modular redundant supermodule, with three inputs, three voters, and three outputs.

The numerical effect of fail-vote TMR is impressive. If the reliability of a single module at time T is 0.999, Equation 2.5.1 says that the reliability of a fail-vote TMR supermodule at that same time is 0.999997. TMR has reduced the probability of failure from one in a thousand to three in a million. This analysis explains why airplanes intended to fly across the ocean have more than one engine. Suppose that the rate of engine failures is such that a single-engine plane would fail to complete one out of a thousand trans-Atlantic flights. Suppose also that a 3-engine plane can continue flying as long as any 2 engines are operating, but it is too heavy to fly with only 1 engine. In 3 flights out of a thousand, one of the three engines will fail, but if engine failures are independent, in 999 out of each thousand first-engine failures, the remaining 2 engines allow the plane to limp home successfully.

Although TMR has greatly improved reliability, it has not made a comparable impact on MTTF. In fact, the MTTF of a fail-vote TMR supermodule can be smaller than the MTTF of the original, single-replica module. The exact effect depends on the failure process of the replicas, so for illustration consider a memoryless failure process, not because it is realistic but because it is mathematically tractable. Suppose that airplane engines have an MTTF of 6,000 hours, they fail independently, the mechanism of engine failure is memoryless, and (since this is a fail-vote design) we need at least 2 operating engines to get home. When flying with three engines, the plane accumulates 6,000 hours of engine running time in only 2,000 hours of flying time, so from the point of view of the airplane as a whole, 2,000 hours is the expected time to the first engine failure. While flying with the remaining two engines, it will take another 3,000 flying hours to accumulate 6,000 more engine hours. Because the failure process is memoryless we can calculate the MTTF of the 3-engine plane by adding:

Mean time to first failure	2000 hours (three engines)
Mean time from first to second failure	3000 hours (two engines)
Total mean time to system failure	5000 hours

Thus the mean time to system failure is less than the 6,000 hour MTTF of a single engine. What is going on here is that we have actually sacrificed long-term reliability in order to enhance short-term reliability. Figure 2.5.6 illustrates the reliability of our hypothetical airplane during its 6 hours of flight, which amounts to only 0.001 of the single-engine MTTF—the mission time is very short compared with the MTTF and the reliability is far higher. Figure 2.5.7 shows the same curve, but for flight times that are comparable with the MTTF. In this region, if the plane tried to keep flying for 8000 hours (about 1.4 times the single-engine MTTF), a single-engine plane would fail to complete the flight in 3 out of 4 tries, but the 3-engine plane would fail to complete the flight in 5 out of 6 tries. (One should be wary of these calculations because the assumptions of independence and memoryless operation may not be met in practice. Sidebar 2.5.1 elaborates.)

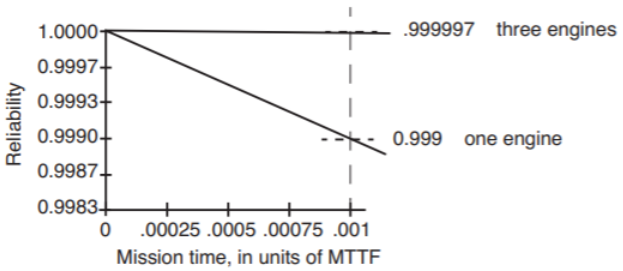


Figure 2.5.6: Reliability with triple modular redundancy, for mission times much less than the MTTF of 6,000 hours. The vertical dotted line represents a six-hour flight.

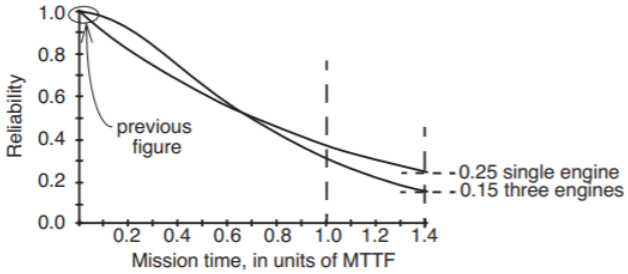


Figure 2.5.7: Reliability with triple modular redundancy, for mission times comparable to the MTTF of 6,000 hours. The two vertical dotted lines represent mission times of 6,000 hours (left) and 8,400 hours (right).

Sidebar 2.5.1

Risks of manipulating MTTFs

The apparently casual manipulation of MTTFs in Sections 2.5.3 and 2.5.4 is justified by assumptions of independence of failures and memoryless processes. But one can trip up by blindly applying this approach without understanding its

limitations. To see how, consider a computer system that has been observed for several years to have a hardware crash an average of every 2 weeks and a software crash an average of every 6 weeks. The operator does not repair the system, but simply restarts it and hopes for the best. The composite MTTF is 1.5 weeks, determined most easily by considering what happens if we run the system for, say, 60 weeks. During that time we expect to see

- 10 software failures
- 30 hardware failures

- 40 system failures in 60 weeks $\rightarrow$ 1.5 weeks between failures

New hardware is installed, identical to the old except that it never fails. The MTTF should jump to 6 weeks because the only remaining failures are software, right?

Perhaps—but *only* if the software failure process is independent of the hardware failure process.

Suppose the software failure occurs because there is a bug (fault) in a clock-updating procedure: The bug always crashes the system exactly 420 hours (2.5 weeks) after it is started—if it gets a chance to run that long. The old hardware was causing crashes so often that the software bug only occasionally had a chance to do its thing—only about once every 6 weeks. Most of the time, the recovery from a hardware failure, which requires restarting the system, had the side effect of resetting the process that triggered the software bug. So, when the new hardware is installed, the system has an MTTF of only 2.5 weeks, much less than hoped.

MTTFs are useful, but one must be careful to understand what assumptions go into their measurement and use.

If we had assumed that the plane could limp home with just one engine, the MTTF would have increased, rather than decreased, but only modestly. Replication provides a dramatic improvement in reliability for missions of duration short compared with the MTTF, but the MTTF itself changes much less. We can verify this claim with a little more analysis, again assuming memoryless failure processes to make the mathematics tractable. Suppose we have an NMR system with the property that it somehow continues to be useful as long as at least one replica is still working. (This system requires using failfast replicas and a cleverer voter, as described in Section 2.5.4 below.) If a single replica has an $MTTF_{replica} = 1$, there are N independent replicas, and the failure process is memoryless, the expected time until the first failure is $MTTF_{replica}/N$, the expected time from then until the second failure is $MTTF_{replica}/(N-1)$, etc., and the expected time until the system of N replicas fails is the sum of these times,

$$MTTF_{system} = 1 + 1/2 + 1/3 + \ldots (1 - N) \tag{2.5.3}$$

which for large N is approximately $\ln(N)$. As we add to the cost by adding more replicas, $MTTF_{system}$ grows disappointingly slowly—proportional to the logarithm of the cost. To multiply the $MTTF_{system}$ by K, the number of replicas required is e^K —the cost grows exponentially. The significant conclusion is that *in systems for which the mission time is long compared with* $MTTF_{replica}$, *simple replication escalates the cost while providing little benefit*. On the other hand, there is a way of making replication effective for long missions, too. The method is to enhance replication by adding **repair**.

2.5.4: Repair

Let us return now to a fail-vote TMR supermodule (that is, it requires that at least two replicas be working) in which the voter has just noticed that one of the three replicas is producing results that disagree with the other two. Since the voter is in a position to report which replica has failed, suppose that it passes such a report along to a repair person who immediately examines the failing replica and either fixes or replaces it. For this approach, the mean time to repair (MTTR) measure becomes of interest. The supermodule fails if either the second or third replica fails before the repair to the first one can be completed. Our intuition is that if the MTTR is small compared with the combined MTTF of the other two replicas, the chance that the supermodule fails will be similarly small.

The exact effect on chances of supermodule failure depends on the shape of the reliability function of the replicas. In the case where the failure and repair processes are both memoryless, the effect is easy to calculate. Since the rate of failure of 1 replica is $1/MTTF$, the rate of failure of 2 replicas is $2/MTTF$. If the repair time is short compared with MTTF, the probability of a

failure of one of the two remaining replicas while waiting a time T for repair of the one that failed is approximately $2T/MTTF$. Since the mean time to repair is MTTR, we have

$$Pr(\text{supermodule fails while waiting for repair}) = \frac{2 \times MTTR}{MTTF} \tag{2.5.4}$$

Continuing our airplane example and temporarily suspending disbelief, suppose that during a long flight we send a mechanic out on the airplane's wing to replace a failed engine. If the replacement takes 1 hour, the chance that one of the other two engines fails during that hour is approximately 1/3000. Moreover, once the replacement is complete, we expect to fly another 2000 hours until the next engine failure. Assuming further that the mechanic is carrying an unlimited supply of replacement engines, completing a 10,000 hour flight—or even a longer one—becomes plausible. The general formula for the MTTF of a fail-vote TMR supermodule with memoryless failure and repair processes is (this formula comes out of the analysis of continuous-transition birth-and-death Markov processes, an advanced probability technique that is beyond our scope):

$$MTTF_{\text{supermodule}} = \frac{MTTF_{replica}}{3} \times \frac{MTTF_{replica}}{2 \times MTTR_{replica}} = \frac{(MTTF_{replica})^2}{6 \times MTTR_{replica}} \tag{2.5.5}$$

Thus, our 3-engine plane with hypothetical in-flight repair has an MTTF of 6 million hours, an enormous improvement over the 6000 hours of a single-engine plane. This equation can be interpreted as saying that, compared with an unreplicated module, the MTTF has been reduced by the usual factor of 3 because there are 3 replicas, but at the same time the availability of repair has increased the MTTF by a factor equal to the ratio of the MTTF of the remaining 2 engines to the MTTR.

Replacing an airplane engine in flight may be a fanciful idea, but replacing a magnetic disk in a computer system on the ground is quite reasonable. Suppose that we store 3 replicas of a set of data on 3 independent hard disks, each of which has an MTTF of 5 years (using as the MTTF the expected operational lifetime, not the "MTTF" derived from the short-term failure rate). Suppose also, that if a disk fails, we can locate, install, and copy the data to a replacement disk in an average of 10 hours. In that case, by Equation 2.5.5, the MTTF of the data is

$$\frac{(MTTF_{replica})^2}{6 \times MTTR_{replica}} = \frac{(5 \text{ years})^2}{6 \cdot (10 \text{ hours})/(8760 \text{ hours/year})} = 3650 \text{ years} \tag{2.5.6}$$

In effect, redundancy plus repair has reduced the probability of failure of this supermodule to such a small value that for all practical purposes, failure can be neglected and the supermodule can operate indefinitely.

Before running out to start a company that sells superbly reliable disk-storage systems, it would be wise to review some of the overly optimistic assumptions we made in getting that estimate of the MTTF, most of which are not likely to be true in the real world:

- *Disks fail independently*. A batch of real world disks may all come from the same vendor, where they acquired the same set of design and manufacturing faults. Or, they may all be in the same machine room, where a single earthquake—which probably has an MTTF of less than 3,650 years—may damage all three.
- *Disk failures are memoryless*. Real-world disks follow a bathtub curve. If, when disk #1 fails, disk #2 has already been in service for three years, disk #2 no longer has an expected operational lifetime of 5 years, so the chance of a second failure while waiting for repair is higher than the formula assumes. Furthermore, when disk #1 is replaced, its chances of failing are probably higher than usual for the first few weeks.
- *Repair is also a memoryless process*. In the real world, if we stock enough spares that we run out only once every 10 years and have to wait for a shipment from the factory, but doing a replacement happens to run us out of stock today, we will probably still be out of stock tomorrow and the next day.
- *Repair is done flawlessly*. A repair person may replace the wrong disk, forget to copy the data to the new disk, or install a disk that hasn't passed burn-in and fails in the first hour.

Each of these concerns acts to reduce the reliability below what might be expected from our overly simple analysis. Nevertheless, NMR with repair remains a useful technique, and in Chapter 4 we will see ways in which it can be applied to disk storage.

One of the most powerful applications of NMR is in the masking of transient errors. When a transient error occurs in one replica, the NMR voter immediately masks it. Because the error is transient, the subsequent behavior of the supermodule is as if repair happened by the next operation cycle. The numerical result is little short of extraordinary. For example, consider a processor arithmetic logic unit (ALU) with a 1 gigahertz clock and which is triply replicated with voters checking its output at the end of each

clock cycle. In Equation 2.5.5 we have $MTTR_{replica} = 1$ (in this application, Equation 2.5.5 is only an approximation because the time to repair is a constant rather than the result of a memoryless process), and $MTTF_{supermodule} = (MTTF_{replica})^2 / 6$ cycles . If $MTTF_{replica}$ is 10^{10} cycles (1 error in 10 billion cycles, which at this clock speed means one error every 10 seconds), $MTTF_{supermodule}$ is $10^{20} / 6$ cycles, about 500 years. TMR has taken three ALUs that were for practical use nearly worthless and created a super-ALU that is almost infallible.

The reason things seem so good is that we are evaluating the chance that two transient errors occur in the same operation cycle. If transient errors really are independent, that chance is small. This effect is powerful, but the leverage works in both directions, thereby creating a potential hazard: it is especially important to keep track of the rate at which transient errors actually occur. If they are happening, say, 20 times as often as hoped, $MTTF_{supermodule}$ will be 1/400 of the original prediction—the super-ALU is likely to fail once per year. That may still be acceptable for some applications, but it is a big change. Also, as usual, the assumption of independence is absolutely critical. If all the ALUs came from the same production line, it seems likely that they will have at least some faults in common, in which case the super-ALU may be just as worthless as the individual ALUs.

Several variations on the simple fail-vote structure appear in practice:

- **Purging.** In an NMR design with a voter, whenever the voter detects that one replica disagrees with the majority, the voter calls for its repair and in addition marks that replica DOWN and ignores its output until hearing that it has been repaired. This technique doesn't add anything to a TMR design, but with higher levels of replication, as long as replicas fail one at a time and any two replicas continue to operate correctly, the supermodule works.
- **Pair-and-compare.** Create a fail-fast module by taking two replicas, giving them the same inputs, and connecting a simple comparator to their outputs. As long as the comparator reports that the two replicas of a pair agree, the next stage of the system accepts the output. If the comparator detects a disagreement, it reports that the module has failed. The major attraction of pair-and-compare is that it can be used to create fail-fast modules starting with easily available commercial, off-the-shelf components, rather than commissioning specialized fail-fast versions. Special high-reliability components typically have a cost that is much higher than off-the-shelf designs, for two reasons. First, since they take more time to design and test, the ones that are available are typically of an older, more expensive technology. Second, they are usually low-volume products that cannot take advantage of economies of large-scale production. These considerations also conspire to produce long delivery cycles, making it harder to keep spares in stock. An important aspect of using standard, high-volume, low-cost components is that one can afford to keep a stock of spares, which in turn means that MTTR can be made small: just replace a failing replica with a spare (the popular term for this approach is **pair-and-spare**) and do the actual diagnosis and repair at leisure.
- **NMR with fail-fast replicas.** If each of the replicas is itself a fail-fast design (perhaps using pair-and-compare internally), then a voter can restrict its attention to the outputs of only those replicas that claim to be producing good results and ignore those that are reporting that their outputs are questionable. With this organization, a TMR system can continue to operate even if 2 of its 3 replicas have failed, since the 1 remaining replica is presumably checking its own results. An NMR system with repair and constructed of fail-fast replicas is so robust that it is unusual to find examples for which N is greater than 2.

Figure 2.5.8 compares the ability to continue operating until repair arrives of 5MR designs that use fail-vote, purging, and fail-fast replicas. The observant reader will note that this chart can be deemed guilty of a misleading comparison, since it claims that the 5MR system continues working when only one fail-fast replica is still running. But if that fail-fast replica is actually a pair-and-compare module, it might be more accurate to say that there are two still-working replicas at that point.

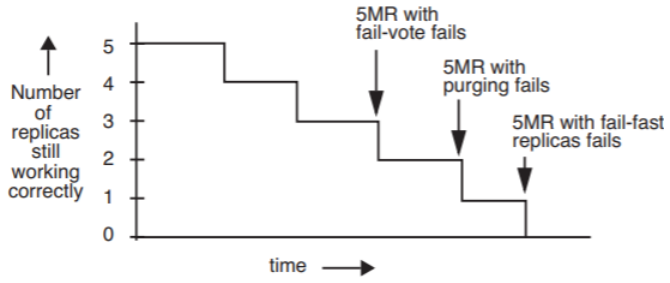


Figure 2.5.8: Failure points of three different 5MR supermodule designs, if repair does not happen in time.

Another technique that takes advantage of repair, can improve availability, and can degrade gracefully (in other words, it can be fail-soft) is called **partition**. If there is a choice of purchasing a system that has either one fast processor or two slower processors, the two-processor system has the virtue that when one of its processors fails, the system can continue to operate with half of its

usual capacity until someone can repair the failed processor. An electric power company, rather than installing a single generator of capacity K megawatts, may install N generators of capacity K/N megawatts each.

When equivalent modules can easily share a load, partition can extend to what is called $\mathbf{N+1}$ **redundancy**. Suppose a system has a load that would require the capacity of N equivalent modules. The designer partitions the load across $N+1$ or more modules. Then, if any one of the modules fails, the system can carry on at full capacity until the failed module can be repaired.

$N+1$ redundancy is most applicable to modules that are completely interchangeable, can be dynamically allocated, and are not used as storage devices. Examples are processors, dial-up modems, airplanes, and electric generators. Thus, one extra airplane located at a busy hub can mask the failure of any single plane in an airline's fleet. When modules are not completely equivalent (for example, electric generators come in a range of capacities, but can still be interconnected to share load), the design must ensure that the spare capacity is greater than the capacity of the largest individual module. For devices that provide storage, such as a hard disk, it is also possible to apply partition and $N+1$ redundancy with the same goals, but it requires a greater level of organization to preserve the stored contents when a failure occurs, for example by using RAID, as was described in Section 2.5.1, or some more general replica management system such as those discussed in Section 4.4.7.

For some applications an occasional interruption of availability is acceptable, while in others every interruption causes a major problem. When repair is part of the fault tolerance plan, it is sometimes possible, with extra care and added complexity, to design a system to provide **continuous operation**. Adding this feature requires that when failures occur, one can quickly identify the failing component, remove it from the system, repair it, and reinstall it (or a replacement part) all without halting operation of the system. The design required for continuous operation of computer hardware involves connecting and disconnecting cables and turning off power to some components but not others, without damaging anything. When hardware is designed to allow connection and disconnection from a system that continues to operate, it is said to allow **hot swap**.

In a computer system, continuous operation also has significant implications for the software. Configuration management software must anticipate hot swap so that it can stop using hardware components that are about to be disconnected, as well as discover newly attached components and put them to work. In addition, maintaining state is a challenge. If there are periodic consistency checks on data, those checks (and repairs to data when the checks reveal inconsistencies) must be designed to work correctly even though the system is in operation and the data is perhaps being read and updated by other users at the same time.

Overall, continuous operation is not a feature that should be casually added to a list of system requirements. When someone suggests it, it may be helpful to point out that it is much like trying to keep an airplane flying indefinitely. Many large systems that appear to provide continuous operation are actually designed to stop occasionally for maintenance.

This page titled 2.5: Systematically Applying Redundancy is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

2.6: Applying Redundancy to Software and Data

The examples of redundancy and replication in the previous sections all involve hardware. A seemingly obvious next step is to apply the same techniques to software and to data. In the case of software the goal is to reduce the impact of programming errors, while in the case of data the goal is to reduce the impact of any kind of hardware, software, or operational error that might affect its integrity. This section begins the exploration of several applicable techniques: N-version programming, valid construction, and building a firewall to separate stored state into two categories: a state whose integrity must be preserved and a state that can casually be abandoned because it is easy to reconstruct.

2.6.1: Tolerating Software Faults

Simply running three copies of the same buggy program is likely to produce three identical incorrect results. NMR requires independence among the replicas, so the designer needs a way of introducing that independence. An example of a way of introducing independence is found in the replication strategy for the root name servers of the Internet Domain Name System (DNS). Over the years, slightly different implementations of the DNS software have evolved for different operating systems, so the root name server replicas intentionally employ these different implementations to reduce the risk of replicated errors.

To try to harness this idea more systematically, one can commission several teams of programmers and ask each team to write a complete version of an application according to a single set of specifications. Then, run these several versions in parallel and compare their outputs. The hope is that the inevitable programming errors in the different versions will be independent and voting will produce a reliable system. Experiments with this technique, known as **N-version programming**, suggest that the necessary independence is hard to achieve. Different programmers may be trained in similar enough ways that they make the same mistakes. Use of the same implementation language may encourage the same errors. Ambiguities in the specification may be misinterpreted in the same way by more than one team and the specification itself may contain errors. Finally, it is hard to write a specification in enough detail that the outputs of different implementations can be expected to be bit-for-bit identical. The result is that after much effort, the technique may still mask only a certain class of bugs and leave others unmasked. Nevertheless, there are reports that N-version programming has been used, apparently with success, in at least two safety-critical aerospace systems: the flight control system of the Boeing 777 aircraft (with $N = 3$) and the on-board control system for the Space Shuttle (with $N = 2$).

Incidentally, the strategy of employing multiple design teams can also be applied to hardware replicas, with a goal of increasing the independence of the replicas by reducing the chance of replicated design errors and systematic manufacturing defects.

Much of software engineering is devoted to a different approach: devising specification and programming techniques that avoid faults in the first place and test techniques that systematically root out faults so that they can be repaired once and for all before deploying the software. This approach, sometimes called **valid construction**, can dramatically reduce the number of software faults in a delivered system, but because it is difficult both to completely specify and to completely test a system, some faults inevitably remain. Valid construction is based on the observation that software, unlike hardware, is not subject to wear and tear, so if it is once made correct, it should stay that way. Unfortunately, this observation can turn out to be wishful thinking, first because it is hard to make software correct, and second because it is nearly always necessary to make changes after installing a program because the requirements, the environment surrounding the program, or both, have changed. There is thus a potential for tension between valid construction and the principle that one should *design for iteration*.

Worse, later maintainers and reworkers often do not have a complete understanding of the ground rules that went into the original design, so their work is likely to introduce new faults for which the original designers did not anticipate providing tests. Even if the original design is completely understood, when a system is modified to add features that were not originally planned, the original ground rules may be subjected to some violence. Software faults more easily creep into areas that lack systematic design.

2.6.2: Tolerating Software (and other) Faults by Separating State

Designers of reliable systems usually assume that, despite the best efforts of programmers there will always be a residue of software faults, just as there is also always a residue of hardware, operation, and environment faults. The response is to develop a strategy for tolerating all of them. Software adds the complication that the current state of a running program tends to be widely distributed. Parts of that state may be in non-volatile storage, while other parts are in temporary variables held in volatile memory locations, processor registers, and kernel tables. This wide distribution of state makes containment of errors problematic. As a

result, when an error occurs, any strategy that involves stopping some collection of running threads, tinkering to repair the current state (perhaps at the same time replacing a buggy program module), and then resuming the stopped threads is usually unrealistic.

In the face of these observations, a programming discipline has proven to be effective: systematically divide the current state of a running program into two mutually exclusive categories and separate the two categories with a firewall. The two categories are:

- State that the system can safely abandon in the event of a failure.
- State whose integrity the system should preserve despite failure.

Upon detecting a failure, the plan becomes to abandon all state in the first category and instead concentrate just on maintaining the integrity of the data in the second category. An important part of the strategy is an important sweeping simplification: classify the state of running threads (that is, the thread table, stacks, and registers) as abandonable. When a failure occurs, the system abandons the thread or threads that were running at the time and instead expects a restart procedure, the system operator, or the individual user to start a new set of threads with a clean slate. The new thread or threads can then, working with only the data found in the second category, verify the integrity of that data and return to normal operation. The primary challenge then becomes to build a firewall that can protect the integrity of the second category of data despite the failure.

The designer can base a natural firewall on the common implementations of volatile (e.g., CMOS memory) and non-volatile (e.g., magnetic disk) storage. As it happens, writing to non-volatile storage usually involves mechanical movement such as rotation of a disk platter, so most transfers move large blocks of data to a limited region of addresses, using a `GET` / `PUT` interface. On the other hand, volatile storage technologies typically provide a `READ` / `WRITE` interface that allows rapid-fire writes to memory addresses chosen at random, so failures that originate in or propagate to software tend to quickly and untraceably corrupt random-access data. By the time an error is detected the software may thus have already damaged a large and unidentifiable part of the data in volatile memory. The `GET` / `PUT` interface instead acts as a bottleneck on the rate of spread of data corruption. The goal can be succinctly stated: to detect failures and stop the system before it reaches the next `PUT` operation, thus making the volatile storage medium the error containment boundary. It is only incidental that volatile storage usually has a `READ` / `WRITE` interface, while non-volatile storage usually has a `GET` / `PUT` interface, but because that is usually true it becomes a convenient way to implement and describe the firewall.

This technique is widely used in systems whose primary purpose is to manage long-lived data. In those systems, two aspects are involved:

- Prepare for failure by recognizing that all state in volatile memory devices can vanish at any instant, without warning. When it does vanish, automatically launch new threads that start by restoring the data in non-volatile storage to a consistent, easily described state. The techniques to do this restoration are called **recovery**. Doing recovery systematically involves atomicity, which is explored in Chapter 3.
- Protect the data in non-volatile storage using replication, thus creating the class of storage known as **durable** storage. Replicating data can be a straightforward application of redundancy, so we will begin the topic in this chapter. However, there are more effective designs that make use of atomicity and geographical separation of replicas, so we will revisit durability in Chapter 4.

When the volatile storage medium is CMOS RAM and the non-volatile storage medium is magnetic disk, following this programming discipline is relatively straightforward because the distinctively different interfaces make it easy to remember where to place data. But when a one-level store is in use, giving the appearance of random access to all storage, or the non-volatile medium is flash memory, which allows fast random access, it may be necessary for the designer to explicitly specify both the firewall mechanism and which data items are to reside on each side of the firewall.

A good example of the firewall strategy can be found in most implementations of Internet Domain Name System servers. In a typical implementation the server stores the authoritative name records for its domain on magnetic disk, and copies those records into volatile CMOS memory either at system startup or the first time it needs a particular record. If the server fails for any reason, it simply abandons the volatile memory and restarts. In some implementations, the firewall is reinforced by not having any `PUT` operations in the running name server. Instead, the service updates the authoritative name records using a separate program that runs when the name server is off-line.

In addition to employing independent software implementations and a firewall between categories of data, DNS also protects against environmental faults by employing geographical separation of its replicas, a topic that is explored more deeply in Section 4.3. The three techniques, taken together, make DNS quite fault-tolerant.

2.6.3: Durability and Durable Storage

For the discipline just described to work, we need to make the result of a `PUT` operation durable. But first we must understand just what "durable" means. **Durability** is a specification of how long the result of an action must be preserved after the action completes. One must be realistic in specifying durability because there is no such thing as perfectly durable storage in which the data will be remembered forever. However, by choosing enough genuinely independent replicas, and with enough care in management, one can meet any reasonable requirement.

Durability specifications can be roughly divided into four categories, according to the length of time that the application requires that data survive. Although there are no bright dividing lines, as one moves from one category to the next the techniques used to achieve durability tend to change.

- *Durability no longer than the lifetime of the thread that created the data.* For this case, it is usually adequate to place the data in volatile memory.

For example, an action such as moving the gearshift may require changing the operating parameters of an automobile engine. The result must be reliably remembered, but only until the next shift of gears or the driver switches the engine off.

The operations performed by calls to the kernel of an operating system provide another example. The `CHDIR` procedure of the UNIX kernel changes the working directory of the currently running process. The kernel state variable that holds the name of the current working directory is a value in volatile RAM that does not need to survive longer than this process.

For a third example, the registers and cache of a hardware processor usually provide just the first category of durability. If there is a failure, the plan is to abandon those values along with the contents of volatile memory, so there is no need for a higher level of durability.

- *Durability for times which are short compared to the expected operational lifetime of nonvolatile storage media such as magnetic disk or flash memory*. A designer typically implements this category of durability by writing one copy of the data in the nonvolatile storage medium.

Returning to the automotive example, there may be operating parameters such as engine timing that, once calibrated, should be durable at least until the next tune-up, not just for the life of one engine use session. Data stored in a cache that writes through to a non-volatile medium has about this level of durability. As a third example, a remote procedure call protocol that identifies duplicate messages by recording nonces might write old nonce values (see Section 1.6.3) to a non-volatile storage medium, knowing that the real goal is not to remember the nonces forever, but rather to make sure that the nonce record outlasts the longest retry timer of any client. Finally, text editors and word-processing systems typically write temporary copies on magnetic disk of the material currently being edited so that if there is a system crash or power failure the user does not have to repeat the entire editing session. These temporary copies need to survive only until the end of the current editing session.

- *Durability for times comparable to the expected operational lifetime of non-volatile storage media*. Because actual non-volatile media lifetimes vary quite a bit around the expected lifetime, implementation generally involves placing replicas of the data on independent instances of the non-volatile media.

This category of durability is the one that is usually called **durable storage** and it is the category for which the next section of this chapter develops techniques for implementation. Users typically expect files stored in their file systems and data managed by a database management system to have this level of durability. Section 4.4 revisits the problem of creating durable storage when replicas are geographically separated.

- *Durability for many multiples of the expected operational lifetime of non-volatile storage media*.

This highest level of durability is known as **preservation**, and is the specialty of archivists. In addition to making replicas and keeping careful records, it involves copying data from one non-volatile medium to another before the first one deteriorates or becomes obsolete. Preservation also involves (sometimes heroic) measures to preserve the ability to correctly interpret idiosyncratic formats created by software that has long since become obsolete. Although important, it is a separate topic, so preservation is not discussed any further here.

2.6.4: Magnetic Disk Fault Tolerance

In principle, durable storage can be constructed starting with almost any storage medium, but it is most straightforward to use non-volatile devices. Magnetic disks are widely used as the basis for durable storage because of their low cost, large capacity and non-volatility—they retain their memory when power is turned off or is accidentally disconnected. Even if power is lost during a write operation, at most a small block of data surrounding the physical location that was being written is lost, and disks can be designed with enough internal power storage and data buffering to avoid even that loss. In its raw form, a magnetic disk is remarkably reliable, but it can still fail in various ways and much of the complexity in the design of disk systems consists of masking these failures.

Conventionally, magnetic disk systems are designed in three nested layers. The innermost layer is the spinning disk itself, which provides what we will call **raw storage**. The next layer is a combination of hardware and firmware of the disk controller that provides for detecting the failures in the raw storage layer; it creates **fail-fast storage**. Finally, the hard disk firmware adds a third layer that takes advantage of the detection features of the second layer to create a substantially more reliable storage system, known as **careful storage**. Most disk systems stop there, but high-availability systems add a fourth layer to create **durable storage**. This section develops a disk failure model and explores error masking techniques for all four layers.

In early disk designs, the disk controller presented more or less the raw disk interface, and the fail-fast and careful layers were implemented in a software component of the operating system called the disk driver. Over the decades, first the fail-fast layer and more recently part or all of the careful layer of disk storage have migrated into the firmware of the disk controller to create what is known in the trade as a "hard drive." A hard drive usually includes a RAM buffer to hold a copy of the data going to and from the disk, both to avoid the need to match the data rate to and from the disk head with the data rate to and from the system memory, and also to simplify retries when errors occur. RAID systems, which provide a form of durable storage, generally are implemented as an additional hardware layer that incorporates mass-market hard drives. One reason for this move of error masking from the operating system into the disk controller is that as computational power has gotten cheaper, the incremental cost of a more elaborate firmware design has dropped. A second reason may explain the obvious contrast with the lack of enthusiasm for memory parity checking hardware that is mentioned in Section 2.9.1. A transient memory error is all but indistinguishable from a program error, so the hardware vendor is not likely to be blamed for it. On the other hand, most disk errors have an obvious source, and hard errors are not transient. Because blame is easy to place, disk vendors have a strong motivation to include error masking in their designs.

2.6.4.1: Magnetic Disk Fault Modes

There are several considerations when it comes to disk reliability:

- Disks are high precision devices made to close tolerances. Defects in manufacturing a recording surface typically show up in the field as a sector that does not reliably record data. Such defects are a source of hard errors. Deterioration of the surface of a platter with age can cause a previously good sector to fail. Such loss is known as **decay** and, since any data previously recorded there is lost forever, decay is another example of hard error.
- Since a disk is mechanical, it is subject to wear and tear. Although a modern disk is a sealed unit, deterioration of its component materials as they age can create dust. The dust particles can settle on a magnetic surface, where they may interfere either with reading or writing. If interference is detected, then rereading or rewriting that area of the surface, perhaps after jiggling the seek arm back and forth, may succeed in getting past the interference, so the fault may be transient. Another source of transient faults is electrical noise spikes. Because disk errors caused by transient faults can be masked by retry, they fall in the category of soft errors.
- If a running disk is bumped, the shock may cause a head to hit the surface of a spinning platter, causing what is known as a head crash. A head crash not only may damage the head and destroy the data at the location of impact, it also creates a cloud of dust that interferes with the operation of heads on other platters. A head crash generally results in several sectors decaying simultaneously. A set of sectors that tend to all fail together is known as a **decay set**. A decay set may be quite large, for example all the sectors on one drive or on one disk platter.
- As electronic components in the disk controller age, clock timing and signal detection circuits can go out of tolerance, causing previously good data to become unreadable, or bad data to be written, either intermittently or permanently. In consequence, electronic component tolerance problems can appear either as soft or hard errors.
- The mechanical positioning systems that move the seek arm and that keep track of the rotational position of the disk platter can fail in such a way that the heads read or write the wrong track or sector within a track. This kind of fault is known as a **seek error**.

2.6.4.2: System Faults

In addition to failures within the disk subsystem, there are at least two threats to the integrity of the data on a disk that arise from outside the disk subsystem:

- If the power fails in the middle of a disk write, the sector being written may end up being only partly updated. After the power is restored and the system restarts, the next reader of that sector may find that the sector begins with the new data, but ends with the previous data.
- If the operating system fails during the time that the disk is writing, the data being written could be affected, even if the disk is perfect and the rest of the system is fail-fast. The reason is that all the contents of volatile memory, including the disk buffer, are inside the fail-fast error containment boundary and thus at risk of damage when the system fails. As a result, the disk channel may correctly write on the disk what it reads out of the disk buffer in memory, but the faltering operating system may have accidentally corrupted the contents of that buffer after the application called `PUT` . In such cases, the data that ends up on the disk will be corrupted, but there is no sure way in which the disk subsystem can detect the problem.

2.6.4.3: Raw Disk Storage

Our goal is to devise systematic procedures to mask as many of these different faults as possible. We start with a model of disk operation from a programmer's point of view. The raw disk has, at least conceptually, a relatively simple interface: There is an operation to seek to a (numbered) track, an operation that writes data on the track and an operation that reads data from the track. The failure model is simple: all errors arising from the failures just described are untolerated. (In the procedure descriptions, arguments are call-by-reference, and `GET` operations read from the disk into the argument named `data` .) The raw disk layer implements these storage access procedures and failure tolerance model:

```
RAW_SEEK (track)    // Move read/write head into position.
RAW_PUT (data)      // Write entire track.
RAW_GET (data)      // Read entire track.
```

- error-free operation: `RAW_SEEK` moves the seek arm to position track. `RAW_GET` returns whatever was most recently written by `RAW_PUT` at position track.
- untolerated error: On any given attempt to read from or write to a disk, dust particles on the surface of the disk or a temporarily high noise level may cause data to be read or written incorrectly. (soft error)
- untolerated error: A spot on the disk may be defective, so all attempts to write to any track that crosses that spot will be written incorrectly. (hard error)
- untolerated error: Information previously written correctly may decay, so `RAW_GET` returns incorrect data. (hard error)
- untolerated error: When asked to read data from or write data to a specified track, a disk may correctly read or write the data, but on the wrong track. (seek error)
- untolerated error: The power fails during a `RAW_PUT` with the result that only the first part of *`data`* ends up being written on *`track`* . The remainder of *`track`* may contain older data.
- untolerated error: The operating system crashes during a `RAW_PUT` and scribbles over the disk buffer in volatile storage, so `RAW_PUT` writes corrupted data on one track of the disk.

2.6.4.4: Fail-Fast Disk Storage

The fail-fast layer is the place where the electronics and microcode of the disk controller divide the raw disk track into sectors. Each sector is relatively small, individually protected with an error-detection code, and includes, in addition to a fixed-sized space for data, a sector and track number. The error-detection code enables the disk controller to return a status code on `FAIL_FAST_GET` that tells whether a sector read correctly or incorrectly, and the sector and track numbers enable the disk controller to verify that the seek ended up on the correct track. The `FAIL_FAST_PUT` procedure not only writes the data, but it verifies that the write was successful by reading the newly written sector on the next rotation and comparing it with the data still in the write buffer. The sector thus becomes the minimum unit of reading and writing, and the disk address becomes the pair `{track, sector_number}` . For performance enhancement, some systems allow the caller to bypass the verification step of `FAIL_FAST_PUT` . When the client chooses this bypass, write failures become indistinguishable from decay events.

There is always a possibility that the data on a sector is corrupted in such a way that the error-detection code accidentally verifies. For completeness, we will identify that case as an untolerated error, but point out that the error-detection code should be powerful enough that the probability of this outcome is negligible.

The fail-fast layer implements these storage access procedures and failure tolerance model:

```
status ← FAIL_FAST_SEEK (track)
status ← FAIL_FAST_PUT (data, sector_number)
status ← FAIL_FAST_GET (data, sector_number)
```

- error-free operation: `FAIL_FAST_SEEK` moves the seek arm to track. `FAIL_FAST_GET` returns whatever was most recently written by `FAIL_FAST_PUT` at *`sector_number`* on *`track`* and returns `status = OK` .
- detected error: `FAIL_FAST_GET` reads the data, checks the error-detection code and finds that it does not verify. The cause may a soft error, a hard error due to decay, or a hard error because there is a bad spot on the disk and the invoker of a previous `FAIL_FAST_PUT` chose to bypass verification. `FAIL_FAST_GET` does not attempt to distinguish these cases; it simply reports the error by returning `status = BAD` .
- detected error: `FAIL_FAST_PUT` writes the data, on the next rotation reads the data back, checks the error-detection code, finds that it does not verify, and reports the error by returning `status = BAD` .
- detected error: `FAIL_FAST_SEEK` moves the seek arm, reads the permanent track number in the first sector that comes by, discovers that it does not match the requested track number (or that the sector checksum does not verify), and reports the error by returning `status = BAD` .
- detected error: The caller of `FAIL_FAST_PUT` tells it to bypass the verification step, so `FAIL_FAST_PUT` always reports `status = OK` even if the sector was not written correctly. But a later caller of `FAIL_FAST_GET` that requests that sector should detect any such error.
- detected error: The power fails during a `FAIL_FAST_PUT` with the result that only the first part of *`data`* ends up
- being written on *`sector`* . The remainder of *`sector`* may contain older data. Any later call of `FAIL_FAST_GET` for that sector should discover that the sector checksum fails to verify and will thus return `status = BAD` . Many (but not all) disks are designed to mask this class of failure by maintaining a reserve of power that is sufficient to complete any current sector write, in which case loss of power would be a tolerated failure.
- untolerated error: The operating system crashes during a `FAIL_FAST_PUT` and scribbles over the disk buffer in volatile storage, so `FAIL_FAST_PUT` writes corrupted data on one sector of the disk.
- untolerated error: The data of some sector decays in a way that is undetectable—the checksum accidentally verifies. (Probability should be negligible.)

2.6.4.5: Careful Disk Storage

The fail-fast disk layer detects but does not mask errors. It leaves masking to the careful disk layer, which is also usually implemented in the firmware of the disk controller. The careful layer checks the value of `status` following each disk `SEEK` , `GET` and `PUT` operation, retrying the operation several times if necessary, a procedure that usually recovers from seek errors and soft errors caused by dust particles or a temporarily elevated noise level. Some disk controllers seek to a different track and back in an effort to dislodge the dust. The careful storage layer implements these storage procedures and failure tolerance model:

```
status ← CAREFUL_SEEK (track)
status ← CAREFUL_PUT (data, sector_number)
status ← CAREFUL_GET (data, sector_number)
```

- error-free operation: `CAREFUL_SEEK` moves the seek arm to track. `CAREFUL_GET` returns whatever was most recently written by `CAREFUL_PUT` at *`sector_number`* on *`track`* . All three return `status = OK` .
- tolerated error: *Soft read, write, or seek error.* `CAREFUL_SEEK` , `CAREFUL_GET` and `CAREFUL_PUT` mask these errors by repeatedly retrying the operation until the fail-fast layer stops detecting an error, returning with `status = OK` . The careful storage layer counts the retries, and if the retry count exceeds some limit, it gives up and declares the problem to be a hard error.
- detected error: *Hard error.* The careful storage layer distinguishes hard from soft errors by their persistence through several attempts to read, write, or seek, and reports them to the caller by setting `status = BAD` . (But also see the note on **revectoring** below.)
- detected error: The power fails during a `CAREFUL_PUT` with the result that only the first part of *`data`* ends up being written on *`sector`* . The remainder of *`sector`* may contain older data. Any later call of `CAREFUL_GET` for that sector should

discover that the sector checksum fails to verify and will thus return `status = BAD` . (Assuming that the fail-fast layer does not tolerate power failures.)

- untolerated error: *Crash corrupts data.* The system crashes during `CAREFUL_PUT` and corrupts the disk buffer in volatile memory, so `CAREFUL_PUT` correctly writes to the disk sector the corrupted data in that buffer. The sector checksum of the fail-fast layer cannot detect this case.
- untolerated error: The data of some sector decays in a way that is undetectable—the checksum accidentally verifies. (Probability should be negligible)

Figure 2.6.1 exhibits algorithms for `CAREFUL_GET` and `CAREFUL_PUT` . The procedure `CAREFUL_GET` , by repeatedly reading any data with `status = BAD` , masks soft read errors. Similarly, `CAREFUL_PUT` retries repeatedly if the verification done by `FAIL_FAST_PUT` fails, thereby masking soft write errors, whatever their source.

```
1  procedure CAREFUL_GET (data, sector_number)
2      for i from 1 to NTRIES do
3          if FAIL_FAST_GET (data, sector_number) = OK then
4              return OK
5      return BAD

6  procedure CAREFUL_PUT (data, sector_number)
7      for i from 1 to NTRIES do
8          if FAIL_FAST_PUT (data, sector_number) = OK then
9              return OK
10     return BAD
```

Figure 2.6.1: Procedures that implement careful disk storage.

The careful layer of most disk controller designs includes one more feature: if `CAREFUL_PUT` detects a hard error while writing a sector, it may instead write the data on a spare sector elsewhere on the same disk and add an entry to an internal disk mapping table so that future instances of `GET` and `PUT` specifying that sector instead use the spare. This mechanism is called **revectoring**, and most disk designs allocate a batch of spare sectors for this purpose. The spares are not usually counted in the advertised disk capacity, but the manufacturer's advertising department does not usually ignore the resulting increase in the expected operational lifetime of the disk. For clarity of the discussion we omit that feature.

As indicated in the failure tolerance analysis, there are still two modes of failure that remain unmasked: a crash during `CAREFUL_PUT` may undetectably corrupt one disk sector, and a hard error arising from a bad spot on the disk or a decay event may detectably corrupt any number of disk sectors.

2.6.4.6: Durable Storage: RAID 1

For durability, the additional requirement is to mask decay events, which the careful storage layer only detects. The primary technique is that the PUT procedure should write several replicas of the data, taking care to place the replicas on different physical devices with the hope that the probability of disk decay in one replica is independent of the probability of disk decay in the next one, and the number of replicas is large enough that when a disk fails there is enough time to replace it before all the other replicas fail. Disk system designers call these replicas **mirrors**. A carefully designed replica strategy can create storage that guards against premature disk failure and that is durable enough to substantially exceed the expected operational lifetime of any single physical disk. Errors on reading are detected by the fail-fast layer, so it is not usually necessary to read more than one copy unless that copy turns out to be bad. Since disk operations may involve more than one replica, the track and sector numbers are sometimes encoded into a virtual sector number and the durable storage layer automatically performs any needed seeks. The durable storage layer implements these storage access procedures and failure tolerance model:

```
status ← DURABLE_PUT (data, virtual_sector_number)
status ← DURABLE_GET (data, virtual_sector_number)
```

- error-free operation: `DURABLE_GET` returns whatever was most recently written by `DURABLE_PUT` at `virtual_sector_number` with `status = OK` .
- tolerated error: Hard errors reported by the careful storage layer are masked by reading from one of the other replicas. The result is that the operation completes with `status = OK` .
- untolerated error: A decay event occurs on the same sector of all the replicas, and the operation completes with `status = BAD` .

- untolerated error: The operating system crashes during a DURABLE_PUT and scribbles over the disk buffer in volatile storage, so DURABLE_PUT writes corrupted data on all mirror copies of that sector.
- untolerated error: The data of some sector decays in a way that is undetectable—the checksum accidentally verifies. (Probability should be negligible)

In this accounting there is no mention of soft errors or positioning errors because they were all masked by a lower layer.

One configuration of RAID, known as "RAID 1," implements exactly this form of durable storage. RAID 1 consists of a tightly-managed array of identical replica disks in which DURABLE_PUT (*data, sector_number*) writes *data* at the same *sector_number* of each disk and DURABLE_GET reads from whichever replica copy has the smallest expected latency, which includes queuing time, seek time, and rotation time. With RAID, the decay set is usually taken to be an entire hard disk. If one of the disks fails, the next DURABLE_GET that tries to read from that disk will detect the failure, mask it by reading from another replica, and put out a call for repair. Repair consists of first replacing the disk that failed and then copying all of the disk sectors from one of the other replica disks.

2.6.4.7: Improving on RAID 1

Even with RAID 1, an untolerated error can occur if a rarely-used sector decays, and before that decay is noticed all other copies of that same sector also decay. When there is finally a call for that sector, all fail to read and the data is lost. A closely related scenario is that a sector decays and is eventually noticed, but the other copies of that same sector decay before repair of the first one is completed. One way to reduce the chances of these outcomes is to implement a clerk that periodically reads all replicas of every sector, to check for decay. If CAREFUL_GET reports that a replica of a sector is unreadable at one of these periodic checks, the clerk immediately rewrites that replica from a good one. If the rewrite fails, the clerk calls for immediate revectoring of that sector or, if the number of revectorings is rapidly growing, replacement of the decay set to which the sector belongs. The period between these checks should be short enough that the probability that *all* replicas have decayed since the previous check is negligible. By analyzing the statistics of experience for similar disk systems, the designer chooses such a period, T_d. This approach leads to the following failure tolerance model:

```
status ← MORE_DURABLE_PUT (data, virtual_sector_number)
status ← MORE_DURABLE_GET (data, virtual_sector_number)
```

- error-free operation: MORE_DURABLE_GET returns whatever was most recently written by MORE_DURABLE_PUT at *virtual_sector_number* with *status* = OK .
- tolerated error: Hard errors reported by the careful storage layer are masked by reading from one of the other replicas. The result is that the operation completes with *status* = OK .
- tolerated error: data of a single decay set decays, is discovered by the clerk, and is repaired, all within T_d seconds of the decay event.
- untolerated error: The operating system crashes during a DURABLE_PUT and scribbles over the disk buffer in volatile storage, so DURABLE_PUT writes corrupted data on all mirror copies of that sector.
- untolerated error: all decay sets fail within T_d seconds. (With a conservative choice of T_d, the probability of this event should be negligible.)
- untolerated error: The data of some sector decays in a way that is undetectable—the checksum accidentally verifies. (With a good quality checksum, the probability of this event should be negligible.)

A somewhat less effective alternative to running a clerk that periodically verifies integrity of the data is to notice that the bathtub curve of Figure 2.3.1 applies to magnetic disks, and simply adopt a policy of systematically replacing the individual disks of the RAID array well before they reach the point where their conditional failure rate is predicted to start climbing. This alternative is not as effective for two reasons: First, it does not catch and repair random decay events, which instead accumulate. Second, it provides no warning if the actual operational lifetime is shorter than predicted (for example, if one happens to have acquired a bad batch of disks).

2.6.4.8: Detecting Errors Caused by System Crashes

With the addition of a clerk to watch for decay, there is now just one remaining untolerated error that has a significant probability: the hard error created by an operating system crash during CAREFUL_PUT . Since that scenario corrupts the data before the disk subsystem sees it, the disk subsystem has no way of either detecting or masking this error. Help is needed from outside the disk subsystem—either the operating system or the application. The usual approach is that either the system or, even better, the

application program, calculates and includes an end-to-end checksum with the data before initiating the disk write. Any program that later reads the data verifies that the stored checksum matches the recalculated checksum of the data. The end-to-end checksum thus monitors the integrity of the data as it passes through the operating system buffers and also while it resides in the disk subsystem.

Sidebar 2.6.1

Are disk system checksums a wasted effort?

From the previous paragraph, an *end-to-end argument* suggests that an end-to-end checksum is always needed to protect data on its way to and from the disk subsystem, and that the fail-fast checksum performed inside the disk system thus may not be essential.

However, the disk system checksum cleanly subcontracts one rather specialized job: correcting burst errors of the storage medium. In addition, the disk system checksum provides a handle for disk-layer erasure code implementations such as RAID, as was described in Section 2.5.1. Thus the disk system checksum, though superficially redundant, actually turns out to be quite useful.

The end-to-end checksum allows only detecting this class of error. Masking is another matter—it involves a technique called **recovery**, which is one of the topics of the next chapter.

Table 2.6.1 summarizes where failure tolerance is implemented in the several disk layers. The hope is that the remaining untolerated failures are so rare that they can be neglected. If they are not, the number of replicas could be increased until the probability of untolerated failures is negligible.

Table 2.6.1: Summary of disk failure tolerance models. Each entry shows the effect of this error at the interface between the named layer and the next higher layer. With careful design, the probability of the two failures marked with an asterisk should be negligible. Masking of corruption caused by system crashes is discussed in Chapter 3.

	raw layer	fail-fast layer	careful layer	durable layer	more durable layer
soft read, write, or seek error	failure	detected	masked		
hard read, write error	failure	detected	detected	masked	
power failure interrupts a write	failure	detected	detected	masked	
single data decay	failure	detected	detected	masked	
multiple data decay spaced in time	failure	detected	detected	detected	masked
multiple data decay within T_d	failure	detected	detected	detected	failure*
undetectable decay	failure	failure	failure	failure	failure*
system crash corrupts write buffer	failure	failure	failure	failure	detected

2.6.4.9: Still More Threats to Durability

The various procedures described above create storage that is durable in the face of individual disk decay but not in the face of other threats to data integrity. For example, if the power fails in the middle of a `MORE_DURABLE_PUT` , some replicas may contain old versions of the data, some may contain new versions, and some may contain corrupted data, so it is not at all obvious

how MORE_DURABLE_GET should go about meeting its specification. The solution is to make MORE_DURABLE_PUT atomic, which is one of the topics of Chapter 3.

RAID systems usually specify that a successful return from a PUT confirms that writing of all of the mirror replicas was successful. That specification in turn usually requires that the multiple disks be physically co-located, which in turn creates a threat that a single physical disaster—fire, earthquake, flood, civil disturbance, etc.—might damage or destroy all of the replicas.

Since magnetic disks are quite reliable in the short term, a different strategy is to write only one replica at the time that MORE_DURABLE_PUT is invoked and write the remaining replicas at a later time. Assuming there are no inopportune failures in the short run, the results gradually become more durable as more replicas are written. Replica writes that are separated in time are less likely to have replicated failures because they can be separated in physical location, use different disk driver software, or be written to completely different media such as magnetic tape. On the other hand, separating replica writes in time increases the risk of inconsistency among the replicas. Implementing storage that has durability that is substantially beyond that of RAID 1 and MORE_DURABLE_PUT/GET generally involves use of geographically separated replicas and systematic mechanisms to keep those replicas coordinated, a challenge that Chapter 4 discusses in depth.

Perhaps the most serious threat to durability is that although different storage systems have employed each of the failure detection and masking techniques discussed in this section, it is all too common to discover that a typical off-the-shelf personal computer file system has been designed using an overly simple disk failure model and thus misses some—or even many—straightforward failure masking opportunities.

This page titled 2.6: Applying Redundancy to Software and Data is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

2.7: Wrapping Up Reliability

2.7.1: Design Strategies and Design Principles

Standing back from the maze of detail about redundancy, we can identify and abstract three particularly effective design strategies:

- **N-modular redundancy** is a simple but powerful tool for masking failures and increasing availability, and it can be used at any convenient level of granularity.
- **Fail-fast modules** provide a sweeping simplification of the problem of containing errors. When containment can be described simply, reasoning about fault tolerance becomes easier.
- **Pair-and-compare** allows fail-fast modules to be constructed from commercial, off-the-shelf components.

Standing back still further, it is apparent that several general design principles are directly applicable to fault tolerance. In the formulation of the fault-tolerance design process in Section 2.2.2, we invoked *be explicit*, *design for iteration*, *keep digging*, and the *safety margin principle*, and in exploring different fault tolerance techniques we have seen several examples of *adopt sweeping simplifications*. One additional design principle that applies to fault tolerance (and also, as we will see in Chapter 5, to security) comes from experience, as documented in the case studies of Section 2.9:

Avoid rarely used components

Deterioration and corruption accumulate unnoticed—until the next use.

Whereas redundancy can provide masking of errors, redundant components that are used only when failures occur are much more likely to cause trouble than redundant components that are regularly exercised in normal operation. The reason is that failures in regularly exercised components are likely to be immediately noticed and fixed. Failures in unused components may not be noticed until a failure somewhere else happens. But then there are two failures, which may violate the design assumptions of the masking plan. This observation is especially true for software, where rarely-used recovery procedures often accumulate unnoticed bugs and incompatibilities as other parts of the system evolve. The alternative of periodic testing of rarely-used components to lower their failure latency is a band-aid that rarely works well.

In applying these design principles, it is important to consider the threats, the consequences, the environment, and the application. Some faults are more likely than others, some failures are more disruptive than others, and different techniques may be appropriate in different environments. A computer-controlled radiation therapy machine, a deep-space probe, a telephone switch, and an airline reservation system all need fault tolerance, but in quite different forms. The radiation therapy machine should emphasize fault detection and fail-fast design, to avoid injuring patients. Masking faults may actually be a mistake. It is likely to be safer to stop, find their cause, and fix them before continuing operation. The deep-space probe, once the mission begins, needs to concentrate on failure masking to ensure mission success. The telephone switch needs many nines of availability because customers expect to always receive a dial tone, but if it occasionally disconnects one ongoing call, that customer will simply redial without thinking much about it. Users of the airline reservation system might tolerate short gaps in availability, but the durability of its storage system is vital. At the other extreme, most people find that a digital watch has an MTTF that is long compared with the time until the watch is misplaced, becomes obsolete, goes out of style, or is discarded. Consequently, no provision for either error masking or repair is really needed. Some applications have built-in redundancy that a designer can exploit. In a video stream, it is usually possible to mask the loss of a single video frame by just repeating the previous frame.

2.7.2: How about the End-to-End Argument?

There is a potential tension between error masking and an *end-to-end argument*. An end-to-end argument suggests that a subsystem need not do anything about errors and should not do anything that might compromise other goals such as low latency, high throughput, or low cost. The subsystem should instead let the higher layer system of which it is a component take care of the problem, because only the higher layer knows whether or not the error matters and what is the best course of action to take.

There are two counter arguments to that line of reasoning:

- Ignoring an error allows it to propagate, thus contradicting the modularity goal of error containment. This observation points out an important distinction between error detection and error masking. Error detection and containment must be performed where the error happens, so that the error does not propagate wildly. Error masking, in contrast, presents a design choice:

masking can be done locally or the error can be handled by reporting it at the interface (that is, by making the module design fail-fast) and allowing the next higher layer to decide what masking action—if any—to take.

- The lower layer may know the nature of the error well enough that it can mask it far more efficiently than the upper layer. The specialized burst error correction codes used on DVDs come to mind. They are designed specifically to mask errors caused by scratches and dust particles, rather than random bit-flips. So we have a trade-off between the cost of masking the fault locally and the cost of letting the error propagate and handling it in a higher layer.

These two points interact: When an error propagates it can contaminate otherwise correct data, which can increase the cost of masking and perhaps even render masking impossible. The result is that when the cost is small, error masking is usually done locally. (That is assuming that masking is done at all. Many personal computer designs omit memory error masking. Section 2.9.1 discusses some of the reasons for this design decision.)

A closely related observation is that when a lower layer masks a fault it is important that it also report the event to a higher layer, so that the higher layer can keep track of how much masking is going on and thus how much failure tolerance there remains. Reporting to a higher layer is a key aspect of the *safety margin principle*.

2.7.3: A Caution on the Use of Reliability Calculations

Reliability calculations seem to be exceptionally vulnerable to the garbage-in, garbage-out syndrome. It is all too common that calculations of mean time to failure are undermined because the probabilistic models are not supported by good statistics on the failure rate of the components, by measures of the actual load on the system or its components, or by accurate assessment of independence between components.

For computer systems, back-of-the-envelope calculations are often more than sufficient because they are usually at least as accurate as the available input data, which tends to be rendered obsolete by rapid technology change. Numbers predicted by formula can generate a false sense of confidence. This argument is much weaker for technologies that tend to be stable (for example, production lines that manufacture glass bottles). So reliability analysis is not a waste of time, but one must be cautious in applying its methods to computer systems.

2.7.4: Where to Learn More about Reliable Systems

Our treatment of fault tolerance has explored only the first layer of fundamental concepts. There is much more to the subject. For example, we have not considered another class of fault that combines the considerations of fault tolerance with those of security: faults caused by inconsistent, perhaps even malevolent, behavior. These faults have the characteristic that they generate inconsistent error values, possibly error values that are specifically designed by an attacker to confuse or confound fault tolerance measures. These faults are called **Byzantine faults**, recalling the reputation of ancient Byzantium for malicious politics. Here is a typical Byzantine fault: suppose that an evil spirit occupies one of the three replicas of a TMR system, waits for one of the other replicas to fail, and then adjusts its own output to be identical to the incorrect output of the failed replica. A voter accepts this incorrect result and the error propagates beyond the intended containment boundary. In another kind of Byzantine fault, a faulty replica in an NMR system sends different result values to each of the voters that are monitoring its output. Malevolence is not required—any fault that is not anticipated by a fault detection mechanism can produce Byzantine behavior. There has recently been considerable attention to techniques that can tolerate Byzantine faults. Because the tolerance algorithms can be quite complex, we defer the topic to advanced study.

We also have not explored the full range of reliability techniques that one might encounter in practice. For an example that has not yet been mentioned, Sidebar 2.7.1 describes the **heartbeat**, a popular technique for detecting failures of active processes.

Sidebar 2.7.1

Detecting failures with heartbeats

An activity such as a Web server is usually intended to keep running indefinitely. If it fails (perhaps by crashing) its clients may notice that it has stopped responding, but clients are not typically in a position to restart the server. Something more systematic is needed to detect the failure and initiate recovery. One helpful technique is to program the thread that should be performing the activity to send a periodic signal to another thread (or a message to a monitoring service) that says, in

effect, "I'm still OK". The periodic signal is known as a **heartbeat** and the observing thread or service is known as a **watchdog**.

The watchdog service sets a timer, and on receipt of a heartbeat message it restarts the timer. If the timer ever expires, the watchdog assumes that the monitored service has gotten into trouble and it initiates recovery. One limitation of this technique is that if the monitored service fails in such a way that the only thing it does is send heartbeat signals, the failure will go undetected.

As with all fixed timers, choosing a good heartbeat interval is an engineering challenge. Setting the interval too short wastes resources sending and responding to heartbeat signals. Setting the interval too long delays detection of failures. Since detection is a prerequisite to repair, a long heartbeat interval increases MTTR and thus reduces availability.

This chapter has oversimplified some ideas. For example, the definition of availability proposed in Section 2.3 of this chapter is too simple to adequately characterize many large systems. If a bank has hundreds of automatic teller machines, there will probably always be a few teller machines that are not working at any instant. For this case, an availability measure based on the percentage of transactions completed within a specified response time would probably be more appropriate.

A rapidly moving but in-depth discussion of fault tolerance can be found in Chapter 3 of the book *Transaction Processing: Concepts and Techniques*, by Jim Gray and Andreas Reuter. A broader treatment, with case studies, can be found in the book *Reliable Computer Systems: Design and Evaluation*, by Daniel P. Siewiorek and Robert S. Swarz. Byzantine faults are an area of ongoing research and development, and the best source is current professional literature.

This chapter has concentrated on general techniques for achieving reliability that are applicable to hardware, software, and complete systems. Looking ahead, Chapter 3 and Chapter 4 revisit reliability in the context of specific software techniques that permit reconstruction of stored state following a failure when there are several concurrent activities. Chapter 5, on securing systems against malicious attack, introduces a redundancy scheme known as defense in depth that can help both to contain and to mask errors in the design or implementation of individual security mechanisms.

This page titled 2.7: Wrapping Up Reliability is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

2.8: Application - A Fault Tolerance Model for CMOS RAM

This section develops a fault tolerance model for words of CMOS random access memory, first without and then with a simple error-correction code, comparing the probability of error in the two cases.

CMOS RAM is both low in cost and extraordinarily reliable, so much so that error masking is often not implemented in mass production systems such as television sets and personal computers. But some systems, for example life-support, air traffic control, or banking systems, cannot afford to take unnecessary risks. Such systems usually employ the same low-cost memory technology but add incremental redundancy.

A common failure of CMOS RAM is that noise intermittently causes a single bit to read or write incorrectly. If intermittent noise affected only reads, then it might be sufficient to detect the error and retry the read. But the possibility of errors on writes suggests using a forward error-correction code.

We start with a fault tolerance model that applies when reading a word from memory without error correction. The model assumes that errors in different bits are independent and it assigns p as the (presumably small) probability that any individual bit is in error. The notation $\mathrm{O}(p^n)$ means terms involving p^n and higher, presumably negligible, powers. Here are the possibilities and their associated probabilities:

Table 2.8.1: Fault tolerance model for raw CMOS random access memory

			Probability
error-free case:		all 32 bits are correct	$(1-p)^{32} = 1 - \mathrm{O}(p)$
errors:			
	untolerated:	one bit is in error	$32p(1-p)^{31} = \mathrm{O}(p)$
	untolerated:	two bits are in error	$(31 \cdot 32/2)p^2(1-p)^{30} = \mathrm{O}(p^2)$
	untolerated:	three or more bits are in error	$(30 \cdot 31 \cdot 32/3 \cdot 2)p^3(1-p)^{29} + \cdots + p^$

The coefficients 32, $(31 \cdot 32)/2$, etc. arise by counting the number of ways that one, two, etc., bits could be in error.

Suppose now that the 32-bit block of memory is encoded using a code of Hamming distance 3, as described in Section 2.5.1. Such a code allows any single-bit error to be corrected and any double-bit error to be detected. After applying the decoding algorithm, the fault tolerance model changes to:

Table 2.8.2: Fault tolerance model for CMOS memory with error correction

			Probability
error-free case:		all 32 bits are correct	$(1-p)^{32} = 1 - \mathrm{O}(p)$
errors:			
	tolerated:	one bit is in error	$32p(1-p)^{31} = \mathrm{O}(p)$
	detected:	two bits are in error	$(31 \cdot 32/2)p^2(1-p)^{30} = \mathrm{O}(p^2)$
	untolerated:	three or more bits are in error	$(30 \cdot 31 \cdot 32/3 \cdot 2)p^3(1-p)^{29} + \cdots + p^$

The interesting change is in the probability that the decoded value is correct. That probability is the sum of the probabilities that there were no errors and that there was one, tolerated error:

$$\begin{aligned} Prob(\text{decoded value is correct}) &= (1-p)^{32} + 32p(1-p)^{31} \\ &= (1 - 32p + (31 \cdot 32/2)p^2 + \ldots) + (32p + (32 \cdot -31)p^2 + \ldots \\ &= (1 - \mathrm{O}(p^2)) \end{aligned}$$

The decoding algorithm has thus eliminated the errors that have probability of order p. It has not eliminated the two-bit errors, which have probability of order p^2, but for two-bit errors the algorithm is fail-fast, so a higher-level procedure has an opportunity to recover, perhaps by requesting retransmission of the data. The code is not helpful if there are errors in three or more bits, which situation has probability of order p^3, but presumably the designer has determined that probabilities of that order are negligible. If they are not, the designer should adopt a more powerful error-correction code.

With this model in mind, one can review the two design questions suggested in Section 2.4.2. The first question is whether the estimate of bit error probability is realistic and if it is realistic to suppose that multiple bit errors are statistically independent of one another. (Error independence appeared in the analysis in the claim that the probability of an n-bit error has the order of the nth power of the probability of a one-bit error.) Those questions concern the real world and the accuracy of the designer's model of it. For example, this failure model doesn't consider power failures, which might take all the bits out at once, or a driver logic error that might take out all of the even-numbered bits. It also ignores the possibility of faults that lead to errors in the logic of the error-correction circuitry itself.

The second question is whether the coding algorithm actually corrects all one-bit errors and detects all two-bit errors. That question is explored by examining the mathematical structure of the error-correction code and is quite independent of anybody's estimate or measurement of real-world

failure types and rates. There are many off-the-shelf coding algorithms that have been thoroughly analyzed and for which the answer to this question is yes.

This page titled 2.8: Application - A Fault Tolerance Model for CMOS RAM is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

2.9: War Stories - Fault-Tolerant Systems That Failed

2.9.1: Adventures with Error Correction[1]

The designers of the computer systems at the Xerox Palo Alto Research Center in the early 1970s encountered a series of experiences with error-detecting and error-correcting memory systems. From these experiences follow several lessons, some of which are far from intuitive, and all of which still apply several decades later.

MAXC. One of the first projects undertaken in the newly-created Computer Systems Laboratory was to build a time-sharing computer system, named MAXC. A brand new 1024-bit memory chip, the Intel 1103, had just appeared on the market, and it promised to be a compact and economical choice for the main memory of the computer. But since the new chip had unknown reliability characteristics, the MAXC designers implemented the memory system using a few extra bits for each 36-bit word, in the form of a single-error-correction, double-error-detection code.

Experience with the memory in MAXC was favorable. The memory was solidly reliable—so solid that no errors in the memory system were ever reported.

The Alto. When the time came to design the Alto personal workstation, the same Intel memory chips still appeared to be the preferred component. Because these chips had performed so reliably in MAXC, the designers of the Alto memory decided to relax a little, omitting error correction. However, they were still conservative enough to provide error detection, in the form of one parity bit for each 16-bit word of memory.

This design choice seemed to be a winner because the Alto memory systems also performed flawlessly, at least for the first several months. Then, mysteriously, the operating system began to report frequent memory-parity failures.

Some background: the Alto started life with an operating system and applications that used a simple typewriter-style interface. The display was managed with a character-by-character teletype emulator. But the purpose of the Alto was to experiment with better things. One of the first steps in that direction was to implement the first what-you-seeis-what-you-get editor, named Bravo. Bravo took full advantage of the bit-map display, filling it not only with text, but also with lines, buttons, and icons. About half the memory system was devoted to display memory. Curiously, the installation of Bravo coincided with the onset of memory parity errors.

It turned out that the Intel 1103 chips were pattern-sensitive—certain read/write sequences of particular bit patterns could cause trouble, probably because those pattern sequences created noise levels somewhere on the chip that systematically exceeded some critical threshold. The Bravo editor's display management was the first application that generated enough different patterns to have an appreciable probability of causing a parity error. It did so, frequently.

Lesson 2.9.1*a*:

There is no such thing as a small change in a large system. A new piece of software can bring down a piece of hardware that is thought to be working perfectly. You are never quite sure just how close to the edge of the cliff you are standing.

Lesson 2.9.1*b*:

Experience is a primary source of information about failures. It is nearly impossible, without specific prior experience, to predict what kinds of failures you will encounter in the field.

Back to MAXC. This circumstance led to a more careful review of the situation on MAXC. MAXC, being a heavily used server, would be expected to encounter at least some of this pattern sensitivity. It was discovered that although the error-correction circuits had been designed to report both corrected errors and uncorrectable errors, the software logged only uncorrectable errors and corrected errors were being ignored. When logging of corrected errors was implemented, it turned out that the MAXC's Intel 1103's were actually failing occasionally, and the error-correction circuitry was busily setting things right.

Lesson 2.9.1*c*:

Whenever systems implement automatic error masking, it is important to follow the *safety margin principle*, by tracking how often errors are successfully masked. Without this information, one has no way of knowing whether the system is operating

with a large or small safety margin for additional errors. Otherwise, despite the attempt to put some guaranteed space between yourself and the edge of the cliff, you may be standing on the edge again.

The Alto 2. In 1975, it was time to design a follow-on workstation, the Alto 2. A new generation of memory chips, this time with 4096 bits, was now available. Since it took up much less space and promised to be cheaper, this new chip looked attractive, but again there was no experience with its reliability. The Alto 2 designers, having been made wary by the pattern sensitivity of the previous generation chips, again resorted to a single-error-correction, double-error-detection code in the memory system.

Once again, the memory system performed flawlessly. The cards passed their acceptance tests and went into service. In service, not only were no double-bit errors detected, only rarely were single-bit errors being corrected. The initial conclusion was that the chip vendors had worked the bugs out and these chips were really good.

About two years later, someone discovered an implementation mistake. In one quadrant of each memory card, neither error correction nor error detection was actually working. All computations done using memory in the misimplemented quadrant were completely unprotected from memory errors.

Lesson 2.9.1*d*:

Never assume that the hardware actually does what it says in the specifications.

Lesson 2.9.1*e*:

It is harder than it looks to test the fault tolerance features of a fault-tolerant system.

One might conclude that the intrinsic memory chip reliability had improved substantially—so much that it was no longer necessary to take heroic measures to achieve system reliability. Certainly the chips were better, but they weren't perfect. The other effect here is that errors often don't lead to failures. In particular, a wrong bit retrieved from memory does not necessarily lead to an observed failure. In many cases a wrong bit doesn't matter; in other cases it does but no one notices; in still other cases, the failure is blamed on something else.

Lesson 2.9.1*f*:

Just because it seems to be working doesn't mean that it actually is.

The bottom line. One of the designers of MAXC and the Altos, Butler Lampson, suggests that the possibility that a failure is blamed on something else can be viewed as an opportunity, and it may be one of the reasons that PC manufacturers often do not provide memory parity checking hardware. First, the chips are good enough that errors are rare. Second, if you provide parity checks, consider who will be blamed when the parity circuits report trouble: the hardware vendor. Omitting the parity checks probably leads to occasional random behavior, but occasional random behavior is indistinguishable from software error and is usually blamed on the software.

Lesson 2.9.1*g*:

In Lampson's words: "Beauty is in the eye of the beholder. The various parties involved in the decisions about how much failure detection and recovery to implement do not always have the same interests."

[1] These experiences were reported by Butler Lampson, one of the designers of the MAXC computer and the Alto personal workstations at Xerox Palo Alto Research Center.

2.9.2: Risks of Rarely-Used Procedures: The National Archives

The National Archives and Record Administration of the United States government has the responsibility, among other things, of advising the rest of the government how to preserve electronic records such as e-mail messages for posterity. Quite separate from that responsibility, the organization also operates an e-mail system at its Washington, D.C. headquarters for a staff of about 125

people; about 10,000 messages a month pass through this system. To ensure that no messages are lost, it arranged with an outside contractor to perform daily incremental backups and to make periodic complete backups of its e-mail files. On the chance that something may go wrong, the system has audit logs that track actions regarding incoming and outgoing mail as well as maintenance on files.

Over the weekend of June 18–21, 1999, the e-mail records for the previous four months (an estimated 43,000 messages) disappeared. No one has any idea what went wrong—the files may have been deleted by a disgruntled employee or a runaway housecleaning program, or the loss may have been caused by a wayward system bug. In any case, on Monday morning when people came to work, they found that the files were missing.

On investigation, the system managers reported that the audit logs had been turned off because they were reducing system performance, so there were no clues available to diagnose what went wrong. Moreover, since the contractor's employees had never gotten around to actually performing the backup part of the contract, there were no backup copies. It had not occurred to the staff of the Archives to verify the existence of the backup copies, much less to test them to see if they could actually be restored. They assumed that since the contract required it, the work was being done.

The contractor's project manager and the employee responsible for making backups were immediately replaced. The Assistant Archivist reports that backup systems have now been beefed up to guard against another mishap, but he added that the safest way to save important messages is to print them out.[2]

Lesson 2.9.2:

Avoid rarely used components. Rarely used failure-tolerance mechanisms, such as restoration from backup copies, must be tested periodically. If they are not, there is not much chance that they will work when an emergency arises. Fire drills (in this case, performing a restoration of all files from a backup copy) seem disruptive and expensive, but they are not nearly as disruptive and expensive as the discovery, too late, that the backup system isn't really operating. Even better, design the system so that all the components are exposed to day-to-day use, so that failures can be noticed before they cause real trouble.

[2] "Archives Loses 43,000 E-Mails; officials can't explain summer erasure; backup system failed." *The Washington Post*, Thursday, January 6, 2000, page A17.

2.9.3: Non-independent Replicas and Backhoe Fade

In Eagan, Minnesota, Northwest airlines operated a computer system named WorldFlight that managed the Northwest flight dispatching database, provided weight-and-balance calculations for pilots, and managed e-mail communications between the dispatch center and all Northwest airplanes. It also provided data to other systems that managed passenger check-in and the airline's Web site. Since many of these functions involved communications, Northwest contracted with U.S. West, the local telephone company at that time, to provide these communications in the form of fiber-optic links to airports that Northwest serves, to government agencies such as the Weather Bureau and the Federal Aviation Administration, and to the Internet. Because these links were vital, Northwest paid U.S. West extra to provide each primary link with a backup secondary link. If a primary link to a site failed, the network control computers automatically switched over to the secondary link to that site.

At 2:05 p.m. on March 23, 2000, all communications to and from WorldFlight dropped out simultaneously. A contractor who was boring a tunnel (for fiber optic lines for a different telephone company) at the nearby intersection of Lone Oak and Pilot Knob roads accidentally bored through a conduit containing six cables carrying the U.S. West fiber-optic and copper lines. In a tongue-in-cheek analogy to the fading in and out of long-distance radio signals, this kind of communications disruption is known in the trade as "backhoe fade." WorldFlight immediately switched from the primary links to the secondary links, only to find that they were not working, either. It seems that the primary and secondary links were routed through the same conduit, and both were severed.

Pilots resorted to manual procedures for calculating weight and balance, and radio links were used by flight dispatchers in place of the electronic message system, but about 125 of Northwest's 1700 flights had to be canceled because of the disruption, about the same number that are canceled when a major snowstorm hits one of Northwest's hubs. Much of the ensuing media coverage concentrated on whether or not the contractor had followed "dig-safe" procedures that are intended to prevent such mistakes. But a news release from Northwest at 5:15 p.m. blamed the problem entirely on U.S. West. "For such contingencies, U.S. West provides to Northwest a complete redundancy plan. The U.S. West redundancy plan also failed."[3]

In a similar incident, the ARPAnet, a predecessor to the Internet, had seven separate trunk lines connecting routers in New England to routers elsewhere in the United States. All the trunk lines were purchased from a single long-distance carrier, AT&T. On December 12, 1986, all seven trunk lines went down simultaneously when a contractor accidentally severed a single fiber-optic cable running from White Plains, New York to Newark, New Jersey.[4]

A complication for communications customers who recognize this problem and request information about the physical location of their communication links is that in the name of security, communications companies sometimes refuse to reveal it.

Lesson 2.9.3:

The calculation of mean time to failure of a redundant system depends critically on the assumption that failures of the replicas are independent. If they aren't independent, then the replication may be a waste of effort and money, while producing a false complacency. This incident also illustrates why it can be difficult to test fault tolerance measures properly. What appears to be redundancy at one level of abstraction turns out not to be redundant at a lower level of abstraction.

---

[3] Tony Kennedy. "Cut cable causes cancellations, delays for Northwest Airlines." *Minneapolis Star Tribune*, March 22, 2000.

[4] Peter G. Neumann. *Computer Related Risks* (Addison-Wesley, New York, 1995), page 14.

2.9.4: Human Error May Be the Biggest Risk

Telehouse was an East London "telecommunications hotel," a seven-story building housing communications equipment for about 100 customers, including most British Internet companies, many British and international telephone companies, and dozens of financial institutions. It was designed to be one of the most secure buildings in Europe, safe against "fire, flooding, bombs, and sabotage." Accordingly, Telehouse had extensive protection against power failure, including two independent connections to the national electric power grid, a room full of batteries, and two diesel generators, along with systems to detect failures in supply and automatically cut over from one backup system to the next, as needed.

On May 8, 1997, all the computer systems went off line for lack of power. According to Robert Bannington, financial director of Telehouse, "It was due to human error." That is, someone pulled the wrong switch. The automatic power supply cutover procedures did not trigger because they were designed to deploy on failure of the outside power supply, and the sensors correctly observed that the outside power supply was intact.[5]

Lesson 2.9.4*a*:

The first step in designing a fault-tolerant system is to identify each potential fault and evaluate the risk that it will happen. People are part of the system, and mistakes made by authorized operators are typically a bigger threat to reliability than trees falling on power lines.

Anecdotes concerning failures of backup power supply systems seem to be common. Here is a typical report of an experience in a Newark, New Jersey, hospital operating room that was equipped with three backup generators: "On August 14, 2003, at 4:10 pm EST, a widespread power grid failure caused our hospital to suffer a total OR power loss, regaining partial power in 4 hours and total restoration 12 hours later... When the backup generators initially came on-line, all ORs were running as usual. Within 20 minutes, one parallel-linked generator caught fire from an oil leak. After being subjected to twice its rated load, the second in-line generator quickly shut down... Hospital engineering, attempting load-reduction to the single surviving generator, switched many hospital circuit breakers off. Main power was interrupted to the OR."[6]

Lesson 2.9.4*b*:

A backup generator is another example of a rarely used component that may not have been maintained properly. The last two sentences of that report reemphasize Lesson 2.9.4*a*.

For yet another example, the M.I.T. Information Services and Technology staff posted the following system services notice on April 2, 2004: "We suffered a power failure in W92 shortly before 11 AM this morning. Most services should be restored now, but some are still being recovered. Please check back here for more information as it becomes available." A later posting reported:

"Shortly after 10 AM Friday morning the routine test of the W92 backup generator was started. Unknown to us was that the transition of the computer room load from commercial power to the backup generator resulted in a power surge within the computer room's Uninterruptable [sic] Power Supply (UPS). This destroyed an internal surge protector, which started to smolder. Shortly before 11 AM the smoldering protector triggered the VESDA® smoke sensing system within the computer room. This sensor triggered the fire alarm, and as a safety precaution forced an emergency power down of the entire computer room."[7]

Lesson 2.9.4*c*:

A failure masking system not only can fail, it can cause a bigger failure than the one it is intended to mask.

---

[5] Robert Uhlig. "Engineer pulls plug on secure bunker." *Electronic Telegraph*, (9 May 1997).

[6] Ian E. Kirk, M.D. and Peter L. Fine, M.D. "Operating by Flashlight: Power Failure and Safety Lessons from the August, 2003 Blackout." *Abstracts of the Annual Meeting of the American Society of Anesthesiologists*, October 2005.

[7] Private internal communication.

2.9.5: Introducing a Single Point of Failure

"[Rabbi Israel Meir HaCohen Kagan described] a real-life situation in his town of Radin, Poland. He lived at the time when the town first purchased an electrical generator and wired all the houses and courtyards with electric lighting. One evening something broke within the machine, and darkness descended upon all of the houses and streets, and even in the synagogue.

"So he pointed out that before they had electricity, every house had a kerosene light—and if in one particular house the kerosene ran out, or the wick burnt away, or the glass broke, that only that one house would be dark. But when everyone is dependent upon one machine, darkness spreads over the entire city if it breaks for any reason."[8]

Lesson 2.9.5:

Centralization may provide economies of scale, but it can also reduce robustness—a single failure can interfere with many unrelated activities. This phenomenon is commonly known as introducing a single point of failure. By carefully adding redundancy to a centralized design one may be able to restore some of the lost robustness, but it takes planning and adds to the cost.

---

[8] Chofetz Chaim (the Rabbi Israel Meir HaCohen Kagan of Radin), paraphrased by Rabbi Yaakov Menken, in a discussion of lessons from the Torah in *Project Genesis Lifeline*. http://www.torah.org/learning/lifeline/5758/reeh.html. Suggested by David Karger.

2.9.6: Multiple Failures: The SOHO Mission Interruption

"Contact with the SOlar Heliospheric Observatory (SOHO) spacecraft was lost in the early morning hours of June 25, 1998, Eastern Daylight Time (EDT), during a planned period of calibrations, maneuvers, and spacecraft reconfigurations. Prior to this the SOHO operations team had concluded two years of extremely successful science operations.

"…The Board finds that the loss of the SOHO spacecraft was a direct result of operational errors, a failure to adequately monitor spacecraft status, and an erroneous decision which disabled part of the on-board autonomous failure detection. Further, following the occurrence of the emergency situation, the Board finds that insufficient time was taken by the operations team to fully assess the spacecraft status prior to initiating recovery operations. The Board discovered that a number of factors contributed to the circumstances that allowed the direct causes to occur."[9]

In a tour-de-force of the *keep digging principle*, the report of the investigating board quoted above identified five distinct direct causes of the loss: two software errors, a design feature that unintentionally amplified the effect of one of the software errors, an incorrect diagnosis by the ground staff, and a violated design assumption. It then goes on to identify three indirect causes in the spacecraft design process: lack of change control, missing risk analysis for changes, and insufficient communication of changes,

and then three indirect causes in operations procedures: failure to follow planned procedures, to evaluate secondary telemetry data, and to question telemetry discrepancies.

Lesson 2.9.6:

Complex systems fail for complex reasons. In systems engineered for reliability, it usually takes several component failures to cause a system failure. Unfortunately, when some of the components are people, multiple failures are all too common.

[9] Massimo Trella and Michael Greenfield. *Final Report of the SOHO Mission Interruption Joint NASA/ESA Investigation Board* (August 31, 1998). National Aeronautics and Space Administration and European Space Agency. http://sohowww.nascom.nasa.gov/whatsnew/ SOHO_final_report.html.

This page titled 2.9: War Stories - Fault-Tolerant Systems That Failed is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

2.10: Exercises

Exercise 2.10.1

1999–3–01

Failures are

A. Faults that are latent.

B. Errors that are contained within a module.

C. Errors that propagate out of a module.

D. Faults that turn into errors.

Exercise 2.10.2

2006–2–3

Ben Bitdiddle has been asked to perform a deterministic computation to calculate the orbit of a near-Earth asteroid for the next 500 years, to find out whether or not the asteroid will hit the Earth. The calculation will take roughly two years to complete, and Ben wants be be sure that the result will be correct. He buys 30 identical computers and runs the same program with the same inputs on all of them. Once each hour the software pauses long enough to write all intermediate results to a hard disk on that computer. When the computers return their results at the end of the two years, a voter selects the majority answer. Which of the following failures can this scheme tolerate, assuming the voter works correctly?

A. The software carrying out the deterministic computation has a bug in it, causing the program to compute the wrong answer for certain inputs.

B. Over the course of the two years, cosmic rays corrupt data stored in memory at twelve of the computers, causing them to return incorrect results.

C. Over the course of the two years, on 24 different days the power fails in the computer room. When the power comes back on, each computer reboots and then continues its computation, starting with the state it finds on its hard disk.

Exercise 2.10.3

1997–0–01

Ben Bitdiddle has seven smoke detectors installed in various places in his house. Since the fire department charges $100 for responding to a false alarm, Ben has connected the outputs of the smoke detectors to a simple majority voter, which in turn can activate an automatic dialer that calls the fire department. Ben returns home one day to find his house on fire, and the fire department has not been called. There is smoke at every smoke detector. What did Ben do wrong?

A. He should have used fail-fast smoke detectors.

B. He should have used a voter that ignores failed inputs from fail-fast sources.

C. He should have used a voter that ignores non-active inputs.

D. He should have done both A and B.

E. He should have done both A and C.

Exercise 2.10.4

2005–2–5

You will be flying home from a job interview in Silicon Valley. Your travel agent gives you the following choice of flights:

A. Flight A uses a plane whose mean time to failure (MTTF) is believed to be 6,000 hours. With this plane, the flight is scheduled to take 6 hours.

B. Flight B uses a plane whose MTTF is believed to be 5,000 hours. With this plane, the flight takes 5 hours.

The agent assures you that both planes' failures occur according to memoryless random processes (not a "bathtub" curve). Assuming that model, which flight should you choose to minimize the chance of your plane failing during the flight?

Exercise 2.10.5

1995–3–1a

(Note: solving this problem is best done with use of probability through the level of Markov chains.) You are designing a computer system to control the power grid for the Northeastern United States. If your system goes down, the lights go out and civil disorder—riots, looting, fires, etc.—will ensue. Thus, you have set a goal of having a system MTTF of at least 100 years (about 10^6 hours). For hardware you are constrained to use a building block computer that has a MTTF of 1000 hours and a MTTR of 1 hour. Assuming that the building blocks are fail-fast, memoryless, and fail independently of one another, how can you arrange to meet your goal?

Exercise 2.10.6

1985–0–1

The town council wants to implement a municipal network to connect the local area networks in the library, the town hall, and the school. They want to minimize the chance that any building is completely disconnected from the others. They are considering two network topologies:

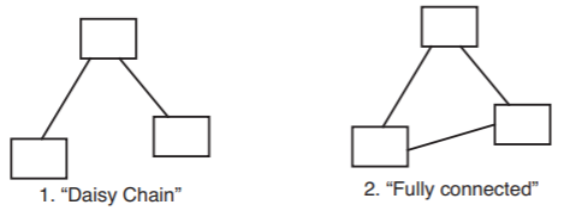


Figure 2.10.1: "Daisy chain" vs fully connected network topologies, each connecting three local area networks.

Each link in the network has a failure probability of p.

a) What is the probability that the daisy chain network is connecting all the buildings?

b) What is the probability that the fully connected network is connecting all the buildings?

c) The town council has a limited budget, with which it can buy either a daisy chain network with two high reliability links $(p = .000001)$, or a fully connected network with three low-reliability links $(p = .0001)$. Which should they purchase?

Exercise 2.10.7

2001–3–05

Figure 2.5.8 shows the failure points of three different 5MR supermodule designs, if repair does not happen in time. Draw the corresponding figure for the same three different TMR supermodule designs.

Exercise 2.10.8

1976–0–3

An astronomer calculating the trajectory of Pluto has a program that requires the execution of 10^{13} machine operations. The fastest processor available in the lab runs only 10^9 operations per second and, unfortunately, has a probability of 10^{-12} of failing on any one operation. (The failure process is memoryless.) The good news is that the processor is fail-fast, so when a failure occurs it stops dead in its tracks and starts ringing a bell. The bad news is that when it fails, it loses all state, so whatever it was doing is lost, and has to be started over from the beginning.

Seeing that in practical terms, the program needs to run for about 3 hours, and the machine has an MTTF of only 1/10 of that time, Louis Reasoner and Ben Bitdiddle have proposed two ways to organize the computation:

- Louis says run it from the beginning and hope for the best. If the machine fails, just try again; keep trying until the calculation successfully completes.
- Ben suggests dividing the calculation into ten equal-length segments; if the calculation gets to the end of a segment, it writes its state out to the disk. When a failure occurs, restart from the last state saved on the disk.

Saving state and restart both take zero time. What is the ratio of the expected time to complete the calculation under the two strategies?

Warning: A straightforward solution to this problem involves advanced probability techniques.

Exercise 2.10.9

2005–0–1

Draw a figure, similar to that of Figure 2.5.3, that shows the recovery procedure for one sector of a 5-disk RAID 4 system when disk 2 fails and is replaced.

Exercise 2.10.10

1997–3–01, 2000–3–01, 1998–3–01, 2000–3–01

Louis Reasoner has just read an advertisement for a RAID controller that provides a choice of two configurations. According to the advertisement, the first configuration is exactly the RAID 4 system described in Section 2.5.1. The advertisement goes on to say that the configuration called RAID 5 has just one difference: in an N-disk configuration, the parity block, rather than being written on disk N, is written on the disk number $(1+$ `sector_address` modulo $N)$. Thus, for example, in a five-disk system, the parity block for sector 18 would be on disk 4 (because $1 + (18 \text{ modulo } 5) = 4$), while the parity block for sector 19 would be on disk 5 (because $1 + (19 \text{ modulo } 5) = 5$). Louis is hoping you can help him understand why this idea might be a good one.

a) RAID 5 has the advantage over RAID 4 that

A. It tolerates single-drive failures.

B. Read performance in the absence of errors is enhanced.

C. Write performance in the absence of errors is enhanced.

D. Locating data on the drives is easier.

E. Allocating space on the drives is easier.

F. It requires less disk space.

G. There's no real advantage; it's just another advertising gimmick.

b) Is there any workload for which RAID 4 has better write performance than RAID 5?

c) Louis is also wondering about whether he might be better off using a RAID 1 system (see Section 2.6.4.6). How does the number of disks required compare between RAID 1 and RAID 5?

d) Out of RAID 1 and RAID 5, which has better performance for a workload consisting of small reads and small writes?

Exercise 2.10.11

2002–3–01

A system administrator notices that a file service disk is failing for two unrelated reasons. Once every 30 days, on average, vibration due to nearby construction breaks the disk's arm. Once every 60 days, on average, a power surge destroys the disk's electronics. The system administrator fixes the disk instantly each time it fails. The two failure modes are independent of each other, and independent of the age of the disk. What is the mean time to failure of the disk?

This page titled 2.10: Exercises is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

CHAPTER OVERVIEW

3: Atomicity - All-or-nothing and Before-or-after

3.1: Overview
3.2: Atomicity
3.3: All-Or-Nothing Atomicity I - Concepts
3.4: All-Or-Nothing Atomicity II - Pragmatics
3.5: Before-Or-After Atomicity I - Concepts
3.6: Before-Or-After Atomicity II - Pragmatics
3.7: Atomicity Across Layers and Multiple Sites
3.8: A More Complete Model of Disk Failure (Advanced Topic)
3.9: Case Studies - Machine Language Atomicity
3.10: Exercises

This page titled 3: Atomicity - All-or-nothing and Before-or-after is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

3.1: Overview

This chapter explores two closely related system engineering design strategies. The first is **all-or-nothing atomicity**, a design strategy for masking failures that occur while interpreting programs. The second is **before-or-after atomicity**, a design strategy for coordinating concurrent activities. Chapter 2 introduced failure masking, but did not show how to mask failures of running programs. This chapter explores ways to systematically synthesize a design that provides both the all-or-nothing property needed for failure masking and the before-or-after property needed for coordination.

Many useful applications can benefit from atomicity. For example, suppose that you are trying to buy a toaster from an Internet store. You click on the button that says "purchase," but before you receive a response the power fails. You would like to have some assurance that, despite the power failure, either the purchase went through properly or that nothing happened at all. You don't want to find out later that your credit card was charged but the Internet store didn't receive word that it was supposed to ship the toaster. In other words, you would like to see that the action initiated by the "purchase" button be all-or-nothing despite the possibility of failure. And if the store has only one toaster in stock and two customers both click on the "purchase" button for a toaster at about the same time, one of the customers should receive a confirmation of the purchase, and the other should receive a "sorry, out of stock" notice. It would be problematic if both customers received confirmations of purchase. In other words, both customers would like to see that the activity initiated by their own click of the "purchase" button occur either completely before or completely after any other, concurrent click of a "purchase" button.

The single conceptual framework of atomicity provides a powerful way of thinking about both all-or-nothing failure masking and before-or-after sequencing of concurrent activities. **Atomicity** is the performing of a sequence of steps, called **actions**, so that they appear to be done as a single, indivisible step, known in operating system and architecture literature as an **atomic action** and in database management literature as a **transaction**. When a fault causes a failure in the middle of a correctly designed atomic action, it will appear to the invoker of the atomic action that the atomic action either completed successfully or did nothing at all—thus an atomic action provides all-or-nothing atomicity. Similarly, when several atomic actions are going on concurrently, each atomic action will appear to take place either completely before or completely after every other atomic action—thus an atomic action provides before-or-after atomicity. Together, all-or-nothing atomicity and before-or-after atomicity provide a particularly strong form of modularity: they hide the fact that the atomic action is actually composed of multiple steps.

The result is a *sweeping simplification* in the description of the possible states of a system. This simplification provides the basis for a methodical approach to recovery from failures and coordination of concurrent activities that simplifies design, simplifies understanding for later maintainers, and simplifies verification of correctness. These desiderata are particularly important because errors caused by mistakes in coordination usually depend on the relative timing of external events and among different threads. When a timing-dependent error occurs, the difficulty of discovering and diagnosing it can be orders of magnitude greater than that of finding a mistake in a purely sequential activity. The reason is that even a small number of concurrent activities can have a very large number of potential real time sequences. It is usually impossible to determine which of those many potential sequences of steps preceded the error, so it is effectively impossible to reproduce the error under more carefully controlled circumstances. Since debugging this class of error is so hard, techniques that ensure correct coordination *a priori* are particularly valuable.

The remarkable thing is that the same systematic approach of atomicity to failure recovery also applies to coordination of concurrent activities. In fact, since one must be able to deal with failures while at the same time coordinating concurrent activities, any attempt to use different strategies for these two problems requires that the strategies be compatible. Being able to use the same strategy for both is another *sweeping simplification*.

Atomic actions are a fundamental building block that is widely applicable in computer system design. Atomic actions are found in database management systems, in register management for pipelined processors, in file systems, in change-control systems used for program development, and in many everyday applications such as word processors and calendar managers.

Sidebar 3.1.1

Actions and transactions

The terminology used by system designers to discuss atomicity can be confusing because the concept was identified and developed independently by database designers and by hardware architects.

An action that changes several data values can have any or all of at least four independent properties: it can be **all-or-nothing** (either all or none of the changes happen), it can be **before-or-after** (the changes all happen either before or after

every concurrent action), it can be **constraint-maintaining** (the changes maintain some specified invariant), and it can be **durable** (the changes last as long as they are needed).

Designers of database management systems customarily are concerned only with actions that are both all-or-nothing and before-or-after, and they describe such actions as **transactions**. In addition, they use the term **atomic** primarily in reference to all-or-nothing atomicity. On the other hand, hardware processor architects customarily use the term **atomic** to describe an action that exhibits before-or-after atomicity.

This book does not attempt to change these common usages. Instead, it uses the qualified terms "all-or-nothing atomicity" and "before-or-after atomicity." The unqualified term "atomic" may imply all-or-nothing, or before-or-after, or both, depending on the context. The text uses the term "transaction" to mean an action that is both all-or-nothing and before-or-after.

All-or-nothing atomicity and before-or-after atomicity are universally defined properties of actions, while constraints are properties that different applications define in different ways. Durability lies somewhere in between because different applications have different durability requirements. At the same time, implementations of constraints and durability usually have a prerequisite of atomicity. Since the atomicity properties are modularly separable from the other two, this chapter focuses just on atomicity. Chapter 4 then explores how a designer can use transactions to implement constraints and enhance durability.

The sections of this chapter define atomicity, examine some examples of atomic actions, and explore systematic ways of achieving atomicity: **version histories**, **logging**, and **locking protocols**. Chapter 4 then explores some applications of atomicity. Case studies at the end of both chapters provide real-world examples of atomicity as a tool for creating useful systems.

This page titled 3.1: Overview is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

3.2: Atomicity

Atomicity is a property required in several different areas of computer system design. These areas include managing a database, developing a hardware architecture, specifying the interface to an operating system, and more generally in software engineering. The table below suggests some of the kinds of problems to which atomicity is applicable. In this chapter we will encounter examples of both kinds of atomicity in each of these different areas.

Table 3.2.1: Some problems in computer system design where atomicity is applicable.

Area	All-or-nothing Atomicity	Before-and-after Atomicity
database management	updating more than one record	records shared between threads
hardware architecture	handling interrupts and exceptions	register renaming
operating systems	supervisor call interface	printer queue
software engineering	handling faults in layers	bounded buffer

3.2.1: All-or-Nothing Atomicity in a Database

As a first example, consider a database of bank accounts. We define a procedure named `TRANSFER` that debits one account and credits a second account, both of which are stored on disk, as follows:

```
1     procedure TRANSFER (debit_account, credit_account, amount)
2         GET (dbdata, debit_account)
3         dbdata ← dbdata - amount
4         PUT (dbdata, debit_account)
5         GET (crdata, credit_account)
6         crdata ← crdata + amount
7         PUT (crdata, credit_account)
```

where *debit_account* and *credit_account* identify the records for the accounts to be debited and credited, respectively.

Suppose that the system crashes while executing the `PUT` instruction on line 4. Even if we use the `MORE_DURABLE_PUT` described in Section 2.6.4, a system crash at just the wrong time may cause the data written to the disk to be scrambled, and the value of `debit_account` lost. We would prefer that either the data be completely written to the disk or nothing be written at all. That is, we want the `PUT` instruction to have the all-or-nothing atomicity property. Section 3.3.1 will describe a way to do that.

There is a further all-or-nothing atomicity requirement in the `TRANSFER` procedure. Suppose that the `PUT` on line 4 is successful but that while executing line 5 or line 6 the power fails, stopping the computer in its tracks. When power is restored, the computer restarts, but volatile memory, including the state of the thread that was running the `TRANSFER` procedure, has been lost. If someone now inquires about the balances in *debit_account* and in *credit_account* things will not add up properly, because *debit_account* has a new value but *credit_account* has an old value. One might suggest postponing the first `PUT` to be just before the second one, but that just reduces the window of vulnerability, it does not eliminate it—the power could still fail in between the two `PUT` s. To eliminate the window, we must somehow arrange that the two `PUT` instructions, or perhaps even the entire `TRANSFER` procedure, be done as an all-or-nothing atomic action. In Section 3.3.3 we will devise a `TRANSFER` procedure that has the all-or-nothing property, and in Section 3.4 we will see some additional ways of providing the property.

3.2.2: All-or-Nothing Atomicity in the Interrupt Interface

A second application for all-or-nothing atomicity is in the processor instruction set interface as seen by a thread. A thread normally performs actions one after another, as directed by the instructions of the current program, but that certain events may catch the attention of the thread's interpreter, causing the interpreter, rather than the program, to supply the next instruction. When such an event happens, a different program, running in an interrupt thread, takes control.

If the event is a signal arriving from outside the interpreter, the interrupt thread may simply invoke a thread management primitive such as `ADVANCE` to alert some other thread about the event. For example, an I/O operation that the other thread was waiting for may now have completed. The interrupt handler then returns control to the interrupted thread. This example requires before-or-after atomicity between the interrupt thread and the interrupted thread. If the interrupted thread was in the midst of a call to the thread manager, the invocation of `ADVANCE` by the interrupt thread should occur either before or after that call.

Another possibility is that the interpreter has detected that something is going wrong in the interrupted thread. In that case, the interrupt event invokes an exception handler, which runs in the environment of the original thread. (Sidebar 3.2.1 offers some examples.) The exception handler either adjusts the environment to eliminate some problem (such as a missing page) so that the original thread can continue, or it declares that the original thread has failed and terminates it. In either case, the exception handler will need to examine the state of the action that the original thread was performing at the instant of the interruption—was that action finished, or is it in a partially done state?

Sidebar 3.2.1

Events that might lead to invoking an exception handler

1. A hardware fault occurs:
   - The processor detects a memory parity fault.
   - A sensor reports that the electric power has failed; the energy left in the power supply may be just enough to perform a graceful shutdown.
2. A hardware or software interpreter encounters something in the program that is clearly wrong:
   - The program tried to divide by zero.
   - The program supplied a negative argument to a square root function.
3. Continuing requires some resource allocation or deferred initialization:
   - The running thread encountered a missing-page exception in a virtual memory system.
   - The running thread encountered an indirection exception, indicating that it encountered an unresolved procedure linkage in the current program.
4. More urgent work needs to take priority, so the user wishes to terminate the thread:
   - This program is running much longer than expected.
   - The program is running normally, but the user suddenly realizes that it is time to catch the last train home.
5. The user realizes that something is wrong and decides to terminate the thread:
   - Calculating e, the program starts to display $3.1415\ldots$
   - The user asked the program to copy the wrong set of files.
6. Deadlock:
   - Thread A has acquired the scanner, and is waiting for memory to become free; thread B has acquired all available memory, and is waiting for the scanner to be released. Either the system notices that this set of waits cannot be resolved or, more likely, a timer that should never expire eventually expires. The system or the timer signals an exception to one or both of the deadlocked threads.

Ideally, the handler would like to see an all-or-nothing report of the state: either the instruction that caused the exception completed or it didn't do anything. An all-or-nothing report means that the state of the original thread is described entirely with values belonging to the layer in which the exception handler runs. An example of such a value is the program counter, which identifies the next instruction that the thread is to execute. An in-the-middle report would mean that the state description involves values of a lower layer, probably the operating system or the hardware processor itself. In that case, knowing the next instruction is only part of the story; the handler would also need to know which parts of the current instruction were executed and which were not. An example might be an instruction that increments an address register, retrieves the data at that new address, and adds that data value to the value in another register. If retrieving the data causes a missing-page exception, the description of the current state is that the address register has been incremented but the retrieval and addition have not yet been performed. Such an in-the-middle report is problematic because after the handler retrieves the missing page it cannot simply tell the processor to jump to the instruction that failed—that would increment the address register again, which is not what the programmer expected. Jumping to the next instruction isn't right, either, because that would omit the addition step. An all-or-nothing report is preferable because it avoids the

need for the handler to peer into the details of the next lower layer. Modern processor designers are generally careful to avoid designing instructions that don't have the all-or-nothing property. As will be seen shortly, designers of higher-layer interpreters must be similarly careful.

Sections 3.2.3 and 3.2.4 explore the case in which the exception terminates the running thread, thus creating a fault. Section 3.2.5 examines the case in which the interrupted thread continues, oblivious (one hopes) to the interruption.

3.2.3: All-or-Nothing Atomicity in a Layered Application

A third example of all-or-nothing atomicity lies in the challenge presented by a fault in a running program: at the instant of the fault, the program is typically in the middle of doing something, and it is usually not acceptable to leave things half-done. Our goal is to obtain a more graceful response, and the method will be to require that some sequence of actions behave as an atomic action with the all-or-nothing property. Atomic actions are closely related to the modularity that arises when things are organized in layers. Layered components have the feature that a higher layer can completely hide the existence of a lower layer. This hiding feature makes layers exceptionally effective at error containment and for systematically responding to faults.

To see why, view the layered structure of the calendar management program found in Figure 3.2.1. The calendar program implements each request of the user by executing a sequence of Java language statements. Ideally, the user will never notice any evidence of the composite nature of the actions implemented by the calendar manager. Similarly, each statement of the Java language is implemented by several actions at the hardware layer. Again, if the Java interpreter is carefully implemented, the composite nature of the implementation in terms of machine language will be completely hidden from the Java programmer.

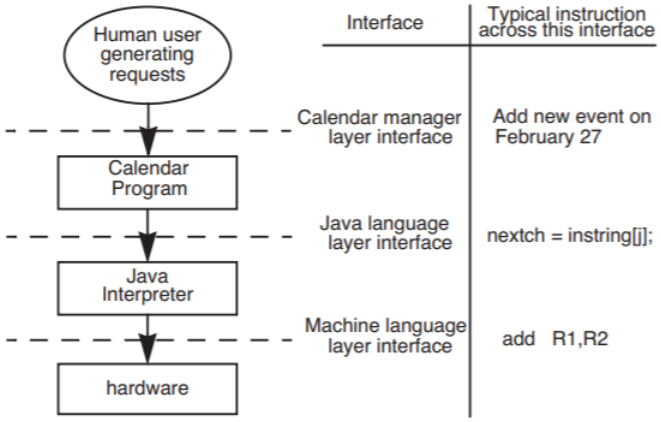


Figure 3.2.1: An application system with three layers of interpretation. If the user has requested an action that will fail, but the failure will be discovered at the lowest layer. A graceful response involves atomicity at each interface.

Now consider what happens if the hardware processor detects a condition that should be handled as an exception—for example, a register overflow. The machine is in the middle of interpreting an action at the machine language layer interface—an ADD instruction somewhere in the middle of the Java interpreter program. That ADD instruction is itself in the middle of interpreting an action at the Java language interface—a Java expression to scan an array. That Java expression in turn is in the middle of interpreting an action at the user interface—a request from the user to add a new event to the calendar. The report "Overflow exception caused by the ADD instruction at location 41574" is not intelligible to the user at the user interface; that description is meaningful only at the machine language interface. Unfortunately, the implication of being "in the middle" of higher-layer actions is that the only accurate description of the current state of affairs is in terms of the progress of the machine language program.

The actual state of affairs in our example as understood by an all-seeing observer might be the following: the register overflow was caused by adding one to a register that contained a two's complement negative one at the machine language layer. That machine language add instruction was part of an action to scan an array of characters at the Java layer and a zero means that the scan has reached the end of the array. The array scan was embarked upon by the Java layer in response to the user's request to add an event on February 31. The highest-level interpretation of the overflow exception is "You tried to add an event on a non-existent date." We want to make sure that this report goes to the end user, rather than the one about register overflow. In addition, we want to be able to assure the user that this mistake has not caused an empty event to be added somewhere else in the calendar or otherwise led to

any other changes to the calendar. Since the system couldn't do the requested change, it should do nothing but report the error. Either a low-level error report or muddled data would reveal to the user that the action was composite.

With the insight that in a layered application, we want a fault detected by a lower layer to be contained in a particular way we can now propose a more formal definition of all-or-nothing atomicity:

Definition: All-or-nothing Atomicity

A sequence of steps is an **all-or-nothing action** if, from the point of view of its invoker, the sequence always either

- **completes**,

or

- aborts in such a way that it appears that the sequence had never been undertaken in the first place. That is, it **backs out**.

In a layered application, the idea is to design each of the actions of each layer to be all-or-nothing. That is, whenever an action of a layer is carried out by a sequence of actions of the next lower layer, the action either completes what it was asked to do or else it backs out, acting as though it had not been invoked at all. When control returns to a higher layer after a lower layer detects a fault, the problem of being "in the middle" of an action thus disappears.

In our calendar management example, we might expect that the machine language layer would complete the add instruction but signal an overflow exception; the Java interpreter layer would, upon receiving the overflow exception might then decide that its array scan has ended, and return a report of "scan complete, value not found" to the calendar management layer; the calendar manager would take this not-found report as an indication that it should back up, completely undo any tentative changes, and tell the user that the request to add an event on that date could not be accomplished because the date does not exist.

Thus some layers run to completion, while others back out and act as though they had never been invoked, but either way the actions are all-or-nothing. In this example, the failure would probably propagate all the way back to the human user to decide what to do next. A different failure (e.g. "there is no room in the calendar for another event") might be intercepted by some intermediate layer that knows of a way to mask it (e.g., by allocating more storage space). In that case, the all-or-nothing requirement is that the layer that masks the failure find that the layer below has either never started what was to be the current action, or else it has completed the current action but has not yet undertaken the next one.

All-or-nothing atomicity is not usually achieved casually, but rather by careful design and specification. Designers often get it wrong. An unintelligible error message is the typical symptom that a designer got it wrong. To gain some insight into what is involved, let us examine some examples.

3.2.4: Some Actions With and Without the All-or-Nothing Property

Actions that lack the all-or-nothing property have frequently been discovered upon adding multilevel memory management to a computer architecture, especially to a processor that is highly pipelined. In this case, the interface that needs to be all-or-nothing lies between the processor and the operating system. Unless the original machine architect designed the instruction set with missing-page exceptions in mind, there may be cases in which a missing-page exception can occur "in the middle" of an instruction, after the processor has overwritten some register or after later instructions have entered the pipeline. When such a situation arises, the later designer who is trying to add the multilevel memory feature is trapped. The instruction cannot run to the end because one of the operands it needs is not in real memory. While the missing page is being retrieved from secondary storage, the designer would like to allow the operating system to use the processor for something else (perhaps even to run the program that fetches the missing page), but reusing the processor requires saving the state of the currently executing program, so that it can be restarted later when the missing page is available. The problem is how to save the next-instruction pointer.

If every instruction is an all-or-nothing action, the operating system can simply save as the value of the next-instruction pointer the address of the instruction that encountered the missing page. The resulting saved state description shows that the program is between two instructions, one of which has been completely executed, and the next one of which has not yet begun. Later, when the page is available, the operating system can restart the program by reloading all of the registers and setting the program counter to the place indicated by the next-instruction pointer. The processor will continue, starting with the instruction that previously encountered the missing page exception; this time it should succeed. On the other hand, if even one instruction of the instruction set lacks the all-or-nothing property, when an interrupt happens to occur during the execution of that instruction it is not at all obvious

how the operating system can save the processor state for a future restart. Designers have come up with several techniques to retrofit the all-or-nothing property at the machine language interface. Section 3.9 describes some examples of machine architectures that had this problem and the techniques that were used to add virtual memory to them.

A second example is the supervisor call (SVC). The SVC instruction changes both the program counter and the processor mode bit (and in systems with virtual memory, other registers such as the page map address register), so it needs to be all-or-nothing to ensure that all (or none) of the intended registers change. Beyond that, the SVC invokes some complete kernel procedure. The designer would like to arrange that the entire call (the combination of the SVC instruction and the operation of the kernel procedure itself) be an all-or-nothing action. An all-or-nothing design allows the application programmer to view the kernel procedure as if it is an extension of the hardware. That goal is easier said than done, since the kernel procedure may detect some condition that prevents it from carrying out the intended action. Careful design of the kernel procedure is thus required.

Consider an SVC to a kernel `READ` procedure that delivers the next typed keystroke to the caller. The user may not have typed anything yet when the application program calls `READ` , so the the designer of `READ` must arrange to wait for the user to type something. By itself, this situation is not especially problematic, but it becomes more so when there is also a user-provided exception handler. Suppose, for example, a thread timer can expire during the call to `READ` and the user-provided exception handler is to decide whether or not the thread should continue to run a while longer. The scenario, then, is the user program calls `READ` , it is necessary to wait, and while waiting, the timer expires and control passes to the exception handler. Different systems choose one of three possibilities for the design of the `READ` procedure, the last one of which is not an all-or-nothing design:

1. *An all-or-nothing design that implements the "nothing" option (blocking read):* Seeing no available input, the kernel procedure first adjusts return pointers ("push the PC back") to make it appear that the application program called `AWAIT` is just ahead of its call to the kernel `READ` procedure and then it transfers control to the kernel `AWAIT` entry point. When the user finally types something, causing `AWAIT` to return, the user's thread re-executes the original kernel call to `READ` , this time finding the typed input. With this design, if a timer exception occurs while waiting, when the exception handler investigates the current state of the thread it finds the answer "the application program is between instructions; its next instruction is a call to `READ` ." This description is intelligible to a user-provided exception handler, and it allows that handler several options. One option is to continue the thread, meaning go ahead and execute the call to `READ` . If there is still no input, `READ` will again push the PC back and transfer control to `AWAIT` . Another option is for the handler to save this state description with a plan of restoring a future thread to this state at some later time.
2. *An all-or-nothing design that implements the "all" option (non-blocking read):* Seeing no available input, the kernel immediately returns to the application program with a zero-length result, expecting that the program will look for and properly handle this case. The program would probably test the length of the result and if zero, call `AWAIT` itself or it might find something else to do instead. As with the previous design, this design ensures that at all times the user-provided timer exception handler will see a simple description of the current state of the thread—it is between two user program instructions. However, some care is needed to avoid a race between the call to `AWAIT` and the arrival of the next typed character.
3. *A blocking read design that is neither "all" nor "nothing" and therefore not atomic:* The kernel `READ` procedure itself calls `AWAIT` , blocking the thread until the user types a character. Although this design seems conceptually simple, the description of the state of the thread from the point of view of the timer exception handler is not simple. Rather than being "between two user instructions," it is "waiting for something to happen in the middle of a user call to kernel procedure `READ` ." The option of saving this state description for future use has been foreclosed. To start another thread with this state description, the exception handler would need to be able to request "start this thread just after the call to `AWAIT` in the middle of the kernel `READ` entry." But allowing that kind of request would compromise the modularity of the user-kernel interface. The user-provided exception handler could equally well make a request to restart the thread anywhere in the kernel, thus bypassing its gates and compromising its security.

The first and second designs correspond directly to the two options in the definition of an all-or-nothing action, and indeed some operating systems offer both options. In the first design the kernel program acts in a way that appears that the call had never taken place, while in the second design the kernel program runs to completion every time it is called. Both designs make the kernel procedure an all-or-nothing action, and both lead to a user-intelligible state description—the program is between two of its instructions—if an exception should happen while waiting.

3.2.5: Before-or-After Atomicity: Coordinating Concurrent Threads

We can express opportunities for concurrency by creating threads, the goal of concurrency being to improve performance by running several things at the same time. Moreover, Section 3.2.2 above pointed out that interrupts can also create concurrency. Concurrent threads do not represent any special problem until their paths cross. The way that paths cross can always be described in terms of shared, writable data: concurrent threads happen to take an interest in the same piece of writable data at about the same time. It is not even necessary that the concurrent threads be running simultaneously; if one is stalled (perhaps because of an interrupt) in the middle of an action, a different, running thread can take an interest in the data that the stalled thread was, and will sometime again be, working with.

From the point of view of the programmer of an application, there are two quite different kinds of concurrency coordination requirements: **sequence coordination** and **before-or-after atomicity**. Sequence coordination is a constraint of the type "Action W must happen before action X." For correctness, the first action must complete before the second action begins. For example, reading of typed characters from a keyboard must happen before running the program that presents those characters on a display. As a general rule, when writing a program one can anticipate the sequence coordination constraints, and the programmer knows the identity of the concurrent actions. Sequence coordination thus is usually explicitly programmed, using either special language constructs or shared variables such as eventcounts.

In contrast, before-or-after atomicity is a more general constraint that several actions that concurrently operate on the same data should not interfere with one another. We define before-or-after atomicity as follows:

Definition: Before-or-after Atomicity

Concurrent actions have the **before-or-after** property if their effect from the point of view of their invokers is the same as if the actions occurred either completely before or completely after one another.

Before-or-after actions can be created with explicit locks and a thread manager that implements the procedures `ACQUIRE` and `RELEASE`. Programming correct before-or-after actions, such as coordinating a bounded buffer with several producers or several consumers, can be a tricky proposition. To be confident of correctness, one needs to establish a compelling argument that every action that touches a shared variable follows the locking protocol.

One thing that makes before-or-after atomicity different from sequence coordination is that the programmer of an action that must have the before-or-after property does not necessarily know the identities of all the other actions that might touch the shared variable. This lack of knowledge can make it problematic to coordinate actions by explicit program steps. Instead, what the programmer needs is an automatic, implicit mechanism that ensures proper handling of every shared variable. This chapter will describe several such mechanisms. Put another way, correct coordination requires discipline in the way concurrent threads read and write shared data.

Applications for before-or-after atomicity in a computer system abound. In an operating system, several concurrent threads may decide to use a shared printer at about the same time. It would not be useful for printed lines of different threads to be interleaved in the printed output. Moreover, it doesn't really matter which thread gets to use the printer first; the primary consideration is that one use of the printer be complete before the next begins, so the requirement is to give each print job the before-or-after atomicity property.

For a more detailed example, let us return to the banking application and the `TRANSFER` procedure. This time the account balances are held in shared memory variables (recall that the declaration keyword **reference** means that the argument is call-by-reference, so that `TRANSFER` can change the values of those arguments):

```
procedure TRANSFER (reference debit_account, reference credit_account, amount)
    debit_account ← debit_account - amount
    credit_account ← credit_account + amount
```

Despite their unitary appearance, a program statement such as " `X ← X + Y` " is actually composite: it involves reading the values of `X` and `Y`, performing an addition, and then writing the result back into `X`. If a concurrent thread reads and changes the value of `X` between the read and the write done by this statement, that other thread may be surprised when this statement overwrites its change.

Suppose this procedure is applied to accounts `A` (initially containing $300) and `B` (initially containing $100) as in

```
TRANSFER (A, B, $10)
```

We expect account `A` , the debit account, to end up with $290, and account `B` , the credit account, to end up with $110. Suppose, however, a second, concurrent thread is executing the statement

```
TRANSFER (B, C, $25)
```

where account `C` starts with $175. When both threads complete their transfers, we expect `B` to end up with $85 and `C` with $200. Further, this expectation should be fulfilled no matter which of the two transfers happens first. But the variable *`credit_account`* in the first thread is bound to the same object (account `B`) as the variable *`debit_account`* in the second thread. The risk to correctness occurs if the two transfers happen at about the same time. To understand this risk, consider Figure 3.2.2, which illustrates several possible time sequences of the `READ` and `WRITE` steps of the two threads with respect to variable `B` .

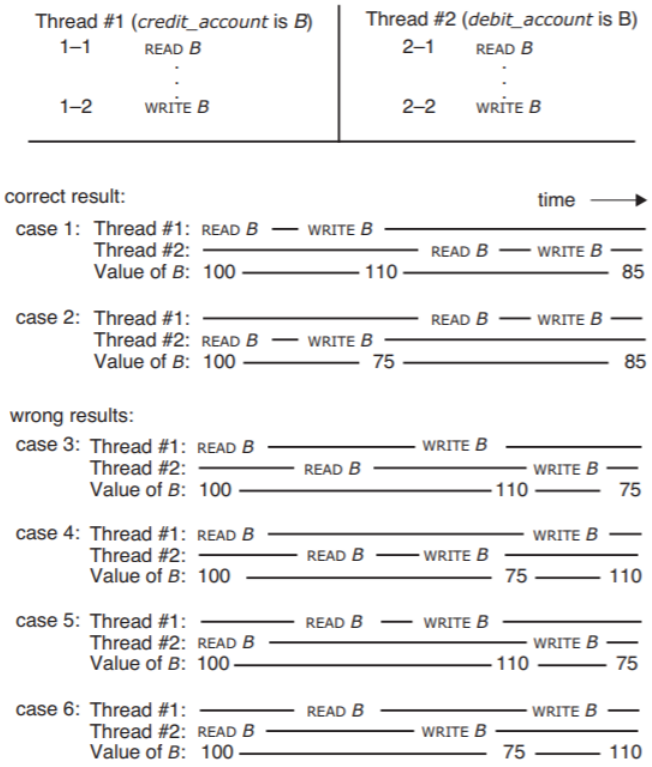


Figure 3.2.2: Six possible histories of variable `B` if two threads that share `B` do not coordinate their concurrent activities.

With each time sequence the figure shows the history of values of the cell containing the balance of account `B` . If both steps 1–1 and 1–2 precede both steps 2–1 and 2–2, (or vice versa) the two transfers will work as anticipated, and `B` ends up with $85. If, however, step 2–1 occurs after step 1–1, but before step 1–2, a mistake will occur: one of the two transfers will not affect account `B` , even though it should have. The first two cases illustrate histories of shared variable `B` in which the answers are the correct result; the remaining four cases illustrate four different sequences that lead to two incorrect values for `B` .

Thus our goal is to ensure that one of the first two time sequences actually occurs. One way to achieve this goal is that the two steps 1–1 and 1–2 should be atomic, and the two steps 2–1 and 2–2 should similarly be atomic. In the original program, the steps

```
debit_account ← debit_account - amount
```

and

```
credit_account ← credit_account + amount
```

should each be atomic. There should be no possibility that a concurrent thread that intends to change the value of the shared variable *`debit_account`* read its value between the `READ` and `WRITE` steps of this statement.

3.2.6: Correctness and Serialization

The notion that the first two sequences of Figure 3.2.2 are correct and the other four are wrong is based on our understanding of the banking application. It would be better to have a more general concept of correctness that is independent of the application. Application independence is a modularity goal: we want to be able to make an argument for correctness of the mechanism that provides before-or-after atomicity without getting into the question of whether or not the application using the mechanism is correct.

There is such a correctness concept: coordination among concurrent actions can be considered to be correct *if every result is guaranteed to be one that could have been obtained by some purely serial application of those same actions*.

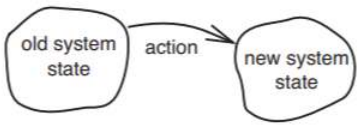


Figure 3.2.3: A single action takes a system from one state to another state.

The reasoning behind this concept of correctness involves several steps. Consider Figure 3.2.3, which shows, abstractly, the effect of applying some action, whether atomic or not, to a system: the action changes the state of the system. Now, if we are sure that:

1. the old state of the system was correct from the point of view of the application, and
2. the action, performing all by itself, correctly transforms any correct old state to a correct new state,

then we can reason that the new state must also be correct. This line of reasoning holds for any application-dependent definition of "correct" and "correctly transform", so our reasoning method is independent of those definitions and thus of the application.

The corresponding requirement when several actions act concurrently, as in Figure 3.2.4, is that the resulting new state ought to be one of those that would have resulted from some serialization of the several actions, as in Figure 3.2.5. This correctness criterion means that concurrent actions are correctly coordinated if their result is guaranteed to be one that would have been obtained by *some* purely serial application of those same actions. So long as the only coordination requirement is before-or-after atomicity, any serialization will do.

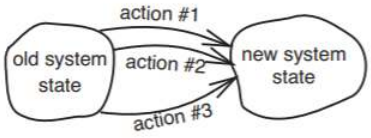


Figure 3.2.4: When several actions act concurrently, they together produce a new state. If the actions are before-or-after and the old state was correct, the new state will be correct.

Moreover, we do not even need to insist that the system actually traverse the intermediate states along any particular path of Figure 3.2.5—it may instead follow the dotted trajectory through intermediate states that are not by themselves correct, according to the application's definition. As long as the intermediate states are not visible above the implementing layer, and the system is guaranteed to end up in one of the acceptable final states, we can declare the coordination to be correct because there exists a trajectory that leads to that state for which a correctness argument could have been applied to every step.

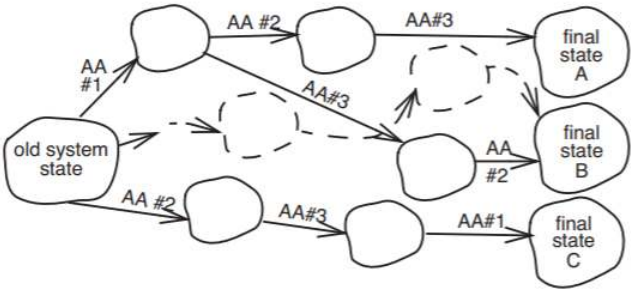


Figure 3.2.5: We insist that the final state be one that could have been reached by some serialization of the atomic actions, but we don't care which serialization. In addition, we do not need to insist that the intermediate states ever actually exist. The actual state trajectory could be that shown by the dotted lines, *but only if there is no way of observing the intermediate states from the outside*.

Since our definition of before-or-after atomicity is that each before-or-after action act as though it ran either completely before or completely after each other before-or-after action, before-or-after atomicity leads directly to this concept of correctness. Put another way, before-or-after atomicity has the effect of serializing the actions, so it follows that before-or-after atomicity guarantees correctness of coordination. A different way of expressing this idea is to say that when concurrent actions have the before-or-after property, they are serializable: there exists some serial order of those concurrent transactions that would, if followed, lead to the same ending state.* Thus in Figure 3.2.2, the sequences of case 1 and case 2 could result from a serialized order, but the actions of cases 3 through 6 could not.

In the example of Figure 3.2.2, there were only two concurrent actions and each of the concurrent actions had only two steps. As the number of concurrent actions and the number of steps in each action grows there will be a rapidly growing number of possible orders in which the individual steps can occur, but only some of those orders will ensure a correct result. Since the purpose of concurrency is to gain performance, one would like to have a way of choosing from the set of correct orders the one correct order that has the highest performance. As one might guess, making that choice can in general be quite difficult. In Section 3.5 and Section 3.6 of this chapter we will encounter several programming disciplines that ensure choice from a subset of the possible orders, all members of which are guaranteed to be correct but which, unfortunately, may not include the correct order that has the highest performance.

In some applications it is appropriate to use a correctness requirement that is stronger than serializability. For example, the designer of a banking system may want to avoid anachronisms by requiring what might be called external time consistency: if there is any external evidence (such as a printed receipt) that before-or-after action T_1 ended before before-or-after action T_2 began, the serialization order of T_1 and T_2 inside the system should be that T_1 precedes T_2. For another example of a stronger correctness requirement, a processor architect may require sequential consistency: when the processor concurrently performs multiple instructions from the same instruction stream, the result should be as if the instructions were executed in the original order specified by the programmer.

Returning to our example, a real funds-transfer application typically has several distinct before-or-after atomicity requirements. Consider the following auditing procedure; its purpose is to verify that the sum of the balances of all accounts is zero (in double-entry bookkeeping, accounts belonging to the bank, such as the amount of cash in the vault, have negative balances):

```
procedure AUDIT()
    sum ← 0
    for each W in bank.accounts
        sum ← sum + W.balance
    if (sum ≠ 0) call for investigation
```

Suppose that `AUDIT` is running in one thread at the same time that another thread is transferring money from account `A` to account `B` . If `AUDIT` examines account `A` before the transfer and account `B` after the transfer, it will count the transferred amount twice and thus will compute an incorrect answer. So the entire auditing procedure should occur either before or after any individual transfer: we want it to be a before-or-after action.

There is yet another before-or-after atomicity requirement: if `AUDIT` should run after the statement in `TRANSFER` that consists of

```
debit_account ← debit_account - amount
```

but before the statement

```
credit_account ← credit_account + amount
```

it will calculate a sum that does not include `amount` ; we therefore conclude that the two balance updates should occur either completely before or completely after any `AUDIT` action. Put another way, `TRANSFER` should be a before-or-after action.

---

* The general question of whether or not a collection of existing transactions isserializable is an advanced topic that is addressed in database management. Problem Set 20 explores one method of answering this question.

3.2.7: All-or-Nothing and Before-or-After Atomicity

We now have seen examples of two forms of atomicity: all-or-nothing and before-or-after. These two forms have a common underlying goal: to hide the internal structure of an action. With that insight, it becomes apparent that atomicity is really a unifying concept:

Definition: Atomicity

An action is atomic if there is no way for a higher layer to discover the internal structure of its implementation.

This description is really the fundamental definition of atomicity. From it, one can immediately draw two important consequences, corresponding to all-or-nothing atomicity and to before-or-after atomicity:

1. From the point of view of a procedure that invokes an atomic action, the atomic action always appears either to complete as anticipated, or to do nothing. This consequence is the one that makes atomic actions useful in recovering from failures.
2. From the point of view of a concurrent thread, an atomic action acts as though it occurs either *completely before* or *completely after* every other concurrent atomic action. This consequence is the one that makes atomic actions useful for coordinating concurrent threads.

These two consequences are not fundamentally different. They are simply two perspectives, the first from other modules within the thread that invokes the action, the second from other threads. Both points of view follow from the single idea that the internal structure of the action is not visible outside of the module that implements the action. Such hiding of internal structure is the essence of modularity, but atomicity is an exceptionally strong form of modularity. Atomicity hides not just the details of which steps form the atomic action, but also the very fact that it has structure. There is a kinship between atomicity and other system-building techniques such as data abstraction and client/server organization. Data abstraction has the goal of hiding the internal structure of data; client/server organization has the goal of hiding the internal structure of major subsystems. Similarly, atomicity has the goal of hiding the internal structure of an action. All three are methods of enforcing industrial-strength modularity, and thereby of guaranteeing absence of unanticipated interactions among components of a complex system.

We have used phrases such as "from the point of view of the invoker" several times, suggesting that there may be another point of view from which internal structure *is* apparent. That other point of view is seen by the implementer of an atomic action, who is often painfully aware that an action is actually composite, and who must do extra work to hide this reality from the higher layer and from concurrent threads. Thus the interfaces between layers are an essential part of the definition of an atomic action, and they provide an opportunity for the implementation of an action to operate in any way that ends up providing atomicity.

There is one more aspect of hiding the internal structure of atomic actions: atomic actions can have benevolent side effects. A common example is an audit log, where atomic actions that run into trouble record the nature of the detected failure and the recovery sequence for later analysis. One might think that when a failure leads to backing out, the audit log should be rolled back too, but rolling it back would defeat its purpose—the whole point of an audit log is to record details about the failure. The important point is that the audit log is normally a private record of the layer that implemented the atomic action; in the normal course of operation it is not visible above that layer, so there is no requirement to roll it back. (A separate atomicity requirement is to ensure that the log entry that describes a failure is complete and not lost in the ensuing recovery.)

Another example of a benevolent side effect is performance optimization. For example, in a high-performance data management system, when an upper layer atomic action asks the data management system to insert a new record into a file, the data management system may decide as a performance optimization that now is the time to rearrange the file into a better physical order. If the atomic action fails and aborts, it need ensure only that the newly-inserted record be removed; the file does not need to be

restored to its older, less efficient, storage arrangement. Similarly, a lower-layer cache that now contains a variable touched by the atomic action does not need to be cleared and a garbage collection of heap storage does not need to be undone. Such side effects are not a problem, as long as they are hidden from the higher-layer client of the atomic action except perhaps in the speed with which later actions are carried out, or across an interface that is intended to report performance measures or failures.

This page titled 3.2: Atomicity is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

3.3: All-Or-Nothing Atomicity I - Concepts

Section 3.2 of this chapter defined the goals of all-or-nothing atomicity and before-or-after atomicity, and provided a conceptual framework that at least in principle allows a designer to decide whether or not some proposed algorithm correctly coordinates concurrent activities. However, it did not provide any examples of actual implementations of either goal. This section of the chapter, together with the next one, describe some widely applicable techniques of systematically implementing **all-or-nothing** atomicity. Later sections of the chapter will do the same for before-or-after atomicity.

Many of the examples employ the technique called **bootstrapping**, a method that resembles inductive proof. Bootstrapping means to first look for a systematic way to reduce a general problem to some much-narrowed particular version of that same problem. Then, solve the narrow problem using some specialized method that might work only for that case because it takes advantage of the specific situation. The general solution then consists of two parts: a special-case technique plus a method that systematically reduces the general problem to the special case. This chapter uses bootstrapping several times. The first example starts with the special case and then introduces a way to reduce the general problem to that special case. The reduction method, called the **version history**, is used only occasionally in practice, but once understood it becomes easy to see why the more widely used reduction methods that will be described in Section 3.4 work.

3.3.1: Achieving All-or-Nothing Atomicity: `ALL_OR_NOTHING_PUT`

The first example is of a scheme that does an all-or-nothing update of a single disk sector. The problem to be solved is that if a system crashes in the middle of a disk write (for example, the operating system encounters a bug or the power fails), the sector that was being written at the instant of the failure may contain an unusable muddle of old and new data. The goal is to create an all-or-nothing `PUT` with the property that when `GET` later reads the sector, it always returns either the old or the new data, but never a muddled mixture.

To make the implementation precise, we develop a disk fault tolerance model that is a slight variation of the one introduced in Chapter 2, taking as an example application a calendar management program for a personal computer. The user is hoping that, if the system fails while adding a new event to the calendar, when the system later restarts the calendar will be safely intact. Whether or not the new event ended up in the calendar is less important than that the calendar not be damaged by inopportune timing of the system failure. This system comprises a human user, a display, a processor, some volatile memory, a magnetic disk, an operating system, and the calendar manager program. We model this system in several parts:

Overall system fault tolerance model.

- error-free operation: All work goes according to expectations. The user initiates actions such as adding events to the calendar and the system confirms the actions by displaying messages to the user.
- tolerated error: The user who has initiated an action notices that the system failed before it confirmed completion of the action and, when the system is operating again, checks to see whether or not it actually performed that action.
- untolerated error: The system fails without the user noticing, so the user does not realize that he or she should check or retry an action that the system may not have completed.

The tolerated error specification means that, to the extent possible, the entire system is fail-fast: if something goes wrong during an update, the system stops before taking any more requests, and the user realizes that the system has stopped. One would ordinarily design a system such as this one to minimize the chance of the untolerated error: for example, by requiring supervision by a human user. The human user then is in a position to realize (perhaps from lack of response) that something has gone wrong. After the system restarts, the user knows to inquire whether or not the action completed. This design strategy should be familiar from our study of best-effort networks in Chapter 1. The lower layer (the computer system) is providing a best-effort implementation. A higher layer (the human user) supervises and, when necessary, retries. For example, suppose that the human user adds an appointment to the calendar but just as he or she clicks "save" the system crashes. The user doesn't know whether or not the addition actually succeeded, so when the system comes up again the first thing to do is open up the calendar to find out what happened.

Processor, memory, and operating system fault tolerance model.

This part of the model just specifies more precisely the intended fail-fast properties of the hardware and operating system:

- error-free operation: The processor, memory, and operating system all follow their specifications.

- detected error: Something fails in the hardware or operating system. The system is fail-fast: the hardware or operating system detects the failure and restarts from a clean slate *before* initiating any further `PUT` s to the disk.
- untolerated error: Something fails in the hardware or operating system. The processor muddles along and `PUT` s corrupted data to the disk before detecting the failure.

The primary goal of the processor/memory/operating-system part of the model is to detect failures and stop running before any corrupted data is written to the disk storage system. The importance of detecting failure before the next disk write lies in error containment: if the goal is met, the designer can assume that the only values potentially in error must be in processor registers and volatile memory, and the data on the disk should be safe, with the exception described in Section 2.6.4.2: if there was a `PUT` to the disk in progress at the time of the crash, the failing system may have corrupted the disk buffer in volatile memory, and consequently corrupted the disk sector that was being written.

The recovery procedure can thus depend on the disk storage system to contain only uncorrupted information, or at most one corrupted disk sector. In fact, after restart the disk will contain the *only* information. "Restarts from a clean slate" means that the system discards all state held in volatile memory. This step brings the system to the same state as if a power failure had occurred, so a single recovery procedure will be able to handle both system crashes and power failures. Discarding volatile memory also means that all currently active threads vanish, so everything that was going on comes to an abrupt halt and will have to be restarted.

Disk storage system fault tolerance model.

Implementing all-or-nothing atomicity involves some steps that resemble the decay masking of `MORE_DURABLE_PUT/GET` in Chapter 2—in particular, the algorithm will write multiple copies of data. To clarify how the all-or-nothing mechanism works, we temporarily back up to `CAREFUL_PUT/GET` (see Section 2.6.4.5), which masks soft disk errors but not hard disk errors or disk decay. To simplify further, we pretend for the moment that a disk never decays and that it has no hard errors. (Since this perfect-disk assumption is obviously unrealistic, we will reverse it in Section 3.8, which describes an algorithm for all-or-nothing atomicity despite disk decay and hard errors.)

With the perfect-disk assumption, only one thing can go wrong: a system crash at just the wrong time. The fault tolerance model for this simplified careful disk system then becomes:

- error-free operation: `CAREFUL_GET` returns the result of the most recent call to `CAREFUL_PUT` at *`sector_number`* on *`track`* , with *`status`* `= OK` .
- detectable error: The operating system crashes during a `CAREFUL_PUT` and corrupts the disk buffer in volatile storage, and `CAREFUL_PUT` writes corrupted data on one sector of the disk.

We can classify the error as "detectable" if we assume that the application has included with the data an end-to-end checksum, calculated before calling `CAREFUL_PUT` and thus before the system crash could have corrupted the data.

The change in this revision of the careful storage layer is that when a system crash occurs, one sector on the disk may be corrupted, but the client of the interface is confident that (1) that sector is the only one that may be corrupted and (2) if it has been corrupted, any later reader of that sector will detect the problem. Between the processor model and the storage system model, all anticipated failures now lead to the same situation: the system detects the failure, resets all processor registers and volatile memory, forgets all active threads, and restarts. No more than one disk sector is corrupted.

Our problem is now reduced to providing the all-or-nothing property: the goal is to create **all-or-nothing disk storage**, which guarantees either to change the data on a sector completely and correctly or else appear to future readers not to have touched it at all. Here is one simple, but somewhat inefficient, scheme that makes use of virtualization: assign, for each data sector that is to have the all-or-nothing property, three physical disk sectors, identified as `S1` , `S2` , and `S3` . The three physical sectors taken together are a virtual "all-or-nothing sector." At each place in the system where this disk sector was previously used, replace it with the all-or-nothing sector, identified by the triple { `S1, S2, S3` }. We start with an almost correct all-or-nothing implementation named `ALMOST_ALL_OR_NOTHING_PUT` , find a bug in it, and then fix the bug, finally creating a correct `ALL_OR_NOTHING_PUT` .

When asked to write data, `ALMOST_ALL_OR_NOTHING_PUT` writes it three times, on `S1` , `S2` , and `S3` , in that order, each time waiting until the previous write finishes so that if the system crashes only one of the three sectors will be affected. To read data, `ALL_OR_NOTHING_GET` reads all three sectors and compares their contents. If the contents of `S1` and `S2` are identical, `ALL_OR_NOTHING_GET` returns that value as the value of the all-or-nothing sector. If `S1` and `S2` differ, `ALL_OR_NOTHING_GET` returns the contents of `S3` as the value of the all-or-nothing sector. Figure 3.3.1 shows this almost correct pseudocode.

```
1  procedure ALMOST_ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)
2      CAREFUL_PUT (data, all_or_nothing_sector.S1)
3      CAREFUL_PUT (data, all_or_nothing_sector.S2)          // Commit point.
4      CAREFUL_PUT (data, all_or_nothing_sector.S3)

5  procedure ALL_OR_NOTHING_GET (reference data, all_or_nothing_sector)
6      CAREFUL_GET (data1, all_or_nothing_sector.S1)
7      CAREFUL_GET (data2, all_or_nothing_sector.S2)
8      CAREFUL_GET (data3, all_or_nothing_sector.S3)
9      if data1 = data2 then data ← data1                    // Return new value.
10     else data ← data3                                     // Return old value.
```

Figure 3.3.1: Algorithms for ALMOST_ALL_OR_NOTHING_PUT and ALL_OR_NOTHING_GET .

Let's explore how this implementation behaves on a system crash. Suppose that at some previous time a record has been correctly stored in an all-or-nothing sector (in other words, all three copies are identical), and someone now updates it by calling ALL_OR_NOTHING_PUT . The goal is that even if a failure occurs in the middle of the update, a later reader can always be ensured of getting some complete, consistent version of the record by invoking ALL_OR_NOTHING_GET .

Suppose that ALMOST_ALL_OR_NOTHING_PUT were interrupted by a system crash some time before it finishes writing sector S2 , and thus corrupts either S1 or S2 . In that case, when ALL_OR_NOTHING_GET reads sectors S1 and S2 , they will have different values, and it is not clear which one to trust. Because the system is fail-fast, sector S3 would not yet have been touched by ALMOST_ALL_OR_NOTHING_PUT , so it still contains the previous value. Returning the value found in S3 thus has the desired effect of ALMOST_ALL_OR_NOTHING_PUT having done nothing.

Now, suppose that ALMOST_ALL_OR_NOTHING_PUT were interrupted by a system crash some time after successfully writing sector S2 . In that case, the crash may have corrupted S3 , but S1 and S2 both contain the newly updated value. ALL_OR_NOTHING_GET returns the value of S1 , thus providing the desired effect of ALMOST_ALL_OR_NOTHING_PUT having completed its job.

So what's wrong with this design? ALMOST_ALL_OR_NOTHING_PUT assumes that all three copies are identical when it starts. But a previous failure can violate that assumption. Suppose that ALMOST_ALL_OR_NOTHING_PUT is interrupted while writing S3 . The next thread to call ALL_OR_NOTHING_GET finds *data1 = data2* , so it uses *data1* , as expected. The new thread then calls ALMOST_ALL_OR_NOTHING_PUT , but is interrupted while writing S2 . Now, S1 doesn't equal S2 , so the next call to ALMOST_ALL_OR_NOTHING_PUT returns the damaged S3 .

The fix for this bug is for ALL_OR_NOTHING_PUT to guarantee that the three sectors be identical before updating. It can provide this guarantee by invoking a procedure named CHECK_AND_REPAIR as in Figure 3.3.2. CHECK_AND_REPAIR simply compares the three copies and, if they are not identical, it forces them to be identical.

```
1  procedure ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)
2      CHECK_AND_REPAIR (all_or_nothing_sector)
3      ALMOST_ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)

4  procedure CHECK_AND_REPAIR (all_or_nothing_sector) // Ensure copies match.
5      CAREFUL_GET (data1, all_or_nothing_sector.S1)
6      CAREFUL_GET (data2, all_or_nothing_sector.S2)
7      CAREFUL_GET (data3, all_or_nothing_sector.S3)
8      if (data1 = data2) and (data2 = data3) return     // State 1 or 7, no repair
9      if (data1 = data2)
10         CAREFUL_PUT (data1, all_or_nothing_sector.S3) return    // State 5 or 6.
11     if (data2 = data3)
12         CAREFUL_PUT (data2, all_or_nothing_sector.S1) return    // State 2 or 3.
13     CAREFUL_PUT (data1, all_or_nothing_sector.S2)      // State 4, go to state 5
14     CAREFUL_PUT (data1, all_or_nothing_sector.S3)      // State 5, go to state 7
```

Figure 3.3.2: Algorithms for ALL_OR_NOTHING_PUT and CHECK_AND_REPAIR .

To see how this works, assume that someone calls ALL_OR_NOTHING_PUT at a time when all three of the copies do contain identical values, which we designate as "old." Because ALL_OR_NOTHING_PUT writes "new" values into S1 , S2 , and S3 one at a time and in order, even if there is a crash, at the next call to ALL_OR_NOTHING_PUT there are only seven possible data states for CHECK_AND_REPAIR to consider:

	data state 1	data state 2	data state 3	data state 4	data state 5	data state 6	data state 7

	data state 1	data state 2	data state 3	data state 4	data state 5	data state 6	data state 7
sector S1	old	bad	new	new	new	new	new
sector S2	old	old	old	bad	new	new	new
sector S3	old	old	old	old	old	bad	new

The way to read this table is as follows: if all three sectors S1 , S2 , and S3 contain the "old" value, the data is in state 1. Now, if CHECK_AND_REPAIR discovers that all three copies are identical (line 8 in Figure 3.3.2), the data is in state 1 or state 7 so CHECK_AND_REPAIR simply returns. Failing that test, if the copies in sectors S1 and S2 are identical (line 9), the data must be in state 5 or state 6, so CHECK_AND_REPAIR forces sector S3 to match the others and returns (line 10). If the copies in sectors S2 and S3 are identical the data must be in state 2 or state 3 (line 11), so CHECK_AND_REPAIR forces sector S1 to match and returns (line 12). The only remaining possibility is that the data is in state 4, in which case sector S2 is surely bad, but sector S1 contains a new value and sector S3 contains an old one. The choice of which to use is arbitrary; as shown the procedure copies the new value in sector S1 to both sectors S2 and S3 .

What if a failure occurs while running CHECK_AND_REPAIR ? That procedure systematically drives the state either forward from state 4 toward state 7, or backward from state 3 toward state 1. If CHECK_AND_REPAIR is itself interrupted by another system crash, rerunning it will continue from the point at which the previous attempt left off.

We can make several observations about the algorithm implemented by ALL_OR_NOTHING_GET and ALL_OR_NOTHING_PUT :

1. This all-or-nothing atomicity algorithm assumes that only one thread at a time tries to execute either ALL_OR_NOTHING_GET or ALL_OR_NOTHING_PUT . This algorithm implements all-or-nothing atomicity but not before-or-after atomicity.
2. CHECK_AND_REPAIR is **idempotent**. That means that a thread can start the procedure, execute any number of its steps, be interrupted by a crash, and go back to the beginning again any number of times with the same ultimate result, as far as a later call to ALL_OR_NOTHING_GET is concerned.
3. The completion of the CAREFUL_PUT on line 3 of ALMOST_ALL_OR_NOTHING_PUT , marked "commit point," exposes the new data to future ALL_OR_NOTHING_GET actions. Until that step begins execution, a call to ALL_OR_NOTHING_GET sees the old data. After line 3 completes, a call to ALL_OR_NOTHING_GET sees the new data.
4. Although the algorithm writes three replicas of the data, the primary reason for the replicas is not to provide durability as described in Section 2.6. Instead, the reason for writing three replicas, one at a time and in a particular order, is to ensure observance at all times and under all failure scenarios of the *golden rule of atomicity*, which is the subject of the next section.

There are several ways of implementing all-or-nothing disk sectors. Near the end of Chapter 2 we introduced a fault tolerance model for decay events that did not mask system crashes, and applied the technique known as RAID to mask decay to produce durable storage. Here we started with a slightly different fault tolerance model that omits decay, and we devised techniques to mask system crashes and produce all-or-nothing storage. What we really should do is start with a fault tolerance model that considers both system crashes and decay, and devise storage that is both all-or-nothing and durable. Such a model, devised by Xerox Corporation researchers Butler Lampson and Howard Sturgis, is the subject of Section 3.8, together with the more elaborate recovery algorithms it requires. That model has the additional feature that it needs only two physical sectors for each all-or-nothing sector.

3.3.2: Systematic Atomicity: Commit and the Golden Rule

The example of ALL_OR_NOTHING_PUT and ALL_OR_NOTHING_GET demonstrates an interesting special case of all-or-nothing atomicity, but it offers little guidance on how to systematically create a more general all-or-nothing action. From the example, our calendar program now has a tool that allows writing individual sectors with the all-or-nothing property, but that is not the same as safely adding an event to a calendar, since adding an event probably requires rearranging a data structure, which in turn may involve writing more than one disk sector. We could do a series of ALL_OR_NOTHING_PUT s to the several sectors, to ensure that each sector is itself written in an all-or-nothing fashion, but a crash that occurs after writing one and before writing the

next would leave the overall calendar addition in a partly-done state. To make the entire calendar addition action all-or-nothing we need a generalization.

Ideally, one might like to be able to take any arbitrary sequence of instructions in a program, surround that sequence with some sort of `begin` and `end` statements as in Figure 3.3.3, and expect that the language compilers and operating system will perform some magic that makes the surrounded sequence into an all-or-nothing action. Unfortunately, no one knows how to do that. But we can come close, if the programmer is willing to make a modest concession to the requirements of all-or-nothing atomicity. This concession is expressed in the form of a discipline on the constituent steps of the all-or-nothing action.

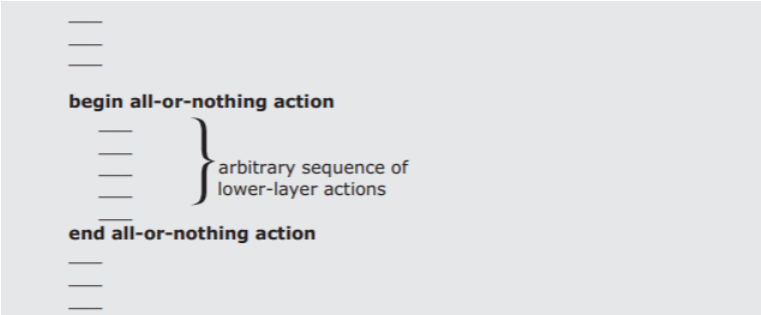


Figure 3.3.3: Imaginary semantics for painless programming of all-or-nothing actions.

The discipline starts by identifying some single step of the sequence as the **commit point**. The all-or-nothing action is thus divided into two phases, a **pre-commit phase** and a **post-commit phase**, as suggested by Figure 3.3.4. During the pre-commit phase, the disciplining rule of design is that no matter what happens, it must be possible to back out of this all-or-nothing action in a way that leaves no trace. During the post-commit phase the disciplining rule of design is that no matter what happens, the action must run to the end successfully. Thus an all-or-nothing action can have only two outcomes. If the all-or-nothing action starts and then, without reaching the commit point, backs out, we say that it **aborts**. If the all-or-nothing action passes the commit point, we say that it **commits**.

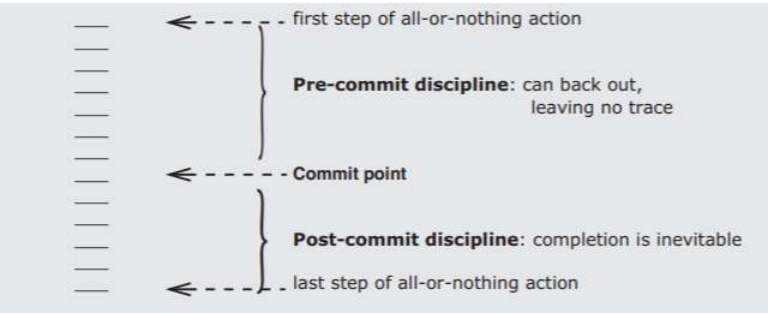


Figure 3.3.4: The commit point of an all-or-nothing action.

We can make several observations about the restrictions of the pre-commit phase. The pre-commit phase must identify all the resources needed to complete the all-or-nothing action, and establish their availability. The names of data should be bound, permissions should be checked, the pages to be read or written should be in memory, removable media should be mounted, stack space must be allocated, etc. In other words, all the steps needed to anticipate the severe run-to-the-end-without-faltering requirement of the post-commit phase should be completed during the pre-commit phase. In addition, the pre-commit phase must maintain the ability to abort at any instant. Any changes that the pre-commit phase makes to the state of the system must be undoable in case this all-or-nothing action aborts. Usually, this requirement means that shared resources, once reserved, cannot be released until the commit point is passed. The reason is that if an all-or-nothing action releases a shared resource, some other, concurrent thread may capture that resource. If the resource is needed in order to undo some effect of the all-or-nothing action, releasing the resource is tantamount to abandoning the ability to abort. Finally, the reversibility requirement means that the all-or-nothing action should not do anything externally visible, for example printing a check or firing a missile, prior to the commit point. (It is possible, though more complicated, to be slightly less restrictive. Sidebar 3.3.1 explores that possibility.)

Sidebar 3.3.1

Cascaded aborts

(Temporary) sweeping simplification. In this initial discussioin of commit points, we are intentionally avoiding a more complex and harder-to-design possibility. Some systems allow other, concurrent activities to see pending results, and they may even allow externally visible actions before commit. Those systems must therefore be prepared to track down and abort those concurrent activities (this tracking down is called **cascaded abort**) or perform **compensating** external actions (e.g., send a letter requesting return of the check or apologizing for the missile firing). The discussion of layers and multiple sites in Chapter 4 introduces a simple version of cascaded abort.

In contrast, the post-commit phase can expose results, it can release reserved resources that are no longer needed, and it can perform externally visible actions such as printing a check, opening a cash drawer, or drilling a hole. But it cannot try to acquire additional resources because an attempt to acquire might fail, and the post-commit phase is not permitted the luxury of failure. The post-commit phase must confine itself to finishing just the activities that were planned during the pre-commit phase.

It might appear that if a system fails before the post-commit phase completes, all hope is lost, so the only way to ensure all-or-nothing atomicity is to always make the commit step the last step of the all-or-nothing action. Often, that is the simplest way to ensure all-or-nothing atomicity, but the requirement is not actually that stringent. An important feature of the post-commit phase is that it is hidden inside the layer that implements the all-or-nothing action, so a scheme that ensures that the post-commit phase completes after a system failure is acceptable, so long as this delay is hidden from the invoking layer. Some all-or-nothing atomicity schemes thus involve a guarantee that a cleanup procedure will be invoked following every system failure, or as a prelude to the next use of the data, before anyone in a higher layer gets a chance to discover that anything went wrong. This idea should sound familiar: the implementation of `ALL_OR_NOTHING_PUT` in Figure 3.3.2 used this approach, by always running the cleanup procedure named `CHECK_AND_REPAIR` before updating the data.

A popular technique for achieving all-or-nothing atomicity is called the **shadow copy**. It is used by text editors, compilers, calendar management programs, and other programs that modify existing files, to ensure that following a system failure the user does not end up with data that is damaged or that contains only some of the intended changes:

- Pre-commit: Create a complete duplicate working copy of the file that is to be modified. Then, make all changes to the working copy.
- Commit point: Carefully exchange the working copy with the original. Typically this step is bootstrapped, using a lower-layer `RENAME` entry point of the file system that provides certain atomic-like guarantees.
- Post-commit: Release the space that was occupied by the original.

The `ALL_OR_NOTHING_PUT` algorithm of Figure 3.3.2 can be seen as a particular example of the shadow copy strategy, which itself is a particular example of the general pre-commit/post-commit discipline. The commit point occurs at the instant when the new value of `S2` is successfully written to the disk. During the pre-commit phase, while `ALL_OR_NOTHING_PUT` is checking over the three sectors and writing the shadow copy `S1` , a crash will leave no trace of that activity (that is, no trace that can be discovered by a later caller of `ALL_OR_NOTHING_GET`). The post-commit phase of `ALL_OR_NOTHING_PUT` consists of writing `S3` .

From these examples we can extract an important design principle:

The Golden Rule of Atomicity

Never modify the only copy!

In order for a composite action to be all-or-nothing, there must be some way of reversing the effect of each of its pre-commit phase component actions, so that if the action does not commit it is possible to back out. As we continue to explore implementations of all-or-nothing atomicity, we will notice that correct implementations always reduce at the end to making a shadow copy. The reason is that this structure ensures that the implementation follows the golden rule.

3.3.3: Systematic All-or-Nothing Atomicity: Version Histories

This section develops a scheme to provide all-or-nothing atomicity in the general case of a program that modifies arbitrary data structures. It will be easy to see why the scheme is correct, but the mechanics can interfere with performance. Section 3.4 of this chapter then introduces a variation on the scheme that requires more thought to see why it is correct, but that allows higher-performance implementations. As before, we concentrate for the moment on all-or-nothing atomicity. While some aspects of before-or-after atomicity will also emerge, we leave a systematic treatment of that topic for discussion in Section 3.4 and Section 3.5 of this chapter. Thus the model to keep in mind in this section is that only a single thread is running. If the system crashes, after a restart the original thread is gone—recall from Chapter 2 the *sweeping simplification* that threads are included in the volatile state that is lost on a crash and only durable state survives. After the crash, a new, different thread comes along and attempts to look at the data. The goal is that the new thread should always find that the all-or-nothing action that was in progress at the time of the crash either never started or completed successfully.

In looking at the general case, a fundamental difficulty emerges: random-access memory and disk usually appear to the programmer as a set of named, shared, and rewritable storage cells, called **cell storage**. Cell storage has semantics that are actually quite hard to make all-or-nothing because the act of storing destroys old data, thus potentially violating the *golden rule of atomicity*. If the all-or-nothing action later aborts, the old value is irretrievably gone; at best it can only be reconstructed from information kept elsewhere. In addition, storing data reveals it to the view of later threads, whether or not the all-or-nothing action that stored the value reached its commit point. If the all-or-nothing action happens to have exactly one output value, then writing that value into cell storage can be the mechanism of committing, and there is no problem. But if the result is supposed to consist of several output values, all of which should be exposed simultaneously, it is harder to see how to construct the all-or-nothing action. Once the first output value is stored, the computation of the remaining outputs has to be successful; there is no going back. If the system fails and we have not been careful, a later thread may see some old and some new values.

These limitations of cell storage did not plague the shopkeepers of Padua, who in the 14th century invented double-entry bookkeeping. Their storage medium was leaves of paper in bound books, and they made new entries with quill pens. They never erased or even crossed out entries that were in error; when they made a mistake they made another entry that reversed the mistake, thus leaving a complete history of their actions, errors, and corrections in the book. It wasn't until the 1950s, when programmers began to automate bookkeeping systems, that the notion of overwriting data emerged. Up until that time, if a bookkeeper collapsed and died while making an entry, it was always possible for someone else to seamlessly take over the books. This observation about the robustness of paper systems suggests that there is a form of the *golden rule of atomicity* that might allow one to be systematic: never erase anything.

Examining the shadow copy technique used by the text editor provides a second useful idea. The essence of the mechanism that allows a text editor to make several changes to a file, yet not reveal any of the changes until it is ready, is this: the only way another prospective reader of a file can reach it is by name. Until commit time the editor works on a copy of the file that is either not yet named or has a unique name not known outside the thread, so the modified copy is effectively invisible. Renaming the new version is the step that makes the entire set of updates simultaneously visible to later readers.

These two observations suggest that all-or-nothing actions would be better served by a model of storage that behaves differently from cell storage: instead of a model in which a store operation overwrites old data, we instead create a new, tentative version of the data, such that the tentative version remains invisible to any reader outside this all-or-nothing action until the action commits. We can provide such semantics, even though we start with traditional cell memory, by interposing a layer between the cell storage and the program that reads and writes data. This layer implements what is known as **journal storage**. The basic idea of journal storage is straightforward: we associate with every named variable not a single cell, but a list of cells in non-volatile storage; the values in the list represent the history of the variable. Figure 3.3.5 illustrates.

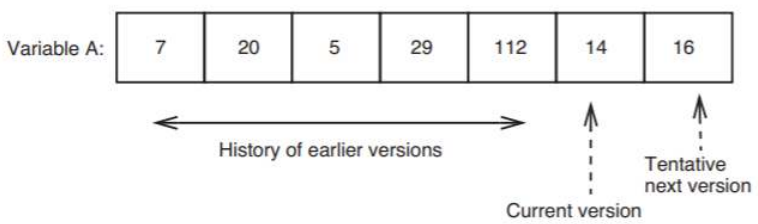


Figure 3.3.5: Version history of a variable in journal storage.

Whenever any action proposes to write a new value into the variable, the journal storage manager appends the prospective new value to the end of the list. Clearly this approach, being history-preserving, offers some hope of being helpful because if an all-or-nothing action aborts, one can imagine a systematic way to locate and discard all of the new versions it wrote. Moreover, we can tell the journal storage manager to expect to receive tentative values, but to ignore them unless the all-or-nothing action that created them commits. The basic mechanism to accomplish such an expectation is quite simple; the journal storage manager should make a note, next to each new version, of the identity of the all-or-nothing action that created it. Then, at any later time, it can discover the status of the tentative version by inquiring whether or not the all-or-nothing action ever committed.

Figure 3.3.6 illustrates the overall structure of such a journal storage system, implemented as a layer that hides a cell storage system. (To reduce clutter, this journal storage system omits calls to create new and delete old variables.) In this particular model, we assign to the journal storage manager most of the job of providing tools for programming all-or-nothing actions. Thus the implementer of a prospective all-or-nothing action should begin that action by invoking the journal storage manager entry `NEW_ACTION`, and later complete the action by invoking either `COMMIT` or `ABORT`. If, in addition, actions perform all reads and writes of data by invoking the journal storage manager's `READ_CURRENT_VALUE` and `WRITE_NEW_VALUE` entries, our hope is that the result will automatically be all-or-nothing with no further concern of the implementer.

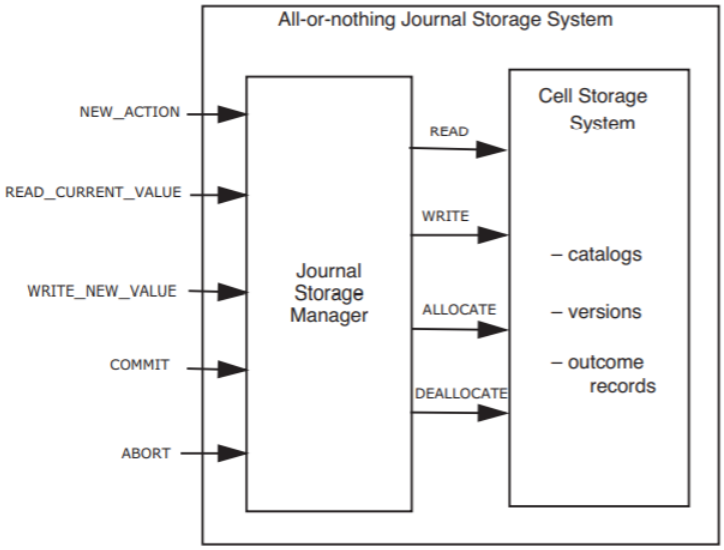
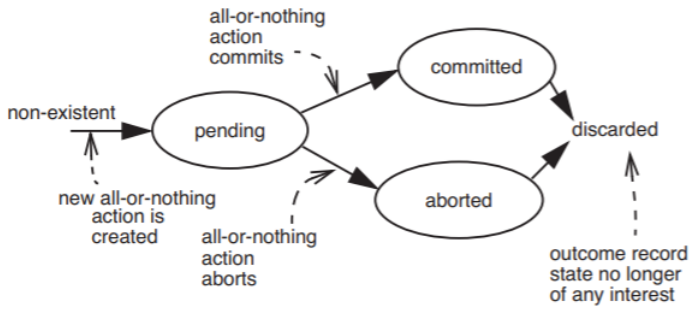


Figure 3.3.6: Interface to and internal organization of an all-or-nothing storage system based on version histories and journal storage.

How could this automatic all-or-nothing atomicity work? The first step is that the journal storage manager, when called at `NEW_ACTION`, should assign a nonce identifier to the prospective all-or-nothing action, and create, in non-volatile cell storage, a record of this new identifier and the state of the new all-or-nothing action. This record is called an outcome record; it begins its existence in the state `PENDING`; depending on the outcome it should eventually move to one of the states `COMMITTED` or `ABORTED`, as suggested by Figure 3.3.7. No other state transitions are possible, except to discard the outcome record once there is no further interest in its state. Figure 3.3.8 illustrates implementations of the three procedures `NEW_ACTION`, `COMMIT`, and `ABORT`.



https://eng.libretexts.org/@go/page/58508

Figure 3.3.7: The allowed state transitions of an outcome record.

```
1  procedure NEW_ACTION ()
2      id ← NEW_OUTCOME_RECORD ()
3      id.outcome_record.state ← PENDING
4      return id

5  procedure COMMIT (reference id)
6      id.outcome_record.state ← COMMITTED

7  procedure ABORT (reference id)
8      id.outcome_record.state ← ABORTED
```

Figure 3.3.8: The procedures NEW_ACTION , COMMIT , and ABORT .

When an all-or-nothing action calls the journal storage manager to write a new version of some data object, that action supplies the identifier of the data object, a tentative new value for the new version, and the identifier of the all-or-nothing action. The journal storage manager calls on the lower-level storage management system to allocate in nonvolatile cell storage enough space to contain the new version; it places in the newly allocated cell storage the new data value and the identifier of the all-or-nothing action. Thus the journal storage manager creates a version history as illustrated in Figure 3.3.9. Now, when someone proposes to read a data value by calling READ_CURRENT_VALUE , the journal storage manager can review the version history, starting with the latest version and return the value in the most recent committed version. By inspecting the outcome records, the journal storage manager can ignore those versions that were written by all-or-nothing actions that aborted or that never committed.

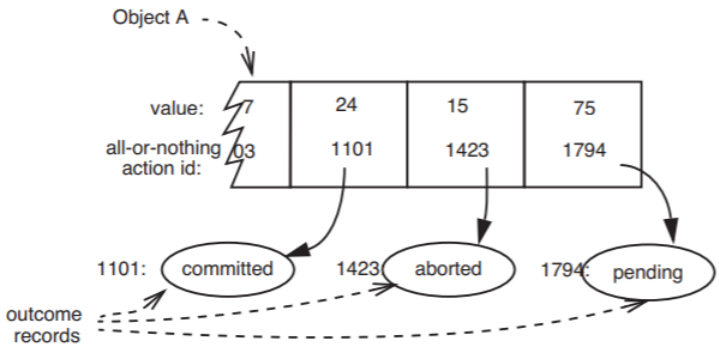


Figure 3.3.9: Portion of a version history, with outcome records. Some thread has recently called WRITE_NEW_VALUE specifying *data_id* = *A* , *new_value* = 75 , and *client_id* = 1794 . A caller to READ_CURRENT_VALUE will read the value 24 for A.

The procedures READ_CURRENT_VALUE and WRITE_NEW_VALUE thus follow the algorithms of Figure 3.3.10. The important property of this pair of algorithms is that if the current all-or-nothing action is somehow derailed before it reaches its call to COMMIT , the new version it has created is invisible to invokers of READ_CURRENT_VALUE . (They are also invisible to the all-or-nothing action that wrote them. Since it is sometimes convenient for an all-or-nothing action to read something that it has tentatively written, a different procedure, named READ_MY_PENDING_VALUE , identical to READ_CURRENT_VALUE except for a different test on line 6, could do that.) Moreover if, for example, all-or-nothing action 99 crashes while partway through changing the values of nineteen different data objects, all nineteen changes would be invisible to later invokers of READ_CURRENT_VALUE . If all-or-nothing action 99 does reach its call to COMMIT , that call commits the entire set of changes simultaneously and atomically, at the instant that it changes the outcome record from PENDING to COMMITTED . Pending versions would also be invisible to any concurrent action that reads data with READ_CURRENT_VALUE , a feature that will prove useful when we introduce concurrent threads and discuss before-or-after atomicity, but for the moment our only concern is that a system crash may prevent the current thread from committing or aborting, and we want to make sure that a later thread doesn't encounter partial results. As in the case of the calendar manager of Section 3.3.1 above, we assume that when a crash occurs, any all-or-nothing action that was in progress at the time was being supervised by some outside agent who realizes that a crash has occurred, uses READ_CURRENT_VALUE to find out what happened, and if necessary initiates a replacement all-or-nothing action.

```
1  procedure READ_CURRENT_VALUE (data_id, caller_id)
2      starting at end of data_id repeat until beginning
3          v ← previous version of data_id        // Get next older version
4          a ← v.action_id  // Identify the action a that created it
5          s ← a.outcome_record.state             // Check action a's outcome record
6          if s = COMMITTED then
7              return v.value
8          else skip v                            // Continue backward search
9      signal ("Tried to read an uninitialized variable!")

10 procedure WRITE_NEW_VALUE (reference data_id, new_value, caller_id)
11     if caller_id.outcome_record.state = PENDING
12         append new version v to data_id
13         v.value ← new_value
14         v.action_id ← caller_id
15     else signal ("Tried to write outside of an all-or-nothing action!")
```

Figure 3.3.10: Algorithms followed by READ_CURRENT_VALUE and WRITE_NEW_VALUE . The parameter *caller_id* is the action identifier returned by NEW_ACTION . In this version, only WRITE_NEW_VALUE uses *caller_id* . Later, READ_CURRENT_VALUE will also use it. *Note:* The last line of psuedocode in the figure (" **else signal** ") should be numbered 15 and aligned with the " **if** " on line 11.

```
1  procedure TRANSFER (reference debit_account, reference credit_account,
                                                                  amount)
2      my_id ← NEW_ACTION ()
3      xvalue ← READ_CURRENT_VALUE (debit_account, my_id)
4      xvalue ← xvalue - amount
5      WRITE_NEW_VALUE (debit_account, xvalue, my_id)
6      yvalue ← READ_CURRENT_VALUE (credit_account, my_id)
7      yvalue ← yvalue + amount
8      WRITE_NEW_VALUE (credit_account, yvalue, my_id)
9      if xvalue > 0 then
10         COMMIT (my_id)
11     else
12         ABORT (my_id)
13         signal("Negative transfers are not allowed.")
```

Figure 3.3.11: An all-or-nothing TRANSFER procedure, based on journal storage. (This program assumes that it is the only running thread. Making the transfer procedure a before-or-after action because other threads might be updating the same accounts concurrently requires additional mechanism that is discussed later in this chapter.)

Figure 3.3.11 shows the TRANSFER procedure of Section 3.2.5 reprogrammed as an all-or-nothing (but not, for the moment, before-or-after) action using the version history mechanism. This implementation of TRANSFER is more elaborate than the earlier one—it tests to see whether or not the account to be debited has enough funds to cover the transfer and if not it aborts the action. The order of steps in the transfer procedure is remarkably unconstrained by any consideration other than calculating the correct answer. The reading of *credit_account* , for example, could casually be moved to any point between NEW_ACTION and the place where *yvalue* is recalculated. We conclude that the journal storage system has made the pre-commit discipline much less onerous than we might have expected.

There is still one loose end: it is essential that updates to a version history and changes to an outcome record be all-or-nothing. That is, if the system fails while the thread is inside WRITE_NEW_VALUE , adjusting structures to append a new version, or inside COMMIT while updating the outcome record, the cell being written must not be muddled; it must either stay as it was before the crash or change to the intended new value. The solution is to design all modifications to the internal structures of journal storage so that they can be done by overwriting a single cell. For example, suppose that the name of a variable that has a version history refers to a cell that contains the address of the newest version, and that versions are linked from the newest version backwards, by address references. Adding a version consists of allocating space for a new version, reading the current address of the prior version, writing that address in the backward link field of the new version, and then updating the descriptor with the address of the new version. That last update can be done by overwriting a single cell. Similarly, updating an outcome record to change it from PENDING to COMMITTED can be done by overwriting a single cell.

As a first bootstrapping step, we have reduced the general problem of creating all-or-nothing actions to the specific problem of doing an all-or-nothing overwrite of one cell. As the remaining bootstrapping step, recall that we already know two ways to do a

single-cell all-or-nothing overwrite: apply the ALL_OR_NOTHING_PUT procedure of Figure 3.3.2. (If there is concurrency, updates to the internal structures of the version history also need before-or-after atomicity. Section 3.5 will explore methods of providing it.)

3.3.4: How Version Histories are Used

The careful reader will note two possibly puzzling things about the version history scheme just described. Both will become less puzzling when we discuss concurrency and before-or-after atomicity in Section 3.5 of this chapter:

1. Because READ_CURRENT_VALUE skips over any version belonging to another all-or-nothing action whose OUTCOME record is not COMMITTED , it isn't really necessary to change the OUTCOME record when an all-or-nothing action aborts; the record could just remain in the PENDING state indefinitely. However, when we introduce concurrency, we will find that a pending action may prevent other threads from reading variables for which the pending action created a new version, so it will become important to distinguish aborted actions from those that really are still pending.
2. As we have defined READ_CURRENT_VALUE , versions older than the most recent committed version are inaccessible and they might just as well be discarded. Discarding could be accomplished either as an additional step in the journal storage manager, or as part of a separate garbage collection activity. Alternatively, those older versions may be useful as an historical record, known as an archive, with the addition of timestamps on commit records and procedures that can locate and return old values created at specified times in the past. For this reason, a version history system is sometimes called a **temporal database** or is said to provide **time domain addressing**. The banking industry abounds in requirements that make use of history information, such as reporting a consistent sum of balances in all bank accounts, paying interest on the fifteenth on balances as of the first of the month, or calculating the average balance last month. Another reason for not discarding old versions immediately will emerge when we discuss concurrency and before-or-after atomicity: concurrent threads may, for correctness, need to read old versions even after new versions have been created and committed.

Direct implementation of a version history raises concerns about performance: rather than simply reading a named storage cell, one must instead make at least one indirect reference through a descriptor that locates the storage cell containing the current version. If the cell storage device is on a magnetic disk, this extra reference is a potential bottleneck, though it can be alleviated with a cache. A bottleneck that is harder to alleviate occurs on updates. Whenever an application writes a new value, the journal storage layer must allocate space in unused cell storage, write the new version, and update the version history descriptor so that future readers can find the new version. Several disk writes are likely to be required. These extra disk writes may be hidden inside the journal storage layer and with added cleverness may be delayed until commit and batched, but they still have a cost. When storage access delays are the performance bottleneck, extra accesses slow things down.

In consequence, version histories are used primarily in low-performance applications. One common example is found in revision management systems used to coordinate teams doing program development. A programmer "checks out" a group of files, makes changes, and then "checks in" the result. The check-out and check-in operations are all-or-nothing, and check-in makes each changed file the latest version in a complete history of that file in case a problem is discovered later. (The check-in operation also verifies that no one else changed the files while they were checked out, which catches some, but not all, coordination errors.) A second example is that some interactive applications such as word processors or image editing systems provide a “deep undo” feature, which allows a user who decides that his or her recent editing is misguided to step backwards to reach an earlier, satisfactory state. A third example appears in file systems that automatically create a new version every time any application opens an existing file for writing; when the application closes the file, the file system tags a number suffix to the name of the previous version of the file and moves the original name to the new version. These interfaces employ version histories because users find them easy to understand and they provide all-or-nothing atomicity in the face of both system failures and user mistakes. Most such applications also provide an archive that is useful for reference and that allows going back to a known good version.

Applications requiring high performance are a different story. They, too, require all-or-nothing atomicity, but they usually achieve it by applying a specialized technique called a **log**. Logs are our next topic.

This page titled 3.3: All-Or-Nothing Atomicity I - Concepts is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

3.4: All-Or-Nothing Atomicity II - Pragmatics

Database management applications such as airline reservation systems or banking systems usually require high performance as well as all-or-nothing atomicity, so their designers use streamlined atomicity techniques. The foremost of these techniques sharply separates the reading and writing of data from the failure recovery mechanism. The idea is to minimize the number of storage accesses required for the most common activities (application reads and updates). The trade-off is that the number of storage accesses for rarely-performed activities (failure recovery, which one hopes is actually exercised only occasionally, if at all) may not be minimal. The technique is called **logging**. Logging is also used for purposes other than atomicity, several of which Sidebar 3.4.1 describes.

Sidebar 3.4.1

The many uses of logs

A log is an object whose primary usage method is to append a new record. Log implementations normally provide procedures to read entries from oldest to newest or in reverse order, but there is usually not any procedure for modifying previous entries. Logs are used for several quite distinct purposes, and this range of purposes sometimes gets confused in real-world designs and implementations. Here are some of the most common uses for logs:

1. **Atomicity log.** If one logs the component actions of an all-or-nothing action, together with sufficient before and after information, then a crash recovery procedure can undo (and thus roll back the effects of) all-or-nothing actions that didn't get a chance to complete, or finish all-or-nothing actions that committed but that didn't get a chance to record all of their effects.
2. **Archive log.** If the log is kept indefinitely, it becomes a place where old values of data and the sequence of actions taken by the system or its applications can be kept for review. There are many uses for archive information: watching for failure patterns, reviewing the actions of the system preceding and during a security breach, recovery from application-layer mistakes (e.g., a clerk incorrectly deleted an account), historical study, fraud control, and compliance with record-keeping requirements.
3. Performance log. Most mechanical storage media have much higher performance for sequential access than for random access. Since logs are written sequentially, they are ideally suited to such storage media. It is possible to take advantage of this match to the physical properties of the media by structuring data to be written in the form of a log. When combined with a cache that eliminates most disk reads, a performance log can provide a significant speed-up. As will be seen in the accompanying text, an atomicity log is usually also a performance log.
4. Durability log. If the log is stored on a non-volatile medium—say magnetic tape—that fails in ways and at times that are independent from the failures of the cell storage medium—which might be magnetic disk, then the copies of data in the log are replicas that can be used as backup in case of damage to the copies of the data in cell storage. This kind of log helps implement durable storage. Any log that uses a non-volatile medium, whether intended for atomicity, archiving or performance, typically also helps support durability.

It is essential to have these various purposes—all-or-nothing atomicity, archive, performance, and durable storage—distinct in one's mind when examining or designing a log implementation because they lead to different priorities among design trade-offs. When archive is the goal, low cost of the storage medium is usually more important than quick access because archive logs are large but, in practice, infrequently read. When durable storage is the goal, it may be important to use storage media with different physical properties, so that failure modes will be as independent as possible. When all-or-nothing atomicity or performance is the purpose, minimizing mechanical movement of the storage device becomes a high priority. Because of the competing objectives of different kinds of logs, as a general rule, it is usually a wise move to implement separate, dedicated logs for different functions.

3.4.1: Atomicity Logs

The basic idea behind atomicity logging is to combine the all-or-nothing atomicity of journal storage with the speed of cell storage, by having the application twice record every change to data. The application first **logs** the change in journal storage, and then it **installs** the change in cell storage*. One might think that writing data twice must be more expensive than writing it just once into a version history, but the separation permits specialized optimizations that can make the overall system faster.

The first recording, to journal storage, is optimized for fast writing by creating a single, interleaved version history of all variables, known as a **log**. The information describing each data update forms a record that the application appends to the end of the log. Since there is only one log, a single pointer to the end of the log is all that is needed to find the place to append the record of a change of any variable in the system. If the log medium is magnetic disk, and the disk is used only for logging, and the disk storage management system allocates sectors contiguously, the disk seek arm will need to move only when a disk cylinder is full, thus eliminating most seek delays. As we will see, recovery does involve scanning the log, which is expensive, but recovery should be a rare event. Using a log is thus an example of following the hint to optimize for the common case.

The second recording, to cell storage, is optimized to make reading fast: the application installs by simply overwriting the previous cell storage record of that variable. The record kept in cell storage can be thought of as a cache that, for reading, bypasses the effort that would be otherwise be required to locate the latest version in the log. In addition, by not reading from the log the logging disk's seek arm can remain in position, ready for the next update. The two steps, `LOG` and `INSTALL` , become a different implementation of the `WRITE_NEW_VALUE` interface of Figure 3.3.6. Figure 3.4.1 illustrates this two-step implementation.

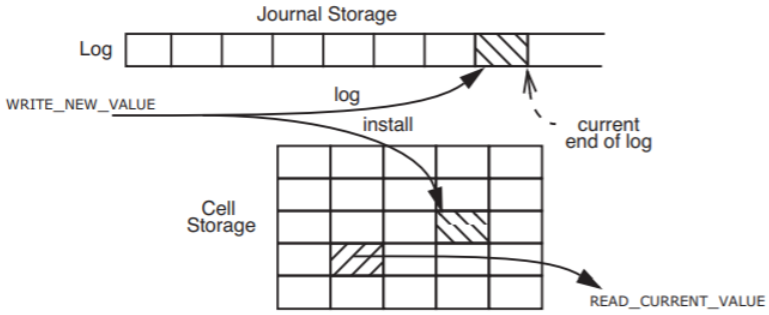


Figure 3.4.1: Logging for all-or-nothing atomicity. The application performs `WRITE_NEW_VALUE` by first appending a record of the new value to the log in journal storage, and then installing the new value in cell storage by overwriting. The application performs `READ_CURRENT_VALUE` by reading just from cell storage.

The underlying idea is that the log is the authoritative record of the outcome of the action. Cell storage is merely a reference copy; if it is lost, it can be reconstructed from the log. The purpose of installing a copy in cell storage is to make both logging and reading faster. By recording data twice, we obtain high performance in writing, high performance in reading, and all-or-nothing atomicity, all at the same time.

There are three common logging configurations, shown in Figure 3.4.2. In each of these three configurations, the log resides in non-volatile storage. For the **in-memory database**, cell storage resides entirely in some volatile storage medium. In the second common configuration, cell storage resides in non-volatile storage along with the log. Finally, high-performance database management systems usually blend the two preceding configurations by implementing a cache for cell storage in a volatile medium, and a potentially independent multilevel memory management algorithm moves data between the cache and non-volatile cell storage.

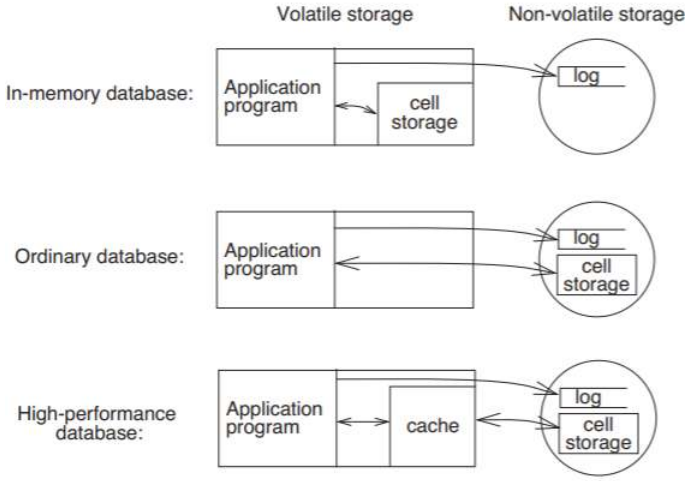


Figure 3.4.2: Three common logging configurations. Arrows show data flow as the application reads, logs, and installs data.

Recording everything twice adds one significant complication to all-or-nothing atomicity because the system can crash between the time a change is logged and the time it is installed. To maintain all-or-nothing atomicity, logging systems follow a protocol that has two fundamental requirements. The first requirement is a constraint on the order of logging and installing. The second requirement is to run an explicit **recovery** procedure after every crash. (We saw a preview of the strategy of using a recovery procedure in Figure 3.3.2, which used a recovery procedure named `CHECK_AND_REPAIR` .)

---

* A hardware architect would say "…it graduates the change to cell storage." This text, somewhat arbitrarily, chooses to use the database management term "install."

3.4.2: Logging Protocols

There are several kinds of atomicity logs that vary in the order in which things are done and in the details of information logged. However, all of them involve the ordering constraint implied by the numbering of the arrows in Figure 3.4.1. The constraint is a version of the *golden rule of atomicity* (never modify the only copy), known as the **write-ahead-log** (WAL) protocol:

> **Write-ahead-log Protocol:**
>
> Log the update *before* installing it.

The reason is that logging appends, but installing overwrites. If an application violates this protocol by installing an update before logging it and then for some reason must abort, or the system crashes, there is no systematic way to discover the installed update and, if necessary, reverse it. The write-ahead-log protocol ensures that if a crash occurs, a recovery procedure can, by consulting the log, systematically find all completed and intended changes to cell storage and either restore those records to old values or set them to new values, as appropriate to the circumstance.

The basic element of an atomicity log is the **log record**. Before an action that is to be all-or-nothing installs a data value, it appends to the end of the log a new record of type `CHANGE` containing, in the general case, three pieces of information (we will later see special cases that allow omitting item 2 or item 3):

1. The identity of the all-or-nothing action that is performing the update.
2. A component action that, if performed, installs the intended value in cell storage. This component action is a kind of an insurance policy in case the system crashes. If the all-or-nothing action commits, but then the system crashes before the action has a chance to perform the install, the recovery procedure can perform the install on behalf of the action. Some systems call this component action the **do** action, others the **redo** action. For mnemonic compatibility with item 3, this text calls it the redo action.
3. A second component action that, if performed, reverses the effect on cell storage of the planned install. This component action is known as the **undo** action because if, after doing the install, the all-or-nothing action aborts or the system crashes, it may be necessary for the recovery procedure to reverse the effect of (undo) the install.

An application appends a log record by invoking the lower-layer procedure `LOG` , which itself must be atomic. The `LOG` procedure is another example of bootstrapping. Starting with, for example, the `ALL_OR_NOTHING_PUT` described earlier in this chapter, a log designer creates a generic `LOG` procedure, and using the `LOG` procedure an application programmer then can implement all-or-nothing atomicity for any properly designed composite action.

As we saw in Figure 3.4.1, `LOG` and `INSTALL` are the logging implementation of the `WRITE_NEW_VALUE` part of the interface of Figure 3.3.6, and `READ_CURRENT_VALUE` is simply a `READ` from cell storage. We also need a logging implementation of the remaining parts of the Figure 3.3.6 interface. The way to implement `NEW_ACTION` is to log a `BEGIN` record that contains just the new all-or-nothing action's identity. As the all-or-nothing action proceeds through its pre-commit phase, it logs `CHANGE` records. To implement `COMMIT` or `ABORT` , the all-or-nothing action logs an `OUTCOME` record that becomes the authoritative indication of the outcome of the all-or-nothing action. The instant that the all-or-nothing action logs the `OUTCOME` record is its commit point. As an example, Figure 3.4.3 shows our by-now-familiar `TRANSFER` action implemented with logging.

```
1  procedure TRANSFER (debit_account, credit_account, amount)
2      my_id ← LOG (BEGIN_TRANSACTION)
3      dbvalue.old ← GET (debit_account)
4      dbvalue.new ← dbvalue.old - amount
5      crvalue.old ← GET (credit_account)
6      crvalue.new ← crvalue.old + amount
7      LOG (CHANGE, my_id,
8          "PUT (debit_account, dbvalue.new)",          //redo action
9          "PUT (debit_account, dbvalue.old)" )         //undo action
10     LOG ( CHANGE, my_id,
11         "PUT (credit_account, crvalue.new)"          //redo action
12         "PUT (credit_account, crvalue.old)")         //undo action
13     PUT (debit_account, dbvalue.new)                 // install
14     PUT (credit_account, crvalue.new)                // install
15     if dbvalue.new > 0 then
16         LOG ( OUTCOME, COMMIT, my_id)
17     else
18         LOG (OUTCOME, ABORT, my_id)
19         signal("Action not allowed. Would make debit account negative.")
20     LOG (END_TRANSACTION, my_id)
```

Figure 3.4.3: An all-or-nothing TRANSFER procedure, implemented with logging.

Because the log is the authoritative record of the action, the all-or-nothing action can perform installs to cell storage at any convenient time that is consistent with the write-ahead-log protocol, either before or after logging the OUTCOME record. The final step of an action is to log an END record, again containing just the action's identity, to show that the action has completed all of its installs. (Logging all four kinds of activity— BEGIN , CHANGE , OUTCOME , and END —is more general than sometimes necessary. As we will see, some logging systems can combine, e.g., OUTCOME and END , or BEGIN with the first CHANGE .) Figure 3.4.4 shows examples of three log records, two typical CHANGE records of an all-or-nothing TRANSFER action, interleaved with the OUTCOME record of some other, perhaps completely unrelated, all-or-nothing action.

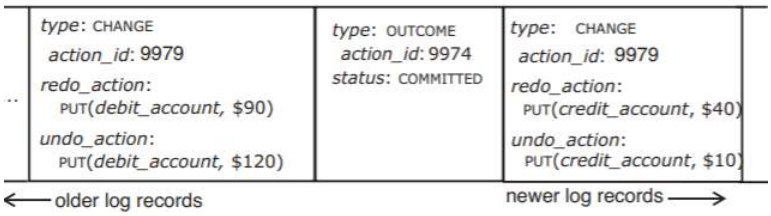


Figure 3.4.4: An example of a section of an atomicity log, showing two CHANGE records for a TRANSFER action that has *action_id* 9979 and the OUTCOME record of a different all-or-nothing action.

One consequence of installing results in cell storage is that for an all-or-nothing action to abort it may have to do some clean-up work. Moreover, if the system involuntarily terminates a thread that is in the middle of an all-or-nothing action (because, for example, the thread has gotten into a deadlock or an endless loop) some entity other than the hapless thread must clean things up. If this clean-up step were omitted, the all-or-nothing action could remain pending indefinitely. The system cannot simply ignore indefinitely pending actions because all-or-nothing actions initiated by other threads are likely to want to use the data that the terminated action changed. (This is actually a before-or-after atomicity concern, one of the places where all-or-nothing atomicity and before-or-after atomicity intersect.)

If the action being aborted did any installs, those installs are still in cell storage, so simply appending to the log an OUTCOME record saying that the action aborted is not enough to make it appear to later observers that the all-or-nothing action did nothing. The solution to this problem is to execute a generic ABORT procedure. The ABORT procedure restores to their old values all cell storage variables that the all-or-nothing action installed. The ABORT procedure simply scans the log backwards looking for log entries created by this all-or-nothing action; for each CHANGE record it finds, it performs the logged *undo_action* , thus restoring the old values in cell storage. The backward search terminates when the ABORT procedure finds that all-or-nothing action's BEGIN record. Figure 3.4.5 illustrates.

```
1  procedure ABORT (action_id)
2      starting at end of log repeat until beginning
3          log_record ← previous record of log
4          if log_record.id = action_id then
5              if (log_record.type = OUTCOME)
6                  then signal ("Can't abort an already completed action.")
7              if (log_record.type = CHANGE)
8                  then perform undo_action of log_record
9              if (log_record.type = BEGIN)
10                 then break repeat
11     LOG (action_id, OUTCOME, ABORTED)                    // Block future undos.
12     LOG (action_id, END)
```

Figure 3.4.5: Generic `ABORT` procedure for a logging system. The argument `action_id` identifies the action to be aborted. An atomic action calls this procedure if it decides to abort. In addition, the operating system may call this procedure if it decides to terminate the action—for example, to break a deadlock or because the action is running too long. The `LOG` procedure must itself be atomic.

The extra work required to undo cell storage installs when an all-or-nothing action aborts is another example of *optimizing for the common case*: one expects that most all-or-nothing actions will commit, and that aborted actions should be relatively rare. The extra effort of an occasional roll back of cell storage values will (one hopes) be more than repaid by the more frequent gains in performance on updates, reads, and commits.

3.4.3: Recovery Procedures

The write-ahead log protocol is the first of the two required protocol elements of a logging system. The second required protocol element is that, following every system crash, the system must run a recovery procedure before it allows ordinary applications to use the data. The details of the recovery procedure depend on the particular configuration of the journal and cell storage with respect to volatile and non-volatile memory.

Consider first recovery for the in-memory database of Figure 3.4.2. Since a system crash may corrupt anything that is in volatile memory, including both the state of cell storage and the state of any currently running threads, restarting a crashed system usually begins by resetting all volatile memory. The effect of this reset is to abandon both the cell storage version of the database and any all-or-nothing actions that were in progress at the time of the crash. On the other hand, the log, since it resides on non-volatile journal storage, is unaffected by the crash and should still be intact.

The simplest recovery procedure performs two passes through the log. On the first pass, it scans the log backward from the last record, so the first evidence it will encounter of each all-or-nothing action is the last record that the all-or-nothing action logged. A backward log scan is sometimes called a LIFO (for last-in, first-out) log review. As the recovery procedure scans backward, it collects in a set the identity and completion status of every all-or-nothing action that logged an `OUTCOME` record before the crash. These actions, whether committed or aborted, are known as **winners**.

When the backward scan is complete the set of winners is also complete, and the recovery procedure begins a forward scan of the log. The reason the forward scan is needed is that restarting after the crash completely reset the cell storage. During the forward scan the recovery procedure performs, in the order found in the log, all of the `REDO` actions of every winner whose `OUTCOME` record says that it `COMMITTED` . Those `REDO` s reinstall all committed values in cell storage, so at the end of this scan, the recovery procedure has restored cell storage to a desirable state. This state is as if every all-or-nothing action that committed before the crash had run to completion, while every all-or-nothing action that aborted or that was still pending at crash time had never existed. The database system can now open for regular business. Figure 3.4.6 illustrates.

```
1  procedure RECOVER () // Recovery procedure for a volatile, in-memory database.
2      winners ← NULL
3      starting at end of log repeat until beginning
4          log_record ← previous record of log
5          if (log_record.type = OUTCOME)
6              then winners ← winners + log_record                    // Set addition.

7      starting at beginning of log repeat until end
8          log_record ← next record of log
9          if (log_record.type= CHANGE)
10             and (outcome_record ← find (log_record.action_id) in winners)
11             and (outcome_record.status = COMMITTED) then
12             perform log_record.redo_action
```

Figure 3.4.6: An idempotent redo-only recovery procedure for an in-memory database. Because RECOVER writes only to volatile storage, if a crash occurs while it is running it is safe to run it again.

This recovery procedure emphasizes the point that a log can be viewed as an authoritative version of the entire database, sufficient to completely reconstruct the reference copy in cell storage.

There exist cases for which this recovery procedure may be overkill, when the durability requirement of the data is minimal. For example, the all-or-nothing action may have been to make a group of changes to soft state in volatile storage. If the soft state is completely lost in a crash, there would be no need to redo installs because the definition of soft state is that the application is prepared to construct new soft state following a crash. Put another way, given the options of "all" or "nothing," when the data is all soft state "nothing" is always an appropriate outcome after a crash.

A critical design property of the recovery procedure is that, if there should be another system crash during recovery, it must still be possible to recover. Moreover, it must be possible for any number of crash-restart cycles to occur without compromising the correctness of the ultimate result. The method is to design the recovery procedure to be **idempotent**—that is, design it so that if it is interrupted and restarted from the beginning it will produce exactly the same result as if it had run to completion to begin with. With the in-memory database configuration, this goal is an easy one: just make sure that the recovery procedure modifies only volatile storage. Then, if a crash occurs during recovery, the loss of volatile storage automatically restores the state of the system to the way it was when the recovery started, and it is safe to run it again from the beginning. If the recovery procedure ever finishes, the state of the cell storage copy of the database will be correct, no matter how many interruptions and restarts intervened.

The ABORT procedure similarly needs to be idempotent because if an all-or-nothing action decides to abort and, while running ABORT , some timer expires, the system may decide to terminate and call ABORT for that same all-or-nothing action. The version of abort in Figure 3.4.5 will satisfy this requirement if the individual undo actions are themselves idempotent.

3.4.4: Other Logging Configurations: Non-Volatile Cell Storage

Placing cell storage in volatile memory is a sweeping simplification that works well for small and medium-sized databases, but some databases are too large for that to be practical, so the designer finds it necessary to place cell storage on some cheaper, non-volatile storage medium such as magnetic disk, as in the second configuration of Figure 3.4.2. But with a non-volatile storage medium, installs survive system crashes, so the simple recovery procedure used with the in-memory database would have two shortcomings:

1. If, at the time of the crash, there were some pending all-or-nothing actions that had installed changes, those changes will survive the system crash. The recovery procedure must reverse the effects of those changes, just as if those actions had aborted.
2. That recovery procedure reinstalls the entire database, even though in this case much of it is probably intact in non-volatile storage. If the database is large enough that it requires non-volatile storage to contain it, the cost of unnecessarily reinstalling it in its entirety at every recovery is likely to be unacceptable.
   In addition, reads and writes to non-volatile cell storage are likely to be slow, so it is nearly always the case that the designer installs a cache in volatile memory, along with a multilevel memory manager, thus moving to the third configuration of Figure 3.4.2. But that addition introduces yet another shortcoming:
3. In a multilevel memory system, the order in which data is written from volatile levels to non-volatile levels is generally under control of a multilevel memory manager, which may, for example, be running a least-recently-used algorithm. As a result, at the instant of the crash some things that were thought to have been installed may not yet have migrated to the non-volatile memory.

To postpone consideration of this shortcoming, let us for the moment assume that the multilevel memory manager implements a write-through cache. (Section 3.4.6, below, will return to the case where the cache is not write-through.) With a write-through cache, we can be certain that everything that the application program has installed has been written to non-volatile storage. This assumption temporarily drops the third shortcoming out of our list of concerns and the situation is the same as if we were using the "Ordinary Database" configuration of Figure 3.4.2 with no cache. But we still have to do something about the first two shortcomings, and we also must make sure that the modified recovery procedure is still idempotent.

To address the first shortcoming, that the database may contain installs from actions that should be undone, we need to modify the recovery procedure of Figure 3.4.6. As the recovery procedure performs its initial backward scan, rather than looking for winners, it instead collects in a set the identity of those all-or-nothing actions that were still in progress at the time of the crash. The actions in this set are known as **losers**, and they can include both actions that committed and actions that did not. Losers are easy to identify because the first log record that contains their identity that is encountered in a backward scan will be something other than an `END` record. To identify the losers, the pseudocode keeps track of which actions logged an `END` record in an auxiliary list named `completeds` . When `RECOVER` comes across a log record belong to an action that is not in completed, it adds that action to the set named `losers` . In addition, as it scans backwards, whenever the recovery procedure encounters a `CHANGE` record belonging to a loser, it performs the `UNDO` action listed in the record. In the course of the LIFO log review, all of the installs performed by losers will thus be rolled back and the state of the cell storage will be as if the all-or-nothing actions of losers had never started. Next, `RECOVER` performs the forward log scan of the log, performing the redo actions of the all-or-nothing actions that committed, as shown in Figure 3.4.7. Finally, the recovery procedure logs an `END` record for every all-or-nothing action in the list of losers. This `END` record transforms the loser into a completed action, thus ensuring that future recoveries will ignore it and not perform its undos again. For future recoveries to ignore aborted losers is not just a performance enhancement, it is essential, to avoid incorrectly undoing updates to those same variables made by future all-or-nothing actions.

```
1  procedure RECOVER ()// Recovery procedure for non-volatile cell memory
2      completeds ← NULL
3      losers ← NULL
4      starting at end of log repeat until beginning
5          log_record ← previous record of log
6          if (log_record.type = END)
7              then completeds ← completeds + log_record          // Set addition.
8          if (log_record.action_id is not in completeds) then
9              losers ← losers + log_record                 // Add if not already in set.
10             if (log_record.type = CHANGE) then
11                 perform log_record.undo_action

12     starting at beginning of log repeat until end
13         log_record ← next record of log
14         if (log_record.type = CHANGE)
15             and (log_record.action_id.status = COMMITTED) then
16             perform log_record.redo_action

17     for each log_record in losers do
18         log (log_record.action_id, END)                    // Show action completed.
```

Figure 3.4.7: An idempotent undo/redo recovery procedure for a system that performs installs to non-volatile cell memory. In this recovery procedure, `losers` are all-or-nothing actions that were in progress at the time of the crash.

As before, the recovery procedure must be idempotent, so that if a crash occurs during recovery the system can just run the recovery procedure again. In addition to the technique used earlier of placing the temporary variables of the recovery procedure in volatile storage, each individual undo action must also be idempotent. For this reason, both redo and undo actions are usually expressed as **blind writes**. A blind write is a simple overwriting of a data value without reference to its previous value. Because a blind write is inherently idempotent, no matter how many times one repeats it, the result is always the same. Thus, if a crash occurs part way through the logging of `END` records of losers, immediately rerunning the recovery procedure will still leave the database correct. Any losers that now have `END` records will be treated as completed on the rerun, but that is OK because the previous attempt of the recovery procedure has already undone their installs.

As for the second shortcoming, that the recovery procedure unnecessarily redoes every install, even installs not belong to losers, we can significantly simplify (and speed up) recovery by analyzing why we have to redo any installs at all. The reason is that, although the WAL protocol requires logging of changes to occur before install, there is no necessary ordering between commit and install. Until a committed action logs its `END` record, there is no assurance that any particular install of that action has actually happened yet. On the other hand, any committed action that has logged an `END` record has completed its installs. The conclusion is that the

recovery procedure does not need to redo installs for any committed action that has logged its END record. A useful exercise is to modify the procedure of Figure 3.4.7 to take advantage of that observation.

It would be even better if the recovery procedure never had to redo *any* installs. We can arrange for that by placing another requirement on the application: it must perform all of its installs *before* it logs its OUTCOME record. That requirement, together with the write-through cache, ensures that the installs of every completed all-or-nothing action are safely in non-volatile cell storage and there is thus never a need to perform any redo actions. (It also means that there is no need to log an END record.) The result is that the recovery procedure needs only to undo the installs of losers, and it can skip the entire forward scan, leading to the simpler recovery procedure of Figure 3.4.8. This scheme, because it requires only undos, is sometimes called **undo logging** or **rollback recovery.** A property of rollback recovery is that for completed actions, cell storage is just as authoritative as the log. As a result, one can garbage-collect the log, discarding the log records of completed actions. The now much smaller log may then be able to fit in a faster storage medium for which the durability requirement is only that it outlast pending actions.

```
1  procedure RECOVER ()              // Recovery procedure for rollback recovery.
2      completeds ← NULL
3      losers ← NULL
4      starting at end of log repeat until beginning          // Perform undo scan.
5          log_record ← previous record of log
6          if (log_record.type = OUTCOME)
7              then completeds ← completeds + log_record      // Set addition.
8          if (log_record.action_id is not in completeds) then
9              losers ← losers + log_record                   // New loser.
10             if (log_record.type = CHANGE) then
11                 perform log_record.undo_action

12     for each log_record in losers do
13         log (log_record.action_id, OUTCOME, ABORT)         // Block future undos.
```

Figure 3.4.8: An idempotent undo-only recovery procedure for rollback logging.

There is an alternative, symmetric constraint used by some logging systems. Rather than requiring that all installs be done *before* logging the OUTCOME record, one can instead require that all installs be done *after* recording the OUTCOME record. With this constraint, the set of CHANGE records in the log that belong to that all-or-nothing action become a description of its intentions. If there is a crash before logging an OUTCOME record, we know that no installs have happened, so the recovery never needs to perform any undos. On the other hand, it may have to perform installs for all-or-nothing actions that committed. This scheme is called **redo logging** or **roll-forward recovery**. Furthermore, because we are uncertain about which installs actually have taken place, the recovery procedure must perform *all* logged installs for all-or-nothing actions that did not log an END record. Any all-or-nothing action that logged an END record must have completed all of its installs, so there is no need for the recovery procedure to perform them. The recovery procedure thus reduces to doing installs just for all-or-nothing actions that were interrupted between the logging of their OUTCOME and END records. Recovery with redo logging can thus be quite swift, though it does require both a backward and forward scan of the entire log.

We can summarize the procedures for atomicity logging as follows:

- Log to journal storage before installing in cell storage (WAL protocol)
- If all-or-nothing actions perform *all* installs to non-volatile storage before logging their OUTCOME record, then recovery needs only to undo the installs of incomplete uncommitted actions. (rollback/undo recovery)
- If all-or-nothing actions perform *no* installs to non-volatile storage before logging their OUTCOME record, then recovery needs only to redo the installs of incomplete committed actions. (roll-forward/redo recovery)
- If all-or-nothing actions are not disciplined about when they do installs to nonvolatile storage, then recovery needs to both redo the installs of incomplete committed actions *and* undo the installs of incomplete uncommitted ones.

In addition to reading and updating memory, an all-or-nothing action may also need to send messages, for example, to report its success to the outside world. The action of sending a message is just like any other component action of the all-or-nothing action. To provide all-or-nothing atomicity, message sending can be handled in a way analogous to memory update. That is, log a CHANGE record with a redo action that sends the message. If a crash occurs after the all-or-nothing action commits, the recovery procedure will perform this redo action along with other redo actions that perform installs. In principle, one could also log an *undo_action* that sends a compensating message ("Please ignore my previous communication!"). However, an all-or-nothing action will usually be careful not to actually send any messages until after the action commits, so roll-forward recovery applies. For

this reason, a designer would not normally specify an undo action for a message or for any other action that has outside-world visibility such as printing a receipt, opening a cash drawer, drilling a hole, or firing a missile.

Incidentally, although much of the professional literature about database atomicity and recovery uses the terms "winner" and "loser" to describe the recovery procedure, different recovery systems use subtly different definitions for the two sets, depending on the exact logging scheme, so it is a good idea to review those definitions carefully.

3.4.5: Checkpoints

Constraining the order of installs to be all before or all after the logging of the `OUTCOME` record is not the only thing we could do to speed up recovery. Another technique that can shorten the log scan is to occasionally write some additional information, known as a **checkpoint**, to non-volatile storage. Although the principle is always the same, the exact information that is placed in a checkpoint varies from one system to another. A checkpoint can include information written either to cell storage or to the log (where it is known as a **checkpoint record**) or both.

Suppose, for example, that the logging system maintains in volatile memory a list of identifiers of all-or-nothing actions that have started but have not yet recorded an `END` record, together with their pending/committed/aborted status, keeping it up to date by observing logging calls. The logging system then occasionally logs this list as a `CHECKPOINT` record. When a crash occurs sometime later, the recovery procedure begins a LIFO log scan as usual, collecting the sets of completed actions and losers. When it comes to a `CHECKPOINT` record it can immediately fill out the set of losers by adding those all-or-nothing actions that were listed in the checkpoint that did not later log an `END` record. This list may include some all-or-nothing actions listed in the `CHECKPOINT` record as `COMMITTED` , but that did not log an `END` record by the time of the crash. Their installs still need to be performed, so they need to be added to the set of losers. The LIFO scan continues, but only until it has found the `BEGIN` record of every loser.

With the addition of `CHECKPOINT` records, the recovery procedure becomes more complex, but is potentially shorter in time and effort:

1. Do a LIFO scan of the log back to the last `CHECKPOINT` record, collecting identifiers of losers and undoing all actions they logged.
2. Complete the list of losers from information in the checkpoint.
3. Continue the LIFO scan, undoing the actions of losers, until every `BEGIN` record belonging to every loser has been found.
4. Perform a forward scan from that point to the end of the log, performing any committed actions belonging to all-or-nothing actions in the list of losers that logged an `OUTCOME` record with status `COMMITTED` .

In systems in which long-running all-or-nothing actions are uncommon, step 3 will typically be quite brief or even empty, greatly shortening recovery. A good exercise is to modify the recovery program of Figure 3.4.7 to accommodate checkpoints.

Checkpoints are also used with in-memory databases, to provide durability without the need to reprocess the entire log after every system crash. A useful checkpoint procedure for an in-memory database is to make a snapshot of the complete database, writing it to one of two alternating (for all-or-nothing atomicity) dedicated non-volatile storage regions, and then logging a `CHECKPOINT` record that contains the address of the latest snapshot. Recovery then involves scanning the log back to the most recent `CHECKPOINT` record, collecting a list of committed all-or-nothing actions, restoring the snapshot described there, and then performing redo actions of those committed actions from the `CHECKPOINT` record to the end of the log. The main challenge in this scenario is dealing with update activity that is concurrent with the writing of the snapshot. That challenge can be met either by preventing all updates for the duration of the snapshot or by applying more complex before-or-after atomicity techniques such as those described in later sections of this chapter.

3.4.6: What if the Cache is Not Write-Through? (Advanced Topic)

Between the log and the write-through cache, the logging configurations just described require two synchronous writes to non-volatile storage for every data update, with attendant delays waiting for the writes to complete. Since the original reason for introducing a log was to increase performance, these two synchronous write delays usually become the system performance bottleneck. Designers who are interested in maximizing performance would prefer to use a cache that is not write-through, so that writes can be deferred until a convenient time when they can be done in batches. Unfortunately, the application then loses control

of the order in which things are actually written to nonvolatile storage. Loss of control of order has a significant impact on our all-or-nothing atomicity algorithms, since they require, for correctness, constraints on the order of writes and certainty about which writes have been done.

The first concern is for the log itself, because the write-ahead log protocol requires that appending a `CHANGE` record to the log precede the corresponding install in cell storage. One simple way to enforce the WAL protocol is to make just log writes write-through, but allow cell storage writes to occur whenever the cache manager finds it convenient. However, this relaxation means that if the system crashes there is no assurance that any particular install has actually migrated to non-volatile storage. The recovery procedure, assuming the worst, cannot take advantage of checkpoints and must again perform installs starting from the beginning of the log. To avoid that possibility, the usual design response is to flush the cache as part of logging each checkpoint record. Unfortunately, flushing the cache and logging the checkpoint must be done as a before-or-after action to avoid getting tangled with concurrent updates, which creates another design challenge. This challenge is surmountable, but the complexity is increasing.

Some systems pursue performance even farther. A popular technique is to write the log to a volatile buffer, and force that entire buffer to non-volatile storage only when an all-or-nothing action commits. This strategy allows batching several `CHANGE` records with the next `OUTCOME` record in a single synchronous write. Although this step would appear to violate the write-ahead log protocol, that protocol can be restored by making the cache used for cell storage a bit more elaborate; its management algorithm must avoid writing back any install for which the corresponding log record is still in the volatile buffer. The trick is to number each log record in sequence, and tag each record in the cell storage cache with the sequence number of its log record. Whenever the system forces the log, it tells the cache manager the sequence number of the last log record that it wrote, and the cache manager is careful never to write back any cache record that is tagged with a higher log sequence number.

We have in this section seen some good examples of the *law of diminishing returns* at work: schemes that improve performance sometimes require significantly increased complexity. Before undertaking any such scheme, it is essential to evaluate carefully how much extra performance one stands to gain.

This page titled 3.4: All-Or-Nothing Atomicity II - Pragmatics is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

3.5: Before-Or-After Atomicity I - Concepts

The mechanisms developed in the previous sections of this chapter provide atomicity in the face of failure, so that other atomic actions that take place after the failure and subsequent recovery find that an interrupted atomic action apparently either executed all of its steps or none of them. This and the next section investigate how to also provide atomicity of concurrent actions, known as **before-or-after atomicity**. In this development we will provide *both* all-or-nothing atomicity *and* before-or-after atomicity, so we will now be able to call the resulting atomic actions **transactions**.

Concurrency atomicity requires additional mechanisms because when an atomic action installs data in cell storage, that data is immediately visible to all concurrent actions. Even though the version history mechanism can hide pending changes from concurrent atomic actions, they can read other variables that the first atomic action plans to change. Thus, the composite nature of a multiple-step atomic action may still be discovered by a concurrent atomic action that happens to look at the value of a variable in the midst of execution of the first atomic action. This indicates that making a composite action atomic with respect to concurrent threads—that is, making it a **before-or-after action**—requires further effort.

Recall that Section 3.2.6 defined the operation of concurrent actions to be correct *if every result is guaranteed to be one that could have been obtained by some purely serial application of those same actions*. Thus, we are looking for techniques that guarantee production of the same result as if concurrent actions had been applied serially yet maximize the performance that can be achieved by allowing concurrency.

In this section, we explore three successively better before-or-after atomicity schemes, where "better" means that the scheme allows more concurrency. To illustrate the concepts we return to version histories, which allow a straightforward and compelling correctness argument for each scheme. Because version histories are rarely used in practice, in the following Section 3.6 we examine a somewhat different approach: locks, which are widely used because they can provide higher performance, but for which correctness arguments are more difficult.

3.5.1: Achieving Before-or-After Atomicity: Simple Serialization

A version history assigns a unique identifier to each atomic action so that it can link tentative versions of variables to the action's outcome record. Suppose that we require that the unique identifiers be consecutive integers, which we interpret as serial numbers, and we modify the procedure `BEGIN_TRANSACTION` by adding enforcement of the following simple serialization rule: each newly created transaction n must, before reading or writing any data, wait until the preceding transaction $n-1$ has either committed or aborted. (To ensure that there is always a transaction $n-1$, assume that the system was initialized by creating a transaction number zero with an `OUTCOME` record in the committed state.) Figure 3.5.1 shows this version of `BEGIN_TRANSACTION`. The scheme forces all transactions to execute in the serial order in which threads happen to invoke `BEGIN_TRANSACTION`. Since that order is a possible serial order of the various transactions, by definition simple serialization will produce transactions that are serialized and thus are correct before-or-after actions. Simple serialization trivially provides before-or-after atomicity, and the transaction is still all-or-nothing, so the transaction is now atomic both in the case of failure and in the presence of concurrency.

```
1  procedure BEGIN_TRANSACTION ()
2      id ← NEW_OUTCOME_RECORD (PENDING)          // Create, initialize, assign id.
3      previous_id ← id - 1
4      wait until previous_id.outcome_record.state ≠ PENDING
5      return id
```

Figure 3.5.1: `BEGIN_TRANSACTION` with the simple serialization discipline to achieve before-or-after atomicity. In order that there be an `id - 1` for every value of `id`, startup of the system must include creating a dummy transaction with `id = 0` and `id.outcome_record.state` set to `COMMITTED`. Pseudocode for the procedure `NEW_OUTCOME_RECORD` appears in Figure 3.5.6.

Simple serialization provides before-or-after atomicity by being too conservative: it prevents all concurrency among transactions, even if they would not interfere with one another. Nevertheless, this approach actually has some practical value—in some applications it may be just the right thing to do, on the basis of simplicity. Concurrent threads can do much of their work in parallel because simple serialization comes into play only during those times that threads are executing transactions, which they generally would be only at the moments they are working with shared variables. If such moments are infrequent or if the actions that need before-or-after atomicity all modify the same small set of shared variables, simple serialization is likely to be just about as effective as any other scheme. In addition, by looking carefully at why it works, we can discover less conservative approaches that allow

more concurrency, yet still have compelling arguments that they preserve correctness. Put another way, the remainder of study of before-or-after atomicity techniques is fundamentally nothing but invention and analysis of increasingly effective—and increasingly complex—performance improvement measures.

The version history provides a useful representation for this analysis. Figure 3.5.2 illustrates in a single figure the version histories of a banking system consisting of four accounts named A , B , C , and D , during the execution of six transactions, with serial numbers 1 through 6. The first transaction initializes all the objects to contain the value 0 and the following transactions transfer various amounts back and forth between pairs of accounts.

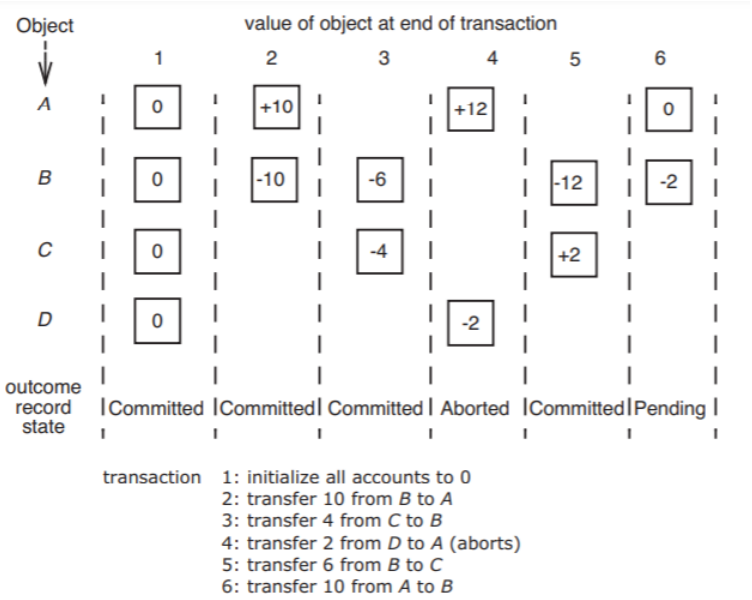


Figure 3.5.2: Version history of a banking system.

This figure provides a straightforward interpretation of why simple serialization works correctly. Consider transaction 3, which must read and write objects B and C in order to transfer funds from one to the other. The way for transaction 3 to produce results as if it ran after transaction 2 is for all of 3's input objects to have values that include all the effects of transaction 2—if transaction 2 commits, then any objects that it changed and that 3 uses should have new values; if transaction 2 aborts, then any objects it tentatively changed and 3 uses should contain the values that they had when transaction 2 started. Since in this example transaction 3 reads B and transaction 2 creates a new version of B , it is clear that for transaction 3 to produce a correct result it must wait until transaction 2 either commits or aborts. Simple serialization requires that wait, and thus ensures correctness.

Figure 3.5.2 also provides some clues about how to increase concurrency. Looking at transaction 4 (the example shows that transaction 4 will ultimately abort for some reason, but suppose we are just starting transaction 4 and don't know that yet), it is apparent that simple serialization is too strict. Transaction 4 reads values only from A and D , yet transaction 3 has no interest in either object. Thus the values of A and D will be the same whether or not transaction 3 commits, and a discipline that forces 4 to wait for 3's completion delays 4 unnecessarily. On the other hand, transaction 4 does use an object that transaction 2 modifies, so transaction 4 must wait for transaction 2 to complete. Of course, simple serialization guarantees that, since transaction 4 can't begin till transaction 3 completes and transaction 3 couldn't have started until transaction 2 completed.

These observations suggest that there may be other, more relaxed, disciplines that can still guarantee correct results. They also suggest that any such discipline will probably involve detailed examination of exactly which objects each transaction reads and writes.

Figure 3.5.2 represents the state history of the entire system in serialization order, but the slightly different representation of Figure 3.5.3 makes that state history more explicit. In Figure 3.5.3 it appears that each transaction has perversely created a new version of every object, with unchanged values in dotted boxes for those objects it did not actually change. This representation emphasizes that the vertical slot for, say, transaction 3 is in effect a reservation in the state history for every object in the system; transaction 3 has an opportunity to propose a new value for any object, if it so wishes.

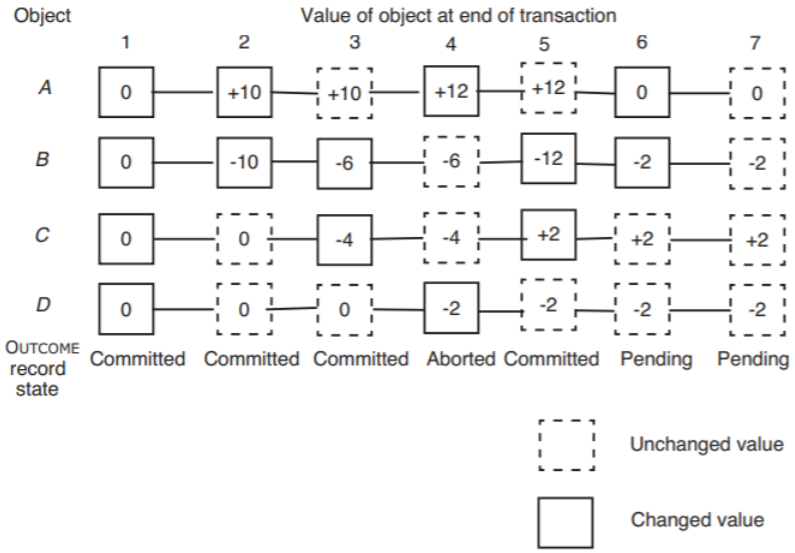


Figure 3.5.3: System state history with unchanged values shown.

The reason that the system state history is helpful to the discussion is that as long as we eventually end up with a state history that has the values in the boxes as shown, the actual order in real time in which individual object values are placed in those boxes is unimportant. For example, in Figure 3.5.3, transaction 3 could create its new version of object `C` before transaction 2 creates its new version of `B` . We don't care when things happen, as long as the result is to fill in the history with the same set of values that would result from strictly following this serial ordering. Making the actual time sequence unimportant is exactly our goal, since that allows us to put concurrent threads to work on the various transactions. There are, of course, constraints on time ordering, but they become evident by examining the state history.

Figure 3.5.3 allows us to see just what time constraints must be observed in order for the system state history to record this particular sequence of transactions. In order for a transaction to generate results appropriate for its position in the sequence, it should use as its input values the latest versions of all of its inputs. If Figure 3.5.3 were available, transaction 4 could scan back along the histories of its inputs `A` and `D` , to the most recent solid boxes (the ones created by transactions 2 and 1, respectively) and correctly conclude that if transactions 2 and 1 have committed then transaction 4 can proceed—even if transaction 3 hasn't gotten around to filling in values for `B` and `C` and hasn't decided whether or not it should commit.

This observation suggests that any transaction has enough information to ensure before-or-after atomicity with respect to other transactions if it can discover the dotted-versus-solid status of those version history boxes to its left. The observation also leads to a specific before-or-after atomicity discipline that will ensure correctness. We call this discipline **mark-point**.

3.5.2: The Mark-Point Discipline

Concurrent threads that invoke `READ_CURRENT_VALUE` as implemented in Figure 3.3.10 can not see a pending version of any variable. That observation is useful in designing a before-or-after atomicity discipline because it allows a transaction to reveal all of its results at once simply by changing the value of its `OUTCOME` record to `COMMITTED` . But in addition to that we need a way for later transactions that need to read a pending version to wait for it to become committed. The way to do that is to modify `READ_CURRENT_VALUE` to wait for, rather than skip over, pending versions created by transactions that are earlier in the sequential ordering (that is, they have a smaller *`caller_id`*), as implemented in lines 4–9 of Figure 3.5.4. Because, with concurrency, a transaction later in the ordering may create a new version of the same variable before this transaction reads it, `READ_CURRENT_VALUE` still skips over any versions created by transactions that have a larger *`caller_id`* . Also, as before, it may be convenient to have a `READ_MY_VALUE` procedure (not shown) that returns pending values previously written by the running transaction.

```
1  procedure READ_CURRENT_VALUE (data_id, this_transaction_id)
2      starting at end of data_id repeat until beginning
3          v ← previous version of data_id
4          last_modifier ← v.action_id
5          if last_modifier ≥ this_transaction_id then skip v        // Keep searching
6          wait until (last_modifier.outcome_record.state ≠ PENDING)
7          if (last_modifier.outcome_record.state = COMMITTED)
8              then return v.state
9              else skip v                                           // Resume search
10     signal ("Tried to read an uninitialized variable")
```

Figure 3.5.4: READ_CURRENT_VALUE for the mark-point discipline.This form of the procedure skips all versions created by transactions later than the calling transaction, and it waits for a pending version created by an earlier transaction until that earlier transaction commits or aborts.

Adding the ability to wait for pending versions in READ_CURRENT_VALUE is the first step; to ensure correct before-or-after atomicity we also need to arrange that all variables that are needed as inputs for a transaction, but that earlier, not-yet-committed transactions plan to modify, have pending versions. To do that we call on the application programmer (for example, the programmer of the TRANSFER transaction) do a bit of extra work: each transaction should create new, pending versions of every variable it intends to modify, and announce when it is finished doing so. Creating a pending version has the effect of marking those variables that are not ready for reading by later transactions, so we will call the point at which a transaction has created them all the **mark point** of the transaction. The transaction announces that it has passed its mark point by calling a procedure named MARK_POINT_ANNOUNCE , which simply sets a flag in the outcome record for that transaction.

The mark-point discipline then is that no transaction can begin reading its inputs until the preceding transaction has reached its mark point or is no longer pending. This discipline requires that each transaction identify which data it will update. If the transaction has to modify some data objects before it can discover the identity of others that require update, it could either delay setting its mark point until it does know all of the objects it will write (which would, of course, also delay all succeeding transactions) or use the more complex discipline described in the next section.

For example, in Figure 3.5.3, the boxes under newly arrived transaction 7 are all dotted; transaction 7 should begin by marking the ones that it plans to make solid. For convenience in marking, we split the WRITE_NEW_VALUE procedure of Figure 3.3.10 into two parts, named NEW_VERSION and WRITE_VALUE , as in Figure 3.5.5. Marking then consists simply of a series of calls to NEW_VERSION . When finished marking, the transaction calls MARK_POINT_ANNOUNCE . It may then go about its business, reading and writing values as appropriate to its purpose.

```
1  procedure NEW_VERSION (reference data_id, this_transaction_id)
2      if this_transaction_id.outcome_record.mark_state = MARKED then
3          signal ("Tried to create new version after announcing mark point!")
4      append new version v to data_id
5      v.value ← NULL
6      v.action_id ← transaction_id

7  procedure WRITE_VALUE (reference data_id, new_value, this_transaction_id)
8      starting at end of data_id repeat until beginning
9          v ← previous version of data_id
10         if v.action_id = this_transaction_id
11             v.value ← new_value
12             return
13     signal ("Tried to write without creating new version!"))
```

Figure 3.5.5: Mark-point discipline versions of NEW_VERSION and WRITE_VALUE .

Finally, we enforce the mark point discipline by putting a test and, depending on its outcome, a wait in BEGIN_TRANSACTION , as in Figure 3.5.6, so that no transaction may begin execution until the preceding transaction either reports that it has reached its mark point or is no longer PENDING . Figure 3.5.6 also illustrates an implementation of MARK_POINT_ANNOUNCE . No changes are needed in procedures ABORT and COMMIT as shown in Figure 3.3.8, so they are not repeated here.

```
1  procedure BEGIN_TRANSACTION ()
2      id ← NEW_OUTCOME_RECORD (PENDING)
3      previous_id ← id - 1
4      wait until (previous_id.outcome_record.mark_state = MARKED)
5          or (previous_id.outcome_record.state ≠ PENDING)
6      return id

7  procedure NEW_OUTCOME_RECORD (starting_state)
8      ACQUIRE (outcome_record_lock)        // Make this a before-or-after action.
9      id ← TICKET (outcome_record_sequencer)
10     allocate id.outcome_record
11     id.outcome_record.state ← starting_state
12     id.outcome_record.mark_state ← NULL
13     RELEASE (outcome_record_lock)
14     return id

15 procedure MARK_POINT_ANNOUNCE (reference this_transaction_id)
16     this_transaction_id.outcome_record.mark_state ← MARKED
```

Figure 3.5.6: The procedures BEGIN_TRANSACTION , NEW_OUTCOME_RECORD , and MARK_POINT_ANNOUNCE for the mark-point discipline. BEGIN_TRANSACTION presumes that there is always a preceding transaction, so the system should be initialized by calling NEW_OUTCOME_RECORD to create an empty initial transaction whose *starting_state* is COMMITTED and immediately calling MARK_POINT_ANNOUNCE for the empty transaction.

Because no transaction can start until the previous transaction reaches its mark point, all transactions earlier in the serial ordering must also have passed their mark points, so every transaction earlier in the serial ordering has already created all of the versions that it ever will. Since READ_CURRENT_VALUE now waits for earlier, pending values to become committed or aborted, it will always return to its client a value that represents the final outcome of all preceding transactions. All input values to a transaction thus contain the committed result of all transactions that appear earlier in the serial ordering, just as if it had followed the simple serialization discipline. The result is thus guaranteed to be exactly the same as one produced by a serial ordering, no matter in what real time order the various transactions actually write data values into their version slots. The particular serial ordering that results from this discipline is, as in the case of the simple serialization discipline, the ordering in which the transactions were assigned serial numbers by NEW_OUTCOME_RECORD .

There is one potential interaction between all-or-nothing atomicity and before-or-after atomicity. If pending versions survive system crashes, at restart the system must track down all PENDING transaction records and mark them ABORTED to ensure that future invokers of READ_CURRENT_VALUE do not wait for the completion of transactions that have forever disappeared.

The mark-point discipline provides before-or-after atomicity by bootstrapping from a more primitive before-or-after atomicity mechanism. As usual in bootstrapping, the idea is to reduce some general problem—here, that problem is to provide before-or-after atomicitiy for arbitrary application programs—to a special case that is amenable to a special-case solution—here, the special case is construction and initialization of a new outcome record. The procedure NEW_OUTCOME_RECORD in Figure 3.5.6 must itself be a before-or-after action because it may be invoked concurrently by several different threads and it must be careful to give out different serial numbers to each of them. It must also create completely initialized outcome records, with value and mark_state set to PENDING and NULL , respectively, because a concurrent thread may immediately need to look at one of those fields. To achieve

before-or-after atomicity, NEW_OUTCOME_RECORD bootstraps from the TICKET procedure to obtain the next sequential serial number, and it uses ACQUIRE and RELEASE to make its initialization steps a before-or-after action. Those procedures in turn bootstrap from still lower-level before-or-after atomicity mechanisms, so we have three layers of bootstrapping.

We can now reprogram the funds TRANSFER procedure of Figure 3.3.10 to be atomic under both failure and concurrent activity, as in Figure 3.5.7. The major change from the earlier version is addition of lines 4 through 6, in which TRANSFER calls NEW_VERSION to mark the two variables that it intends to modify and then calls MARK_POINT_ANNOUNCE . The interesting observation about this program is that most of the work of making actions before-or-after is actually carried out in the called procedures. The only effort or thought required of the application programmer is to identify and mark, by creating new versions, the variables that the transaction will modify.

```
1  procedure TRANSFER (reference debit_account, reference credit_account,
2                                                                amount)
3      my_id ← BEGIN_TRANSACTION ()
4      NEW_VERSION (debit_account, my_id)
5      NEW_VERSION (credit_account, my_id)
6      MARK_POINT_ANNOUNCE (my_id);
7      xvalue ← READ_CURRENT_VALUE (debit_account, my_id)
8      xvalue ← xvalue - amount
9      WRITE_VALUE (debit_account, xvalue, my_id)
10     yvalue ← READ_CURRENT_VALUE (credit_account, my_id)
11     yvalue ← yvalue + amount
12     WRITE_VALUE (credit_account, yvalue, my_id)
13     if xvalue > 0 then
14         COMMIT (my_id)
15     else
16         ABORT (my_id)
17         signal("Negative transfers are not allowed.")
```

Figure 3.5.7: An implementation of the funds transfer procedure that uses the mark point discipline to ensure that it is atomic both with respect to failure and with respect to concurrent activity.

The delays (which under the simple serialization discipline would all be concentrated in `BEGIN_TRANSACTION`) are distributed under the mark-point discipline. Some delays may still occur in `BEGIN_TRANSACTION` , waiting for the preceding transaction to reach its mark point. But if marking is done before any other calculations, transactions are likely to reach their mark points promptly, and thus this delay should be not as great as waiting for them to commit or abort. Delays can also occur at any invocation of `READ_CURRENT_VALUE` , but only if there is really something that the transaction must wait for, such as committing a pending version of a necessary input variable. Thus the overall delay for any given transaction should never be more than that imposed by the simple serialization discipline, and one might anticipate that it will often be less.

A useful property of the mark-point discipline is that it never creates deadlocks. Whenever a wait occurs it is a wait for some transaction *earlier* in the serialization. That transaction may in turn be waiting for a still earlier transaction, but since no one ever waits for a transaction later in the ordering, progress is guaranteed. The reason is that at all times there must be some earliest pending transaction. The ordering property guarantees that this earliest pending transaction will encounter no waits for other transactions to complete, so it, at least, can make progress. When it completes, some other transaction in the ordering becomes earliest, and it now can make progress. Eventually, by this argument, every transaction will be able to make progress. This kind of reasoning about progress is a helpful element of a before-or-after atomicity discipline. In Section 3.6 of this chapter we will encounter before-or-after atomicity disciplines that are correct in the sense that they guarantee the same result as a serial ordering, but they do not guarantee progress. Such disciplines require additional mechanisms to ensure that threads do not end up deadlocked, waiting for one another forever.

Two other minor points are worth noting. First, if transactions wait to announce their mark point until they are ready to commit or abort, the mark-point discipline reduces to the simple serialization discipline. That observation confirms that one discipline is a relaxed version of the other. Second, there are at least two opportunities in the mark-point discipline to discover and report protocol errors to clients. A transaction should never call `NEW_VERSION` after announcing its mark point. Similarly, `WRITE_VALUE` can report an error if the client tries to write a value for which a new version was never created. Both of these error-reporting opportunities are implemented in the pseudocode of Figure 3.5.5.

3.5.3: Optimistic Atomicity: Read-Capture (Advanced Topic)

Both the simple serialization and mark-point disciplines are concurrency control methods that may be described as **pessimistic**. That means that they presume that interference between concurrent transactions is likely, and they actively prevent any possibility of interference by imposing waits at any point where interference might occur. In doing so, they also may prevent some concurrency that would have been harmless to correctness. An alternative scheme, called **optimistic** concurrency control, is to presume that interference between concurrent transactions is unlikely, and allow them to proceed without waiting. Then, watch for actual interference, and if it happens take some recovery action: for example, aborting an interfering transaction and making it restart. (There is a popular tongue-in-cheek characterization of the difference: pessimistic = "ask first", optimistic = "apologize later".) The goal of optimistic concurrency control is to increase concurrency in situations where actual interference is rare.

The system state history of Figure 3.5.3 suggests an opportunity to be optimistic. We could allow transactions to write values into the system state history in any order and at any time, but with the risk that some attempts to write may be met with the response "Sorry, that write would interfere with another transaction. You must abort, abandon this serialization position in the system state history, obtain a later serialization, and rerun your transaction from the beginning."

A specific example of this approach is the **read-capture** discipline. Under the read-capture discipline, there is an option, but not a requirement, of advance marking. Eliminating the requirement of advance marking has the advantage that a transaction does not need to predict the identity of every object it will update—it can discover the identity of those objects as it works. Instead of advance marking, whenever a transaction calls `READ_CURRENT_VALUE` , that procedure makes a mark at this thread's position in the version history of the object it read. This mark tells potential version-inserters earlier in the serial ordering but arriving later in real time that they are no longer allowed to insert—they must abort and try again, using a later serial position in the version history. Had the prospective version inserter gotten there sooner, before the reader had left its mark, the new version would have been acceptable, and the reader would have instead waited for the version inserter to commit, and taken that new value instead of the earlier one. Read-capture gives the reader the power of extending validity of a version through intervening transactions, up to the reader's own serialization position. This view of the situation is illustrated in Figure 3.5.8, which has the same version history as did Figure 3.5.3.

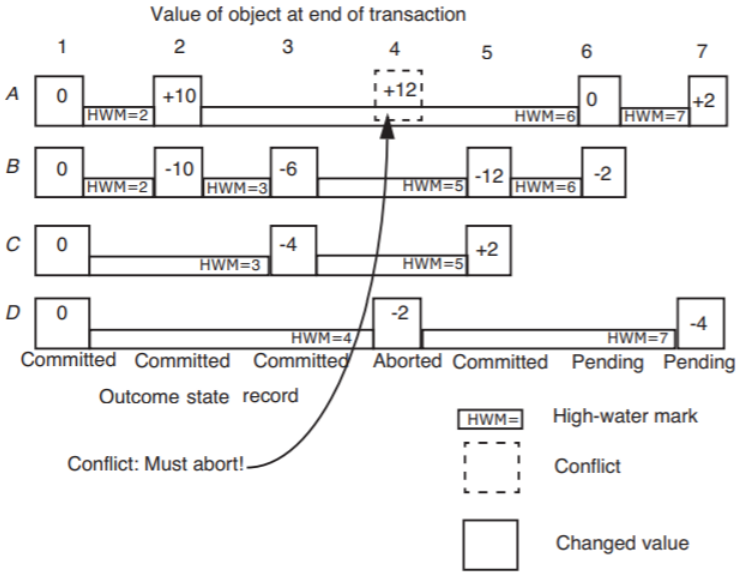


Figure 3.5.8: Version history with high-water marks and the read-capture discipline. First, transaction 6, which is running concurrently with transaction 4, reads variable `A` , thus extending the high-water mark of `A` to 6. Then, transaction 4 (which intends to transfer 2 from `D` to `A`) encounters a conflict when it tries to create a new version of `A` and discovers that the high-water mark of `A` has already been set by transaction 6, so 4 aborts and returns as transaction 7. Transaction 7 retries transaction 4, extending the high-water marks of `A` and `D` to 7.

The key property of read-capture is illustrated by an example in Figure 3.5.8. Transaction 4 was late in creating a new version of object `A` ; by the time it tried to do the insertion, transaction 6 had already read the old value (+10) and thereby extended the validity of that old value to the beginning of transaction 6. Therefore, transaction 4 had to be aborted; it has been reincarnated to try again as transaction 7. In its new position as transaction 7, its first act is to read object `D` , extending the validity of its most recent committed value (zero) to the beginning of transaction 7. When it tries to read object A, it discovers that the most recent version is still uncommitted, so it must wait for transaction 6 to either commit or abort. Note that if transaction 6 should now decide to create a new version of object `C` , it can do so without any problem, but if it should try to create a new version of object `D` , it would run into a conflict with the old, now extended version of `D` , and it would have to abort.

Read-capture is relatively easy to implement in a version history system. We start, as shown in Figure 3.5.9, by adding a new step (at line 9) to `READ_CURRENT_VALUE` . This new step records with each data object a **high-water mark**—the serial number of the highest-numbered transaction that has ever read a value from this object's version history. The high-water mark serves as a warning to other transactions that have earlier serial numbers but are late in creating new versions. The warning is that someone later in the serial ordering has already read a version of this object from earlier in the ordering, so it is too late to create a new

version now. We guarantee that the warning is heeded by adding a step to NEW_VERSION (at line 14), which checks the high-water mark for the object to be written, to see if any transaction with a higher serial number has already read the current version of the object. If not, we can create a new version without concern. But if the transaction serial number in the high-water mark is greater than this transaction's own serial number, this transaction must abort, obtain a new, higher serial number, and start over again.

```
1   procedure READ_CURRENT_VALUE (reference data_id, value, caller_id)
2       starting at end of data_id repeat until beginning
3           v ← previous version of data_id
4           if v.action_id ≥ caller_id then skip v
5           examine v.action_id.outcome_record
6               if PENDING then
7                   WAIT for v.action_id to COMMIT or ABORT
8                   if COMMITTED then
9                       v.high_water_mark ← max(v.high_water_mark, caller_id)
10                      return v.value
11                  else skip v                          // Continue backward search
12      signal ("Tried to read an uninitialized variable!")

13  procedure NEW_VERSION (reference data_id, caller_id)
14      if (caller_id < data_id.high_water_mark)        // Conflict with later reader.
15      or (caller_id < (LATEST_VERSION[data_id].action_id))     // Blind write conflict.
16      then ABORT this transaction and terminate this thread
17      add new version v at end of data_id
18      v.value ← 0
19      v.action_id ← caller_id

20  procedure WRITE_VALUE (reference data_id, new_value, caller_id)
21      locate version v of data_id.history such that v.action_id = caller_id
22              (if not found, signal ("Tried to write without creating new version!"))
23      v.value ← new_value
```

Figure 3.5.9: Read-capture forms of READ_CURRENT_VALUE , NEW_VERSION , and WRITE_VALUE .

We have removed all constraints on the real-time sequence of the constituent steps of the concurrent transaction, so there is a possibility that a high-numbered transaction will create a new version of some object, and then later a low-numbered transaction will try to create a new version of the same object. Since our NEW_VERSION procedure simply tacks new versions on the end of the object history, we could end up with a history in the wrong order. The simplest way to avoid that mistake is to put an additional test in NEW_VERSION (at line 15), to ensure that every new version has a client serial number that is larger than the serial number of the next previous version. If not, NEW_VERSION aborts the transaction, just as if a read-capture conflict had occurred. (This test aborts only those transactions that perform conflicting **blind writes**, which are uncommon. If either of the conflicting transactions reads the value before writing it, the setting and testing of *high_water_mark* will catch and prevent the conflict.)

The first question one must raise about this kind of algorithm is whether or not it actually works: is the result always the same as some serial ordering of the concurrent transactions? Because the read-capture discipline permits greater concurrency than does mark-point, the correctness argument is a bit more involved. The induction part of the argument goes as follows:

1. The WAIT for PENDING values in READ_CURRENT_VALUE ensures that if any pending transaction $k < n$ has modified any value that is later read by transaction n, transaction n will wait for transaction k to commit or abort.
2. The setting of the high-water mark when transaction n calls READ_CURRENT_VALUE , together with the test of the high-water mark in NEW_VERSION , ensures that if any transaction $j < n$ tries to modify any value after transaction n has read that value, transaction j will abort and not modify that value.
3. Therefore, every value that READ_CURRENT_VALUE returns to transaction n will include the final effect of all preceding transactions $1 \ldots n-1$.
4. Therefore, every transaction n will act as if it serially follows transaction $n-1$.

Optimistic coordination disciplines such as read-capture have the possibly surprising effect that something done by a transaction later in the serial ordering can cause a transaction earlier in the ordering to abort. This effect is the price of optimism; to be a good candidate for an optimistic discipline, an application probably should not have a lot of data interference.

A subtlety of read-capture is that it is necessary to implement bootstrapping before-or-after atomicity in the procedure NEW_VERSION , by adding a lock and calls to ACQUIRE and RELEASE , because NEW_VERSION can now be called by two concurrent threads that happen to add new versions to the same variable at about the same time. In addition, NEW_VERSION

must be careful to keep versions of the same variable in transaction order, so that the backward search performed by READ_CURRENT_VALUE works correctly.

There is one final detail, an interaction with all-or-nothing recovery. High water marks should be stored in volatile memory, so that following a crash (which has the effect of aborting all pending transactions) the high water marks automatically disappear and thus don't cause unnecessary aborts.

3.5.4: Does Anyone Actually Use Version Histories for Before-or-After Atomicity?

The answer is yes, but the most common use is in an application not likely to be encountered by a software specialist. Legacy processor architectures typically provide a limited number of registers (the "architectural registers") in which the programmer can hold temporary results, but modern large scale integration technology allows space on a physical chip for many more physical registers than the architecture calls for. More registers generally allow better performance, especially in multiple-issue processor designs, which execute several sequential instructions concurrently whenever possible. To allow use of the many physical registers, a register mapping scheme known as **register renaming** implements a version history for the architectural registers. This version history allows instructions that would interfere with each other only because of a shortage of registers to execute concurrently.

For example, Intel Pentium processors, which are based on an x86 instruction set architecture, have only eight architectural registers. The Pentium 4 has 128 physical registers, and a register renaming scheme based on a circular reorder buffer. A reorder buffer resembles a direct hardware implementation of the procedures NEW_VERSION and WRITE_VALUE of Figure 3.5.5. As each instruction issues (which corresponds to BEGIN_TRANSACTION), it is assigned the next sequential slot in the reorder buffer. The slot is a map that maintains a correspondence between two numbers: the number of the architectural register that the programmer specified to hold the output value of the instruction, and the number of one of the 128 physical registers, the one that will actually hold that output value. Since machine instructions have just one output value, assigning a slot in the reorder buffer implements in a single step the effect of both NEW_OUTCOME_RECORD and NEW_VERSION . Similarly, when the instruction commits, it places its output in that physical register, thereby implementing WRITE_VALUE and COMMIT as a single step.

Figure 3.5.10 illustrates register renaming with a reorder buffer. In the program sequence of that example, instruction n uses architectural register five to hold an output value that instruction $n+1$ will use as an input. Instruction $n+2$ loads architectural register five from memory. Register renaming allows there to be two (or more) versions of register five simultaneously, one version (in physical register 42) containing a value for use by instructions n and $n+1$ and the second version (in physical register 29) to be used by instruction $n+2$. The performance benefit is that instruction $n+2$ (and any later instructions that write into architectural register 5) can proceed concurrently with instructions n and $n+1$. An instruction following instruction $n+2$ that requires the new value in architectural register five as an input uses a hardware implementation of READ_CURRENT_VALUE to locate the most recent preceding mapping of architectural register five in the reorder buffer. In this case that most recent mapping is to physical register 29. The later instruction then stalls, waiting for instruction $n+2$ to write a value into physical register 29. Later instructions that reuse architectural register five for some purpose that does not require that version can proceed concurrently.

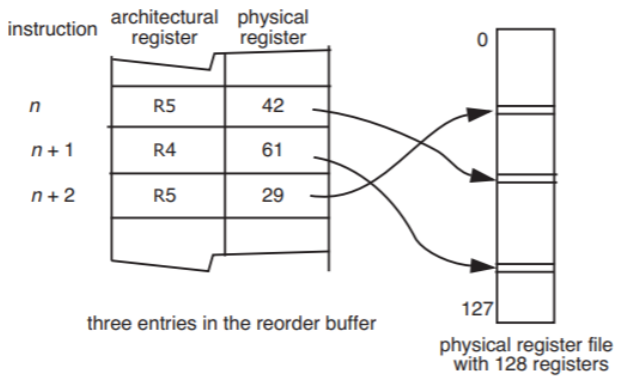


Figure 3.5.10: Example showing how a reorder buffer maps architectural register numbers to physical register numbers. The program sequence corresponding to the three entries is:

```
n          R5 ← R4 × R2           // Write a result in register five.
n + 1      R4 ← R5 + R1           // Use result in register five.
n + 2      R5 ← READ (117492)     // Write content of a memory cell in register five.
```

Instructions n and $n+2$ both write into register R5, so R5 has two versions, with mappings to physical registers 42 and 29, respectively. Instruction $n+2$ can thus execute concurrently with instructions n and $n+1$.

Although register renaming is conceptually straightforward, the mechanisms that prevent interference when there are dependencies between instructions tend to be more intricate than either of the mark-point or read-capture disciplines, so this description has been oversimplified. For more detail, the reader should consult a textbook on processor architecture: for example, *Computer Architecture, a Quantitative Approach*, by Hennessy and Patterson*.

The Oracle database management system offers several before-or-after atomicity methods, one of which it calls "serializable," though the label may be a bit misleading. This method uses a before-or-after atomicity scheme that the database literature calls **snapshot isolation**. The idea is that when a transaction begins, the system conceptually takes a snapshot of every committed value and the transaction reads all of its inputs from that snapshot. If two concurrent transactions (which might start with the same snapshot) modify the same variable, the first one to commit wins; the system aborts the other one with a "serialization error." This scheme effectively creates a limited variant of a version history that, in certain situations, does not always ensure that concurrent transactions are correctly coordinated.

Another specialized variant implementation of version histories, known as **transactional memory**, is a discipline for creating atomic actions from arbitrary instruction sequences that make multiple references to primary memory. Transactional memory was first suggested in 1993, and with widespread availability of multicore processors has become the subject of quite a bit of recent research interest because it allows the application programmer to use concurrent threads without having to deal with locks. The discipline is to mark the beginning of an instruction sequence that is to be atomic with a "begin transaction" instruction, direct all ensuing `STORE` instructions to a hidden copy of the data that concurrent threads cannot read, and at end of the sequence check to see that nothing read or written during the sequence was modified by some other transaction that committed first. If the check finds no such earlier modifications, the system commits the transaction by exposing the hidden copies to concurrent threads; otherwise it discards the hidden copies and the transaction aborts. Because it defers all discovery of interference to the commit point, this discipline is even more optimistic than the read-capture discipline described in Section 3.5.3 above, so it is most useful in situations where interference between concurrent threads is possible but unlikely. Transactional memory has been experimentally implemented in both hardware and software. Hardware implementations typically involve tinkering with either a cache or a reorder buffer to make it defer writing hidden copies back to primary memory until commit time, while software implementations create hidden copies of changed variables somewhere else in primary memory. As with instruction renaming, this description of transactional memory is somewhat oversimplified, and the interested reader should consult the literature for fuller explanations.

Other software implementations of version histories for before-or-after atomicity have been explored primarily in research environments. Designers of database systems usually use locks rather than version histories because there is more experience in achieving high performance with locks. Achieving before-or-after atomicity by using locks systematically is the subject of the next section of this chapter.

---

* David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, fourth edition, 2007. ISBN: 978–0–12–370490–0. 704 + various pages (paperback). The cover gives the authors' names in the opposite order.

> This book provides a spectacular tour-de-force that explores much of the design space of current computer architecture. One of the best features is that each area includes a discussion of misguided ideas and their pitfalls. Even though the subject matter gets sophisticated, the book is always readable. The book is opinionated (with a strong bias toward RISC architecture), but nevertheless this is a definitive work on computer organization from the system perspective.

This page titled 3.5: Before-Or-After Atomicity I - Concepts is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

3.6: Before-Or-After Atomicity II - Pragmatics

The previous section showed that a version history system that provides all-or-nothing atomicity can be extended to also provide before-or-after atomicity. When the all-or-nothing atomicity design uses a log and installs data updates in cell storage, other, concurrent actions can again immediately see those updates, so we again need a scheme to provide before-or-after atomicity. When a system uses logs for all-or-nothing atomicity, it usually adopts the mechanism of **locks** for before-or-after atomicity. However, programming with locks is hazardous, and the traditional programming technique of debugging until the answers seem to be correct is unlikely to catch all locking errors. We now revisit locks, this time with the goal of using them in stylized ways that allow us to develop arguments that the locks correctly implement before-or-after atomicity.

3.6.1: Locks

A **lock** is a flag associated with a data object and set by an action to warn other, concurrent, actions not to read or write the object. Conventionally, a locking scheme involves two procedures:

```
ACQUIRE (A.lock)
```

marks a lock variable associated with object `A` as having been acquired. If the object is already acquired, `ACQUIRE` waits until the previous acquirer releases it.

```
RELEASE (A.lock)
```

unmarks the lock variable associated with `A` , perhaps ending some other action's wait for that lock. For the moment, we assume that the semantics of a lock follow a single-acquire protocol: if two or more actions attempt to acquire a lock at about the same time, only one will succeed; the others must find the lock already acquired. In Section 3.6.4 we will consider some alternative protocols: for example, one that permits several readers of a variable as long as there is no one writing it.

The biggest problem with locks is that programming errors can create actions that do not have the intended before-or-after property. Such errors can open the door to races that, because the interfering actions are timing dependent, can make it extremely difficult to figure out what went wrong. Thus a primary goal is that coordination of concurrent transactions should be arguably correct. For locks, the way to achieve this goal is to follow three steps systematically:

- Develop a locking discipline that specifies which locks must be acquired and when.
- Establish a compelling line of reasoning that concurrent transactions that follow the discipline will have the before-or-after property.
- Interpose a lock manager, a program that enforces the discipline, between the programmer and the `ACQUIRE` and `RELEASE` procedures.

Many locking disciplines have been designed and deployed, including some that fail to correctly coordinate transactions (for an example, see Exercise 3.10.5. We examine three disciplines that succeed. Each allows more concurrency than its predecessor, though even the best one is not capable of guaranteeing that concurrency is maximized.

The first, and simplest, discipline that coordinates transactions correctly is the **system-wide lock**. When the system first starts operation, it creates a single lockable variable named, for example, `System` , in volatile memory. The discipline is that every transaction must start with

```
begin_transaction
ACQUIRE (System.lock)
...
```

and every transaction must end with

```
...
RELEASE (System.lock)
end_transaction
```

A system can even enforce this discipline by including the `ACQUIRE` and `RELEASE` steps in the code sequence generated for `begin_transaction` and `end_transaction` , independent of whether the result was `COMMIT` or `ABORT` . Any programmer who creates a new transaction then has a guarantee that it will run either before or after any other transactions.

The systemwide lock discipline allows only one transaction to execute at a time. It serializes potentially concurrent transactions in the order that they call `ACQUIRE` . The systemwide lock discipline is in all respects identical to the simple serialization discipline of Section 3.5.1. In fact, the simple serialization pseudocode

```
id ← NEW_OUTCOME_RECORD ()
preceding_id ← id - 1
wait until preceding_id.outcome_record.value ≠ PENDING
...
COMMIT (id) [or ABORT (id)]
```

and the systemwide lock invocation

```
ACQUIRE (System.lock)
...
RELEASE (System.lock)
```

are actually just two implementations of the same idea.

As with simple serialization, systemwide locking restricts concurrency in cases where it doesn't need to because it locks all data touched by every transaction. For example, if systemwide locking were applied to the funds `TRANSFER` program of Figure 3.3.11, only one transfer could occur at a time, even though any individual transfer involves only two out of perhaps several million accounts, so there would be many opportunities for concurrent, non-interfering transfers. Thus there is an interest in developing less restrictive locking disciplines. The starting point is usually to employ a finer lock **granularity**: lock smaller objects, such as individual data records, individual pages of data records, or even fields within records. The trade-offs in gaining concurrency are first, that when there is more than one lock, more time is spent acquiring and releasing locks and second, correctness arguments become more complex. One hopes that the performance gain from concurrency exceeds the cost of acquiring and releasing the multiple locks. Fortunately, there are at least two other disciplines for which correctness arguments are feasible: **simple locking** and **two-phase locking**.

3.6.2: Simple Locking

The second locking discipline, known as **simple locking**, is similar in spirit to, though not quite identical with, the mark-point discipline. The simple locking discipline has two rules. First, each transaction must acquire a lock for every shared data object it intends to read or write before doing any actual reading and writing. Second, it may release its locks only after the transaction installs its last update and commits or completely restores the data and aborts. Analogous to the mark point, the transaction has what is called a **lock point**: the first instant at which it has acquired all of its locks. The collection of locks it has acquired when it reaches its lock point is called its **lock set**. A lock manager can enforce simple locking by requiring that each transaction supply its intended lock set as an argument to the `begin_transaction` operation, which acquires all of the locks of the lock set, if necessary waiting for them to become available. The lock manager can also interpose itself on all calls to read data and to log changes, to verify that they refer to variables that are in the lock set. The lock manager also intercepts the call to commit or abort (or, if the application uses roll-forward recovery, to log an `END` record) at which time it automatically releases all of the locks of the lock set.

The simple locking discipline correctly coordinates concurrent transactions. We can make that claim using a line of argument analogous to the one used for correctness of the mark-point discipline. Imagine that an all-seeing outside observer maintains an ordered list, where it adds each transaction identifier as soon as the transaction reaches its lock point and removes each transaction from the list when it begins to release its locks. Under the simple locking discipline each transaction has agreed not to read or write anything until that transaction has been added to the observer's list. We also know that all transactions that precede this one in the list must have already passed their lock point. Since no data object can appear in the lock sets of two transactions, no data object in any transaction's lock set appears in the lock set of the transaction preceding it in the list, and by induction to any transaction earlier in the list. Thus all of this transaction’s input values are the same as they will be when the preceding transaction in the list commits or aborts. The same argument applies to the transaction before the preceding one, so all inputs to any transaction are identical to the inputs that would be available if all the transactions ahead of it in the list ran serially, in the order of the list. Thus the simple locking discipline ensures that this transaction runs completely after the preceding one and completely before the next one. Concurrent transactions will produce results as if they had been serialized in the order that they reached their lock points.

As with the mark-point discipline, simple locking can miss some opportunities for concurrency. In addition, the simple locking discipline creates a problem that can be significant in some applications. Because it requires the transaction to acquire a lock on every shared object that it will either read *or* write (recall that the mark-point discipline requires marking only of shared objects that the transaction will write), applications that discover which objects need to be read by reading other shared data objects have no alternative but to lock every object that they *might* need to read. To the extent that the set of objects that an application *might* need to read is larger than the set for which it eventually *does* read, the simple locking discipline can interfere with opportunities for concurrency. On the other hand, when the transaction is straightforward (such as the `TRANSFER` transaction of Figure 3.3.11, which needs to lock only two records, both of which are known at the outset), simple locking can be effective.

3.6.3: Two-Phase Locking

The third locking discipline, called **two-phase locking**, like the read-capture discipline, avoids the requirement that a transaction must know in advance which locks to acquire. Two-phase locking is widely used, but it is harder to argue that it is correct. The two-phase locking discipline allows a transaction to acquire locks as it proceeds, and the transaction may read or write a data object as soon as it acquires a lock on that object. The primary constraint is that the transaction may not release any locks until it passes its lock point. Furthermore, the transaction can release a lock on an object that it only reads any time after it reaches its lock point *if* it will never need to read that object again, even to abort. The name of the discipline comes about because the number of locks acquired by a transaction monotonically increases up to the lock point (the first phase), after which it monotonically decreases (the second phase). Just as with simple locking, two-phase locking orders concurrent transactions so that they produce results as if they had been serialized in the order they reach their lock points. A lock manager can implement two-phase locking by intercepting all calls to read and write data; it acquires a lock (perhaps having to wait) on the first use of each shared variable. As with simple locking, it then holds the locks until it intercepts the call to commit, abort, or log the `END` record of the transaction, at which time it releases them all at once.

The extra flexibility of two-phase locking makes it harder to argue that it guarantees before-or-after atomicity. Informally, once a transaction has acquired a lock on a data object, the value of that object is the same as it will be when the transaction reaches its lock point, so reading that value now must yield the same result as waiting till then to read it. Furthermore, releasing a lock on an object that it hasn't modified must be harmless if this transaction will never look at the object again, even to abort. A formal argument that two-phase locking leads to correct before-or-after atomicity can be found in most advanced texts on concurrency control and transactions. See, for example, *Transaction Processing*, by Gray and Reuter [see Suggestions for Further Reading 2.1.1].

The two-phase locking discipline can potentially allow more concurrency than the simple locking discipline, but it still unnecessarily blocks certain serializable, and therefore correct, action orderings. For example, suppose transaction T_1 reads `X` and writes `Y`, while transaction T_2 just does a (blind) write to `X`. Because the lock sets of T_1 and T_2 intersect at variable `X`, the two-phase locking discipline will force transaction T_2 to run either completely before or completely after T_1. But the sequence

T_1: `READ X`
T_2: `WRITE X`
T_1: `WRITE Y`

in which the write of T_2 occurs between the two steps of T_1 yields the same result as running T_2 completely before T_1, so the result is always correct, even though this sequence would be prevented by two-phase locking. (Under two-phase locking, the earliest that T_1 can release its lock on `X` is after it acquires the lock on `Y`, so it unnecessarily delays the blind write of T_2.) Disciplines that allow all possible concurrency while at the same time ensuring before-or-after atomicity are quite difficult to devise. (Theorists identify the problem as NP-complete.)

There are two interactions between locks and logs that require some thought: (1) individual transactions that abort, and (2) system recovery. Aborts are the easiest to deal with. Since we require that an aborting transaction restore its changed data objects to their original values before releasing any locks, no special account need be taken of aborted transactions. For purposes of before-or-after atomicity they look just like committed transactions that didn't change anything. The rule about not releasing any locks on modified data before the end of the transaction is essential to accomplishing an abort. If a lock on some modified object were released, and then the transaction decided to abort, it might find that some other transaction has now acquired that lock and changed the object again. Backing out an aborted change is likely to be impossible unless the locks on modified objects have been held.

The interaction between log-based recovery and locks is less obvious. The question is whether locks themselves are data objects for which changes should be logged. To analyze this question, suppose there is a system crash. At the completion of crash recovery there should be no pending transactions because any transactions that were pending at the time of the crash should have been rolled back by the recovery procedure, and recovery does not allow any new transactions to begin until it completes. Since locks exist only to coordinate pending transactions, it would clearly be an error if there were locks still set when crash recovery is complete. That observation suggests that locks belong in volatile storage, where they will automatically disappear on a crash, rather than in nonvolatile storage, where the recovery procedure would have to hunt them down to release them. The bigger question, however, is whether or not the log-based recovery algorithm will construct a correct system state—correct in the sense that it could have arisen from some serial ordering of those transactions that committed before the crash.

Continue to assume that the locks are in volatile memory, and at the instant of a crash all record of the locks is lost. Some set of transactions—the ones that logged a `BEGIN` record but have not yet logged an `END` record—may not have been completed. But we know that the transactions that were not complete at the instant of the crash had nonoverlapping lock sets at the moment that the lock values vanished. The recovery algorithm of Figure 3.4.7 will systematically `UNDO` or `REDO` installs for the incomplete transactions, but every such `UNDO` or `REDO` must modify a variable whose lock was in some transaction's lock set at the time of the crash. Because those lock sets must have been non-overlapping, those particular actions can safely be redone or undone without concern for before-or-after atomicity during recovery. Put another way, the locks created a particular serialization of the transactions and the log has captured that serialization. Since `RECOVER` performs `UNDO` actions in reverse order as specified in the log, and it performs `REDO` actions in forward order, again as specified in the log, `RECOVER` reconstructs exactly that same serialization. Thus even a recovery algorithm that reconstructs the entire database from the log is guaranteed to produce the same serialization as when the transactions were originally performed. So long as no new transactions begin until recovery is complete, there is no danger of miscoordination, despite the absence of locks during recovery.

3.6.4: Performance Optimizations

Most logging-locking systems are substantially more complex than the description so far might lead one to expect. The complications primarily arise from attempts to gain performance. In Section 3.4.6 we saw how buffering of disk I/O in a volatile memory cache, to allow reading, writing, and computation to go on concurrently, can complicate a logging system. Designers sometimes apply two performance-enhancing complexities to locking systems: physical locking and adding lock compatibility modes.

A performance-enhancing technique driven by buffering of disk I/O and physical media considerations is to choose a particular lock granularity known as **physical locking**. If a transaction makes a change to a six-byte object in the middle of a 1000-byte disk sector, or to a 1500-byte object that occupies parts of two disk sectors, there is a question about which "variable" should be locked: the object, or the disk sector(s)? If two concurrent threads make updates to unrelated data objects that happen to be stored in the same disk sector, then the two disk writes must be coordinated. Choosing the right locking granularity can make a big performance difference.

Locking application-defined objects without consideration of their mapping to physical disk sectors is appealing because it is understandable to the application writer. For that reason, it is usually called **logical locking**. In addition, if the objects are small, it apparently allows more concurrency: if another transaction is interested in a different object that is in the same disk sector, it could proceed in parallel. However, a consequence of logical locking is that logging must also be done on the same logical objects. Different parts of the same disk sector may be modified by different transactions that are running concurrently, and if one transaction commits but the other aborts, neither the old nor the new disk sector is the correct one to restore following a crash; the log entries must record the old and new values of the individual data objects that are stored in the sector. Finally, recall that a high-performance logging system with a cache must, at commit time, force the log to disk and keep track of which objects in the cache it is safe to write to disk without violating the write-ahead log protocol. So logical locking with small objects can escalate cache record-keeping.

Backing away from the details, high-performance disk management systems typically require that the argument of a `PUT` call be a block whose size is commensurate with the size of a disk sector. Thus the real impact of logical locking is to create a layer between the application and the disk management system that presents a logical, rather than a physical, interface to its transaction clients; such things as data object management and garbage collection within disk sectors would go into this layer. The alternative is to tailor the logging and locking design to match the native granularity of the disk management system. Since matching the

logging and locking granularity to the disk write granularity can reduce the number of disk operations, both logging changes to and locking blocks that correspond to disk sectors rather than individual data objects is a common practice.

Another performance refinement appears in most locking systems: the specification of **lock compatibility modes**. The idea is that when a transaction acquires a lock, it can specify what operation (for example, `READ` or `WRITE`) it intends to perform on the locked data item. If that operation is compatible—in the sense that the result of concurrent transactions is the same as some serial ordering of those transactions—then this transaction can be allowed to acquire a lock even though some other transaction has already acquired a lock on that same data object.

The most common example involves replacing the single-acquire locking protocol with the **multiple-reader, single-writer protocol**. According to this protocol, one can allow any number of readers to simultaneously acquire read-mode locks for the same object. The purpose of a read-mode lock is to ensure that no other thread can change the data while the lock is held. Since concurrent readers do not present an update threat, it is safe to allow any number of them. If another transaction needs to acquire a write-mode lock for an object on which several threads already hold read-mode locks, that new transaction will have to wait for all of the readers to release their read-mode locks. There are many applications in which a majority of data accesses are for reading, and for those applications the provision of read-mode lock compatibility can reduce the amount of time spent waiting for locks by orders of magnitude. At the same time, the scheme adds complexity, both in the mechanics of locking and also in policy issues, such as what to do if, while a prospective writer is waiting for readers to release their read-mode locks, another thread calls to acquire a read-mode lock. If there is a steady stream of arriving readers, a writer could be delayed indefinitely.

This description of performance optimizations and their complications is merely illustrative, to indicate the range of opportunities and kinds of complexity that they engender. There are many other performance-enhancement techniques, some of which can be effective, and others that are of dubious value; most have different values depending on the application. For example, some locking disciplines compromise before-or-after atomicity by allowing transactions to read data values that are not yet committed. As one might expect, the complexity of reasoning about what can or cannot go wrong in such situations escalates. If a designer intends to implement a system using performance enhancements such as buffering, lock compatibility modes, or compromised before-or-after atomicity, it would be advisable to study carefully the book by Gray and Reuter, as well as existing systems that implement similar enhancements.

3.6.5: Deadlock; Making Progress

The possibility of **deadlock** is an inevitable consequence of using locks to coordinate concurrent activities. Any number of concurrent transactions can get hung up in a deadlock, either waiting for one another, or simply waiting for a lock to be released by some transaction that is already deadlocked. Deadlock leaves us a significant loose end: correctness arguments ensure us that any transactions that complete will produce results as though they were run serially, but they say nothing about whether or not any transaction will ever complete. In other words, our system may ensure *correctness*, in the sense that no wrong answers ever come out, but it does not ensure *progress*—no answers may come out at all.

As with methods for concurrency control, methods for coping with deadlock can also be described as pessimistic or optimistic. Pessimistic methods take *a priori* action to prevent deadlocks from happening. Optimistic methods allow concurrent threads to proceed, detect deadlocks if they happen, and then take action to fix things up. Here are some of the most popular methods:

1. **Lock ordering** (pessimistic). Number the locks uniquely, and require that transactions acquire locks in ascending numerical order. With this plan, when a transaction encounters an already-acquired lock, it is always safe to wait for the transaction that has already acquired that lock. Said transaction cannot be waiting for any locks that this transaction has already acquired, all of which are lower in number. There is thus a guarantee that somewhere, at least one transaction (the one holding the highest-numbered lock) can always make progress. When that transaction finishes, it will release all of its locks, and some other transaction will become the one that is guaranteed to be able to make progress. A generalization of lock ordering that may eliminate some unnecessary waits is to arrange the locks in a lattice and require that they be acquired in some lattice traversal order. The trouble with lock ordering, as with simple locking, is that some applications may not be able to predict all of the locks they need before acquiring the first one.
2. **Backing out** (optimistic): An elegant strategy devised by Andre Bensoussan in 1966 allows a transaction to acquire locks in any order, but if it encounters an already-acquired lock with a number lower than one it has previously acquired itself, the transaction must back up (in terms of this chapter, `UNDO` previous actions) just far enough to release its higher-numbered locks, wait for the lower-numbered lock to become available, acquire that lock, and then `REDO` the backed-out actions.

3. **Timer expiration** (optimistic). When a new transaction begins, the lock manager sets an interrupting timer to a value somewhat greater than the time it should take for the transaction to complete. If a transaction gets into a deadlock, its timer will expire, at which point the system aborts that transaction, rolling back its changes and releasing its locks in the hope that the other transactions involved in the deadlock may be able to proceed. If not, another one will time out, releasing further locks. Timing out deadlocks is effective, though it has the usual defect: it is difficult to choose a suitable timer value that keeps things moving along but also accommodates normal delays and variable operation times. If the environment or system load changes, it may be necessary to readjust all such timer values, an activity that can be a real nuisance in a large system.
4. **Cycle detection** (optimistic). Maintain, in the lock manager, a wait-for graph that shows which transactions have acquired which locks and which transactions are waiting for which locks. Whenever another transaction tries to acquire a lock, finds it is already locked, and proposes to wait, the lock manager examines the graph to see if waiting would produce a cycle, and thus a deadlock. If it would, the lock manager selects some cycle member to be a victim, and unilaterally aborts that transaction so that the others may continue. The aborted transaction then retries in the hope that the other transactions have made enough progress to be out of the way and another deadlock will not occur.

When a system uses lock ordering, backing out, or cycle detection, it is common to also set a timer as a safety net because a hardware failure or a programming error such as an endless loop can create a progress-blocking situation that none of the deadlock detection methods can catch.

Since a deadlock detection algorithm can introduce an extra reason to abort a transaction, one can envision pathological situations where the algorithm aborts every attempt to perform some particular transaction, no matter how many times its invoker retries. Suppose, for example, that two threads named Alphonse and Gaston get into a deadlock trying to acquire locks for two objects named Apple and Banana: Alphonse acquires the lock for Apple, Gaston acquires the lock for Banana, Alphonse tries to acquire the lock for Banana and waits, then Gaston tries to acquire the lock for Apple and waits, creating the deadlock. Eventually, Alphonse times out and begins rolling back updates in preparation for releasing locks. Meanwhile, Gaston times out and does the same thing. Both restart, and they get into another deadlock, with their timers set to expire exactly as before, so they will probably repeat the sequence forever. Thus we still have no guarantee of progress. This is an emergent property called livelock, since formally no deadlock ever occurs and both threads are busy doing something that looks superficially useful.

One way to deal with livelock is to apply a randomized version of a technique familiar from Section 1.7.2: **exponential random backoff**. When a timer expiration leads to an abort, the lock manager clears the locks and then delays that thread for a random length of time, chosen from some starting interval, in the hope that the randomness will change the relative timing of the livelocked transactions enough that on the next try one will succeed and then the other can then proceed without interference. If the transaction again encounters interference, it tries again, but on each retry not only does the lock manager choose a new random delay, but it also increases the interval from which the delay is chosen by some multiplicative constant, typically 2. Since on each retry there is an increased probability of success, one can push this probability as close to unity as desired by continued retries, with the expectation that the interfering transactions will eventually get out of one another's way. A useful property of exponential random backoff is that if repeated retries continue to fail it is almost certainly an indication of some deeper problem—perhaps a programming mistake or a level of competition for shared variables that is intrinsically so high that the system should be redesigned.

The design of more elaborate algorithms or programming disciplines that guarantee progress is a project that has only modest potential payoff, and an *end-to-end argument* suggests that it may not be worth the effort. In practice, systems that would have frequent interference among transactions are not usually designed with a high degree of concurrency anyway. When interference is not frequent, simple techniques such as safety-net timers and exponential random backoff not only work well, but they usually must be provided anyway, to cope with any races or programming errors such as endless loops that may have crept into the system design or implementation. Thus a more complex progress-guaranteeing discipline is likely to be redundant, and only rarely will it get a chance to promote progress.

This page titled 3.6: Before-Or-After Atomicity II - Pragmatics is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

3.7: Atomicity Across Layers and Multiple Sites

There remain some important gaps in our exploration of atomicity. First, in a layered system, a transaction implemented in one layer may consist of a series of component actions of a lower layer that are themselves atomic. The question is how the commitment of the lower-layer transactions should relate to the commitment of the higher layer transaction. If the higher-layer transaction decides to abort, the question is what to do about lower-layer transactions that may have already committed. There are two possibilities:

- Reverse the effect of any committed lower-layer transactions with an `UNDO` action. This technique requires that the results of the lower-layer transactions be visible only within the higher-layer transaction.
- Somehow delay commitment of the lower-layer transactions and arrange that they actually commit at the same time that the higher-layer transaction commits.

Up to this point, we have assumed the first possibility. In this section we explore the second one.

Another gap is that, as described so far, our techniques to provide atomicity all involve the use of shared variables in memory or storage (for example, pointers to the latest version, outcome records, logs, and locks) and thus implicitly assume that the composite actions that make up a transaction all occur in close physical proximity. When the composing actions are physically separated, communication delay, communication reliability, and independent failure make atomicity both more important and harder to achieve.

We will edge up on both of these problems by first identifying a common subproblem: implementing nested transactions. We will then extend the solution to the nested transaction problem to create an agreement protocol, known as **two-phase commit**, that coordinates commitment of lower-layer transactions. We can then extend the two-phase commit protocol, using a specialized form of remote procedure call, to coordinate steps that must be carried out at different places. This sequence is another example of bootstrapping; the special case that we know how to handle is the single-site transaction and the more general problem is the multiple-site transaction. As an additional observation, we will discover that multiple-site transactions are quite similar to, but not quite the same as, the **dilemma of the two generals**.

3.7.1: Hierarchical Composition of Transactions

We got into the discussion of transactions by considering that complex interpreters are engineered in layers, and that each layer should implement atomic actions for its next-higher, client layer. Thus transactions are nested, each one typically consisting of multiple lower-layer transactions. This nesting requires that some additional thought be given to the mechanism of achieving atomicity.

Consider again a banking example. Suppose that the `TRANSFER` procedure of Section 3.2.5 is available for moving funds from one account to another, and it has been implemented as a transaction. Suppose now that we wish to create the two application procedures of Figure 3.7.1. The first procedure, `PAY_INTEREST` , invokes `TRANSFER` to move an appropriate amount of money from or to an internal account named bank, the direction and rate depending on whether the customer account balance is positive or negative. The second procedure, `MONTH_END_INTEREST` , fulfills the bank's intention to pay (or extract) interest every month on every customer account by iterating through the accounts and invoking `PAY_INTEREST` on each one.

```
procedure PAY_INTEREST (reference account)
    if account.balance > 0 then
        interest = account.balance * 0.05
        TRANSFER (bank, account, interest)
    else
        interest = account.balance * 0.15
        TRANSFER (account, bank, interest)

procedure MONTH_END_INTEREST:()
    for A ← each customer_account do
        PAY_INTEREST (A)
```

Figure 3.7.1: An example of two procedures, one of which calls the other, yet each should be individually atomic.

It would probably be inappropriate to have two invocations of `MONTH_END_INTEREST` running at the same time, but it is likely that at the same time that `MONTH_END_INTEREST` is running there are other banking activities in progress that are also invoking `TRANSFER` . It is also possible that the `for each` statement inside `MONTH_END_INTEREST` actually runs several instances of its iteration (and thus of `PAY_INTEREST`) concurrently. Thus we have a need for three layers of transactions. The lowest

layer is the TRANSFER procedure, in which debiting of one account and crediting of a second account must be atomic. At the next higher layer, the procedure PAY_INTEREST should be executed atomically, to ensure that some concurrent TRANSFER transaction doesn't change the balance of the account between the positive/negative test and the calculation of the interest amount. Finally, the procedure MONTH_END_INTEREST should be a transaction, to ensure that some concurrent TRANSFER transaction does not move money from an account A to an account B between the interest-payment processing of those two accounts, since such a transfer could cause the bank to pay interest twice on the same funds. Structurally, an invocation of the TRANSFER procedure is nested inside PAY_INTEREST , and one or more concurrent invocations of PAY_INTEREST are nested inside MONTH_END_INTEREST .

The reason that nesting is a potential problem comes from a consideration of the commit steps of the nested transactions. For example, the commit point of the TRANSFER transaction would seem to have to occur either before or after the commit point of the PAY_INTEREST transaction, depending on where in the programming of PAY_INTEREST we place its commit point. Yet either of these positions will cause trouble. If the TRANSFER commit occurs in the pre-commit phase of PAY_INTEREST , then if there is a system crash PAY_INTEREST will not be able to back out as though it hadn’t tried to operate—the values of the two accounts

that TRANSFER changed may have already been used by concurrent transactions to make payment decisions. But if the TRANSFER commit does not occur until the post-commit phase of PAY_INTEREST , there is a risk that the transfer itself can not be completed: for example, because one of the accounts is inaccessible. The conclusion is that somehow the commit point of the nested transaction should coincide with the commit point of the enclosing transaction. A slightly different coordination problem applies to MONTH_END_INTEREST : no instances of TRANSFER called by other transactions should occur while it runs (that is, it should run either before or after any concurrent TRANSFER transactions), but it must be able to do multiple instances of TRANSFER itself, each time it invokes PAY_INTEREST , and its own possibly concurrent transfer actions must be before-or-after actions, since they all involve the account named "bank."

Suppose for the moment that the system provides transactions with version histories. We can deal with nesting problems by extending the idea of an outcome record: we allow outcome records to be organized hierarchically. Whenever we create a nested transaction, we record in its outcome record both the initial state (PENDING) of the new transaction and the identifier of the enclosing transaction. The resulting hierarchical arrangement of outcome records then exactly reflects the nesting of the transactions. A top-layer outcome record would contain a flag to indicate that it is not nested inside any other transaction. When an outcome record contains the identifier of a higher-layer transaction, we refer to it as a dependent outcome record, and the record to which it refers is called its superior. The transactions, whether nested or enclosing, then go about their business, and depending on their success mark their own outcome records COMMITTED or ABORTED , as usual. However, when READ_CURRENT_VALUE (described in Section 3.5.2) examines the status of a version to see whether or not the transaction that created it is COMMITTED , it must additionally check to see if the outcome record contains a reference to a superior outcome record. If so, it must follow the reference and check the status of the superior. If that record says that it, too, is COMMITTED , it must continue following the chain upward, all the way to the highest-layer outcome record if necessary. The transaction in question is actually COMMITTED only if all the records in the chain are in the COMMITTED state. If any record in the chain is ABORTED , this transaction is actually ABORTED , despite the COMMITTED claim in its own outcome record. Finally, if neither of those situations holds, then there must be one or more records in the chain that are still PENDING . The outcome of this transaction remains PENDING until those records become COMMITTED or ABORTED . Thus the outcome of an apparently-COMMITTED dependent outcome record actually depends on the outcomes of all of its ancestors. We can describe this situation by saying that, until all its ancestors commit, this lower-layer transaction is sitting on a knife-edge, at the point of committing but still capable of aborting if necessary. For purposes of discussion we will identify this situation as a distinct virtual state of the outcome record and the transaction, by saying that the transaction is **tentatively committed**.

This hierarchical arrangement has several interesting programming consequences. If a nested transaction has any post-commit steps, those steps cannot proceed until all of the hierarchically higher transactions have committed. For example, if one of the nested transactions opens a cash drawer when it commits, the sending of the release message to the cash drawer must somehow be held up until the highest-layer transaction determines its outcome.

This output visibility consequence is only one example of many relating to the tentatively committed state. The nested transaction, having declared itself tentatively committed, has renounced the ability to abort—the decision is in someone else's hands. It must be able to run to completion *or* to abort, and it must be able to maintain the tentatively committed state indefinitely. Maintaining the ability to go either way can be awkward, since the transaction may be holding locks, keeping pages in memory or tapes mounted,

or reliably holding on to output messages. One consequence is that a designer cannot simply take any arbitrary transaction and blindly use it as a nested component of a larger transaction. At the least, the designer must review what is required for the nested transaction to maintain the tentatively committed state.

Another, more complex, consequence arises when one considers possible interactions among different transactions that are nested within the same higher-layer transaction. Consider our earlier example of TRANSFER transactions that are nested inside PAY_INTEREST , which in turn is nested inside MONTH_END_INTEREST . Suppose that the first time that MONTH_END_INTEREST invokes PAY_INTEREST , that invocation commits, thus moving into the tentatively committed state, pending the outcome of MONTH_END_INTEREST . Then MONTH_END_INTEREST invokes PAY_INTEREST on a second bank account. PAY_INTEREST needs to be able to read as input data the value of the bank's own interest account, which is a pending result of the previous, tentatively COMMITTED , invocation of PAY_INTEREST . The READ_CURRENT_VALUE algorithm, as implemented in Section 3.5.2, doesn't distinguish between reads arising within the same group of nested transactions and reads from some completely unrelated transaction. Figure 3.7.2 illustrates the situation. If the test in READ_CURRENT_VALUE for committed values is extended by simply following the ancestry of the outcome record controlling the latest version, it will undoubtedly force the second invocation of PAY_INTEREST to wait pending the final outcome of the first invocation of PAY_INTEREST . But since the outcome of that first invocation depends on the outcome of MONTH_END_INTEREST and the outcome of MONTH_END_INTEREST currently depends on the success of the second invocation of PAY_INTEREST , we have a built-in cycle of waits that at best can only time out and abort.

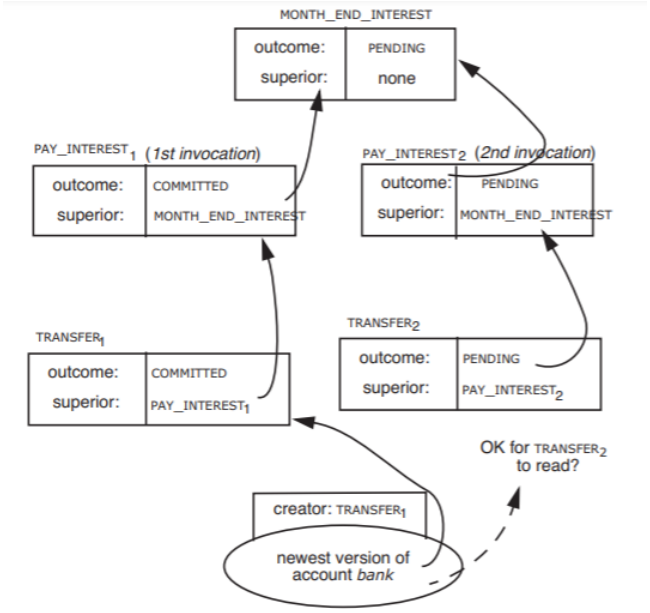


Figure 3.7.2: Transaction $TRANSFER_2$, nested in transaction $PAY_INTEREST_2$, which is nested in transaction MONTH_END_INTEREST , wants to read the current value of account *bank* . But *bank* was last written by transaction $TRANSFER_1$, which is nested in COMMITTED transaction $PAY_INTEREST_1$, which is nested in still- PENDING transaction MONTH_END_INTEREST . Thus this version of *bank* is actually PENDING , rather than COMMITTED as one might conclude by looking only at the outcome of $TRANSFER_1$. However, $TRANSFER_1$ and $TRANSFER_2$ share a common ancestor (namely, MONTH_END_INTEREST), and the chain of transactions leading from bank to that common ancestor is completely committed, so the read of *bank* can—and to avoid a deadlock, must—be allowed.

Since blocking the read would be a mistake, the question of when it might be OK to permit reading of data values created by tentatively COMMITTED transactions requires some further thought. The before-or-after atomicity requirement is that no update made by a tentatively COMMITTED transaction should be visible to any transaction that would survive if for some reason the tentatively COMMITTED transaction ultimately aborts. Within that constraint, updates of tentatively COMMITTED transactions can freely be passed around. We can achieve that goal in the following way: compare the outcome record ancestry of the transaction doing the read with the ancestry of the outcome record that controls the version to be read. If these ancestries do not merge (that is, there is no common ancestor) then the reader must wait for the version's ancestry to be completely committed. If they do merge and all the transactions in the ancestry of the data version that are below the point of the merge are tentatively committed, no wait is necessary. Thus, in Figure 3.7.2, MONTH_END_INTEREST might be running the two (or more)

invocations of PAY_INTEREST concurrently. Each invocation will call CREATE_NEW_VERSION as part of its plan to update the value of account "bank", thereby establishing a serial order of the invocations. When later invocations of PAY_INTEREST call READ_CURRENT_VALUE to read the value of account "bank", they will be forced to wait until all earlier invocations of PAY_INTEREST decide whether to commit or abort.

3.7.2: Two-Phase Commit

Since a higher-layer transaction can comprise several lower-layer transactions, we can describe the commitment of a hierarchical transaction as involving two distinct phases. In the first phase, known variously as the **preparation** or **voting** phase, the higher-layer transaction invokes some number of distinct lower-layer transactions, each of which either aborts or, by committing, becomes tentatively committed. The top-layer transaction evaluates the situation to establish that all (or enough) of the lower-layer transactions are tentatively committed that it can declare the higher-layer transaction a success.

Based on that evaluation, it will either COMMIT or ABORT the higher-layer transaction. Assuming it decides to commit, it enters the second, **commitment** phase, which in the simplest case consists of simply changing its own state from PENDING to COMMITTED or ABORTED . If it is the highest-layer transaction, at that instant all of the lower-layer tentatively committed transactions also become either COMMITTED or ABORTED . If it is itself nested in a still higher-layer transaction, it becomes tentatively committed and its component transactions continue in the tentatively committed state also. We are implementing here a coordination protocol known as **two-phase commit**. When we implement multiple-site atomicity in the next section, the distinction between the two phases will take on additional clarity.

If the system uses version histories for atomicity, the hierarchy of Figure 3.7.2 can be directly implemented by linking outcome records. If the system uses logs, a separate table of pending transactions can contain the hierarchy, and inquiries about the state of a transaction would involve examining this table.

The concept of nesting transactions hierarchically is useful in its own right, but our particular interest in nesting is that it is the first of two building blocks for multiple-site transactions. To develop the second building block, we next explore what makes multiple-site transactions different from single-site transactions.

3.7.3: Multiple-Site Atomicity: Distributed Two-Phase Commit

If a transaction requires executing component transactions at several sites that are separated by a best-effort network, obtaining atomicity is more difficult because any of the messages used to coordinate the transactions of the various sites can be lost, delayed, or duplicated. In Chapter 1 we learned how to design protocols such as a Remote Procedure Call (RPC) for performing an action at another site, with a persistent sender to ensure at-least-once execution and duplicate suppression to ensure at-most-once execution. Unfortunately, neither of these two assurances is exactly what is needed to ensure atomicity of a multiple-site transaction. However, by properly combining a two-phase commit protocol with persistent senders, duplicate suppression, and single-site transactions, we can create a correct multiple-site transaction. We assume that each site, on its own, is capable of implementing local transactions, using techniques such as version histories or logs and locks for all-or-nothing atomicity and before-or-after atomicity. Correctness of the multiple-site atomicity protocol will be achieved if all the sites commit or if all the sites abort; we will have failed if some sites commit their part of a multiple-site transaction while others abort their part of that same transaction.

Suppose the multiple-site transaction consists of a coordinator Alice requesting component transactions X, Y, and Z of worker sites Bob, Charles, and Dawn, respectively. The simple expedient of issuing three remote procedure calls certainly does not produce a transaction for Alice because Bob may do X while Charles may report that he cannot do Y. Conceptually, the coordinator would like to send three messages, to the three workers, like this one to Bob:

```
From: Alice
To: Bob
Re: my transaction 91

    if (Charles does Y and Dawn does Z) then do X, please.
```

and let the three workers handle the details. We need some clue how Bob could accomplish this strange request.

The clue comes from recognizing that the coordinator has created a higher-layer transaction and each of the workers is to perform a transaction that is nested in the higher-layer transaction. Thus, what we need is a distributed version of the two-phase commit protocol. The complication is that the coordinator and workers cannot reliably communicate. The problem thus reduces to constructing a reliable distributed version of the two-phase commit protocol. We can do that by applying persistent senders and duplicate suppression.

Phase one of the protocol starts with coordinator Alice creating a top-layer outcome record for the overall transaction. Then Alice begins persistently sending to Bob an RPC-like message:

```
From: Alice
To: Bob
Re: my transaction 271

    Please do X as part of my transaction.
```

Similar messages go from Alice to Charles and Dawn, also referring to transaction 271, and requesting that they do Y and Z, respectively. As with an ordinary remote procedure call, if Alice doesn't receive a response from one or more of the workers in a reasonable time she resends the message to the non-responding workers as many times as necessary to elicit a response.

A worker site, upon receiving a request of this form, checks for duplicates and then creates a transaction of its own, but it makes the transaction a *nested* one, with its superior being Alice's original transaction. It then goes about doing the pre-commit part of the requested action, reporting back to Alice that this much has gone well:

```
From: Bob
To: Alice
Re: your transaction 271

    My part X is ready to commit.
```

Alice, upon collecting a complete set of such responses then moves to the two-phase commit part of the transaction, by sending messages to each of Bob, Charles, and Dawn saying, e.g.:

```
Two-phase-commit message #1:

    From: Alice
    To: Bob
    Re: my transaction 271

        PREPARE to commit X.
```

Bob, upon receiving this message, commits—but only tentatively—or aborts. Having created durable tentative versions (or logged to journal storage its planned updates) and having recorded an outcome record saying that it is `PREPARED` either to commit or abort, Bob then persistently sends a response to Alice reporting his state:

```
Two-phase-commit message #2:

    From: Bob
    To: Alice
    Re: your transaction 271

        I am PREPARED to commit my part. Have you decided to commit yet? Regards.
```

or alternatively, a message reporting it has aborted. If Bob receives a duplicate request from Alice, his persistent sender sends back a duplicate of the `PREPARED` or `ABORTED` response.

At this point Bob, being in the `PREPARED` state, is out on a limb. Just as in a local hierarchical nesting, Bob must be able either to run to the end or to abort, to maintain that state of preparation indefinitely, and wait for someone else (Alice) to say which. In addition, the coordinator may independently crash or lose communication contact, increasing Bob's uncertainty. If the coordinator goes down, all of the workers must wait until it recovers; in this protocol, the coordinator is a single point of failure.

As coordinator, Alice collects the response messages from her several workers (perhaps re-requesting `PREPARED` responses several times from some worker sites). If all workers send `PREPARED` messages, phase one of the two-phase commit is complete. If any worker responds with an abort message, or doesn't respond at all, Alice has the usual choice of aborting the entire transaction

or perhaps trying a different worker site to carry out that component transaction. Phase two begins when Alice commits the entire transaction by marking her own outcome record `COMMITTED` .

Once the higher-layer outcome record is marked as `COMMITTED` or `ABORTED` , Alice sends a completion message back to each of Bob, Charles, and Dawn:

```
Two-phase-commit message #3
    From: Alice
    To: Bob
    Re: my transaction 271

        My transaction committed. Thanks for your help.
```

Each worker site, upon receiving such a message, changes its state from `PREPARED` to `COMMITTED` , performs any needed post-commit actions, and exits. Meanwhile, Alice can go about other business, with one important requirement for the future: she must remember, reliably and for an indefinite time, the outcome of this transaction. The reason is that one or more of her completion messages may have been lost. Any worker sites that are in the `PREPARED` state are awaiting the completion message to tell them which way to go. If a completion message does not arrive in a reasonable period of time, the persistent sender at the worker site will resend its `PREPARED` message. Whenever Alice receives a duplicate `PREPARED` message, she simply sends back the current state of the outcome record for the named transaction.

If a worker site that uses logs and locks crashes, the recovery procedure at that site has to take three extra steps. First, it must classify any `PREPARED` transaction as a tentative winner that it should restore to the `PREPARED` state. Second, if the worker is using locks for before-or-after atomicity, the recovery procedure must reacquire any locks the `PREPARED` transaction was holding at the time of the failure. Finally, the recovery procedure must restart the persistent sender, to learn the current status of the higher-layer transaction. If the worker site uses version histories, only the last step, restarting the persistent sender, is required.

Since the workers act as persistent senders of their `PREPARED` messages, Alice can be confident that every worker will eventually learn that her transaction committed. But since the persistent senders of the workers are independent, Alice has no way of ensuring that they will act simultaneously. Instead, Alice can only be certain of eventual completion of her transaction. This distinction between simultaneous action and eventual action is critically important, as will soon be seen.

If all goes well, two-phase commit of N worker sites will be accomplished in $3N$ messages, as shown in Figure 3.7.3: for each worker site a `PREPARE` message, a `PREPARED` message in response, and a `COMMIT` message. This $3N$ message protocol is complete and sufficient, although there are several variations one can propose.

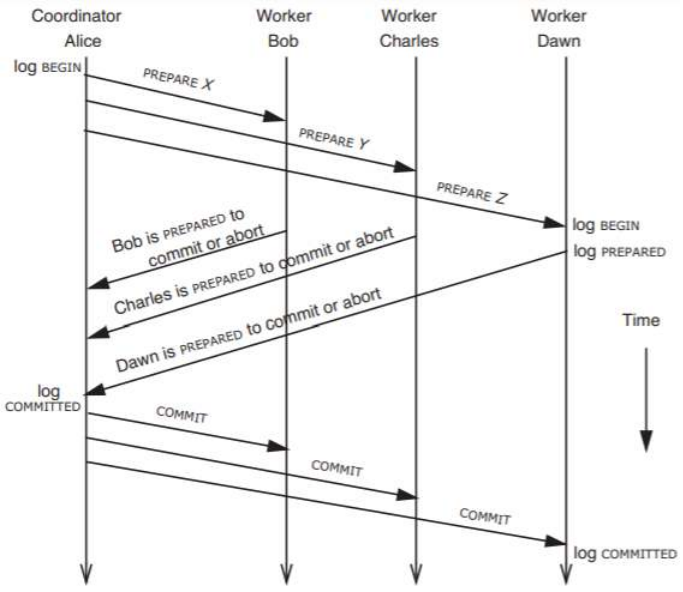


Figure 3.7.3: Timing diagram for distributed two-phase commit, using $3N$ messages. (The initial RPC request and response messages are not shown.) Each of the four participants maintains its own version history or recovery log. The diagram shows log entries made by the coordinator and by one of the workers.

An example of a simplifying variation is that the initial RPC request and response could also carry the PREPARE and PREPARED messages, respectively. However, once a worker sends a PREPARED message, it loses the ability to unilaterally abort, and it must remain on the knife edge awaiting instructions from the coordinator. To minimize this wait, it is usually preferable to delay the PREPARE / PREPARED message pair until the coordinator knows that the other workers seem to be in a position to do their parts.

Some versions of the distributed two-phase commit protocol have a fourth acknowledgment message from the worker sites to the coordinator. The intent is to collect a complete set of acknowledgment messages—the coordinator persistently sends completion messages until every site acknowledges. Once all acknowledgments are in, the coordinator can then safely discard its outcome record, since every worker site is known to have gotten the word.

A system that is concerned both about outcome record storage space and the cost of extra messages can use a further refinement, called **presumed commit**. Since one would expect that most transactions commit, we can use a slightly odd but very space-efficient representation for the value COMMITTED of an outcome record: non-existence. The coordinator answers any inquiry about a non-existent outcome record by sending a COMMITTED response. If the coordinator uses this representation, it commits by destroying the outcome record, so a fourth acknowledgment message from every worker is unnecessary. In return for this apparent magic reduction in both message count and space, we notice that outcome records for aborted transactions can not easily be discarded because if an inquiry arrives after discarding, the inquiry will receive the response COMMITTED . The coordinator can, however, persistently ask for acknowledgment of aborted transactions, and discard the outcome record after all these acknowledgments are in. This protocol that leads to discarding an outcome record is identical to the protocol described in Section 1.6.7 to close a stream and discard the record of that stream.

Distributed two-phase commit does not solve all multiple-site atomicity problems. For example, if the coordinator site (in this case, Alice) is aboard a ship that sinks after sending the PREPARE message but before sending the COMMIT or ABORT message, the worker sites are in left in the PREPARED state with no way to proceed. Even without that concern, Alice and her co-workers are standing uncomfortably close to a multiple-site atomicity problem that, at least in principle, cannot be solved. The only thing that rescues them is our observation that the several workers will do their parts eventually, not necessarily simultaneously. If she had required simultaneous action, Alice would have been in trouble.

The unsolvable problem is known as the **dilemma of the two generals**.

3.7.4: The Dilemma of the Two Generals

An important constraint on possible coordination protocols when communication is unreliable is captured in a vivid analogy, called the **dilemma of the two generals**.* Suppose that two small armies are encamped on two mountains outside a city. The city is defended sufficiently well that it can repulse and destroy either one of the two armies. Only if the two armies attack simultaneously can they take the city. Thus the two generals who command the armies desire to coordinate their attack.

The only method of communication between the two generals is to send runners from one camp to the other. But the defenders of the city have sentries posted in the valley separating the two mountains, so there is a chance that the runner, trying to cross the valley, will instead fall into enemy hands, and be unable to deliver the message.

Suppose that the first general sends this message:

```
From: Julius Caesar
To: Titus Labienus
Date: 11 January

    I propose to cross the Rubicon and attack at dawn tomorrow. OK?
```

expecting that the second general will respond either with:

```
From: Titus Labienus
To: Julius Caesar
Date: 11 January

    Yes, dawn on the 12th.
```

or, possibly:

```
From: Titus Labienus
To: Julius Caesar
Date: 11 January

    No. I am awaiting reinforcements from Gaul.
```

Suppose further that the first message does not make it through. In that case, the second general does not march because no request to do so arrives. In addition, the first general does not march because no response returns, and all is well (except for the lost runner).

Now, instead suppose the runner delivers the first message successfully and second general sends the reply "Yes," but that the reply is lost. The first general cannot distinguish this case from the earlier case, so that army will not march. The second general has agreed to march, but knowing that the first general won't march unless the "Yes" confirmation arrives, the second general will not march without being certain that the first general received the confirmation. This hesitation on the part of the second general suggests that the first general should send back an acknowledgment of receipt of the confirmation:

```
From: Julius Caesar
To: Titus Labienus
Date: 11 January

    The die is cast.
```

Unfortunately, that doesn't help, since the runner carrying this acknowledgment may be lost and the second general, not receiving the acknowledgment, will still not march. Thus the dilemma.

We can now leap directly to a conclusion: there is no protocol with a bounded number of messages that can convince both generals that it is safe to march. If there were such a protocol, the *last* message in any particular run of that protocol must be unnecessary to safe coordination because it might be lost, undetectably. Since the last message must be unnecessary, one could delete that message to produce another, shorter sequence of messages that must guarantee safe coordination. We can reapply the same reasoning repeatedly to the shorter message sequence to produce still shorter ones, and we conclude that if such a safe protocol exists it either generates message sequences of zero length or else of unbounded length. A zero-length protocol can't communicate anything, and an unbounded protocol is of no use to the generals, who must choose a particular time to march.

A practical general, presented with this dilemma by a mathematician in the field, would reassign the mathematician to a new job as a runner, and send a scout to check out the valley and report the probability that a successful transit can be accomplished within a specified time. Knowing that probability, the general would then send several (hopefully independent) runners, each carrying a copy of the message, choosing a number of runners large enough that the probability is negligible that all of them fail to deliver the message before the appointed time. (The loss of all the runners would be what Chapter 2 called an intolerable error.) Similarly, the second general sends many runners each carrying a copy of either the "Yes" or the "No" acknowledgment. This procedure provides a practical solution of the problem, so the dilemma is of no real consequence. Nevertheless, it is interesting to discover a problem that cannot, in principle, be solved with complete certainty.

We can state the theoretical conclusion more generally and succinctly: if messages may be lost, no bounded protocol can guarantee with complete certainty that both generals know that they will both march at the same time. The best that they can do is accept some non-zero probability of failure equal to the probability of non-delivery of their last message.

It is interesting to analyze just why we can't we use a distributed two-phase commit protocol to resolve the dilemma of the two generals. As suggested at the outset, it has to do with a subtle difference in *when* things may, or must, happen. The two generals require, in order to vanquish the defenses of the city, that they march at the *same* time.

The persistent senders of the distributed two-phase commit protocol ensure that if the coordinator decides to commit, all of the workers will eventually also commit, but there is no assurance that they will do so at the same time. If one of the communication links goes down for a day, when it comes back up the worker at the other end of that link will then receive the notice to commit, but this action may occur a day later than the actions of its colleagues. Thus the problem solved by distributed two-phase commit is slightly relaxed when compared with the dilemma of the two generals. That relaxation doesn't help the two generals, but the relaxation turns out to be just enough to allow us to devise a protocol that ensures correctness.

By a similar line of reasoning, there is no way to ensure with complete certainty that actions will be taken simultaneously at two sites that communicate only via a best-effort network. Distributed two-phase commit can thus safely open a cash drawer of an ATM in Tokyo, with confidence that a computer in Munich will eventually update the balance of that account. But if, for some reason, it

is necessary to open two cash drawers at different sites at the same time, the only solution is either the probabilistic approach or to somehow replace the best-effort network with a reliable one. The requirement for reliable communication is why real estate transactions and weddings (both of which are examples of two-phase commit protocols) usually occur with all of the parties in one room.

---

* The origin of this analogy has been lost, but it was apparently first described in print in 1977 by Jim N. Gray in his "Notes on Database Operating Systems", reprinted in *Operating Systems, Lecture Notes in Computer Science 60*, Springer Verlag, 1978. At about the same time, Danny Cohen described another analogy he called the dating protocol, which is congruent with the dilemma of the two generals.

This page titled 3.7: Atomicity Across Layers and Multiple Sites is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

3.8: A More Complete Model of Disk Failure (Advanced Topic)

Section 3.2 of this chapter developed a failure analysis model for a calendar management program in which a system crash may corrupt at most one disk sector—the one, if any, that was being written at the instant of the crash. That section also developed a masking strategy for that problem, creating all-or-nothing disk storage. To keep that development simple, the strategy ignored decay events. This section revisits that model, considering how to also mask decay events. The result will be all-or-nothing durable storage, meaning that it is both all-or-nothing in the event of a system crash and durable in the face of decay events.

3.8.1: Storage that is Both All-or-Nothing and Durable

In Chapter 2 we learned that to obtain durable storage we should write two or more replicas of each disk sector. In the current chapter we learned that to recover from a system crash while writing a disk sector we should never overwrite the previous version of that sector, we should write a new version in a different place. To obtain storage that is both durable and all-or-nothing we combine these two observations: make more than one replica, and don't overwrite the previous version. One easy way to do that would be to simply build the all-or-nothing storage layer of the current chapter on top of the durable storage layer of Chapter 2. That method would certainly work but it is a bit heavy-handed: with a replication count of just two, it would lead to allocating six disk sectors for each sector of real data. This is a case in which modularity has an excessive cost.

Recall that the parameter that Chapter 2 used to determine frequency of checking the integrity of disk storage was the expected time to decay, T_d. Suppose for the moment that the durability requirement can be achieved by maintaining only two copies. In that case, T_d must be much greater than the time required to write two copies of a sector on two disks. Put another way, a large T_d means that the short-term chance of a decay event is small enough that the designer may be able to safely neglect it. We can take advantage of this observation to devise a slightly risky but far more economical method of implementing storage that is both durable and all-or-nothing with just two replicas. The basic idea is that if we are confident that we have two good replicas of some piece of data for durability, it is safe (for all-or-nothing atomicity) to overwrite one of the two replicas; the second replica can be used as a backup to ensure all-or-nothing atomicity if the system should happen to crash while writing the first one. Once we are confident that the first replica has been correctly written with new data, we can safely overwrite the second one, to regain long-term durability. If the time to complete the two writes is short compared with T_d, the probability that a decay event interferes with this algorithm will be negligible. Figure 3.8.1 shows the algorithm and the two replicas of the data, here named `D0` and `D1` .

```
1  procedure ALL_OR_NOTHING_DURABLE_GET (reference data, atomic_sector)
2      ds ← CAREFUL_GET (data, atomic_sector.D0)
3      if ds = BAD then
4          ds ← CAREFUL_GET (data, atomic_sector.D1)
5      return ds

6  procedure ALL_OR_NOTHING_DURABLE_PUT (new_data, atomic_sector)
7      SALVAGE(atomic_sector)
8      ds ← CAREFUL_PUT (new_data, atomic_sector.D0)
9      ds ← CAREFUL_PUT (new_data, atomic_sector.D1)
10     return ds

11 procedure SALVAGE(atomic_sector)       //Run this program every T_d seconds.
12     ds0 ← CAREFUL_GET (data0, atomic_sector.D0)
13     ds1 ← CAREFUL_GET (data1, atomic_sector.D1)
14     if ds0 = BAD then
15         CAREFUL_PUT (data1, atomic_sector.D0)
16     else if ds1 = BAD then
17         CAREFUL_PUT (data0, atomic_sector.D1)
18     if data0 ≠ data1 then
19         CAREFUL_PUT (data0, atomic_sector.D1)
```



Figure 3.8.1: Data arrangement and algorithms to implement all-or-nothing durable storage on top of the careful storage layer of Figure 2.6.1.

An interesting point is that `ALL_OR_NOTHING_DURABLE_GET` does not bother to check the status returned upon reading `D1` —it just passes the status value along to its caller. The reason is that in the absence of decay, `CAREFUL_GET` has no expected errors when reading data that `CAREFUL_PUT` was allowed to finish writing. Thus the returned status would be `BAD` only in two cases:

1. CAREFUL_PUT of D1 was interrupted in mid-operation, or
2. D1 was subject to an unexpected decay.

The algorithm guarantees that the first case cannot happen. ALL_OR_NOTHING_DURABLE_PUT doesn't begin CAREFUL_PUT on data D1 until after the completion of its CAREFUL_PUT on data D0 . At most one of the two copies could be BAD because of a system crash during CAREFUL_PUT . Thus if the first copy (D0) is BAD , then we expect that the second one (D1) is OK .

The risk of the second case is real, but we have assumed its probability to be small: it arises only if there is a random decay of D1 in a time much shorter than T_d. In reading D1 we have an opportunity to *detect* that error through the status value, but we have no way to recover when both data copies are damaged, so this detectable error must be classified as untolerated. All we can do is pass a status report along to the application so that it knows that there was an untolerated error.

There is one currently unnecessary step hidden in the SALVAGE program: if D0 is BAD , nothing is gained by copying D1 onto D0 since ALL_OR_NOTHING_DURABLE_PUT , which called SALVAGE , will immediately overwrite D0 with new data. The step is included because it allows SALVAGE to be used in a refinement of the algorithm.

In the absence of decay events, this algorithm would be just as good as the all-or-nothing procedures of Figures 3.3.1 and 3.3.2, and it would perform somewhat better, since it involves only two copies. Assuming that errors are rare enough that recovery operations do not dominate performance, the usual cost of ALL_OR_NOTHING_DURABLE_GET is just one disk read, compared with three in the ALL_OR_NOTHING_GET algorithm. The cost of ALL_OR_NOTHING_DURABLE_PUT is two disk reads (in SALVAGE) and two disk writes, compared with three disk reads and three disk writes for the ALL_OR_NOTHING_PUT algorithm.

That analysis is based on a decay-free system. To deal with decay events, thus making the scheme both all-or-nothing *and* durable, the designer adopts two ideas from the discussion of durability in Chapter 2, the second of which eats up some of the better performance:

1. Place the two copies, D0 and D1 , in independent decay sets (for example, write them on two different disk drives, preferably from different vendors).
2. Have a clerk run the SALVAGE program on every atomic sector at least once every T_d seconds.

The clerk running the SALVAGE program performs $2N$ disk reads every T_d seconds to maintain N durable sectors. This extra expense is the price of durability against disk decay. The performance cost of the clerk depends on the choice of T_d, the value of N, and the priority of the clerk. Since the expected operational lifetime of a hard disk is usually several years, setting T_d to a few weeks should make the chance of untolerated failure from decay negligible, especially if there is also an operating practice to routinely replace disks well before they reach their expected operational lifetime. A modern hard disk with a capacity of one terabyte would have about $N = 10^9$ kilobyte-sized sectors. If it takes 10 milliseconds to read a sector, it would take about 2×10^7 seconds, or two days, for a clerk to read all of the contents of two one-terabyte hard disks. If the work of the clerk is scheduled to occur at night, or uses a priority system that runs the clerk when the system is otherwise not being used heavily, that reading can spread out over a few weeks and the performance impact can be minor.

A few paragraphs above, it was mentioned that there is the potential for a refinement: If we also run the SALVAGE program on every atomic sector immediately following every system crash, then it should not be necessary to do it at the beginning of every ALL_OR_NOTHING_DURABLE_PUT . That variation, which is more economical if crashes are infrequent and disks are not too large, is due to Butler Lampson and Howard Sturgis [Suggestions for Further Reading 3.3.3]. It raises one minor concern, because it depends on the rarity of coincidence of two failures: the spontaneous decay of one data replica at about the same time that CAREFUL_PUT crashes in the middle of rewriting the other replica of that same sector. If we are convinced that such a coincidence is rare, we can declare it to be an untolerated error, and we have a self-consistent and more economical algorithm. With this scheme, the cost of ALL_OR_NOTHING_DURABLE_PUT reduces to just two disk writes.

This page titled 3.8: A More Complete Model of Disk Failure (Advanced Topic) is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

3.9: Case Studies - Machine Language Atomicity

3.9.1: Complex Instruction Sets: The General Electric 600 Line

In the early days of mainframe computers, most manufacturers reveled in providing elaborate instruction sets, without paying much attention to questions of atomicity. The General Electric 600 line, which later evolved to be the Honeywell Information System, Inc., 68 series computer architecture, had a feature called "indirect and tally." One could specify this feature by setting to `ON` a one-bit flag (the "tally" flag) stored in an unused high-order bit of any indirect address. The instruction

```
Load register A from Y indirect.
```

was interpreted to mean that the low-order bits of the cell with address Y contain another address, called an indirect address, and that that indirect address should be used to retrieve the operand to be loaded into register A. In addition, if the tally flag in cell Y is `ON`, the processor is to increment the indirect address in Y by one and store the result back in Y. The idea is that the next time Y is used as an indirect address it will point to a different operand—the one in the next sequential address in memory. Thus the indirect and tally feature could be used to sweep through a table. The feature seemed useful to the designers, but in action it was useful only occasionally because most applications were written in higher-level languages and compiler writers found it hard to exploit. On the other hand, the feature gave no end of trouble when virtual memory was retrofitted to the product line.

Suppose that virtual memory is in use, and that the indirect word is located in a page that is in primary memory, but the actual operand is in another page that has been removed to secondary memory. When the above instruction is executed, the processor will retrieve the indirect address in Y, increment it, and store the new value back in Y. Then it will attempt to retrieve the actual operand, at which time it discovers that it is not in primary memory, so it signals a missing-page exception. Since it has already modified the contents of Y (and by now Y may have been read by another processor or even removed from memory by the missing-page exception handler running on another processor), it is not feasible to back out and act as if this instruction had never executed. The designer of the exception handler would like to be able to give the processor to another thread by calling a function such as `AWAIT` while waiting for the missing page to arrive. Indeed, processor reassignment may be the only way to assign a processor to retrieve the missing page. However, to reassign the processor it is necessary to save its current execution state. Unfortunately, its execution state is "halfway through the instruction last addressed by the program counter." Saving this state and later restarting the processor in this state is challenging. The indirect and tally feature was just one of several sources of atomicity problems that cropped up when virtual memory was added to this processor.

The virtual memory designers desperately wanted to be able to run other threads on the interrupted processor. To solve this problem, they extended the definition of the current program state to contain not just the next-instruction counter and the program-visible registers, but also the complete internal state description of the processor—a 216-bit snapshot in the middle of the instruction. By later restoring the processor state to contain the previously saved values of the next-instruction counter, the program-visible registers, and the 216-bit internal state snapshot, the processor could exactly continue from the point at which the missing-page alert occurred. This technique worked but it had two awkward side effects:

1. When a program (or programmer) inquires about the current state of an interrupted processor, the state description includes things not in the programmer's interface; and
2. The system must be careful when restarting an interrupted program to make certain that the stored micro-state description is a valid one. If someone has altered the state description the processor could try to continue from a state it could never have gotten into by itself, which could lead to unplanned behavior, including failures of its memory protection features.

3.9.2: More Elaborate Instruction Sets: The IBM System/370

When IBM developed the System/370 by adding virtual memory to its System/360 architecture, certain System/360 multi-operand character-editing instructions caused atomicity problems. For example, the `TRANSLATE` instruction contains three arguments, two of which are addresses in memory (call them *`string`* and *`table`*) and the third of which, *`length`*, is an 8-bit count that the instruction interprets as the length of *`string`*. `TRANSLATE` takes one byte at a time from *`string`*, uses that byte as an offset in *`table`*, retrieves the byte at the offset, and replaces the byte in *`string`* with the byte it found in *`table`*. The designers had in mind that `TRANSLATE` could be used to convert a character string from one character set to another.

The problem with adding virtual memory is that both *`string`* and *`table`* may be as long as 65,536 bytes, so either or both of those operands may cross not just one, but several page boundaries. Suppose just the first page of *`string`* is in physical memory.

The TRANSLATE instruction works its way through the bytes at the beginning of *string* . When it comes to the end of that first page, it encounters a missing-page exception. At this point, the instruction cannot run to completion because data it requires is missing. It also cannot back out and act as if it never started, because it has modified data in memory by overwriting it. After the virtual memory manager retrieves the missing page, the problem is how to restart the half-completed instruction. If it restarts from the beginning, it will try to convert the already-converted characters, which would be a mistake. For correct operation, the instruction needs to continue from where it left off.

Rather than tampering with the program state definition, the IBM processor designers chose a **dry run** strategy in which the TRANSLATE instruction is executed using a hidden copy of the program-visible registers, making no changes in memory. If one of the operands causes a missing-page exception, the processor can act as if it never tried the instruction, since there is no program-visible evidence that it did. The stored program state shows only that the TRANSLATE instruction is about to be executed. After the processor retrieves the missing page, it restarts the interrupted thread by trying the TRANSLATE instruction from the beginning again, another dry run. If there are several missing pages, several dry runs may occur, each getting one more page into primary memory. When a dry run finally succeeds in completing, the processor runs the instruction once more, this time for real, using the program-visible registers and allowing memory to be updated. Since the System/370 (at the time this modification was made) was a single-processor architecture, there was no possibility that another processor might snatch a page away after the dry run but before the real execution of the instruction. This solution had the side effect of making life more difficult for a later designer with the task of adding multiple processors.

3.9.3: The Apollo Desktop Computer and the Motorola 68000 Microprocessor

When Apollo Computer designed a desktop computer using the Motorola 68000 microprocessor, the designers, who wanted to add a virtual memory feature, discovered that the microprocessor instruction set interface was not atomic. Worse, because it was constructed entirely on a single chip it could not be modified to do a dry run (as in the IBM 370) or to make it store the internal microprogram state (as in the General Electric 600 line). So the Apollo designers used a different strategy: they installed not one, but two Motorola 68000 processors. When the first one encounters a missing-page exception, it simply stops in its tracks, and waits for the operand to appear. The second Motorola 68000 (whose program is carefully planned to reside entirely in primary memory) fetches the missing page and then restarts the first processor.

Other designers working with the Motorola 68000 used a different, somewhat risky trick: modify all compilers and assemblers to generate only instructions that happen to be atomic. Motorola later produced a version of the 68000 in which all internal state registers of the microprocessor could be saved, the same method used in adding virtual memory to the General Electric 600 line.

This page titled 3.9: Case Studies - Machine Language Atomicity is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

3.10: Exercises

Exercise 3.10.1

1995–3–2a…c

Locking up humanities: The registrar's office is upgrading its scheduling program for limited-enrollment humanities subjects. The plan is to make it multithreaded, but there is concern that having multiple threads trying to update the database at the same time could cause trouble. The program originally had just two operations:

```
status ← REGISTER (subject_name)
DROP (subject_name)
```

where `subject_name` was a string such as "21W471". The `REGISTER` procedure checked to see if there is any space left in the subject, and if there was, it incremented the class size by one and returned the status value `ZERO`. If there was no space, it did not change the class size; instead it returned the status value −1. (This is a primitive registration system—it just keeps counts!)

As part of the upgrade, `subject_name` has been changed to a two-component structure:

```
structure subject
    string subject_name
    lock slock
```

and the registrar is now wondering where to apply the locking primitives,

```
ACQUIRE (subject.slock)
RELEASE (subject.slock)
```

Here is a typical application program, which registers the caller for two humanities subjects, `hx` and `hy`:

```
procedure REGISTER_TWO (hx, hy)
    status ← REGISTER (hx)
    if status = 0 then
        status ← REGISTER (hy)
        if status = -1 then
            DROP (hx)
    return status;
```

a) The goal is that the entire procedure `REGISTER_TWO` should have the before-or-after property. Add calls for `ACQUIRE` and `RELEASE` to the `REGISTER_TWO` procedure that obey the **simple locking protocol**.

b) Add calls to `ACQUIRE` and `RELEASE` that obey the **two-phase locking protocol**, while postponing every `ACQUIRE` as late as possible and performing every `RELEASE` as early as possible.

Louis Reasoner has come up with a suggestion that he thinks could simplify the job of programmers creating application programs such as `REGISTER_TWO`. His idea is to revise the two programs `REGISTER` and `DROP` by having them do the `ACQUIRE` and `RELEASE` internally. That is, the procedure

```
procedure REGISTER (subject)
    { current code }
    return status
```

would become instead:

```
procedure REGISTER (subject)
    ACQUIRE (subject.slock)
    { current code }
    RELEASE (subject.slock)
    return status
```

c) As usual, Louis has misunderstood some aspect of the problem. Give a brief explanation of what is wrong with this idea.

Exercise 3.10.2

2006-0-1

Ben and Alyssa are debating a fine point regarding version history transaction disciplines and would appreciate your help. Ben says that under the mark point transaction discipline, every transaction should call `MARK_POINT_ANNOUNCE` as soon as possible, or else the discipline won't work. Alyssa claims that everything will come out correct even if no transaction calls `MARK_POINT_ANNOUNCE`. Who is right?

Exercise 3.10.3

Ben and Alyssa are debating another fine point about the way that the version history transaction discipline bootstraps. The version of `NEW_OUTCOME_RECORD` given in the text uses `TICKET` as well as `ACQUIRE` and `RELEASE`. Alyssa says this is overkill—it should be possible to correctly coordinate `NEW_OUTCOME_RECORD` using just `ACQUIRE` and `RELEASE`. Modify the pseudocode of Figure 3.5.6 to create a version of `NEW_OUTCOME_RECORD` that doesn't need the ticket primitive.

Exercise 3.10.4

1997–0–02

You have been hired by Many-MIPS corporation to help design a new 32-register RISC processor that is to have six-way multiple instruction issue. Your job is to coordinate the interaction among the six arithmetic-logic units (ALUs) that will be running concurrently. Recalling the discussion of coordination, you realize that the first thing you must do is decide what constitutes "correct" coordination for a multiple-instruction-issue system.

Correct coordination for concurrent operations on a database was said to be: *No matter in what order things are actually calculated, the final result is always guaranteed to be one that could have been obtained by some sequential ordering of the concurrent operations*.

You have two goals: (1) maximum performance, and (2) not surprising a programmer who wrote a program expecting it to be executed on a single-instruction-issue machine.

Identify the best coordination correctness criterion for your problem.

A. Multiple instruction issue must be restricted to sequences of instructions that have non-overlapping register sets.

B. No matter in what order things are actually calculated, the final result is always guaranteed to be one that could have been obtained by some sequential ordering of the instructions that were issued in parallel.

C. No matter in what order things are actually calculated, the final result is always guaranteed to be the one that would have been obtained by the original ordering of the instructions that were issued in parallel.

D. The final result must be obtained by carrying out the operations in the order specified by the original program.

E. No matter in what order things are actually calculated, the final result is always guaranteed to be one that could have been obtained by some set of instructions carried out sequentially.

F. The six ALUs do not require any coordination.

Exercise 3.10.5

1982–3–3a,b

In 1968, IBM introduced the Information Management System (IMS) and it soon became one of the most widely used database management systems in the world. In fact, IMS is still in use today. At the time of introduction IMS used a before-or-after atomicity protocol consisting of the following two rules:

- A transaction may read only data that has been written by previously committed transactions.
- A transaction must acquire a lock for every data item that it will write.

Consider the following two transactions, which, for the interleaving shown, both adhere to the protocol:

1	BEGIN (*t1*)	BEGIN (*t2*)
2	ACQUIRE (y.lock)	
3	*temp1* ← *x*	
4		ACQUIRE (*x.lock*)
5		*temp2* ← *y*
6		*x* ← *temp2*
7	*y* ← *temp1*	
8	COMMIT (*t1*)	
9		COMMIT (*t2*)

Previously committed transactions had set $x \leftarrow 3$ and $y \leftarrow 4$.

a) After both transactions complete, what are the values of x and y ? In what sense is this answer wrong?

b) In the mid-1970s, this flaw was noticed, and the before-or-after atomicity protocol was replaced with a better one, despite a lack of complaints from customers. Explain why customers may not have complained about the flaw.

Exercise 3.10.6

1994–3–3

A system that attempts to make actions all-or-nothing writes the following type of records to a log maintained on non-volatile storage:

< STARTED *i* >	action *i* starts
< *i*, *x*, *old*, *new* >	action *i* writes the value *new* over the variable *old* for the variable *x*
< COMMITTED *i* >	action *i* commits
< ABORTED *i* >	action *i* aborts
< CHECKPOINT *i*, *j* >	at this checkpoint, actions *i*, *j*, ... are pending

Actions start in numerical order. A crash occurs, and the recovery procedure finds the following log records starting with the last checkpoint:

```
< CHECKPOINT 17, 51, 52 >
< STARTED 53 >
< STARTED 54 >
< 53, y, 5, 6 >
< 53, x, 6, 4 >
< COMMITTED 53 >
< 54, y, 6, 4 >
< STARTED 55 >
< 55, z, 3, 4 >
< ABORTED 17 >
< 51, q, 1, 9 >
< STARTED 56 >
< 55, y, 4, 3 >
< COMMITTED 54 >
```

```
< 55, y, 3, 7 >
< COMMITTED 51 >
< STARTED 57 >
< 56, x, 9, 2 >
< 56, w, 0, 1 >
< COMMITTED 56 >
< 57, u, 2, 1 >
*************** crash happened here ***************
```

a) Assume that the system is using a rollback recovery procedure. How much farther back in the log should the recovery procedure scan?

b) Assume that the system is using a roll-forward recovery procedure. How much farther back in the log should the recovery procedure scan?

c) Which operations mentioned in this part of the log are winners and which are losers?

d) What are the values of x and y immediately after the recovery procedure finishes? Why?

Exercise 3.10.7

1994–3–4

The log of exercise 3.10.7 contains (perhaps ambiguous) evidence that someone didn't follow coordination rules. What is that evidence?

Exercise 3.10.8

1994–3–5

Roll-forward recovery requires writing the commit (or abort) record to the log *before* doing any installs to cell storage. Identify the best reason for this requirement.

A. So that the recovery manager will know what to undo.

B. So that the recovery manager will know what to redo.

C. Because the log is less likely to fail than the cell storage.

D. To minimize the number of disk seeks required

Exercise 3.10.9

1997–3–03

Two-phase locking within transactions ensures that

A. No deadlocks will occur.

B. Results will correspond to some serial execution of the transactions.

C. Resources will be locked for the minimum possible interval.

D. Neither gas nor liquid will escape.

E. Transactions will succeed even if one lock attempt fails.

Exercise 3.10.10

1994–3–7

Pat, Diane, and Quincy are having trouble using email to schedule meetings. Pat suggests that they take inspiration from the two-phase commit protocol.

a) Which of the following protocols most closely resembles two-phase commit?

1. a. Pat requests everyone's schedule openings.
   b. Everyone replies with a list but does not guarantee to hold all the times available.
   c. Pat inspects the lists and looks for an open time.
      If there is a time,
        Pat chooses a meeting time and sends it to everyone.
      Otherwise
        Pat sends a message canceling the meeting.

2. a-c, as in Protocol 1.
   d. Everyone, if they received the second message,
        acknowledge receipt.
      Otherwise
        send a message to Pat asking what happened.

3. a-c, as in Protocol 1.
   d. Everyone, if their calendar is still open at the chosen time
        Send Pat an acknowledgment.
      Otherwise
        Send Pat apologies.
   e. Pat collects the acknowledgments. If all are positive
        Send a message to everyone saying the meeting is ON.
      Otherwise
        Send a message to everyone saying the meeting is OFF.
   f. Everyone, if they received the ON/OFF message,
        acknowledge receipt.
      Otherwise
        send a message to Pat asking what happened.

4. a-f, as in Protocol 3.
   g. Pat sends a message telling everyone that everyone has confirmed.
   h. Everyone acknowledges the confirmation.

b) For the protocol you selected, which step commits the meeting time?

Exercise 3.10.11

Alyssa P. Hacker needs a transaction processing system for updating information about her collection of 97 cockroaches.*

a) In her first design, Alyssa stores the database on disk. When a transaction commits, it simply goes to the disk and writes its changes in place over the old data. What are the major problems with Alyssa's system?

b) In Alyssa's second design, the only structure she keeps on disk is a log, with a reference copy of all data in volatile RAM. The log records every change made to the database, along with the transaction which the change was a part of. Commit records, also stored in the log, indicate when a transaction commits. When the system crashes and recovers, it replays the log, redoing each committed transaction, to reconstruct the reference copy in RAM. What are the disadvantages of Alyssa's second design?

To speed things up, Alyssa makes an occasional checkpoint of her database. To checkpoint, Alyssa just writes the entire state of the database into the log. When the system crashes, she starts from the last checkpointed state, and then redoes or undoes some transactions to restore her database. Now consider the five transactions in the illustration:

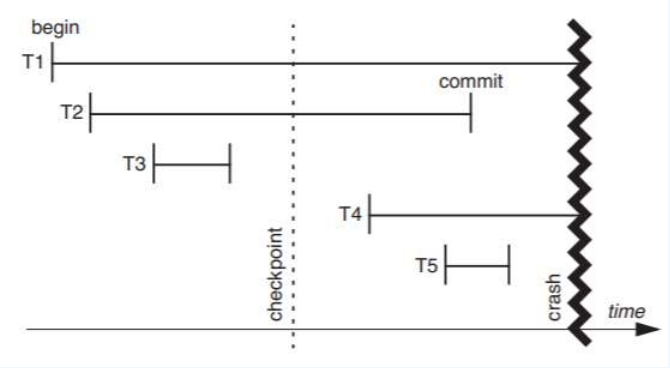


Figure 3.10.1: Transactions T2, T3, and T5 committed before the crash, but T1 and T4 were still pending.

When the system recovers, after the checkpointed state is loaded, some transactions will need to be undone or redone using the log. For each transaction,

c) When the system recovers, after the checkpointed state is loaded, some transactions will need to be undone or redone using the log. For each transaction, mark off in the table whether that transaction needs to be undone, redone, or neither.

	Undone	Redone	Neither
T1			
T2			
T3			
T4			
T5			

d) Now, assume that transactions T2 and T3 were actually *nested* transactions: T2 was nested in T1, and T3 was nested in T2. Again, fill in the table

	Undone	Redone	Neither
T1			
T2			
T3			
T4			
T5			

*Credit for developing exercise 3.10.11goes to Eddie Kohler.

Exercise 3.10.12

2008–3–11

Alice is acting as the coordinator for Bob and Charles in a two-phase commit protocol. Here is a log of the messages that pass among them:

1	Alice ⇒ Bob:	please do X
2	Alice ⇒ Charles:	please do Y
3	Bob ⇒ Alice:	done with X

4	Charles ⇒ Alice:	done with Y
5	Alice ⇒ Bob:	`PREPARE` to commit or abort
6	Alice ⇒ Charles:	`PREPARE` to commit or abort
7	Bob ⇒ Alice:	`PREPARED`
8	Charles ⇒ Alice:	`PREPARED`
9	Alice ⇒ Bob:	`COMMIT`
10	Alice ⇒ Charles:	`COMMIT`

At which points in this sequence is it OK for Bob to abort his part of the transaction?

A. After Bob receives message 1 but before he sends message 3.

B. After Bob sends message 3 but before he receives message 5.

C. After Bob receives message 5 but before he sends message 7.

D. After Bob sends message 7 but before he receives message 9.

E. After Bob receives message 9.

This page titled 3.10: Exercises is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

CHAPTER OVERVIEW

4: Consistency

4.1: Overview
4.2: Constraints and Interface Consistency
4.3: Cache Coherence
4.4: Durable Storage Revisited - Geographically Separated Replicas
4.5: Reconciliation
4.6: Perspectives
4.7: Exercises

This page titled 4: Consistency is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

4.1: Overview

The previous chapter developed **all-or-nothing atomicity** and **before-or-after atomicity**, two properties that define a transaction. This chapter introduces or revisits several applications that can make use of transactions. Section 4.2 introduces constraints and discusses how transactions can be used to maintain invariants and implement memory models that provide interface consistency. Sections 4.3 and 4.4 develop techniques used in two different application areas, **caching** and **geographically distributed replication**, to achieve higher performance and greater durability, respectively. Section 4.5 discusses **reconciliation**, which is a way of restoring the constraint that replicas be identical if their contents should drift apart. Finally, Section 4.6 considers some perspectives relating to Chapters 3 and 4.

This page titled 4.1: Overview is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

4.2: Constraints and Interface Consistency

One common use for transactions is to maintain **constraints**. A constraint is an application-defined requirement that every update to a collection of data preserve some specified invariant. Different applications can have quite different constraints. Here are some typical constraints that a designer might encounter:

- Table management: The variable that tells the number of entries should equal the number of entries actually in the table.
- Double-linked list management: The forward pointer in a list cell, `A` , should refer a list cell whose back pointer refers to `A` .
- Disk storage management: Every disk sector should be assigned either to the free list or to exactly one file.
- Display management: The pixels on the screen should match the description in the display list.
- Replica management: A majority (or perhaps all) of the replicas of the data should be identical.
- Banking: The sum of the balances of all credit accounts should equal the sum of the balances of all debit accounts.
- Process control: At least one of the valves on the boiler should always be open.

As was seen in Chapter 9, maintaining a constraint over data within a single file can be relatively straightforward: for example, by creating a shadow copy. Maintaining constraints across data that is stored in several files is harder, and that is one of the primary uses of transactions. Finally, two-phase commit allows maintaining a constraint that involves geographically separated files despite the hazards of communication.

A constraint usually involves more than one variable data item, in which case an update action by nature must be composite—it requires several steps. In the midst of those steps, the data will temporarily be inconsistent. In other words, there will be times when the data violates the invariant. During those times, there is a question about what to do if someone—another thread or another client—asks to read the data. This question is one of interface, rather than of internal operation. Different designers have developed several data consistency models to deal with this inevitable temporary inconsistency. In this chapter we consider two of those models: **strict consistency** and **eventual consistency**.

The first model, **strict consistency**, hides the constraint violation behind modular boundaries. Strict consistency means that actions outside the transaction performing the update will never see data that is inconsistent with the invariant. Since strict consistency is an interface concept, it depends on actions honoring abstractions: for example, by using only the intended reading and writing operations. Thus, for a cache, read/write coherence is a strict consistency specification: "The result of a `READ` of a named object is always the value that was provided by the most recent `WRITE` to that object." This specification does not demand that the replica in the cache always be identical to the replica in the backing store, it requires only that the cache deliver data at its interface that meets the specification.

Applications can maintain strict consistency by using transactions. If an action is all-or-nothing, the application can maintain the outward appearance of consistency despite failures, and if an action is before-or-after, the application can maintain the outward appearance of consistency despite the existence of other actions concurrently reading or updating the same data. Designers generally strive for strict consistency in any situation where inconsistent results can cause confusion, such as in a multiprocessor system, and in situations where mistakes can have serious negative consequences, such as in banking and safety-critical systems. Section 3.2.6 mentioned two other consistency models, sequential consistency and external time consistency. Both are examples of strict consistency.

The second, more lightweight, way of dealing with temporary inconsistency is called **eventual consistency**. Eventual consistency means that after a data update the constraint may not hold until some unspecified time in the future. An observer may, using the standard interfaces, discover that the invariant is violated, and different observers may even see different results. But the system is designed so that once updates stop occurring, it will make a best effort drive toward the invariant.

Eventual consistency is employed in situations where performance or availability is a high priority and temporary inconsistency is tolerable and can be easily ignored. For example, suppose a Web browser is to display a page from a distant service. The page has both a few paragraphs of text and several associated images. The browser obtains the text immediately, but it will take some time to download the images. The invariant is that the appearance on the screen should match the Web page specification. If the browser renders the text paragraphs first and fills in the images as they arrive, the human reader finds that behavior not only acceptable, but perhaps preferable to staring at the previous screen until the new one is completely ready. When a person can say, "Oh, I see what is happening," eventual consistency is usually acceptable, and in cases such as the Web browser it can even improve human engineering. For a second example, if a librarian catalogs a new book and places it on the shelf but the public version of the library

catalog doesn't include the new book until the next day, there is an observable inconsistency, but most library patrons would find it tolerable and not particularly surprising.

Eventual consistency is sometimes used in replica management because it allows for relatively loose coupling among the replicas, thus taking advantage of independent failure. In some applications, continuous service is a higher priority than always-consistent answers. If a replica server crashes in the middle of an update, the other replicas may be able to continue to provide service, even though some may have been updated and some may have not. In contrast, a strict consistency algorithm may have to refuse to provide service until a crashed replica site recovers, rather than taking a risk of exposing an inconsistency.

The remaining sections of this chapter explore several examples of strict and eventual consistency in action. A cache can be designed to provide either strict or eventual consistency; Section 4.3 provides the details. The Internet Domain Name System, discussed in Section 4.3.2, relies on eventual consistency in updating its caches, with the result that it can on occasion give inconsistent answers. Similarly, for the geographically replicated durable storage of Section 4.4 a designer can choose either a strict or an eventual consistency model. When replicas are maintained on devices that are only occasionally connected, eventual consistency may be the only choice, in which case reconciliation, the topic of Section 4.5, drives occasionally connected replicas toward eventual consistency. The reader should be aware that these examples do not provide a comprehensive overview of consistency; instead, they are intended primarily to create awareness of the issues involved by illustrating a few of the many possible designs.

This page titled 4.2: Constraints and Interface Consistency is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

4.3: Cache Coherence

4.3.1: Coherence, Replication, and Consistency in a Cache

A cache is an example of a multilevel memory system; it can also be thought of as a replication system whose primary goal is performance, rather than reliability. An invariant for a cache is that the replica of every data item in the primary store (that is, the cache) should be identical to the corresponding replica in the secondary memory. Since the primary and secondary stores usually have different latencies, when an action updates a data value, the replica in the primary store will temporarily be inconsistent with the one in the secondary memory. How well the multilevel memory system hides that inconsistency is the question.

A cache can be designed to provide either strict or eventual consistency. Since a cache, together with its backing store, is a memory system, a typical interface specification is that it provide read/write coherence for the entire name space of the cache.

Read/Write Coherence:

The result of a read of a named object is always the value of the most recent write to that object.

Read/write coherence is thus a specification that the cache provide strict consistency.

A write-through cache provides strict consistency for its clients in a straightforward way: it does not acknowledge that a write is complete until it finishes updating both the primary and secondary memory replicas. Unfortunately, the delay involved in waiting for the write-through to finish can be a performance bottleneck, so write-through caches are not popular.

A non-write-through cache acknowledges that a write is complete as soon as the cache manager updates the primary replica, in the cache. The thread that performed the write can go about its business expecting that the cache manager will eventually update the secondary memory replica and the invariant will once again hold. Meanwhile, if that same thread reads the same data object by sending a `READ` request to the cache, it will receive the updated value from the cache, even if the cache manager has not yet restored the invariant. Thus, because the cache manager masks the inconsistency, a non-write-through cache can still provide strict consistency.

On the other hand, if there is more than one cache, or other threads can read directly from the secondary storage device, the designer must take additional measures to ensure that other threads cannot discover the violated constraint. If a concurrent thread reads a modified data object via the same cache, the cache will deliver the modified version, and thus maintain strict consistency. But if a concurrent thread reads the modified data object directly from secondary memory, the result will depend on whether or not the cache manager has done the secondary memory update. If the second thread has its own cache, even a write-through design may not maintain consistency because updating the secondary memory does not affect a potential replica hiding in the second thread's cache. Nevertheless, all is not lost. There are at least three ways to regain consistency, two of which provide strict consistency, when there are multiple caches.

4.3.2: Eventual Consistency with Timer Expiration

The Internet Domain Name System provides an example of an eventual consistency cache that does *not* meet the read/write coherence specification. When a client calls on a DNS server to do a recursive name lookup, if the DNS server is successful in resolving the name it caches a copy of the answer as well as any intermediate answers that it received. Suppose that a client asks some local name server to resolve the name `ginger.pedantic.edu` . In the course of doing so, the local name server might accumulate the following name records in its cache:

`names.edu`	`198.41.0.4`	`name server for .edu`
`ns.pedantic.edu`	`128.32.25.19`	`name server for .pedantic.edu`
`ginger.pedantic.edu`	`128.32.247.24`	`target host name`

If the client then asks for `thyme.pedantic.edu` the local name server will be able to use the cached record for `ns.pedantic.edu` to directly ask that name server, without having to go back up to the root to find `names.edu` and thence

to names.edu to find ns.pedantic.edu .

Now, suppose that a network manager at Pedantic University changes the Internet address of ginger.pedantic.edu to 128.32.201.15 . At some point the manager updates the authoritative record stored in the name server ns.pedantic.edu . The problem is that local DNS caches anywhere in the Internet may still contain the old record of the address of ginger.pedantic.edu. DNS deals with this inconsistency by limiting the lifetime of a cached name record. Every name server record comes with an expiration time, known as the **time-to-live** (TTL), that can range from seconds to months. A typical time-to-live is one hour; it is measured from the moment that the local name server receives the record. So, until the expiration time, the local cache will be inconsistent with the authoritative version at Pedantic University. The system will eventually reconcile this inconsistency. When the time-to-live of that record expires, the local name server will handle any further requests for the name ginger.pedantic.edu by asking ns.pedantic.edu for a new name record. That new name record will contain the new, updated address, so this system provides eventual consistency.

There are two different actions that the network manager at Pedantic University might take to make sure that the inconsistency is not an inconvenience. First, the network manager may temporarily reconfigure the network layer of ginger.pedantic.edu to advertise both the old and the new Internet addresses, and then modify the authoritative DNS record to show the new address. After an hour has passed, all cached DNS records of the old address will have expired, and ginger.pedantic.edu can be reconfigured again, this time to stop advertising the old address. Alternatively, the network manager may have realized this change is coming, so a few hours in advance he or she modifies just the time-to-live of the authoritative DNS record, say to five minutes, without changing the Internet address. After an hour passes, all cached DNS records of this address will have expired, and any currently cached record will expire in five minutes or less. The manager now changes both the Internet address of the machine and also the authoritative DNS record of that address, and within a few minutes everyone in the Internet will be able to find the new address. Anyone who tries to use an old, cached, address will receive no response. But a retry a few minutes later will succeed, so from the point of view of a network client the outcome is similar to the case in which ginger.pedantic.edu crashes and restarts—for a few minutes the server is non-responsive.

There is a good reason for designing DNS to provide eventual, rather than strict, consistency, and for not requiring read/write coherence. Replicas of individual name records may potentially be cached in any name server anywhere in the Internet—there are thousands, perhaps even millions of such caches. Alerting every name server that might have cached the record that the Internet address of ginger.pedantic.edu changed would be a huge effort, yet most of those caches probably don't actually have a copy of this particular record. Furthermore, it turns out not to be that important because, as described in the previous paragraph, a network manager can easily mask any temporary inconsistency by configuring address advertisement or adjusting the time-to-live. Eventual consistency with expiration is an efficient strategy for this job.

4.3.3: Obtaining Strict Consistency with a Fluorescent Pen

In certain special situations, it is possible to regain strict consistency, and thus read/write coherence, despite the existence of multiple, private caches: If only a few variables are actually both shared and writable, mark just those variables with a fluorescent marking pen. The meaning of the mark is "don't cache me." When someone reads a marked variable, the cache manager retrieves it from secondary memory and delivers it to the client, but does not place a replica in the cache. Similarly, when a client writes a marked variable, the cache manager notices the mark in secondary memory and does not keep a copy in the cache. This scheme erodes the performance-enhancing value of the cache, so it would not work well if most variables have don't-cache-me marks.

The World Wide Web uses this scheme for Web pages that may be different each time they are read. When a client asks a Web server for a page that the server has marked "don't cache me," the server adds to the header of that page a flag that instructs the browser and any intermediaries not to cache that page.

The Java language includes a slightly different, though closely related, concept, intended to provide read/write coherence despite the presence of caches, variables in registers, and reordering of instructions, all of which can compromise strict consistency when there is concurrency. The Java memory model allows the programmer to declare a variable to be **volatile**. This declaration tells the compiler to take whatever actions (such as writing registers back to memory, flushing caches, and blocking any instruction reordering features of the processor) might be needed to ensure read/write coherence for the volatile variable within the actual memory model of the underlying system. Where the fluorescent marking pen marks a variable for special treatment by the memory system, the volatile declaration marks a variable for special treatment by the interpreter.

4.3.4: Obtaining Strict Consistency with the Snoopy Cache

The basic idea of most cache coherence schemes is to somehow **invalidate** cache entries whenever they become inconsistent with the authoritative replica. One situation where a designer can use this idea is when several processors share the same secondary memory. If the processors could also share the cache, there would be no problem. However, a shared cache tends to reduce performance, in two ways. First, to minimize latency the designer would prefer to integrate the cache with the processor, but a shared cache eliminates that option. Second, there must be some mechanism that arbitrates access to the shared cache by concurrent processors. That arbitration mechanism must enforce waits that increase access latency even more. Since the main point of a processor cache is to reduce latency, each processor usually has at least a small private cache.

Making the private cache write-through would ensure that the replica in secondary memory tracks the replica in the private cache. But write-through does not update any replicas that may be in the private caches of other processors, so by itself it doesn't provide read/write coherence. We need to add some way of telling those processors to invalidate any replicas their caches hold.

A naive approach would be to run a wire from each processor to the others and specify that whenever a processor writes to memory, it should send a signal on this wire. The other processors should, when they see the signal, assume that something in their cache has changed and, not knowing exactly what, invalidate everything their cache currently holds. Once all caches have been invalidated, the first processor can then confirm completion of its own write. This scheme would work, but it would have a disastrous effect on the cache hit rate. If 20% of processor data references are write operations, each processor will receive signals to invalidate the cache roughly every fifth data reference by each other processor. There would not be much point in having a big cache, since it would rarely have a chance to hold more than half a dozen valid entries.

To avoid invalidating the entire cache, a better idea would be to somehow communicate to the other caches the specific address that is being updated. To rapidly transmit an entire memory address in hardware could require adding a lot of wires. The trick is to realize that there is already a set of wires in place that can do this job: the memory bus. One designs each private cache to actively monitor the memory bus. If the cache notices that anyone else is doing a write operation via the memory bus, it grabs the memory address from the bus and invalidates any copy of data it has that corresponds to that address. A slightly more clever design will also grab the data value from the bus as it goes by and update, rather than invalidate, its copy of that data. These are two variations on what is called the **snoopy cache** [Suggestions for Further Reading 4.1.1]—each cache is snooping on bus activity. Figure 4.3.1 illustrates the snoopy cache.

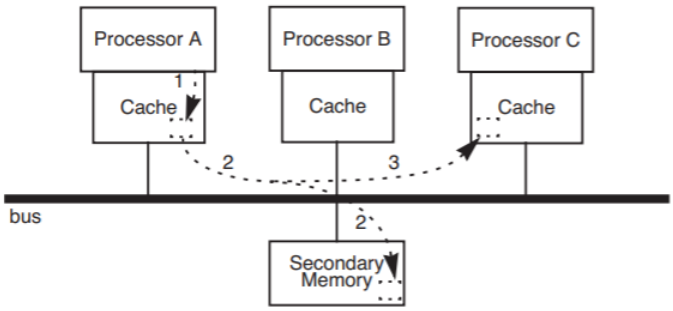


Figure 4.3.1: A configuration for which a snoopy cache can restore strict consistency and read/write coherence. When processor A writes to memory (arrow 1), its write-through cache immediately updates secondary memory using the next available bus cycle (arrow 2). The caches for processors B and C monitor ("snoop on") the bus address lines, and if they notice a bus write cycle for an address they have cached, they update (or at least invalidate) their replica of the contents of that address (arrow 3).

The registers of the various processors constitute a separate concern because they may also contain copies of variables that were in a cache at the time a variable in the cache was invalidated or updated. When a program loads a shared variable into a register, it should be aware that it is shared, and provide coordination, for example through the use of locks, to ensure that no other processor can change (and thus invalidate) a variable that this processor is holding in a register. Locks themselves generally are implemented using write-through, to ensure that cached copies do not compromise the single-acquire protocol.

A small cottage industry has grown up around optimizations of cache coherence protocols for multiprocessor systems both with and without buses, and different designers have invented many quite clever speed-up tricks, especially with respect to locks. Before undertaking a multiprocessor cache design, a prospective processor architect should review the extensive literature of the area. A good place to start is with Chapter 8 of *Computer Architecture: A Quantitative Approach*, by Hennessy and Patterson *.

* See Section 3.5.4 for more information on this source.

This page titled 4.3: Cache Coherence is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

4.4: Durable Storage Revisited - Geographically Separated Replicas

4.4.1: Durable Storage and the Durability Mantra

Chapter 2 demonstrated how to create durable storage using a technique called **mirroring**, and Section 3.8 showed how to give the mirrored replicas the all-or-nothing property when reading and writing. Mirroring is characterized by writing the replicas synchronously—that is, waiting for all or a majority of the replicas to be written before going on to the next action. The replicas themselves are called **mirrors**, and they are usually created on a physical unit basis. For example, one common RAID configuration uses multiple disks, on each of which the same data is written to the same numbered sector, and a write operation is not considered complete until enough mirror copies have been successfully written.

Mirroring helps protect against internal failures of individual disks, but it is not a magic bullet. If the application or operating system damages the data before writing it, all the replicas will suffer the same damage. Also, as shown in the fault tolerance analyses in the previous two chapters, certain classes of disk failure can obscure discovery that a replica was not written successfully. Finally, there is a concern for where the mirrors are physically located.

Placing replicas at the same physical location does not provide much protection against the threat of environmental faults, such as fire or earthquake. Having them all under the same administrative control does not provide much protection against administrative bungling. To protect against these threats, the designer uses a powerful design principle:

The Durability Mantra

> Multiple copies, widely separated and independently administered…
>
> Multiple copies, widely separated and independently administered…

Another formulation of the durability mantra is "lots of copies keep stuff safe" [Suggestions for Further Reading 4.2.3]. The idea is not new: "…let us save what remains; not by vaults and locks which fence them from the public eye and use in consigning them to the waste of time, but by such a multiplication of copies, as shall place them beyond the reach of accident." *

The first step in applying this design principle is to separate the replicas geographically. The problem with separation is that communication with distant points has high latency and is also inherently unreliable. Both of those considerations make it problematic to write the replicas synchronously. When replicas are made asynchronously, one of the replicas (usually the first replica to be written) is identified as the **primary copy**, and the site that writes it is called the **master**. The remaining replicas are called **backup copies**, and the sites that write them are called **slaves**.

The constraint usually specified for replicas is that they should be identical. But when replicas are written at different times, there will be instants when they are not identical; that is, they violate the specified constraint. If a system failure occurs during one of those instants, violation of the constraint can complicate recovery because it may not be clear which replicas are authoritative. One way to regain some simplicity is to organize the writing of the replicas in a way understandable to the application, such as file-by-file or record-by-record, rather than in units of physical storage such as disk sector-by-sector. That way, if a failure does occur during replica writing, it is easier to characterize the state of the replica: some files (or records) of the replica are up to date, some are old, the one that was being written may be damaged, and the application can do any further recovery as needed. Writing replicas in a way understandable to the application is known as making **logical copies**, to contrast it with the **physical copies** usually associated with mirrors. Logical copying has the same attractions as logical locking, and also some of the performance disadvantages, because more software layers must be involved and it may require more disk seek arm movement.

In practice, replication schemes can be surprisingly complicated. The primary reason is that the purpose of replication is to suppress unintended changes to the data caused by random decay. But decay suppression also complicates intended changes, since one must now update more than one copy, while being prepared for the possibility of a failure in the midst of that update. In addition, if updates are frequent, the protocols to perform update must not only be correct and robust, they must also be efficient. Since multiple replicas can usually be read and written concurrently, it is possible to take advantage of that possibility to enhance overall system performance. But performance enhancement can then become a complicating requirement of its own, one that interacts strongly with a requirement for strict consistency.

---

*Letter from Thomas Jefferson to the publisher and historian Ebenezer Hazard, February 18, 1791. Library of Congress, *The Thomas Jefferson Papers Series 1. General Correspondence*. 1651-1827.

4.4.2: Replicated State Machines

Data replicas require a management plan. If the data is written exactly once and never again changed, the management plan can be fairly straightforward: make several copies, put them in different places so they will not all be subject to the same environmental faults, and develop algorithms for reading the data that can cope with loss of, disconnection from, and decay of data elements at some sites.

Unfortunately, most real world data need to be updated, at least occasionally, and update greatly complicates management of the replicas. Fortunately, there exists an easily-described, systematic technique to ensure correct management. Unfortunately, it is surprisingly hard to meet all the conditions needed to make it work.

The systematic technique is a *sweeping simplification* known as the **replicated state machine**. The idea is to identify the data with the state of a finite state machine whose inputs are the updates to be made to the data, and whose operation is to make the appropriate changes to the data, as illustrated in Figure 4.4.1 by maintaining identical data replicas, co-locating with each of those replicas a replica of the state machine, and sending the same inputs to each state machine. Since the state of a finite state machine is at all times determined by its prior state and its inputs, the data of the various replicas will, in principle, perfectly match one another.

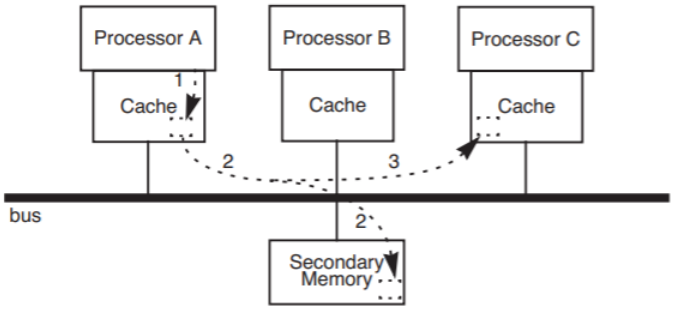


Figure 4.4.1: Replicated state machines. If N identical state machines that all have the same prior state receive and perform the same update requests in the same order, then all N of the machines will enter the same new state.

The concept is sound, but four real-world considerations conspire to make this method harder than it looks:

1. All of the state machine replicas must receive the same inputs, in the same order. Agreeing on the values and order of the inputs at separated sites is known as achieving **consensus**. Achieving consensus among sites that do not have a common clock, that can crash independently, and that are separated by a best-effort communication network is a project in itself. Consensus has received much attention from theorists, who begin by defining its core essence, known as **the consensus problem**: to achieve agreement on a single binary value. There are various algorithms and protocols designed to solve this problem under specified conditions, as well as proofs that with certain kinds of failures consensus is impossible to reach. When conditions permit solving the core consensus problem, a designer can then apply bootstrapping to come to agreement on the complete set of values and order of inputs to a set of replicated state machines.
2. All of the data replicas (in Figure 4.4.1, the "prior state") must be identical. The problem is that random decay events can cause the data replicas to drift apart, and updates that occur when they have drifted can cause them to drift further apart. So there needs to be a plan to check for this drift and correct it. The mechanism that identifies such differences and corrects them is known as **reconciliation**.
3. The replicated state machines must also be identical. This requirement is harder to achieve than it might at first appear. Even if all the sites run copies of the same program, the operating environment surrounding that program may affect its behavior, and there can be transient faults that affect the operation of individual state machines differently. Since the result is again that the data replicas drift apart, the same reconciliation mechanism that fights decay may be able to handle this problem.
4. To the extent that the replicated state machines really are identical, they will contain identical implementation faults. Updates that cause the faults to produce errors in the data will damage all the replicas identically, and reconciliation can neither detect nor correct the errors.

The good news is that the replicated state machine scheme not only is systematic, but it lends itself to modularization. One module can implement the consensus-achieving algorithm; a second set of modules, the state machines, can perform the actual updates; and

a third module responsible for reconciliation can periodically review the data replicas to verify that they are identical and, if necessary, initiate repairs to keep them that way.

4.4.3: Shortcuts to Meet More Modest Requirements

The replicated state machine method is systematic, elegant, and modular, but its implementation requirements are severe. At the other end of the spectrum, some applications can get along with a much simpler method: implement just a **single state machine**. The idea is to carry out all updates at one replica site, generating a new version of the database at that site, and then somehow bring the other replicas into line. The simplest, brute-force scheme is to send a copy of this new version of the data to each of the other replica sites, completely replacing their previous copies. This scheme is a particularly simple example of master/slave replication. One of the things that makes it simple is that there is no need for consultation among sites; the master decides what to do and the slaves just follow along.

The single state machine with brute-force copies works well if:

- The data need to be updated only occasionally.
- The database is small enough that it is practical to retransmit it in its entirety.
- There is no urgency to make updates available, so the master can accumulate updates and perform them in batches.
- The application can get along with temporary inconsistency among the various replicas. Requiring clients to read from the master replica is one way to mask the temporary inconsistency. On the other hand if, for improved performance, clients are allowed to read from any available replica, then during an update a client reading data from a replica that has received the update may receive different answers from another client reading data from a different replica to which the update hasn’t propagated yet.

This method is subject to data decay, just as is the replicated state machine, but the effects of decay are different. Undetected decay of the master replica can lead to a disaster in which the decay is propagated to the slave replicas. On the other hand, since update installs a complete new copy of the data at each slave site, it incidentally blows away any accumulated decay errors in slave replicas, so if update is frequent, it is usually not necessary to provide reconciliation. If updates are so infrequent that replica decay is a hazard, the master can simply do an occasional dummy update with unchanged data to reconcile the replicas.

The main defect of the single state machine is that even though data access can be fault-tolerant—if one replica goes down, the others may still available for reading—data update is not: if the primary site fails, no updates are possible until that failure is detected and repaired. Worse, if the primary site fails while in the middle of sending out an update, the replicas may remain inconsistent until the primary site recovers. This whole approach doesn't work well for some applications, such as a large database with a requirement for strict consistency and a performance goal that can be met only by allowing concurrent reading of the replicas.

Despite these problems, the simplicity is attractive, and in practice many designers try to get away with some variant of the single state machine method, typically tuned up with one or more enhancements:

- The master site can distribute just those parts of the database that changed (the updates are known as "deltas" or "diffs") to the replicas. Each replica site must then run an engine that can correctly update the database using the information in the deltas. This scheme moves back across the spectrum in the direction of the replicated state machine. Though it may produce a substantial performance gain, such a design can end up with the disadvantages of both the single and the replicated state machines.
- Devise methods to reduce the size of the time window during which replicas may appear inconsistent to reading clients. For example, the master could hold the new version of the database in a shadow copy, and ask the slave sites to do the same, until all replicas of the new version have been successfully distributed. Then, short messages can tell the slave sites to make the shadow file the active database. (This model should be familiar: a similar idea was used in the design of the two-phase commit protocol described in Chapter 3.)
- If the database is large, partition it into small regions, each of which can be updated independently. Section 4.4.7, below, explores this idea in more depth. (The Internet Domain Name System is for the most part managed as a large number of small, replicated partitions.)
- Assign a different master to each partition, to distribute the updating work more evenly and increase availability of update.
- Add fault tolerance for data update when a master site fails by using a consensus algorithm to choose a new master site.
- If the application is one in which the data is insensitive to the order of updates, implement a replicated state machine without a consensus algorithm. This idea can be useful if the only kind of update is to add new records to the data and the records are

identified by their contents, rather than by their order of arrival. Members of a workgroup collaborating by email typically see messages from other group members this way. Different users may find that received messages appear in different orders, and may even occasionally see one member answer a question that another member apparently hasn't yet asked, but if the email system is working correctly, eventually everyone sees every message.

- The master site can distribute just its update log to the replica sites. The replica sites can then run `REDO` on the log entries to bring their database copies up to date. Or, the replica site might just maintain a complete log replica rather than the database itself. In the case of a disaster at the master site, one of the log replicas can then be used to reconstruct the database.

This list just touches the surface. There seem to be an unlimited number of variations in application-dependent ways of doing replication.

4.4.4: Maintaining Data Integrity

In updating a replica, many things can go wrong: data records can be damaged or even completely lost track of in memory buffers of the sending or receiving systems, transmission can introduce errors, and operators or administrators can make blunders, to name just some of the added threats to data integrity. The durability mantra suggests imposing physical and administrative separation of replicas to make threats to their integrity more independent, but the threats still exist.

The obvious way to counter these threats to data integrity is to apply the method suggested in Section 3.8.1 to counter spontaneous data decay: plan to periodically compare replicas, doing so often enough that it is unlikely that all of the replicas have deteriorated. However, when replicas are not physically adjacent this obvious method has the drawback that bit-by-bit comparison requires transmission of a complete copy of the data from one replica site to another, an activity that can be time-consuming and possibly expensive.

An alternative and less costly method that can be equally effective is to calculate a **witness** of the contents of a replica and transmit just that witness from one site to another. The usual form for a witness is a hash value that is calculated over the content of the replica, thus attesting to that content. By choosing a good hash algorithm (for example, a cryptographic quality hash such as described in Sidebar 5.9.1) and making the witness sufficiently long, the probability that a damaged replica will have a hash value that matches the witness can be made arbitrarily small. A witness can thus stand in for a replica for purposes of confirming data integrity or detecting its loss.

The idea of using witnesses to confirm or detect loss of data integrity can be applied in many ways. We have already seen checksums used in communications, both for end-to-end integrity verification (Section 1.3.5) and in the link layer (Section 1.4.3); checksums can be thought of as weak witnesses. For another example of the use of witnesses, a file system might calculate a separate witness for each newly written file, and store a copy of the witness in the directory entry for the file. When later reading the file, the system can recalculate the hash and compare the result with the previously stored witness to verify the integrity of the data in the file. Two sites that are supposed to be maintaining replicas of the file system can verify that they are identical by exchanging and comparing lists of witnesses. In Chapter 5 we will see that by separately protecting a witness, one can also counter threats to data integrity that are posed by an adversary.

4.4.5: Replica Reading and Majorities

So far, we have explored various methods of creating replicas, but not how to use them. The simplest plan, with a master/slave system, is to direct all client read and write requests to the primary copy located at the master site, and treat the slave replicas exclusively as backups whose only use is to restore the integrity of a damaged master copy. What makes this plan simple is that the master site is in a good position to keep track of the ordering of read and write requests, and thus enforce a strict consistency specification such as the usual one for memory coherence: that a read should return the result of the most recent write.

A common enhancement to a replica system, intended to increase availability for read requests, is to allow reads to be directed to any replica, so that the data continues to be available even when the master site is down. In addition to improving availability, this enhancement may also have a performance advantage, since the several replicas can probably provide service to different clients at the same time. Unfortunately, the enhancement has the complication that there will be instants during update when the several replicas are not identical, so different readers may obtain different results, a violation of the strict consistency specification. To restore strict consistency, some mechanism that ensures before-or-after atomicity between reads and updates would be needed, and that before-or-after atomicity mechanism will probably erode some of the increased availability and performance.

Both the simple and the enhanced schemes consult only one replica site, so loss of data integrity, for example from decay, must be detected using just information local to that site, perhaps with the help of a witness stored at the replica site. Neither scheme takes

advantage of the data content of the other replicas to verify integrity. A more expensive, but more reliable, way to verify integrity is for the client to also obtain a second copy (or a witness) from a different replica site. If the copy (or witness) from another site matches the data (or a just-calculated hash of the data) of the first site, confidence in the integrity of the data can be quite high. This idea can be carried further to obtain copies or witnesses from several of the replicas, and compare them. Even when there are disagreements, if a majority of the replicas or witnesses agree, the client can still accept the data with confidence, and might in addition report a need for reconciliation.

Some systems push the majority idea further by introducing the concept of a **quorum**. Rather than simply "more than half the replicas," one can define separate read and write quorums, Q_r and Q_w, that have the property that $Q_r + Q_w > N_{replicas}$. This scheme declares a write to be confirmed after writing to at least a write quorum, Q_w, of replicas (while the system continues to try to propagate the write to the remaining replicas), and a read to be successful if at least a read quorum, Q_r, agree on the data or witness value. By varying Q_r and Q_w, one can configure such a system to bias availability in favor of either reads or writes in the face of multiple replica outages. In these terms, the enhanced availability scheme described above is one for which $Q_w = N_{replicas}$ and $Q_r = 1$.

Alternatively, one might run an $N_{replicas} = 5$ system with a rule that requires that all updates be made to at least $Q_w = 4$ of the replicas and that reads locate at least $Q_r = 2$ replicas that agree. This choice biases availability modestly in favor of reading: a successful write requires that at least 4 of the 5 replicas be available, while a read will succeed if only 2 of the replicas are available and agree, and agreement of 2 is ensured if any 3 are available. Or, one might set $Q_w = 2$ and $Q_r = 4$. That configuration would allow someone doing an update to receive confirmation that the update has been accomplished if any two replicas are available for update, but reading would then have to wait at least until the update gets propagated to two more replicas. With this configuration, write availability should be high but read availability might be quite low.

In practice, quorums can actually be quite a bit more complicated. The algorithm as described enhances durability and allows adjusting read versus write availability, but it does not provide either before-or-after or all-or-nothing atomicity, both of which are likely to be required to maintain strict consistency if there is either write concurrency or a significant risk of system crashes. Consider, for example, the system for which $N_{replicas} = 5$, $Q_w = 4$, and $Q_r = 2$. If an updater is at work and has successfully updated two of the replicas, one reader could read the two replicas already written by the updater while another reader might read two of the replicas that the updater hasn't gotten to yet. Both readers would believe they had found a consistent set of replicas, but the read/write coherence specification has not been met. Similarly, with the same system parameters, if an updater crashes after updating two of replicas, a second updater might come along and begin updating a different two of the replicas and then crash. That scenario would leave a muddled set of replicas in which one reader could read the replicas written by the first updater while another reader might read the replicas written by the second updater.

Thus a practical quorum scheme requires some additional before-or-after atomicity mechanism that serializes writes and ensures that no write begins until the previous write has sufficiently propagated to ensure coherence. The complexity of the mechanism depends on the exact system configuration. If all reading and updating originates at a single site, a simple sequencer at that site can provide the needed atomicity. If read requests can come from many different sources but all updates originate at a single site, the updating site can associate a version number with each data update and reading sites can check the version numbers to ensure that they have read the newest consistent set. If updates can originate from many sites, a protocol that provides a distributed sequencer implementation might be used for atomicity. Performance maximization usually is another complicating consideration. The interested reader should consult the professional literature, which describes many (sometimes quite complex) schemes for providing serialization of quorum replica systems. All of these mechanisms are specialized solutions to the generic problem of achieving atomicity across multiple sites, which was discussed at the end of Chapter 3.

4.4.6: Backup

Probably the most widely used replication technique for durable storage that is based on a single state machine is to periodically make backup copies of a complete file system on an independent, removable medium such as magnetic tape, writable video disk (DVD), or removable hard disk. Since the medium is removable, one can make the copy locally and introduce geographic separation later. If a disk fails and must be replaced, its contents can be restored from the most recent removable medium replica. Removable media are relatively cheap, so it is not necessary to recycle previous backup copies immediately. Older backup copies can serve an additional purpose, as protection against human error by acting as archives of the data at various earlier times, allowing retrieval of old data values.

The major downside of this technique is that it may take quite a bit of time to make a complete backup copy of a large storage system. For this reason, refinements such as **incremental backup** (copy only files changed since the last backup) and partial backup (don't copy files that can be easily reconstructed from other files) are often implemented. These techniques reduce the time spent making copies, but they introduce operational complexity, especially at restoration time.

A second problem is that if updates to the data are going on at the same time as backup copying, the backup copy may not be a snapshot at any single instant—it may show some results of a multi-file update but not others. If internal consistency is important, either updates must be deferred during backup or some other scheme, such as logging updates, must be devised. Since complexity also tends to reduce reliability, the designer must use caution when going in this direction.

It is worth repeating that the success of data replication depends on the independence of failures of the copies, and it can be difficult to assess correctly the amount of independence between replicas. To the extent that they are designed by the same designer and are modified by the same software, replicas may be subject to the same design or implementation faults. It is folk wisdom among system designers that the biggest hazard for a replicated system is replicated failures. For example, a programming error in a replicated state machine may cause all of the data replicas to become identically corrupted. In addition, there is more to achieving durable storage than just replication. Because a thread can fail at a time when some invariant on the data is not satisfied, additional techniques are needed to recover the data.

Complexity can also interfere with success of a backup system. Another piece of folk wisdom is that the more elaborate the backup system, the less likely that it actually works. Most experienced computer users can tell tales of the day that the disk crashed, and for some reason the backup copy did not include the most important files. (But the tale usually ends with a story that the owner of those files didn't trust the backup system, and was able to restore those important files from an *ad hoc* copy he or she made independently.)

4.4.7: Partitioning Data

A quite different approach to tolerating failures of storage media is to simply partition the data, thereby making the system somewhat fail-soft. In a typical design, one would divide a large collection of data into several parts, each of about the same size, and place each part on a different physical device. Failure of any one of the devices then compromises availability of only one part of the entire set of data. For some applications this approach can be useful, easy to arrange and manage, easy to explain to users, and inexpensive. Another reason that partition is appealing is that access to storage is often a bottleneck. Partition can allow concurrent access to different parts of the data, an important consideration in high-performance applications such as popular Web servers.

Replication can be combined with partition. Each partition of the data might itself be replicated, with the replicas placed on different storage devices, and each storage device can contain replicas of several of the different partitions. This strategy ensures continued availability if any single storage device fails, and at the same time an appropriate choice of configuration can preserve the performance-enhancing feature of partition.

This page titled 4.4: Durable Storage Revisited - Geographically Separated Replicas is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

4.5: Reconciliation

A typical constraint for replicas is that a majority of them be identical. Unfortunately, various events can cause them to become different: data of a replica can decay, a replicated state machine may experience an error, an update algorithm that has a goal of eventual consistency may be interrupted before it reaches its goal, an administrator of a replica site may modify a file in a way that fails to respect the replication protocol, or a user may want to make an update at a time when some replicas are disconnected from the network. In all of these cases, a need arises for an after-the-fact procedure to discover the differences in the data and to recover consistency. This procedure, called **reconciliation**, makes the replicas identical again.

Although reconciliation is a straightforward concept in principle, in practice three things conspire to make it more complicated than one might hope:

1. For large bodies of data, the most straightforward methods (e.g., compare all the bits) are expensive, so performance enhancements dominate, and complicate, the algorithms.
2. A system crash during a reconciliation can leave a body of data in worse shape than if no reconciliation had taken place. The reconciliation procedure itself must be resilient against failures and system crashes.
3. During reconciliation, one may discover conflicts, which are cases where different replicas have been modified in inconsistent ways. And in addition to files decaying, decay may also strike records kept by the reconciliation system itself.

One way to simplify thinking about reconciliation is to decompose it into two distinct modular components:

1. Detecting differences among the replicas.
2. Resolving the differences so that all the replicas become identical.

At the outset, every difference represents a potential conflict. Depending on how much the reconciliation algorithm knows about the semantics of the replicas, it may be able to algorithmically resolve many of the differences, leaving a smaller set of harder-to-handle conflicts. The remaining conflicts generally require more understanding of the semantics of the data, and ultimately may require a decision to be made on the part of a person. To illustrate this decomposition, the next section examines one widely-implemented reconciliation application, known as **occasionally connected operation**, in some detail.

4.5.1: Occasionally Connected Operation

A common application for reconciliation arises when a person has both a desktop computer and a laptop computer, and needs to work with the same files on both computers. The desktop computer is at home or in an office, while the laptop travels from place to place, and because the laptop is often not network-connected, changes made to a file on one of the two computers can not be automatically reflected in the replica of that file on the other. This scenario is called **occasionally connected operation**. Moreover, while the laptop is disconnected files may change on either the desktop or the laptop (for example, the desktop computer may pick up new incoming mail or do an automatic system update while the owner is traveling with the laptop and editing a report). We are thus dealing with a problem of concurrent update to multiple replicas.

Recall from the discussion in Section 3.5.3 that there are both pessimistic and optimistic concurrency control methods. Either method can be applied to occasionally connected replicas:

- Pessimistic: Before disconnecting, identify all of the files that might be needed in work on the laptop computer and mark them as "checked out" on the desktop computer. The file system on the desktop computer then blocks any attempts to modify checked-out files. A pessimistic scheme makes sense if the traveler can predict exactly which files the laptop should check out and it is likely that someone will also attempt to modify them at the desktop.
- Optimistic: Allow either computer to update any file and, the next time that the laptop is connected, detect and resolve any conflicting updates. An optimistic scheme makes sense if the traveler cannot predict which files will be needed while traveling and there is little chance of conflict anyway.

Either way, when the two computers can again communicate, reconciliation of their replicas must take place. The same need for reconciliation applies to the handheld computers known as "personal digital assistants," which may have replicas of calendars, address books, to-do lists, or databases filled with business cards. The popular term for this kind of reconciliation is "file synchronization." We avoid using that term because "synchronization" has too many other meanings.

The general outline of how to reconcile the replicas seems fairly simple: If a particular file changed on one computer but not on the other, the reconciliation procedure can resolve the difference by simply copying the newer file to the other computer. In the pessimistic case that is all there is to it. If the optimistic scheme is being used, the same file may have changed on both computers.

If so, that difference is a conflict and reconciliation requires more guidance to figure out how the resolve it. For the file application, both the detection step and the resolution step can be fairly simple.

The most straightforward and accurate way to detect differences would be to read both copies of the file and compare their contents, bit by bit, with a record copy that was made at the time of the last reconciliation. If either file does not match the record copy, there is a difference; if both files fail to match the record copy, there is a conflict. But this approach would require maintaining a record copy of the entire file system as well as transmitting all of the data of at least one of the file systems to the place that holds the record copy. Thus there is an incentive to look for shortcuts.

One shortcut is to use a witness in place of the record copy. The reconciliation algorithm can then detect both differences and conflicts by calculating the current hash of a file and comparing it with a witness that was stored at the time of the previous reconciliation. Since a witness is likely to be much smaller than the original file, it does not take much space to store and it is easy to transmit across a network for comparison. The same set of stored witnesses can also support a decay detector that runs in a low-priority thread, continually reading files, recalculating their hash values, and comparing them with the stored witnesses to see if anything has changed.

Since witnesses require a lot of file reading and hash computation, a different shortcut is to just examine the time of last modification of every file on both computers, and compare that with the time of last reconciliation. If either file has a newer modification timestamp, there is a difference, and if both have newer modification timestamps, there is a conflict. This shortcut is popular because most file systems maintain modification timestamps as part of the metadata associated with a file. One requirement of this shortcut is that the timestamp have a resolution fine enough to ensure that every time a file is modified its timestamp increases. Unfortunately, modification timestamps are an approximation to witnesses that have several defects. First, the technique does not discover decay because decay events change file contents without updating modification times. Second, if someone modifies a file, then undoes the changes, perhaps because a transaction was aborted, the file will have a new timestamp and the reconciliation algorithm will consider the file changed, even though it really hasn't. Finally, the system clocks of disconnected computers may drift apart or users may reset system clocks to match their wristwatches (and some file systems allow the user to "adjust" the modification timestamp on a file), so algorithms based on comparing timestamps may come to wrong conclusions as to which of two file versions is "newer." The second defect affects performance rather than correctness, and the impact may be inconsequential, but the first and third defects can create serious correctness problems.

A file system can provide a different kind of shortcut by maintaining a systemwide sequence number, known as a **generation number**. At some point when the replicas are known to be identical, both file systems record as part of the metadata of every file a starting generation number, say zero, and they both set their current systemwide generation numbers to one. Then, whenever a user modifies a file, the file system records in the metadata of that file the current generation number. When the reconciliation program next runs, by examining the generation numbers on each file it can easily determine whether either or both copies of a file were modified since the last reconciliation: if either copy of the file has the current generation number, there is a difference; if both copies of the file have the current generation number, there is a conflict. When the reconciliation is complete and the two replicas are again identical, the file systems both increase their current generation numbers by one in preparation for the next reconciliation. Generation numbers share two of the defects of modification timestamps. First, they do not allow discovery of decay, since decay events change file contents without updating generation numbers. Second, an aborted transaction can leave one or more files with a new generation number even though the file contents haven't really changed. An additional problem that generation numbers do not share with modification timestamps is that implementation of generation numbers is likely to require modifying the file system.

The resolution step usually starts with algorithmic handling of as many detected differences as possible, leaving (one hopes) a short list of conflicts for the user to resolve manually.

4.5.2: A Reconciliation Procedure

To illustrate some of the issues involved in reconciliation, Figure 4.5.1 shows a file reconciliation procedure named `RECONCILE`, which uses timestamps. To simplify the example, files have path names, but there are no directories. The procedure reconciles two sets of files, named *`left`* and *`right`*, which were previously reconciled at *`last_reconcile_time`*, which acts as a kind of generation number. The procedure assumes that the two sets of files were identical at that time, and its goal is to make the two sets identical again, by examining the modification timestamps recorded by the storage systems that hold the files. The function `MODIFICATION_TIME(`*`file`*`)` returns the time of the last modification to *`file`*. The copy operation, in addition to copying a file from one set to another, also copies the time of last modification, if necessary creating a file with the appropriate file name.

```
1  procedure RECONCILE (reference left, reference right,
2                                          reference last_reconcile_time)
3      quiesce all activity on left and right   // Shut down all file-using applications
4      ALL_OR_NOTHING_GET (last_reconcile_time, reconcile_time_sector)
5      left_list ← enumerate(left)
6      right_list ← enumerate(right)
7      common_list ← intersect(left_list, right_list)
8      left_only_list ← remove members of common_list from left_list
9      right_only_list ← remove members of common_list from right_list
10     conflict_list ← NIL

11     for each named_file in common_list do   // Reconcile files found both sides
12         left_new ← (MODIFICATION_TIME (left.named_file) > last_reconcile_time)
13         right_new ← (MODIFICATION_TIME (right.named_file) > last_reconcile_time)
14         if left_new and right_new then
15             add named_file to conflict_list
16         else if left_new then
17             copy named_file from left to right
18         else if right_new then
19             copy named_file from right to left
20         else if MODIFICATION_TIME (left.named_file) ≠
21                 (MODIFICATION_TIME (right.named_file)
22             then TERMINATE ("Something awful has happened.")
23     for each named_file in left_only_list do   // Reconcile files found one side
24         if MODIFICATION_TIME (left.named_file) > last_reconcile_time then
25             copy named_file from left to right
26         else
27             delete left.named_file
28     for each named_file in right_only_list do
29         if MODIFICATION_TIME (right.named_file) > last_reconcile_time then
30             copy named_file from right to left
31         else
32             delete right.named_file

33     for each named_file in conflict_list do          // Handle conflicts
34         MANUALLY_RESOLVE (right.named_file, left.named_file)
35     last_reconcile_time ← NOW ()
36     ALL_OR_NOTHING_PUT (last_reconcile_time, reconcile_time_sector)
37     Allow activity to resume on left and right
```

Figure 4.5.1: A simple reconciliation algorithm.

RECONCILE operates as a transaction. To achieve all-or-nothing atomicity, RECONCILE is constructed to be idempotent; in addition, the **copy** operation must be atomic. To achieve before-or-after atomicity, RECONCILE must run by itself, without anyone else making more changes to files while its executes, so it begins by quiescing all file activity, perhaps by setting a lock that prevents new files from being opened by anyone other than itself, and then waiting until all files opened by other threads have been closed. For durability, reconcile depends on the underlying file system. Its constraint is that when it exits, the two sets *left* and *right* are identical.

RECONCILE prepares for reconciliation by reading from a dedicated disk sector the timestamp of the previous reconciliation and enumerating the names of the files on both sides. From the two enumerations, program lines 7 through 9 create three lists:

- names of files that appear on both sides (*common_list*),
- names of files that appear only on the left (*left_only_list*), and
- names of files that appear only on the right (*right_only_list*).

These three lists drive the rest of the reconciliation. Line 10 creates an empty list named *conflict_list* , which will accumulate names of any files that it cannot algorithmically reconcile.

Next, RECONCILE reviews every file in *common_list* . It starts, on lines 12 and 13, by checking timestamps to see whether either side has modified the file. If both sides have timestamps that are newer than the timestamp of the previous run of the reconciliation program, that indicates that both sides have modified the file, so it adds that file name to the list of conflicts. If only one side has a newer timestamp, it takes the modified version to be the authoritative one and copies it to the other side. (Thus, this program does some difference resolution at the same time that it is doing difference detection. Completely modularizing these two steps would require two passes through the lists of files, and thereby reduce performance.) If both file timestamps are older than the timestamp of the previous run, it checks to make sure that the timestamps on both sides are identical. If they are not, that suggests

that the two file systems were different at the end of the previous reconciliation, perhaps because something went wrong during that attempt to reconcile, so the program terminates with an error message rather than blundering forward and taking a chance on irreparably messing up both file systems.

Having handled the list of names of files found on both sides, `RECONCILE` then considers those files whose names it found on only one side. This situation can arise in three ways:

1. one side deletes an old file,
2. the other side creates a new file, or
3. one side modifies a file that the other side deletes.

The first case is easily identified by noticing that the side that still has the file has not modified it since the previous run of the reconciliation program. For this case `RECONCILE` deletes the remaining copy. The other two cases cannot, without keeping additional state, be distinguished from one another, so `RECONCILE` simply copies the file from one side to the other. A consequence of this choice is that a deleted file will silently reappear if the other side modified it after the previous invocation of `RECONCILE` . An alternative implementation would be to declare a conflict, and ask the user to decide whether to delete or copy the file. With that choice, every newly created file requires manual intervention at the next run of `RECONCILE` . Both implementations create some user annoyance. Eliminating the annoyance is possible but requires an algorithm that remembers additional, per-file state between runs of `RECONCILE` .

Having reconciled all the differences that could be resolved algorithmically, `RECONCILE` asks the user to resolve any remaining conflicts by manual intervention. When the user finishes, `RECONCILE` is ready to commit the transaction, which it does by recording the current time in the dedicated disk sector, in line 36. It then allows file creation activity to resume, and it exits. The two sets of files are again identical.

4.5.3: Improvements

There are several improvements that we could make to this simple reconciliation algorithm to make it more user-friendly or comprehensive. As usual, each improvement adds complexity. Here are some examples:

1. Rather than demanding that the user resolve all remaining conflicts on the spot, it would be possible to simply notify the user that there is a non-empty conflict list and let the user resolve those conflicts at leisure. The main complication this improvement adds is that the user is likely to be modifying files (and changing file modification timestamps) at the same time that other file activity is going on, including activity that may be generating new inconsistencies among the replicas. Changes that the user makes to resolve the conflicts may thus look like new conflicts next time the reconciliation program runs. A second complication is that there is no assurance that the user actually reconciles the conflicts; the conflict list may still be non-empty the next time that the reconciliation program runs, and it must take that possibility into account. A simple response could be for the program to start by checking the previous conflict list to see if it is empty, and if it is not asking the user to take care of it before proceeding.

2. Some of the remaining conflicts may actually be algorithmically resolvable, with the help of an application program that understands the semantics and format of a particular file. Consider, for example, an appointment calendar application that stores the entire appointment book in a single file. If the user adds a 1 p.m. meeting to the desktop replica and a 4 p.m. meeting to the laptop replica, both files would have modification timestamps later than the previous reconciliation, so the reconciliation program would flag these files as a conflict. On the other hand, the calendar application program might be able to resolve the conflict by copying both meeting records to both files. What is needed is for the calendar application to perform the same kind of detection/resolution reconciliation we have already seen, but applied to individual appointment records rather than to the whole file. Any application that maintains suitable metadata (e.g. a record copy, witnesses, a generation number, or a timestamp showing when each entry in its database was last modified) can do such a record-by-record reconciliation. Of course, if the calendar application encounters two conflicting changes to the same appointment record, it probably would refer that conflict to the user for advice. The result of the application-specific reconciliation should be identical files on both replicas with identical modification timestamps.

   Application-specific reconciliation procedures have been designed for many different specialized databases such as address books, to-do lists, and mailboxes; all that is required is that the program designer develop an appropriate reconciliation algorithm. For convenience, it is helpful to integrate these application-specific procedures with the main reconciliation

procedure. The usual method is for such applications to register their reconciliation procedures, along with a list of files or file types that each reconciliation procedure can handle, with the main reconciliation program. The main reconciliation program then adds a step of reviewing its conflict list to see if there is an application-specific program available for each file. If there is, it invokes that program, rather than asking the user to resolve the conflict.

3. As it stands, the reconciliation procedure enumerates only files. If it were to be applied to a file system that has directories, links, and file metadata other than file names and modification times, it might do some unexpected things. For example, the program would handle links badly, by creating a second copy of the linked file, rather than creating a link. Most reconciliation programs have substantial chunks of code devoted to detecting and resolving differences in directories and metadata. Because the semantics of the directory management operations are usually known to the writer of the reconciliation program, many differences between directories can be resolved algorithmically. However, there can still be a residue of conflicts that require user guidance to resolve, such as when a file named A has been created in a directory on one side and a different file named A has been created in the same directory on the other side.

4.5.4: Clock Coordination

This `RECONCILE` program is relatively fragile. It depends, for example, on the timestamps being accurate. If the two sets of files are managed by different computer systems with independent clocks, and someone sets the clock incorrectly on one side, the timestamps on that side will also be incorrect, with the result that `RECONCILE` may not notice a conflict, it may overwrite a new version of a file with an old version, it may delete a file that should not be deleted, or it may incorrectly revive a deleted file. For the same reason, `RECONCILE` must carefully preserve the variable *`last_reconcile_time`* from one run to the next.

Some reconciliation programs try to minimize the possibility of accidental damage by reading the current clock value from both systems, noting the difference, and taking that difference into account. If the difference has not changed since the previous reconciliation, reconcile can simply add (or subtract, as appropriate) the time difference and proceed as usual. If the difference has changed, the amount of the change can be considered a delta of uncertainty; any file whose fate depends on that uncertainty is added to the list of conflicts for the user to resolve manually.

This page titled 4.5: Reconciliation is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

4.6: Perspectives

In Chapters 3 and 4 we have gone into considerable depth on various aspects of atomicity and systematic approaches to providing it. At this point it is appropriate to stand back from the technical details and try to develop some perspective on how all these ideas relate to the real world. The observations of this section are wide-ranging: history, trade-offs, and unexplored topics. Individually these observations appear somewhat disconnected, but in concert they may provide the reader with some preparation for the way that atomicity fits into the practical world of computer system design.

4.6.1: History

Systematic application of atomicity to recovery and to coordination is relatively recent. *Ad hoc* programming of concurrent activities has been common since the late 1950s, when machines such as the IBM 7030 (STRETCH) computer and the experimental TX–0 at M.I.T. used interrupts to keep I/O device driver programs running concurrently with the main computation. The first time-sharing systems (in the early 1960s) demonstrated the need to be more systematic in interrupt management, and many different semantic constructs were developed over the next decade to get a better grasp on coordination problems: Edsger Dijkstra's semaphores, Per Brinch Hansen's message buffers, David Reed and Raj Kanodia's eventcounts, Nico Habermann's path expressions, and Anthony Hoare's monitors are examples. A substantial literature grew up around these constructs, but a characteristic of all of them was a focus on properly coordinating concurrent activities, each of which by itself was assumed to operate correctly. The possibility of failure and recovery of individual activities, and the consequences of such failure and recovery on coordination with other, concurrent activities, was not a focus of attention. Another characteristic of these constructs is that they resemble a machine language, providing low-level tools but little guidance in how to apply them.

Failure recovery was not simply ignored in those early systems, but it was handled quite independently of coordination, again using *ad hoc* techniques. The early timesharing system implementers found that users required a kind of durable storage, in which files could be expected to survive intact in the face of system failures. To this end most time-sharing systems periodically made backup copies of online files, using magnetic tape as the backup medium. The more sophisticated systems developed incremental backup schemes, in which recently created or modified files were copied to tape on an hourly basis, producing an almost-up-to-date durability log. To reduce the possibility that a system crash might damage the on-line disk storage contents, salvager programs were developed to go through the disk contents and repair obvious and common kinds of damage. The user of a modern personal computer will recognize that some of these techniques are still in widespread use.

These *ad hoc* techniques, though adequate for some uses, were not enough for designers of serious database management systems. To meet their requirements, they developed the concept of a transaction, which initially was exactly an all-or-nothing action applied to a database. Recovery logging protocols thus developed in the database environment, and it was some time before it was recognized that recovery semantics had wider applicability.

Within the database world, coordination was accomplished almost entirely by locking techniques that became more and more systematic and automatic, with the realization that the definition of correctness for concurrent atomic actions involved getting the same result as if those atomic actions had actually run one at a time in some serial order. The database world also contributed the concept of maintaining constraints or invariants among different data objects, and the word **transaction** came to mean an action that is both all-or-nothing and before-or-after and that can be used to maintain constraints and provide durability. The database world also developed systematic replication schemes, primarily to enhance reliability and availability, but also to enhance performance.

The understanding of before-or-after atomicity, along with a requirement for hierarchical composition of programs, in turn led to the development of version history (also called **temporal database** or **time domain addressing**) systems. Version histories systematically provide both recovery and coordination with a single mechanism, and they simplify building big atomic actions out of several, independently developed, smaller ones.

This text has reversed this order of development because the relatively simple version history is pedagogically more straightforward, while the higher complexity of the logging/locking approach is easier to grasp after seeing why version histories work. Version histories are used in source code management systems and also in user interfaces that provide an UNDO button, but virtually all commercial database management systems use logs and locking in order to attain maximum performance.

4.6.2: Trade-Offs

An interesting set of trade-offs applies to techniques for coordinating concurrent activities. Figure 4.6.1 suggests that there is a spectrum of coordination possibilities, ranging from totally serialized actions on the left to complete absence of coordination on the right. Starting at the left, we can have great simplicity (for example by scheduling just one thread at a time) but admit no concurrency at all. Moving toward the right, the complexity required to maintain correctness increases but so does the possibility of improved performance, since more and more concurrency is admitted. For example, the mark-point and simple locking disciplines might lie more toward the left end of this spectrum while two-phase locking would be farther to the right. The solid curved line in the figure represents a boundary of increasing minimum complexity, below which that level of coordination complexity can no longer ensure correctness; outcomes that do not correspond to any serial schedule of the same actions become possible. (For purposes of illustration, the figure shows the boundary line as a smooth increasing curve, but that is a gross oversimplification. At the first hint of concurrency, the complexity leaps upward.)

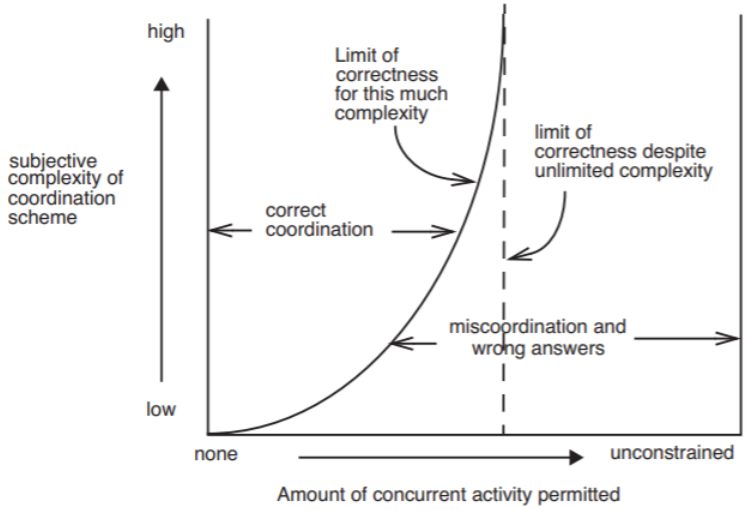


Figure 4.6.1: The trade-off among concurrency, complexity, and correctness. The choice of where in this chart to position a system design depends on the answers to two questions: 1) How frequently will concurrent activities actually interfere with one another? 2) How important are 100% correct results? If interference is rare, it is appropriate to design farther to the right. If correctness is not essential, it may be acceptable to design even to the right of the two correctness boundaries.

Continuing to traverse the concurrency spectrum to the right, one passes a point, indicated by the dashed vertical line, beyond which correctness cannot be achieved no matter how clever or complex the coordination scheme. The closer one approaches this limit from the left, the higher the performance, but at the cost of higher complexity. All of the algorithms explored in Chapters 3 and 4 are intended to operate to the left of the correctness limit, but we might inquire about the possibilities of working on the other side. Such a possibility is not as unthinkable as it might seem at first. If interference between concurrent activities is rare, and the cost of an error is small, one might actually be willing to permit concurrent actions that can lead to certifiably wrong answers. Section 3.6.4 suggested that designers sometimes employ locking protocols that operate in this region.

For example, in an inventory control system for a grocery store, if an occasional sale of a box of cornflakes goes unrecorded because two point-of-sale terminals tried to update the cornflakes inventory concurrently, the resulting slight overstatement of inventory may not be a serious problem. The grocery store must do occasional manual inventory anyway because other boxes of cornflakes are misplaced, damaged, and stolen, and employees sometimes enter wrong numbers when new boxes are delivered. This higher-layer data recovery mechanism will also correct any errors that creep in because of miscoordination in the inventory management system, so its designer might well decide to use a coordination technique that allows maximum concurrency, is simple, catches the most common miscoordination problems, but nevertheless operates below or to the right of the strict correctness line. A decision to operate a data management system in a mode that allows such errors can be made on a rational basis. One would compare the rate at which the system loses track of inventory because of its own coordination errors with the rate at which it loses track because of outside, uncontrolled events. If the latter rate dominates, it is not necessary to press the computer system for better accuracy.

Another plausible example of acceptable operation outside the correctness boundary is the calculation, by the Federal Reserve Bank, of the United States money supply. Although in principle one could program a two-phase locking protocol that includes

every bank account in every bank that contains U.S. funds, the practical difficulty of accomplishing that task with thousands of independent banks distributed over a continent is formidable. Instead, the data is gathered without locking, with only loose coordination and it is almost certain that some funds are counted twice and other funds are overlooked. However, great precision is not essential in the result, so lack of perfect coordination among the many individual bank systems operating concurrently is acceptable.

Although allowing incorrect coordination might appear usable only in obscure cases, it is actually applicable to a wider range of situations than one might guess. In almost all database management applications, the biggest cause of incorrect results is wrong input by human operators. Typically, stored data already has many defects before the transaction programs of the database management system have a chance to "correctly" transform it. Thus the proper perspective is that operation outside of the correctness boundaries of Figure 4.6.1 merely adds to the rate of incorrectness of the database. We are making an *end-to-end argument* here: there may be little point in implementing heroic coordination correctness measures in a lower layer if the higher-layer user of our application makes other mistakes, and has procedures in place to correct them anyway.

With that perspective, one can in principle balance heavy-handed but "correct" transaction coordination schemes against simpler techniques that can occasionally damage the data in limited ways. One piece of evidence that this approach is workable in practice is that many existing data management systems offer optional locking protocols called "cursor stability," "read committed," or "snapshot isolation," all of which are demonstrably incorrect in certain cases. However, the frequency of interacting update actions that actually produce wrong answers is low enough and the benefit in increased concurrency is high enough that users find the trade-off tolerable. The main problem with this approach is that no one has yet found a good way of characterizing (with the goal of limiting) the errors that can result. If you can't bound the maximum damage that could occur, then these techniques may be too risky.

An obvious question is whether or not some similar strategy of operating beyond a correctness boundary applies to atomicity. Apparently not, at least in the area of instruction set design for central processors. Three generations of central processor designers (of the main frame processors of the 1950s and 1960s, the mini-computers of the 1970s, and the one-chip microprocessors of the 1980s) did not recognize the importance of all-or-nothing atomicity in their initial design and were later forced to retrofit it into their architectures in order to accommodate the thread switching that accompanies multilevel memory management.

4.6.3: Directions for Further Study

Chapters 3 and 4 have opened up only the first layer of study of atomicity, transactions, durability, replication, and consistency; there are thick books that explore the details. Among the things we have touched only lightly (or not at all) are distributed atomic actions, hierarchically composing programs with modules that use locks, the systematic use of loose or incorrect coordination, the systematic application of compensation, and the possibility of malicious participants.

Implementing distributed atomic actions efficiently is a difficult problem for which there is a huge literature, with some schemes based on locking, others on timestamp-based protocols or version histories, some on combining the two, and yet others with optimistic strategies. Each such scheme has a set of advantages and disadvantages with respect to performance, availability, durability, integrity, and consistency. No one scheme seems ready to dominate and new schemes appear regularly.

Hierarchical composition—making larger atomic actions out of previously programmed smaller ones—interacts in an awkward way with locking as a before-or-after atomicity technique. The problem arises because locking protocols require a lock point for correctness. Creating an atomic action from two previously independent atomic actions is difficult because each separate atomic action has its own lock point, coinciding with its own commit point. But the higher-layer action must also have a lock point, suggesting that the order of capture and release of locks in the constituent atomic action needs to be changed. Rearrangement of the order of lock capture and release contradicts the usual goal of modular composition, under which one assembles larger systems out of components without having to modify the components. To maintain modular composition, the lock manager needs to know that it is operating in an environment of hierarchical atomic actions. With this knowledge, it can, behind the scenes, systematically rearrange the order of lock release to match the requirements of the action nesting. For example, when a nested atomic action calls to release a lock, the lock manager can simply relabel that lock to show that it is held by the higher-layer, not-yet-committed atomic action in which this one is nested. A systematic discipline of passing locks up and down among nested atomic actions thus can preserve the goal of modular composition, but at a cost in complexity.

Returning to the idea suggested by Figure 4.6.1, the possibility of designing a system that operates in the region of incorrectness is intriguing, but there is one major deterrent: one would like to specify, and thus limit, the nature of the errors that can be caused by

miscoordination. This specification might be on the magnitude of errors, or their direction, or their cumulative effect, or something else. Systematic specification of tolerance of coordination errors is a topic that has not been seriously explored.

Compensation is the way that one deals with miscoordination or with recovery in situations where rolling back an action invisibly cannot be accomplished. Compensation is performing a visible action that reverses all known effects of some earlier, visible action. For example, if a bank account was incorrectly debited, one might later credit it for the missing amount. The usefulness of compensation is limited by the extent to which one can track down and reverse everything that has transpired since the action that needs reversal. In the case of the bank account, one might successfully discover that an interest payment on an incorrect balance should also be adjusted; it might be harder to reverse all the effects of a check that was bounced because the account balance was incorrectly too low. Apart from generalizations along the line of "one must track the flow of information output of any action that is to be reversed" little is known about systematic compensation; it seems to be an application-dependent concept. If committing the transaction resulted in drilling a hole or firing a missile, compensation may not be an applicable concept.

Finally, all of the before-or-after atomicity schemes we explored assume that the various participants are all trying to reach a consistent, correct result. Another area of study explores what happens if one or more of the workers in a multiple-site coordination task decides to mislead the others, such as by sending a message to one site reporting it has committed, while sending a message to another site reporting it has aborted. (This possibility is described colorfully as the **Byzantine Generals' problem**.) The possibility of adversarial participants merges concerns of security with those of atomicity. The solutions so far are based primarily on extension of the coordination and recovery protocols to allow achieving consensus in the face of adversarial behavior. There has been little overlap with the security mechanisms that will be studied in Chapter 5.

One reason for exploring this area of overlap between atomicity and security is the concern that undetected errors in communication links could simulate uncooperative behavior. A second reason is increasing interest in peer-to-peer network communication, which frequently involves large numbers of administratively independent participants who may, either accidentally or intentionally, engage in Byzantine behavior. Another possible source of Byzantine behavior could lie in outsourcing of responsibility for replica storage.

This page titled 4.6: Perspectives is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

4.7: Exercises

Exercise 4.7.1

1994–3–2

You are developing a storage system for a application that demands unusually high reliability, so you have decided to use a three-replica durable storage scheme. You plan to use three ordinary disk drives D1, D2, and D3, and arrange that D2 and D3 store identical mirror copies of each block stored on D1. The disk drives are of a simple design that does not report read errors. That is, they just return data, whether or not it is valid.

a) You initially construct the application so that it writes a block of data to the same sector on all three drives concurrently. After a power failure occurs during the middle of a write, you are unable to reconstruct the correct data for that sector. What is the problem?

b) Describe a modification that solves this problem.

c) One day there is a really awful power glitch that crashes all three disks in such a way that each disk corrupts one random track. Fortunately, the system wasn't writing any data at the time. Describe a procedure for reconstructing the data and explain any cases that your procedure cannot handle.

Exercise 4.7.2

1999–3–04

What assumptions does the design of the `RECONCILE` procedure of Section 4.5.2 make with respect to concurrent updates to different replicas of the same file?

A. It assumes that these conflicts seldom happen.

B. It assumes that these conflicts can be automatically detected.

C. It assumes that all conflicts can be automatically resolved later.

D. It assumes that these conflicts cannot happen.

Exercise 4.7.3

1997–0–03

Mary uses `RECONCILE` to keep the files in her laptop computer coordinated with her desktop computer. However, she is getting annoyed. While she is traveling, she works on her email inbox, reading and deleting messages, and preparing replies, which go into an email outbox. When she gets home, `RECONCILE` always tells her that there is a conflict with the inbox and outbox on her desktop because while she was gone the system added several new messages to the desktop inbox, and it dispatched and deleted any messages that were in the desktop outbox. Her mailer implements the inbox as a single file and the outbox as a single file. Ben suggests that Mary switch to a different mailer, one that implements the inbox and outbox as two directories, and places each incoming or outgoing message in a separate file. Assuming that no one but the mail system touches Mary's desktop mailboxes in her absence, which of the following is the most accurate description of the result?

A. `RECONCILE` will still not be able to reconcile either the inbox or the outbox.

B. `RECONCILE` will be able to reconcile the inbox but not the outbox.

C. `RECONCILE` will be able to reconcile the outbox but not the inbox.

D. `RECONCILE` will be able to reconcile both the inbox and the outbox.

Exercise 4.7.4

2008–3–2

Which of the following statements are true of the `RECONCILE` program of Section 4.5.2?

A. If `RECONCILE` finds that the content of one copy of a file differs from the other copy of the same file, it indicates a conflict.

B. You create a file X with content "a" in file set 1, then create a file X with content "b" in file set 2. You then delete X from host 1, and run `RECONCILE` to synchronize the two file sets. After `RECONCILE` finishes you'll see a file X with content "b" in file set 1.

C. If you accidentally reset `RECONCILE` 's variable named *`last_reconcile_time`* to midnight, January 1, 1900, you are likely to need to resolve many more conflicts when you next run RECONCILE than if you had preserved that variable.

Exercise 4.7.4

Here is a proposed invariant for a file reconciler such as the program `RECONCILE` of Section 4.5.2: At every moment during a run of `RECONCILE` , every file has either its original contents, or its correct final contents. Which of the following statements is true about `RECONCILE` ?

A. `RECONCILE` does not attempt to maintain this invariant.

B. `RECONCILE` maintains this invariant in all cases.

C. `RECONCILE` uses file creation time to determine the most recent version of a file.

D. If the two file sets are on different computers connected by a network, `RECONCILE` would have to send the content of one version of each file over the network to the other computer for comparison.

This page titled 4.7: Exercises is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

CHAPTER OVERVIEW

5: Information Security

5.1: Overview
5.2: Introduction to Secure Systems
5.3: Authenticating Principals
5.4: Authenticating Messages
5.5: Message Confidentiality
5.6: Security Protocols
5.7: Authorization-Controlled Sharing
5.8: Reasoning About Authentication (Advanced Topic)
5.9: Cryptography as a Building Block (Advanced Topic)
5.10: Summary
5.11: Case Study - Transport Layer Security (TLS) for the Web
5.12: War Stories - Security System Breaches
5.13: Exercises

This page titled 5: Information Security is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

5.1: Overview

Secure computer systems ensure that users' privacy and possessions are protected against malicious and inquisitive users. Security is a broad topic, ranging from issues such as not allowing your friend to read your files to protecting a nation's infrastructure against attacks. Defending against an adversary is a **negative** goal. The designer of a computer system must ensure that an adversary cannot breach the security of the system in any way. Furthermore, the designer must make it difficult for an adversary to side-step the security mechanism; one of the simplest ways for an adversary to steal confidential information is to bribe someone on the inside.

Because security is a negative goal, it requires designers to be careful and pay attention to the details. Each detail might provide an opportunity for an adversary to breach the system security. Fortunately, many of the previously-encountered design principles can also guide the designer of secure systems. For example, the principles of the **safety net** approach from Chapter 2, *be explicit* (state your assumptions so that they can be reviewed) and *design for iteration* (assume you will make errors), apply equally, or perhaps even with more force, to security.

The conceptual model for protecting computer systems against adversaries is that some agent presents to a computer system a claimed identity and requests the system to perform some specified action. To achieve security, the system must obtain trustworthy answers to the following three questions before performing the requested action:

1. Authenticity: Is the agent's claimed identity authentic? (Or, is someone masquerading as the agent?)
2. Integrity: Is this request actually the one the agent made? (Or, did someone tamper with it?)
3. Authorization: Has a proper authority granted permission to this agent to perform this action?

The primary underpinning of security of a system is the set of mechanisms that ensures that these questions are answered satisfactorily for every action that the system performs. This idea is known as the principle of complete mediation.

Complete mediation:

For every requested action, check authenticity, integrity, and authorization.

To protect against inside attacks (adversaries who are actually users that have the appropriate permissions, but abuse them) or adversaries who successfully break the security mechanisms, the service must also maintain audit trails of who used the system, what authorization decisions have been made, etc. This information may help determine who the adversary was after the attack, how the adversary breached the security of the system, and bring the adversary to justice. In the end, a primary instrument to deter adversaries is to increase the likelihood of detection and punishment.

The next section provides a general introduction to security. It discusses possible threats (Section 5.2.1), why security is a negative goal (Section 5.2.2), presents the safety net approach (Section 5.2.3), lays out principles for designing secure computer systems (Section 5.2.4), the basic model for structuring secure computer systems (Section 5.2.6), an implementation strategy based on minimizing the trusted computing base (Section 5.2.7), and concludes with a road map for the rest of this chapter (Section 5.2.8). The rest of the chapter works the ideas introduced in Section 5.2 in more detail, but by no means provides a complete treatment of computer security. Computer security is an active area of research with many open problems and the interested reader is encouraged to explore the research literature to get deeper into the topic.

5.1: Overview is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

5.2: Introduction to Secure Systems

We have discussed how to divide a computer system into modules so that errors don't propagate from one module to another, but under the assumption that errors happen *unintentionally*: modules fail to adhere to their contracts because users make mistakes or hardware fails accidentally. As computer systems become more and more deployed for mission-critical applications, however, we require computer systems that can tolerate adversaries. By an **adversary**, we mean a entity that breaks into systems intentionally: for example, to steal information from other users, to blackmail a company, to deny other users access to services, to hack systems for fun or fame, to test the security of a system, etc. An adversary encompasses a wide range of bad guys as well as good guys (e.g., people hired by an organization to test the security of that organization's computers systems). An adversary can be a single person or a group collaborating to break the protection.

Almost all computers are connected to networks, which means that they can be attacked by an adversary from any place in the world. Not only must the security mechanism withstand adversaries who have physical access to the system, but the mechanism also must withstand a 16-year old wizard sitting behind a personal computer in some country one has never heard of. Since most computers are connected through *public* networks (e.g., the Internet), defending against a remote adversary is particularly challenging. Any person who has access to the public network might be able to compromise any computer or router in the network.

Although, in most secure systems, keeping adversaries from doing bad things is the primary objective, there is usually also a need to provide users with different levels of authority. Consider electronic banking. Certainly, a primary objective must be to ensure that no one can steal money from accounts, modify transactions performed over the public networks, or do anything else bad. But in addition, a banking system must enforce other security constraints. For example, the owner of an account should be allowed to withdraw money from the account, but the owner shouldn't be allowed to withdraw money from other accounts. Bank personnel, though, (under some conditions) should be allowed to transfer money between accounts of different users and view any account. Some scheme is needed to enforce the desired authority structure.

In some applications no enforcement mechanism internal to the computer system may be necessary. For instance, an externally administered code of ethics or other mechanisms outside of the computer system may protect the system adequately. On the other hand, with the rising importance of computers and the Internet many systems require some security plan. Examples include file services storing private information, Internet stores, law enforcement information systems, electronic distribution of proprietary software, on-line medical information systems, and government social service data processing systems. These examples span a wide range of needs for organizational and personal privacy.

Not all fields of study use the terms "privacy," "security," and "protection" in the same way. This chapter adopts definitions that are commonly encountered in the computer science literature. The traditional meaning of the term privacy is the ability of an individual to determine if, when, and to whom personal information is to be released (see Sidebar 5.2.1). The term security describes techniques that protect information and information systems against unauthorized access or modification of information, whether in storage, processing, or transit, and against denial of service to authorized users. In this chapter the term protection is used as a synonym for security.

Sidebar 5.2.1

Privacy

The definition of privacy (the ability of an individual to determine if, when, and to whom personal information is to be released) comes from the 1967 book *Privacy and Freedom* by Alan Westin [Suggestions for Further Reading 5.1.1]. Some privacy advocates (see, for example, Suggestions for Further Reading 5.1.3) suggest that with the increased interconnectivity provided by changing technology, Westin's definition now covers only a subset of privacy, and is in need of update. They suggest this broader definition: the ability of an individual to decide how and to what extent personal information can be used by others.

This broader definition includes the original concept, but it also encompasses control over use of information that the individual has agreed to release, but that later can be systematically accumulated from various sources such as public records, grocery store frequent shopper cards, Web browsing logs, online bookseller records about what books that person seems interested in, etc. The reasoning is that modern network and data mining technology add a new dimension to the activities that can constitute an invasion of privacy. The traditional definition implied that privacy can be protected by

> confidentiality and access control mechanisms; the broader definition implies adding accountability for use of information that the individual has agreed to release.

A common goal in a secure system is to enforce some privacy policy. An example of a policy in the banking system is that only an owner and selected bank personnel should have access to that owner's account. The nature of a privacy policy is not a technical question, but a social and political question. To make progress without having to solve the problem of what an acceptable policy is, we focus on the mechanisms to enforce policies. In particular, we are interested in mechanisms that can support a wide variety of policies. Thus, the principle *separate mechanism from policy* is especially important in design of secure systems.

5.2.1: Threat Classification

The design of any security system starts with identifying the threats that the system should withstand. **Threats** are potential security violations caused either by a planned attack by an adversary or unintended mistakes by legitimate users of the system. The designer of a secure computer system must be consider both.

There are three broad categories of threats:

1. Unauthorized information release: an unauthorized person can read and take advantage of information stored in the computer or being transmitted over networks. This category of concern sometimes extends to "traffic analysis," in which the adversary observes only the patterns of information use and from those patterns can infer some information content.
2. Unauthorized information modification: an unauthorized person can make changes in stored information or modify messages that cross a network—an adversary might engage in this behavior to sabotage the system or to trick the receiver of a message to divulge useful information or take unintended action. This kind of violation does not necessarily require that the adversary be able to see the information it has changed.
3. Unauthorized denial of use: an adversary can prevent an authorized user from reading or modifying information, even though the adversary may not be able to read or modify the information. Causing a system "crash," flooding a service with messages, or firing a bullet into a computer are examples of denial of use. This attack is another form of sabotage.

In general, the term "unauthorized" means that release, modification, or denial of use occurs contrary to the intent of the person who controls the information, possibly even contrary to the constraints supposedly enforced by the system.

As mentioned in the overview, a complication in defending against these threats is that the adversary can exploit the behavior of users who are legitimately authorized to use the system but are lax about security. For example, many users aren't security experts and put their computers at risk through surfing the Internet and downloading untrusted, third-party programs voluntarily or even without realizing it. Some users bring their own personal devices and gadgets into their work place; these devices may contain malicious software. Yet other users allow friends and family members to use computers at institutions for personal ends (e.g., storing personal content or playing games). Some employees may be disgruntled with their company and may be willing to collaborate with an adversary.

A legitimate user acting as an adversary is difficult to defend against because the adversary's actions will appear to be legitimate. Because of this difficulty, this threat has its own label, the **insider threat**.

Because there are many possible threats, a broad set of security techniques exists. The following list just provides a few examples (see Anderson's *Security Engineering: A Guide to Building Dependable Distributed Systems*[*] for a wider range of many more examples):

- making credit card information sent over the Internet unreadable by anyone other than the intended recipients,
- verifying the claimed identity of a user, whether local or across a network,
- labeling files with lists of authorized users,
- executing secure protocols for electronic voting or auctions,
- installing a router (in security jargon called a firewall) that filters traffic between a private network and a public network to make it more difficult for outsiders to attack the private network,
- shielding the computer to prevent interception and subsequent interpretation of electromagnetic radiation,
- locking the room containing the computer,
- certifying that the hardware and software are actually implemented as intended,

- providing users with configuration profiles to simplify configuration decisions with secure defaults,
- encouraging legitimate users to follow good security practices,
- monitoring the computer system, keeping logs to provide audit trails, and protecting the logs from tampering.

---

* Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, second edition, 2008. ISBN 978–0–470–06852–6. 1,040 pages.

> This book considers a remarkable range of system security problems, from taxi mileage recorders to nuclear command and control systems. It provides great depth on the mechanics, assuming that the reader already has a high-level picture. The book is sometimes quick in its explanations; the reader must be quite knowledgeable about systems. One of its strengths is that most of the discussions of how to do it are immediately followed by a section titled "What goes wrong", exploring misimplementations, fallacies, and other modes of failure. The first edition is available online.

5.2.2: Security is a Negative Goal

Having a narrow view of security is dangerous because the objective of a secure system is to prevent *all* unauthorized actions. This requirement is a negative kind of requirement. It is hard to prove that this negative requirement has been achieved, for one must demonstrate that *every possible* threat has been anticipated. Therefore, a designer must take a broad view of security and consider any method in which the security scheme can be penetrated or circumvented.

To illustrate the difficulty, consider the positive goal, "Alice can read file x." It is easy to test if a designer has achieved the goal (we ask Alice to try to read the file). Furthermore, if the designer failed, Alice will probably provide direct feedback by sending the designer a message "I can't read x!" In contrast, with a negative goal, such as "Lucifer cannot read file x," the designer must check that all the ways that the adversary Lucifer might be able to read x are blocked, and it's likely that the designer won't receive any direct feedback if the designer slips up. Lucifer won't tell the designer because Lucifer has no reason to and it may not even be in Lucifer's interest.

An example from the field of biology illustrates nicely the difference between proving a positive and proving a negative. Consider the question "Is a species (for example, the Ivory-Billed Woodpecker) extinct?" It is generally easy to prove that a species exists; just exhibit a live example. But to prove that it is extinct requires exhaustively searching the whole world. Since the latter is usually difficult, the most usual answer to proving a negative is "we aren't sure".*

The question "Is a system secure?" has these same three possible outcomes: insecure, secure, or don't know. In order to prove a system is insecure, one must find just one example of a security hole. Finding the hole is usually difficult and typically requires substantial expertise, but once one hole is found it is clear that the system is insecure. In contrast, to prove that a system is secure, one has to show that there is *no* security hole *at all*. Because the latter is so difficult, the typical outcome is "we don't know of any remaining security holes, but we are certain that there are some."

Another way of appreciating the difficulty of achieving a negative goal is to model a computer system as a state machine with states for all the possible configurations in which the system can be and with links between states for transitions between configurations. As shown in Figure 5.2.1, the possible states and links form a graph, with the states as nodes and possible transitions as edges. Assume that the system is in some current state. The goal of an adversary is to force the system from the current state to a state, labeled "Bad" in the figure, that gives the adversary unauthorized access. To defend against the adversary, the security designers must identify and block *every* path that leads to the bad state. But the adversary needs to find only *one* path from the current state to the bad state.

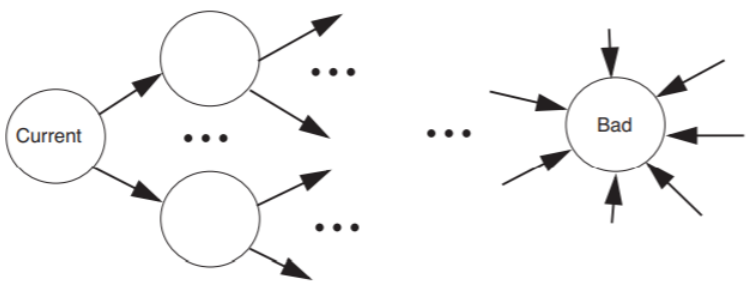


Figure 5.2.1: Modeling a computer systems as a state machine. An adversary's goal is to get the system into a state, labeled "Bad", that gives the adversary unauthorized access.To prevent the adversary from succeeding, *all* paths leading to the bad state must be

blocked off because the adversary needs to find only *one* path to succeed.

---

* The woodpecker was believed to be extinct, but in 2005 a few scientists claimed to have found the bird in Arkansas after a kayaker caught a glimpse in 2004; if true, it is the first confirmed sighting in 60 years.

5.2.3: The Safety Net Approach

To successfully design systems that satisfy negative goals, this chapter adopts the safety net approach of Chapter 2, which in essence guides a designer to be paranoid—never assume the design is right. In the context of security, the two safety net principles *be explicit* and *design for iteration* reinforce this paranoid attitude:

1. Be explicit: Make all assumptions explicit so that they can be reviewed. It may require only *one* hole in the security of the system to penetrate it. The designer must therefore consider any threat that has security implications and make explicit the assumption on which the security design relies. Furthermore, make sure that all assumptions on which the security of the system is based are apparent at all times to all participants. For example, in the context of protocols, the meaning of each message should depend only on the content of the message itself, and should not be dependent on the context of the conversation. If the content of a message depends on its context, an adversary might be able to break the security of a protocol by tricking a receiver into interpreting the message in a different context.
2. Design for iteration: Assume you will make errors. Because the designer must assume that the design itself will contain flaws, the designer must be prepared to iterate the design. When a security hole is discovered, the designer must review the assumptions, if necessary adjust them, and repair the design. When a designer discovers an error in the system, the designer must reiterate the whole design and implementation process.

The safety net approach implies several requirements for the design of a secure system:

- Certify the security of the system. **Certification** involves verifying that the design matches the intended security policy, the implementation matches the design, and the running system matches the implementation, followed up by end-to-end tests by security specialists looking for errors that might compromise security. Certification provides a systematic approach to reviewing the security of a system against the assumptions. Ideally, certification is performed by independent reviewers, and, if possible, using formal tools. One way to make certification manageable is to identify those components that must be trusted to ensure security, minimize their number, and build a wall around them. Below, Section 5.1.8 discusses this idea, known as the trusted computing base, in more detail.

- Maintain audit trails of all authorization decisions. Since the designer must assume that legitimate users might abuse their permissions or an adversary may be masquerading as a legitimate user, the system should maintain an tamper-proof log (so that an adversary cannot erase records) of all authorization decisions made. If, despite all security mechanisms, an adversary (either from the inside or from the outside) succeeds in breaking the security of the system, the log might help in forensics. A forensics expert may be able to use the log to collect evidence that stands in court and help establish the identity of the adversary so that the adversary can be prosecuted after the fact. The log also can be used as a source of feedback that reveals an incorrect assumption, design, or implementation.

- Design the system for feedback. An adversary is unlikely to provide feedback when compromising the system, so it is up to the designer to create ways to obtain feedback. Obtaining feedback starts with stating the assumptions explicitly, so the designer can check the designed, implemented, and operational system against the assumptions when a flaw is identified. This method by itself doesn't identify security weaknesses, and thus the designer must actively look for potential problems. Methods include reviewing audit logs and running programs that alert system administrators about unexpected behavior, such as unusual network traffic (e.g., many requests to a machine that normally doesn't receive many requests), repeated login failures, etc. The designer should also create an environment in which staff and customers are not blamed for system compromises, but instead are rewarded for reporting them, so that they are encouraged to report problems instead of hiding them. Designing for feedback reduces the chance that security holes will slip by unnoticed. Anderson illustrates well through a number of real-world examples how important it is to design for feedback [Suggestions for Further Reading 5.5.5].

As part of the safety net approach, a designer must consider the environment in which the system runs. The designer must secure all communication links (e.g., dial-up modem lines that would otherwise bypass the firewall that filters traffic between a private

network and a public network), prepare for malfunctioning equipment, find and remove back doors that create security problems, provide configuration settings for users that are secure by default, and determine who is trustworthy enough to own a key to the room that protects the most secure part of the system. Moreover, the designer must protect against bribes and worry about disgruntled employees. The security literature is filled with stories of failures because the designers didn't take one of these issues into account.

As another part of the safety net approach, the designer must consider the **dynamics of use**. This term refers to how one establishes and changes the specification of who may obtain access to what. For example, Alice might revoke Bob's permission to read file x. To gain some insight into the complexity introduced by changes to access authorization, consider again the question, "Is there any way that Lucifer could obtain access to file x?" One should check not only whether Lucifer has access to file x, but also whether Lucifer may change the specification of file x's accessibility. The next step is to see if Lucifer can change the specification of who may change the specification of file x's accessibility, etc.

Another problem of dynamics arises when the owner revokes a user's access to a file while that file is being used. Letting the previously authorized user continue until the user is "finished" with the information may be unacceptable if the owner has suddenly realized that the file contains sensitive data. On the other hand, immediate withdrawal of authorization may severely disrupt the user or leave inconsistent data if the user was in the middle of an atomic action. Provisions for the dynamics of use are at least as important as those for static specification of security.

Finally, the safety net approach suggests that a designer should never believe that a system is completely secure. Instead, one must design systems that **defend in depth** by using redundant defenses, a strategy that the Russian army deployed successfully for centuries to defend Russia. For example, a designer might have designed a system that provides end-to-end security over untrusted networks. In addition, the designer might also include a firewall between the trusted and untrusted network for network-level security. The firewall is in principle completely redundant with the end-to-end security mechanisms; if the end-to-end security mechanism works correctly, there is no need for network-level security. For an adversary to break the security of the system, however, the adversary has to find flaws in both the firewall *and* in the end-to-end security mechanisms, and be lucky enough that the first flaw allows exploitation of the second.

The defense-in-depth design strategy offers no guarantees, but it seems to be effective in practice. The reason is that conceptually, the defense-in-depth strategy cuts more edges in the graph of all possible paths from a current state to some undesired state. As a result, an adversary has fewer paths available to get to and exploit the undesired state.

5.2.4: Design Principles

In practice, because security is a negative goal, producing a system that actually does prevent all unauthorized acts has proved to be extremely difficult. Penetration exercises involving many different systems all have shown that users can obtain unauthorized access to these systems. Even if designers follow the safety net approach carefully, design and implementation flaws provide paths that circumvent the intended access constraints. In addition, because computer systems change rapidly or are deployed in new environments for which they were not designed originally, new opportunities for security compromises come about. Section 5.12 provides several war stories about security breaches.

Design and construction techniques that systematically exclude flaws are the topic of much research activity, but no complete method applicable to the design of computer systems exists yet. This difficulty is related to the negative quality of the requirement to prevent all unauthorized actions. In the absence of such methodical techniques, experience has provided several security principles to guide the design towards minimizing the number of security flaws in an implementation. We discuss these principles next.

The design should not be secret:

Open design principle:

Let anyone comment on the design. You need all the help you can get.

Violation of the open design principle has historically proven to almost always lead to flawed designs. The mechanisms should not depend on the ignorance of potential adversaries, but rather on the possession of specific, more easily protected, secret keys or passwords. This decoupling of security mechanisms from security keys permits the mechanisms to be examined by many reviewers

without concern that the review may itself compromise the safeguards. In addition, any skeptical user must be able to review that the system is adequate for the user's purpose. Finally, it is simply not realistic to maintain secrecy of any system that receives wide distribution. However, the open design principle can conflict with other goals, which has led to numerous debates; Sidebar 5.2.2 summarizes some of the arguments.

Sidebar 5.2.2

Should designs and vulnerabilities be public?

The debate of closed versus open designs has been raging literally for ages, and is not unique to computer security. The advocates of closed designs argue that making designs public helps the adversaries, so why do it? The advocates of open designs argue that closed designs don't really provide security because in the long run it is impossible to keep a design secret. The practical result of attempted secrecy is usually that the bad guys know about the flaws but the good guys don't. Open design advocates disparage closed designs by describing them as "security through obscurity."

On the other hand, the open design principle can conflict with the desire to keep a design and its implementation proprietary for commercial or national security reasons. For example, software companies often do not want a competitor to review their software in fear that the competitor can easily learn or copy ideas. Many companies attempt to resolve this conflict by arranging reviews, but restricting who can participate in the reviews. This approach has the danger that not the right people are performing the reviews.

Closely related to the question whether designs should be public or not is the question whether vulnerabilities should be made public or not? Again, the debate about the right answer to this question has been raging for ages, and is perhaps best illustrated by the following quote from a 1853 book* about old-fashioned door locks:

> *A commercial, and in some respects a social doubt has been started within the last year or two, whether or not it is right to discuss so openly the security or insecurity of locks. Many well-meaning persons suppose that the discussion respecting the means for baffling the supposed safety of locks offers a premium for dishonesty, by showing others how to be dishonest. This is a fallacy. Rogues are very keen in their profession, and know already much more than we can teach them respecting their several kinds of roguery.*
>
> *Rogues knew a good deal about lock-picking long before locksmiths discussed it among themselves, as they have lately done. If a lock, let it have been made in whatever country, or by whatever maker, is not so inviolable as it has hitherto been deemed to be, surely it is to the interest of honest persons to know this fact, because the dishonest are tolerably certain to apply the knowledge practically; and the spread of the knowledge is necessary to give fair play to those who might suffer by ignorance.*
>
> *It cannot be too earnestly urged that an acquaintance with real facts will, in the end, be better for all parties.*

Computer security experts generally believe that one should publish vulnerabilities for the reasons stated by Hobbs and that users should know if the system they are using has a problem so they can decide whether or not they care. Companies, however, are typically reluctant to disclose vulnerabilities. For example, a bank has little incentive to advertise successful compromises because it may scare away customers.

To handle this tension, many governments have created laws and organizations that make vulnerabilities public. In California companies must inform their customers if an adversary might have succeeded in stealing customers' private information (e.g., a social security number). The U.S federal government has created the Computer Emergency Response Team (CERT) to document vulnerabilities in software systems and help with the response to these vulnerabilities (see

www.cert.org). When CERT learns about a new vulnerability, it first notifies the vendor, then it waits for some time for the vendor to develop a patch, and then goes public with the vulnerability and the patch.

* A.C Hobbs (Charles Tomlinson, ed.), *Locks and Safes: The Construction of Locks*. Virtue & Co., London, 1853 (revised 1868).

The right people must perform the review because spotting security holes is difficult. Even if the design and implementation are public, that is an insufficient condition for spotting security problems. For example, standard committees are usually open in principle but their openness sometimes has barriers that cause the proposed standard not to be reviewed by the right people. To participate in the design of the WiFi Wired Equivalent Privacy standard required committee members to pay a substantial fee, which apparently discouraged security researchers from participating. When the standard was finalized and security researchers began to examine the standard, they immediately found several problems, one of which is described in Section 5.5.2.

Since it is difficult to keep a secret:

Minimize secrets

Because they probably won't remain secret for long.

Following this principle has the following additional advantage. If the secret is compromised, it must be replaced; if the secret is minimal, then replacing the secret is easier.

An open design that minimizes secrets doesn’t provide security itself. The primary underpinning of the security of a system is, as was mentioned in Section 5.1, the principle of *complete mediation*. This principle forces every access to be explicitly authenticated and authorized, including ones for initialization, recovery, shutdown, and maintenance. It implies that a foolproof method of verifying the authenticity of the origin and data of every request must be devised. This principle applies to a service mediating requests, as well as to a kernel mediating supervisor calls and a virtual memory manager mediating a read request for a byte in memory. This principle also implies that proposals for caching results of an authority check should be examined skeptically; if a change in authority occurs, cached results must be updated.

The human engineering *principle of least astonishment* applies especially to mediation. The mechanism for authorization should be transparent enough to a user that the user has a good intuitive understanding of how the security goals map to the provided security mechanism. It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the security mechanisms correctly. For example, a system should provide intuitive, default settings for security mechanisms so that only the appropriate operations are authorized. If a system administrator or user must first configure or jump through hoops to use a security mechanism, the user won't use it. Also, to the extent that the user's mental image of security goals matches the security mechanisms, mistakes will be minimized. If a user must translate intuitive security objectives into a radically different specification language, errors are inevitable. Ideally, security mechanisms should make a user's computer experience better instead of worse.

Another widely applicable principle, *adopt sweeping simplifications*, also applies to security. The fewer mechanisms that must be right to ensure protection, the more likely the design will be correct:

Economy of mechanism

The less there is, the more likely you will get it right.

Designing a secure system is difficult because every access path must be considered to ensure complete mediation, including ones that are not exercised during normal operation. As a result, techniques such as line-by-line inspection of software and physical examination of hardware implementing security mechanisms may be necessary. For such techniques to be successful, a small and simple design is essential.

Reducing the number of mechanisms necessary helps with verifying the security of a computer system. For the ones remaining, it would be ideal if only a few are common to more than one user and depended on by all users because every shared mechanism might provide unintended communication paths between users. Further, any mechanism serving all users must be certified to the satisfaction of every user, a job presumably harder than satisfying only one or a few users. These observations lead to the following security principle:

Minimize common mechanism

Shared mechanisms provide unwanted communication paths.

This principle helps reduce the number of unintended communication paths and reduces the amount of hardware and software on which all users depend, thus making it easier to verify if there are any undesirable security implications. For example, given the choice of implementing a new function as a kernel procedure shared by all users or as a library procedure that can be handled as though it were the user's own, choose the latter course. Then, if one or a few users are not satisfied with the level of certification of the function, they can provide a substitute or not use it at all. Either way, they can avoid being harmed by a mistake in it. This principle is an *end-to-end argument*.

Complete mediation requires that every request be checked for authorization and only authorized requests be approved. It is important that requests are not authorized accidently. The following security principle helps reduce such mistakes:

Fail-safe defaults

Most users won't change them, so make sure that defaults do something safe.

Access decisions should be based on permission rather than exclusion. This principle means that lack of access should be the default, and the security scheme lists conditions under which access is permitted. This approach exhibits a better failure mode than the alternative approach, where the default is to permit access. A design or implementation mistake in a mechanism that gives explicit permission tends to fail by refusing permission, a safe situation that can be quickly detected. On the other hand, a design or implementation mistake in a mechanism that explicitly excludes access tends to fail by allowing access, a failure that may long go unnoticed in normal use.

To ensure that complete mediation and fail-safe defaults work well in practice, it is important that programs and users have privileges only when necessary. For example, system programs or administrators who have special privileges should have those privileges only when necessary; when they are doing ordinary activities the privileges should be withdrawn. Leaving them in place just opens the door to accidents. These observations suggest the following security principle:

Least-privilege principle:

Don't store lunch in the safe with the jewels.

This principle limits the damage that can result from an accident or an error. Also, if fewer programs have special privileges, less code must be audited to verify the security of a system. The military security rule of "need-to-know" is an example of this principle.

Security experts sometimes use alternative formulations that combine aspects of several principles. For example, the formulation "minimize the attack surface" combines aspects of *economy of mechanism* (a narrow interface with a simple implementation provides fewer opportunities for designer mistakes and thus provides fewer attack possibilities), *minimize secrets* (few opportunities to crack secrets), *least privilege* (run most code with few privileges so that a successful attack does little harm), and *minimize common mechanism* (reduce the number of opportunities of unintended communication paths).

5.2.5: A High d(technology)/dt Poses Challenges for Security

Much software on the Internet and on personal computers fails to follow these principles, even though most of these principles were understood and articulated in the 1970s, before personal computers and the Internet came into existence. The reasons why they weren't followed are different for the Internet and personal computers, but they illustrate how difficult it is to achieve security when the rate of innovation is high.

When the Internet was first deployed, software implementations of the cryptographic techniques necessary to authenticate and protect messages (see Section 5.3 and Section 5.4) were considered but would have increased latency to unacceptable levels. Hardware implementations of cryptographic operations at that time were too expensive, and not exportable because the US government enforced rules to limit the use of cryptography. Since the Internet was originally used primarily by academics—a mostly cooperative community—the resulting lack of security was initially not a serious defect.

In 1994 the Internet was opened to commercial activities. Electronic stores came into existence, and many more computers storing valuable information came online. This development attracted many more adversaries. Suddenly, the designers of the Internet were forced to provide security. Because security was not part of the initial design plan, security mechanisms today have been designed as after-the-fact additions and have been provided in an *ad hoc* fashion instead of following an overall plan based on established security principles.

For different historical reasons, most personal computers came with little internal security and only limited stabs at network security. Yet today personal computers are almost always attached to networks where they are vulnerable. Originally, personal computers were designed as stand-alone devices to be used by a single person (that's why they are called *personal* computers). To keep the cost low, they had essentially no security mechanisms, but because they were used stand-alone, the situation was acceptable. With the arrival of the Internet, the desire to get online exposed their previously benign security problems. Furthermore, because of rapid improvements in technology, personal computers are now the primary platform for all kinds of computing, including most business-related computing. Because personal computers now store valuable information, are attached to networks, and have minimal protection, personal computers have become a prime target for adversaries.

The designers of the personal computer didn't originally foresee that network access would quickly become a universal requirement. When they later did respond to security concerns, the designers tried to add security mechanism quickly. Just getting the hardware mechanisms right, however, took multiple iterations, both because of blunders and because they were after-the-fact add-ons. Today, designers are still trying to figure out how to retrofit the existing personal-computer software and to configure the default settings right for improved security, while they are also being hit with requirements for improved security to handle denial-of-service attacks, phishing attacks*, viruses, worms, malware, and adversaries who try to take over machines without being noticed to create botnets (see Sidebar 5.2.3). As a consequence, there are many *ad hoc* mechanisms found in the field that don't follow the models or principles suggested in this chapter.

Sidebar 5.2.3

Malware: viruses, worms, Trojan horses, logic bombs, bots, etc.

There is a community of programmers that produces **malware**, software designed to run on a computer without the computer owner's intent. Some malware is created as a practical joke, other malware is designed to make money or to sabotage someone; Hafner and Markoff profile a few early high-profile cases of computer break-ins and the perpetrator's motivation*. More recently, there is an industry in creating malware that silently turns a user's computer into a **bot**, a computer controlled by an adversary, which is then used by the adversary to send unsolicited email (spam) on behalf of paying customers, which generates a revenue stream for the adversary [Suggestions for Further Reading 5.6.5].**

Malware uses a combinations of techniques to take control of a user's computer. These techniques include ways to install malware on a user's computer, ways to arrange that the malware will run on the user's computer, ways to replicate the malware on other computers, and ways to do perfidious things. Some of the techniques rely on users' naïvete while others rely on innovative ideas to exploit errors in the software running on the user's computer. As an example of both, in 2000 an adversary constructed the "ILOVEYOU" **virus**, an email message with a malicious executable attachment. The adversary sent the e-mail to a few recipients. When a recipient opened the executable email (attracted by "ILOVEYOU" in the email's subject), the malicious attachment read the recipient's address book and sent itself to the users in the address book. So many users opened the email that it spread rapidly and overwhelmed email servers at many institutions.

The Morris **worm** [Suggestions for Further Reading 5.6.1], created in 1984, is an example of malware that relies only on clever ways to exploit errors in software. The worm exploited various weaknesses in remote computers, among them a buffer overrun (see Sidebar 5.1.7.1) in an email server (sendmail) running on the UNIX operating system, which allowed it to install and run itself on the compromised computer. There it looked for network addresses of computers in configuration files, and then penetrated those computers, and so on. According to its creator it was not intended to create damage but a design error caused it to effectively create a denial-of-service attack. The worm spread so rapidly, infecting some computers multiple times, that it effectively shut down parts of the Internet.

The popular jargon attaches colorful labels to describe different types of malware such as virus, worm, Trojan horse, logic bomb, drive-by download, etc., and new ones appear as new types of malware show up. These labels don't correspond to precise, orthogonal technical concepts, but combine various malware features in different ways. All of them, however,

exploit some weakness in the security of a computer, and the techniques described in this chapter are also relevant in containing malware.

---

* Katie Hafner and John Markoff. *Cyberpunk: Outlaws and Hackers on the Computer Frontier*. Simon & Schuster (Touchstone), 1991, updated June 1995. ISBN 978–0–671–68322–1 (hardcover), 978–0–684–81862–7 (paperback). 368 pages.

This book is a readable, yet thorough, account of the scene at the ethical edges of cyberspace: the exploits of Kevin Mitnick, Hans Hubner, and Robert Tappan Morris. It serves as an example of a view from the media, but an unusually well-informed view.

** Problem Set 30 explores a potential stamp-based solution.

---

* Jargon term for an attack in which an adversary lures a victim to Web site controlled by the adversary; for an example see Suggestions for Further Reading 5.6.6.

5.2.6: Security Model

Although there are many ways to compromise the security of a system, the conceptual model to secure a system is surprisingly simple. To be secure, a system requires *complete mediation*: the system must mediate every action requested, including ones to configure and manage the system. The basic security plan then is that for each requested action the agent requesting the operation proves its identity to the system and then the system decides if the agent is allowed to perform that operation.

This simple model covers a wide range of instances of systems. For example, the agent may be a client in a client/service application, in which case the request is in the form of a message to a service. For another example, the agent may be a thread referring to virtual memory, in which case the request is in the form of a `LOAD` or `STORE` to a named memory cell. In each of these cases, the system must establish the identity of the agent and decide whether to perform the request or not. If all requests are mediated correctly, then the job of the adversary becomes much harder. The adversary must compromise the mediation system, launch an insider attack, or is limited to denial-of-service attacks.

The rest of this section works out the mediation model in more detail, and illustrates it with various examples. Of course a simple conceptual model cannot cover all attacks and all details. And, unfortunately, in security, the devil is often in the details of the implementation: does the system intended to be secure implement the model for all its operations and is the implementation correct? Nevertheless, the model is helpful in framing many security problems and then addressing them.

Agents perform on behalf of some entity that corresponds to a person outside the computer system; we call the representation of such an entity inside the computer system a **principal**. The principal is the unit of authorization in a computer system, and therefore also the unit of accountability and responsibility. Using these terms, mediating an action is asking the question, "Is the principal who requested the action authorized to perform the action?"

The basic approach to mediating every requested action is to ensure that there is really only *one* way to request an action. Conceptually, we want to build a wall around the system with one small opening through which all requested actions pass. Then, for every requested action, the system must answer "Should I perform the action?". To do so a system is typically decomposed in two parts: one part, called a **guard**, that specializes in deciding the answer to the question and a second part that performs the action. (In the literature, a guard that provides complete mediation is usually called a **reference monitor**.)

The guard can clarify the question, "Is the principal who originated the requested action allowed to perform the action?" by obtaining answers to the three subquestions of complete mediation (see Figure 5.2.1). The guard verifies that the message containing the request is authentic (i.e., the request hasn't been modified and that the principal is indeed the source of the request), and that the principal is permitted to perform the requested action on the object (authorization). If so, the guard allows the action; otherwise, it denies the request. The guard also logs all decisions for later audits.

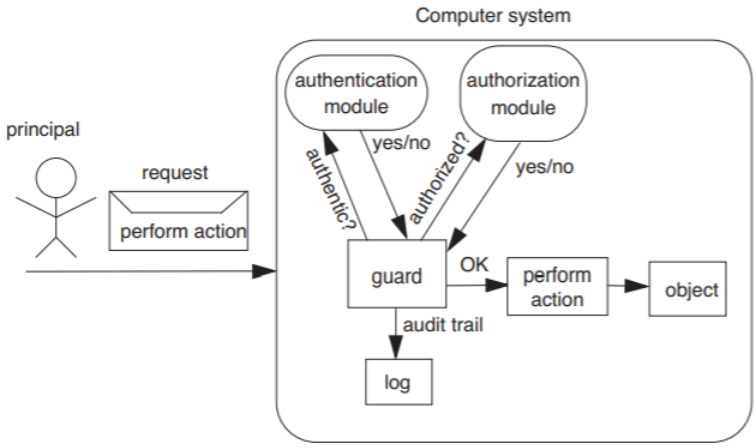


Figure 5.2.2: The security model based on complete mediation. The authenticity question includes both verifying the integrity and the source of the request.

The first two (has the request been modified and what is the source of the request) of the three mediation questions fall in the province of **authentication** of the request. Using an authentication service, the guard verifies the identity of the principal. Using additional information, sometimes part of the request but sometimes communicated separately, the guard verifies the integrity of the request. After answering the authenticity questions, the guard knows who the principal associated with the request is and that no adversary has modified the request.

The third and final question falls in the province of **authorization**. An authorization service allows principals to specify which objects they share with whom. Once the guard has securely established the identity of the principal associated with the request using the authentication service, the guard verifies with the authorization service that the principal has the appropriate authorization, and, if so, allows the requested service to perform the requested action.

The guard approach of complete mediation applies broadly to computer systems. Whether the messages are Web requests for an Internet store, `LOAD` and `STORE` operations to memory, or supervisor calls for the kernel, in all cases the same three questions must be answered by the Web service, virtual memory manager, or kernel, respectively. The implementation of the mechanisms for mediation, however, might be quite different for each case.

Consider an online newspaper. The newspaper service may restrict certain articles to paying subscribers and therefore must authenticate users and authorize requests, which often work as follows. The Web browser sends requests on behalf of an Internet user to the newspaper's Web server. The guard uses the principal's subscriber number and an authenticator (e.g., a password) included in the requests to authenticate the principal associated with the requests. If the principal is a legitimate subscriber and has authorization to read the requested article, the guard allows the request and the server replies with the article. Because the Internet is untrusted, the communications between the Web browser and the server must be protected; otherwise, an adversary can, for example, obtain the subscriber's password. Using **cryptography** one can create a **secure channel** that protects the communications over an untrusted network. Cryptography is a branch of computer science that designs primitives such as ciphers, pseudorandom number generators, and hashes, which can be used to protect messages against a wide range of attacks.

As another example, consider a virtual memory system with one domain per thread. In this case, the processor issues (`LOAD` and `STORE`) instructions on behalf of a thread to a virtual memory manager, which checks if the addresses in the instructions fall in the thread's domain. Conceptually, the processor sends a message across a bus, containing the operation (`LOAD` or `STORE`) and the requested address. This message is accompanied with a principal identifier naming the thread. If the bus is a trusted communication link, then the message doesn't have to be protected. If the bus isn't a secure channel (e.g., a digital rights management application may want to protect against an owner snooping on the bus to steal the copyrighted content), then the message between the processor and memory might be protected using cryptographic techniques. The virtual memory manager plays the role of a guard. It uses the thread identifier to verify if the address falls in the thread's domain and if the thread is authorized to perform the operation. If so, the guard allows the requested operation, and virtual memory manager replies by reading and writing the requested memory location.

Even if the mechanisms for complete mediation are implemented perfectly (i.e., there are no design and implementation errors in the cryptography, password checker, the virtual memory manager, the kernel, etc.), a system may still leave opportunities for an

adversary to break the security of the system. The adversary may be able to circumvent the guard, or launch an insider attack, or overload the system with requests for actions, thus delaying or even denying legitimate principals access. A designer must be prepared for these cases—an example of the paranoid design attitude. We discusses these cases in more detail.

To circumvent the guard, the adversary might create or find another opening in the system. A simple opening for an adversary might be a dial-up modem line that is not mediated. If the adversary finds the phone number (and perhaps the password to dial in), the adversary can gain control over the service. A more sophisticated way to create an opening is a **buffer overrun attack** on services written in the C programming language (see Sidebar 5.2.4), which causes the service to execute a program under the control of the adversary, which then creates an interface for the adversary that is not checked by the system.

Sidebar 5.2.4

Why are buffer overrun bugs so common?

It has become disappointingly common to hear a news report that a new Internet worm is rapidly spreading, and a little research on the World-Wide Web usually turns up as one detail that the worm exploits a **buffer overrun** bug. The reason that buffer overrun bugs are so common is that some widely used programming languages (in particular, C and C++) do not routinely check array bounds. When those languages are used, array bounds checking must be explicitly provided by the programmer. The reason that buffer overrun bugs are so easily exploited arises from an unintentional conspiracy of common system design and implementation practices that allow a buffer overrun to modify critical memory cells.

1. Compilers usually allocate space to store arrays as contiguous memory cells, with the first element at some starting address and successive elements at higher-numbered addresses.
2. Since there usually isn't any hardware support for doing anything different, most operating systems allocate a single, contiguous block of address space for a program and its data. The addresses may be either physical or virtual, but the important thing is that the programming environment is a single, contiguous block of memory addresses.
3. Faced with this single block of memory, programming support systems typically suballocate the address block into three regions: They place the program code in low-numbered addresses, they place static storage (the heap) just above those low-numbered addresses, and they start the stack at the highest-numbered address and grow it down, using lower addresses, toward the heap.

These three design practices, when combined with lack of automatic bounds checking, set the stage for exploitation. For example, historically it has been common for programs written in the C language to use library programs such as

```
GETS (character array reference string_buffer)
```

rather than a more elaborate version of the same program

```
FGETS (character array reference string_buffer, integer string_length, file stream)
```

to move character string data from an incoming stream to a local array, identified by the memory address of string_buffer. The important difference is that `GETS` reads characters until it encounters a new-line character or end of file, while `FGETS` adds an additional stop condition: it stops after reading `string_length` characters, thus providing an explicit array bound check. Using `GETS` rather than `FGETS` is an example of Gabriel's Worse is Better: "it is slightly better to be simple than to be correct." [Suggestions for Further Reading 1.5.1]

A program that is listening on some Internet port for incoming messages allocates a `string_buffer` of size 30 characters, to hold a field from the message, knowing that that field should never be larger. It copies data of the message from the port into `string_buffer` , using `GETS` . An adversary prepares and sends a message in which that field contains a string of, say, 250 characters. `GETS` overruns `string_buffer` .

Because of the compiler practice of placing successive array elements of `string_buffer` in higher-numbered addresses, if the program placed `string_buffer` in the stack the overrun overwrites cells in the stack that have higher-numbered addresses. But because the stack grows toward lower-numbered addresses, the cells overwritten by the buffer overrun are all *older* variables, allocated before `string_buffer` . Typically, an important older variable is the one that holds the return point of the currently running procedure, so the return point is vulnerable. A common exploit is thus to include runnable code in the 250-character string and, knowing stack offsets, smash the return point stack variable to

contain the address of that code. Then, when the thread returns from the current procedure, it unwittingly transfers control to the adversary's code.

By now, many such simple vulnerabilities have been discovered and fixed. But exploiting buffer overruns is not limited to smashing return points in the stack. Any writable variable that contains a jump address and that is located adjacent to a buffer in the stack or the heap may be vulnerable to an overrun of that buffer. The next time that the running thread uses that jump address, the adversary gains control of that thread. The adversary may not even have to supply executable code if he or she can cause the jump to go to some existing code such as a library routine that, with a suitable argument value, can be made to do something bad [Suggestions for Further Reading 11.6.2]. Such attacks require detailed knowledge of the layout and code generation methods used by the compiler on the system being attacked, but adversaries can readily discover that information by examining their own systems at leisure. Problem set 49 explores some of these attacks.

From that discussion one can draw several lessons that invoke security design principles:

1. The root cause of buffer overruns is the use of programming languages that do not provide the *fail-safe default* of automatically checking all array references to verify that they do not exceed the space allocated for the array.

2. *Be explicit.* One can interpret the problem with `GETS` to be that it relies on its context, rather than the program, to tell it exactly what to do. When the context contains contradictions (a string of one size, a buffer of another size) or ambiguities, the library routine may resolve them in an unexpected way. There is a trade-off between convenience and explicitness in programming languages. When security is the goal, a programming language that requires that the programmer be explicit is probably safer.

3. Hardware architecture features can help minimize the impact of common programming errors, and thus make it harder for an adversary to exploit them. Consider, for example, an architecture that provides distinct, hardware-enforced memory segments, using one segment for program code, a second segment for the heap, and a third segment for the stack. Since different segments can have different read, write, and execute permissions, the stack and heap segments might disallow executable instructions, while the program area disallows writing. The *principle of least privilege* suggests that no region of memory should be simultaneously writable and executable. If all buffers are in segments that are not executable, an adversary would find it more difficult to deposit code in the execution environment. Instead, the adversary may have to resort to methods that exploit code already in that execution environment. Even better might be to place each buffer in a separate segment, thus using the hardware to check array bounds.

   Hardware for Multics [Suggestions for Further Reading 3.1.4 and 5.4.1], a system implemented in the 1960s, provided segments. The Multics kernel followed the principle of least privilege in setting up permissions, and the observed result was that addressing errors were virtually always caught by the hardware at the instant they occurred, rather than leading to a later system meltdown. Designers of currently common hardware platforms have recently modified the memory management unit of these platforms to provide similar features, and today's popular operating systems are using the features to provide better protection.

4. Storing a jump address in the midst of writable data is hazardous because it is hard to protect it against either programming errors or intentional attacks. If an adversary can control the value of a jump address, there is likely to be some way that the adversary can exploit it to gain control of the thread. *Complete mediation* suggests that all such jump values should be validated before being used. Designers have devised schemes to try to provide at least partial validation. An example of such a scheme is to store an unpredictable nonce value (a "canary") adjacent to the memory cell that holds the jump address and, before using the jump address, verify that the canary is intact by comparing it with a copy stored elsewhere. Many similar schemes have been devised, but it is hard to devise one that is foolproof. For the most part these schemes do not prevent exploits, they just make the adversary's job harder.

As examples of insider attacks, the adversary may be able to guess a principal's password, may be able to bribe a principal to act on the adversary's behalf, or may be able to trick the principal to run the adversary's program on the principal's computer with the principal's privileges (e.g., the principal opens an executable email attachment sent by the adversary). Or, the adversary may be a legitimate principal who is disgruntled.

Measures against badly behaving principals are also the final line of defense against adversaries who successfully break the security of the system, thus appearing to be legitimate users. The measures include (1) running every requested operation with the least privilege because that minimizes damage that a legitimate principal can do, (2) maintaining an audit trail, of the mediation decisions made for every operation, (3) making copies and archiving data in secure places, and (4) periodically manually reviewing which principals should continue to have access and with what privileges. Of course, the archived data and the audit trail must be maintained securely; an adversary must not be able to modify the archived data or the audit trail. Measures to secure archives and audit trails include designing them to be write once and append-only.

An adversary's goal may be just to deny service to other users. To achieve this goal an adversary could flood a communication link with requests that take enough time of the service that it is unavailable for other users. The challenge in handling a denial-of-service attack is that the messages sent by the adversary may be legitimate requests and the adversary may use many computers to send these legitimate requests (see Suggestions for Further Reading 5.6.4 for an example). There is no single technique that can address denial-of-service attacks. Solutions typically involve several ideas: audit messages to be able to detect and filter bad traffic before it reaches the service, careful design of services to control the resources dedicated to a request and to push work back to the clients, and replicating services (see Section 4.4) to keep the service available during an attack. By replicating the service, an adversary must flood multiple replicas to make the service unavailable. This attack may require so many messages that with careful analysis of audit trails it becomes possible to track down the adversary.

5.2.7: Trusted Computing Base

Implementing the security model of Section 5.2.6 is a negative goal, and therefore difficult. There are no methods to verify correctness of an implementation that is claimed to achieve a negative goal. So, how do we proceed? The basic idea is to minimize the number of mechanisms that need to be correct in order for the system to be secure—the *economy of mechanism principle*, and to follow the safety net approach (*be explicit* and *design for iteration*).

When designing a secure system, we organize the system into two kinds of modules: **untrusted** modules and **trusted** modules. The correctness of the untrusted modules does not affect the security of the whole system. The trusted modules are the part that must work correctly to make the system secure. Ideally, we want the trusted modules to be usable by other untrusted modules, so that the designer of a new module doesn't have to worry about getting the trusted modules right. The collection of trusted modules is usually called the **trusted computing base** (TCB).

Establishing whether or not a module is part of the TCB can be difficult. Looking at an individual module, there isn't any simple procedure to decide whether or not the system's security depends on the correct operation of that module. For example, in UNIX if a module runs on behalf of the superuser principal (see Section 5.7.3), it is likely to be part of the TCB because if the adversary compromises the module, the adversary has full privileges. If the same module runs on behalf of a regular principal, it is often not part of the trusted computing base because it cannot perform privileged operations. But even then the module could be part of the TCB; it may be part of a user-level service (e.g., a Web service) that makes decisions about which clients have access. An error in the module's code may allow an adversary to obtain unauthorized access.

Lacking a systematic decision procedure for deciding if a module is in the TCB, the decision is difficult to make and easy to get wrong, yet a good division is important. A bad division between trusted and untrusted modules may result in a large and complex TCB, making it difficult to reason about the security of the system. If the TCB is large, it also means that ordinary users can make only few changes because ordinary users should only change modules outside the TCB that don't impact security. If ordinary users can change the system in only limited ways, it may make it difficult for them to get their job done in an effective way and result in bad user experiences. A large TCB also means that much of the system can be modified by only trusted principals, limiting the rate at which the system can evolve. The design principles of Section 5.2.4 can guide this part of the design process, but typically the division must be worked out by security experts.

Once the split has been worked out, the challenge becomes one of designing and implementing a TCB. To be successful at this challenge, we want to work in a way that maximizes the chance that the design and implementation of the TCB are correct. To do so, we want to minimize the chance of errors and maximize the rate of discovery of errors. To achieve the first goal, we should minimize the size of the TCB. To achieve the second goal, the design process should include feedback so that we will find errors quickly.

The following method shows how to build such a TCB:

- Specify security requirements for the TCB (e.g., secure communication over untrusted networks). The main reason for this step is to *explicitly* specify assumptions so that we can decide if the assumptions are credible. As part of the requirements, one also specifies the attacks against which the TCB is protected so that the security risks are assessable. By specifying what the TCB does and does not do, we know against which kinds of attacks we are protected and to which kinds we are vulnerable.
- Design a minimal TCB. Use good tools (such as authentication logic, which we will discuss in Section 5.6) to express the design.
- Implement the TCB. It is again important to use good tools. For example, buffer-overrun attacks can be avoided by using a language that checks array bounds.
- Run the TCB and try to break the security.

The hard part in this multistep design method is verifying that the steps are consistent: verifying that the design meets the specification, verifying that the design is resistant to the specified attacks, verifying that the implementation matches the design, and verifying that the system running in the computer is the one that was actually implemented. For example, as Thompson has demonstrated, it is easy for an adversary with compiler expertise to insert a Trojan Horses into a system that is difficult to detect [Suggestions for Further Reading 5.3.3 and 5.3.4].

The problem in computer security is typically *not* one of inventing clever mechanisms and architectures, but rather one of ensuring that the installed system actually meets the design requirements in implementation. Performing such an end-to-end check is difficult. For example, it is common to hire a **tiger team** whose mission is to find loopholes that could be exploited to break the security of the system. The tiger team may be able to find some loopholes, but, unfortunately, cannot provide a guarantee that all loopholes have been found.

The method also clearly states what risks were considered acceptable when the system was designed, because the prospective user must be able to look at the specification to evaluate whether the system meets the requirements. Stating what risks are acceptable is important because much of the design of secure systems is driven by economic constraints. Users may consider a security risk acceptable if the cost of a security failure is small compared to designing a system that negates the risk.

5.2.8: The Road Map for this Chapter

The rest of this chapter follows the security model of Figure 5.2.2. Section 5.3 presents techniques for authenticating principals. Section 5.4 explains how to authenticate messages by using a pair of procedures named `SIGN` and `VERIFY`. Section 5.5 explains how to keep messages confidential using a pair of procedures named `ENCRYPT` and `DECRYPT`. Section 5.6 explains how to set up, for example, an authenticated and secure communication link using security protocols. Section 5.7 discusses different designs for an authorization service. Because authentication is the foundation of security, Section 5.8 discusses how to reason about authenticating principals systematically. The actual implementation of `SIGN`, `VERIFY`, `ENCRYPT`, and `DECRYPT` we outsource to theoreticians specialized in cryptography, but a brief summary of how to implement `SIGN`, `VERIFY`, `ENCRYPT`, and `DECRYPT` is provided in Section 5.9. The case study in Section 5.11 provides a complete example of the techniques discussed in this chapter by describing how authentication and authorization is done in the World-Wide Web. Finally, Section 5.12 concludes the chapter with war stories of security failures, despite the best intentions of the designers; these stories emphasize how difficult it is to achieve a negative goal.

This page titled 5.2: Introduction to Secure Systems is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

5.3: Authenticating Principals

Most security policies involve people. For example, a simple policy might say that only the owner of the file x should be able to read it. In this statement the owner corresponds to a human. To be able to support such a policy the file service must have a way of establishing a secure binding between a user of the service and the origin of a request. Establishing and verifying the binding are topics that fall in the province of authentication.

Returning to our security model, the setup for authentication can be presented pictorially as in Figure 5.3.1. A person (Alice) asks her client computer to send a message "Buy 100 shares of Generic Moneymaking, Inc." to her favorite electronic trading service. An adversary may be able to copy the message, delete it, modify it, or replace it. As explained in Section 5.2, when Alice's trading service receives this message, the guard must establish two important facts related to authenticity:

1. Who is this principal making the request? The guard must establish if the message indeed came from the principal that represents the real-world person "Alice". More generally, the guard must establish the origin of the message.
2. Is this request actually the one that Alice made? Or, for example, has an adversary modified the message? The guard must establish the integrity of the message.

This section provides the techniques to answer these two questions.

5.3.1: Separating Trust from Authenticating Principals

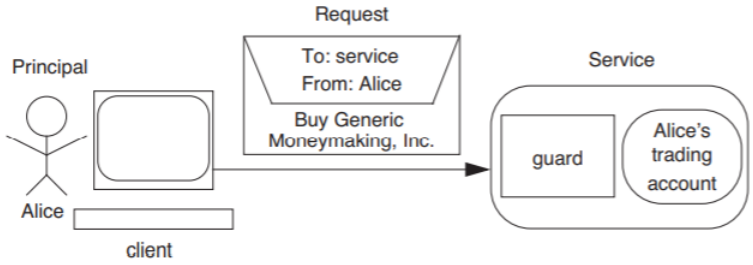


Figure 5.3.1: Authentication model.

Authentication consists of reliably identifying the principal associated with a request. Authentication can be provided by technical means such as passwords and signing messages. The technical means create a chain of evidence that securely connects an incoming request with a principal, perhaps by establishing that a message came from the same principal as a previous message. The technical means may even be able to establish the realworld identity of the principal.

Once the authentication mechanisms have identified the principal, there is a closely related but distinct problem: can the principal be trusted? The authentication means may be able to establish that the real-world identity for a principal is the person "Alice", but other techniques are required to decide whether and how much to trust Alice. The trading service may decide to consider Alice's request because the trading service can, by technical means, establish that Alice's credit card number is valid. To be more precise, the trading service trusts the credit card company to come through with the money and relies on the credit card company to establish the trust that Alice will pay her credit card bill.

The authenticity and trust problems are connected through the name of the principal. The technical means establish the name of the principal. Names for principals come in many flavors: for example, the name might be a symbolic one, like "Alice", a credit card number, a pseudonym, or a cryptographic key. The psychological techniques establish trust in the principal's name. For example, a reporter might trust information from an anonymous informer who has a pseudonym because previous content of the messages connected with the pseudonym has always been correct.

To make the separation of trust from authentication of principals more clear, consider the following example. You hear about an Internet bookstore named "ShopWithUs.com". Initially, you may not be sure what to think about this store. You look at their Web site, you talk to friends who have bought books from them, you hear a respectable person say publicly that this store is where the person buys books, and from all of this information you develop some trust that perhaps this bookstore is for real and is safe to order from. You order one book from ShopWithUs.com and the store delivers it faster than you expected. After a while you are ordering all your books from them because it saves the drive to the local bookstore and you have found that they take defective books back without a squabble.

Developing trust in ShopWithUs.com is the psychological part. The name ShopWithUs.com is the principal identifier that you have learned that you can trust. It is the name you heard from your friends, it is the name that you tell your Web browser, and it is the name that appears on your credit card bill. Your trust is based on that name; when you receive an e-mail offer from "ShopHere.com", you toss it in the trash because, although the name is similar, it does not precisely match the name.

When you actually buy a book at ShopWithUs.com, the authentication of principal comes into play. The mechanical techniques allow you to establish a secure communication link to a Web site that claims to be ShopWithUs.com, and verify that this Web site indeed has the name ShopWithUs.com. The mechanical techniques do not themselves tell you who you are dealing with; they just assure you that whoever it is, it is named ShopWithUs.com. You must decide yourself (the psychological component) who that is and how much to trust them.

In the reverse direction, ShopWithUs.com would like to assure itself that it will be paid for the books it sends. It does so by asking you for a principal identifier—your credit card number—and subcontracting to the credit card company the psychological component of developing trust that you will pay your credit card bills. The secure communication link between your browser and the Web site of ShopWithUs.com assures ShopWithUs.com that the credit card number you supply is securely associated with the transaction, and a similar secure communication link to the credit card company assures ShopWithUs.com that the credit card number is a valid principal identifier.

5.3.2: Authenticating Principals

When the trading service receives the message, the guard knows that the message claims to come from the person named "Alice", but it doesn't know whether or not the claim is true. The guard must verify the claim that the identifier Alice corresponds to the principal who sent the message.

Most authentication systems follow this model: the sender tells the guard its principal identity, and the guard verifies that claim. This verification protocol has two stages:

1. A rendezvous step, in which a real-world person physically visits an authority that configures the guard. The authority checks the identity of the real-world person, creates a principal identifier for the person, and agrees on a method by which the guard can later identify the principal identifier for the person. One must be particularly cautious in checking the real-world identity of a principal because an adversary may be able to fake it.
2. A verification of identity, which occurs at various later times. The sender presents a claimed principal identifier and the guard uses the agreed-upon method to verify the claimed principal identifier. If the guard is able to verify the claimed principal identifier, then the source is authenticated. If not, the guard disallows access and raises an alert.

The verification method the user and guard agree upon during the rendezvous step falls in three broad categories:

- The method uses a unique physical property of the user. For example, faces, voices, fingerprints, etc. are assumed to identify a human uniquely. For some of these properties it is possible to design a verification interface that is acceptable to users: for example, a user speaks a sentence into a microphone and the system compares the voice print with a previous voice print on file. For other properties it is difficult to design an acceptable user interface; for example, a computer system that asks "please give a blood sample" is not likely to sell well. The uniqueness of the physical property and whether it is easy to reproduce (e.g., replaying a recorded voice) determine the strength of this identification approach. Physical identification is sometimes a combination of a number of techniques (e.g., voice and face or iris recognition) and is combined with other methods of verification.
- The method uses something unique the user *has*. The user might have an ID card with an identifier written on a magnetic strip that can be read by a computer. Or, the card might contain a small computer that stores a secret; such cards are called smart cards. The security of this method depends on (1) users not giving their card to someone else or losing it, and (2) an adversary being unable to reproduce a card that contains the secret (e.g., copying the content of the magnetic strip). These constraints are difficult to enforce, since an adversary might bribe the user or physically threaten the user to give the adversary the user's card. It is also difficult to make tamper-proof devices that will not reveal their secret.
- The method uses something that only the user knows. The user remembers a secret string, for example, a password, a personal identification number (PIN) or, as will be introduced in Section 5.4, a cryptographic key. The strength of this method depends on (1) the user not giving away (voluntarily or involuntarily) the password and (2) how difficult it is for an adversary to guess the user's secret. Your mother's maiden name and 4-digit PINs are **weak** secrets.

For example, when Alice created a trading account, the guard might have asked her for a principal identifier and a **password** (a secret character string), which the guard stores. This step is the rendezvous step. Later when Alice sends a message to trade, she includes in the message her claimed principal identifier ("Alice") and her password, which the guard verifies by comparing it with its stored copy. If the password in the message matches, the guard knows that this message came from the principal Alice, assuming that Alice didn't disclose her password to anyone else voluntarily or involuntarily. This step is the verification step.

In real-life authentication we typically use a similar process. For example, we first obtain a passport by presenting ourselves at the passport bureau, where we answer questions, provide evidence of our identity, and a photograph. This step is the rendezvous step. Later, we present the passport at a border station. The border guard examines the information in the passport (height, hair color, etc.) and looks carefully at the photograph. This step is the verification step.

The security of authenticating principals depends on, among other things, how carefully the rendezvous step is executed. As we saw above, a common process is that before a user is allowed to use a computer system, the user must see an administrator in person and prove to the administrator the user's identity. The administrator might ask the prospective user, for example, for a passport or a driver's license. In that case, the administrator relies on the agency that issued the passport or driver's license to do a good job in establishing the identity of the person.

In other applications the rendezvous step is a lightweight procedure and the guard cannot place much trust in the claimed identity of the principal. In the example with the trading service, Alice chooses her principal identifier and password. The service just stores the principal identifier and password in its table, but it has no direct way of verifying Alice's identity; Alice is unlikely to be able to see the system administrator of the trading service in person because she might be at a computer on the other side of the world. Since the trading service cannot verify Alice's identity, the service puts little trust in any claimed connection between the principal identifier and a real-world person. The account exists for the convenience of Alice to review, for example, her trades; when she actually buys something, the service doesn't verify Alice's identity, but instead verifies something else (e.g., Alice's credit card number). The service trusts the credit card company to verify the principal associated with the credit card number. Some credit card companies have weak verification schemes, which can be exploited by adversaries for identity theft.

5.3.3: Cryptographic Hash Functions, Computationally Secure, Window of Validity

The most commonly employed method for verifying identities in computer systems is based on passwords because it has a convenient user interface; users can just type in their name and password on a keyboard. However, there are several weaknesses in this approach. One weakness is that the stored copy of the password becomes an attractive target for adversaries. One way to remove this weakness is to store a cryptographic hash of the password in the password file of the system, rather than the password itself.

A c**ryptographic hash function** maps an arbitrary-sized array of bytes M to a fixed-length value V, and has the following properties:

1. For a given input M, it is easy to compute $V \leftarrow H(M)$, where H is the hash function;
2. It is difficult to compute M knowing only V;
3. It is difficult to find another input M' such that $H(M') = H(M)$;
4. The computed value V is as short as possible, but long enough that H has a low probability of collision: the probability of two different inputs hashing to the same value V must be so low that one can neglect it in practice. A typical size for V is 160 to 256 bits.

The challenge in designing a cryptographic hash function is finding a function that has all these properties. In particular, providing property 3 is challenging. Section 5.9 describes an implementation of the Secure Hash Algorithm (SHA), which is a U.S. government and OECD standard family of hash algorithms.

Cryptographic hash functions, like most cryptographic functions, are computationally secure. They are designed in such a way that it is computationally infeasible to break them, rather than being impossible to break. The idea is that if it takes an unimaginable number of years of computation to break a particular function, then we can consider the function secure.

Computationally security is measured quantified using a **work factor**. For cryptographic hash functions, the work factor is the minimum amount of work required to compute a message M' such that for a given M, $H(M') = H(M)$. Work is measured in primitive operations (e.g., processor cycles). If the work factor is many years, then for all practical purposes, the function is just as

secure as an unbreakable one because in both cases there is probably an easier attack approach based on exploiting human fallibility.

In practice, computationally security is measured by a **historical work factor**. The historical work factor is the work factor based on the current best-known algorithms and current state-of-the-art technology to break a cryptographic function. This method of evaluation runs the risk that an adversary might come up with a better algorithm to break a cryptographic function than the ones that are currently known, and furthermore technology changes may reduce the work factor. Given the complexities of designing and analyzing a cryptographic function, it is advisable to use only ones, such as SHA-256, that have been around long enough that they have been subjected to much careful, public review.

Theoreticians have developed models under which they can make absolute statements about the hardness of some cryptographic functions. Coming up with good models that match practice and the theoretical analysis of security primitives is an active area of research with a tremendous amount of progress in the last three decades, but also with many open problems.

Given that $d(technology)/dt$ is so high in computer systems and cryptography is a fast developing field, it is good practice to consider the window of validity for a specific cryptographic function. The window of validity of a cryptographic function is the minimum of the time-to-compromise of all of its components. The window of validity for cryptographic hash functions is the minimum of the time to compromise the hash algorithm and the time to find a message M' such that for a given M, $H(M') = H(M)$. The window of validity of a password-based authentication system is the minimum of the window of validity of the hashing algorithm, the time to try all possible passwords, and the time to compromise a password.

A challenge in system design is that the window of validity of a cryptographic function may be shorter than the lifetime of the system. For example, SHA, now referred to as "SHA-0" and which produces a 160-bit value for V, was first published in 1993 and was superseded just two years later by SHA-1 to repair a possible weakness. Indeed, in 2004, a cryptographic researcher found a way to systematically derive examples of messages M and M' that SHA-0 hashes to the same value. Research published in 2005 suggest weaknesses in SHA-1, but as of 2007 no one has yet found a systematic way to compromise that widely used hash algorithm (i.e., for a given M no one has yet found a M' that hashes to the same value of $H(M)$). As a precaution, however, the National Institute for Standards and Technology is recommending that by 2010 users switch to versions of SHA (for example, SHA-256) that produce longer values for V. A system designer should be prepared that during the lifetime of a computer system the cryptographic hash function may have to be replaced, perhaps more than once.

5.3.4: Using Cryptographic Hash Functions to Protect Passwords

There are many usages of cryptographic hash functions, and we will see them show up in this chapter frequently. One good use is to protect passwords. The advantage of storing the cryptographic hash of the password in the password file instead of the password itself is that the hash value does not need to be kept secret. For this purpose, the important property of the hash function is the second property in the list in Section 5.3.3, that if the adversary has only the output of a hash function (e.g., the adversary was able to steal the password file), it is difficult to compute a corresponding input. With this scheme, even the system administrator cannot figure out what the user's password is. (Design principle: *Minimize secrets.*)

The verification of identity happens when a user logs onto the computer. When the user types a password, the guard computes the cryptographic hash of the typed password and compares the result with the value stored in the table. If the values match, the verification of identity was successful; if the verification fails, the guard denies access.

The most common attack on this method is a brute-force attack, in which an adversary tries all possible passwords. A brute-force attack can take a long time, so adversaries often use a more sophisticated version of it: a **dictionary attack**, which works well for passwords because users prefer to select an easy-to-remember password. In a dictionary attack, an adversary compiles a list of likely passwords: first names*, last names, street names, city names, words from a dictionary, and short strings of random characters. Names of cartoon characters and rock bands have been shown to be effective guesses in universities.

The adversary either computes the cryptographic hash of these strings and compares the result to the value stored in the computer system (if the adversary has obtained the table), or writes a computer program that repeatedly attempts to log on with each of these strings. A variant of this attack is an attack on a specific person's password. Here the adversary mines all the information one can find (mother's maiden name, daughter's birth date, license plate number, etc.) about that person and tries passwords consisting of that information forwards and backwards. Another variant is of this attack is to try a likely password on each user of a popular

Internet site; if passwords are 20 bits (e.g., a 6 digit PIN), then trying a given PIN as a password for 10,000,000 accounts is likely to yield success for 10 accounts ($10 \times 2^{20} = 10{,}000{,}000$).

Several studies have shown that brute-force and dictionary attacks are effective in practice because passwords are often inherently weak. Users prefer easy-to-remember passwords, which are often short and contain existing words, and thus dictionary attacks work well. System designers have countered this problem in several ways. Some systems force the user to chose a strong password, and require the user to change it frequently. Some systems disable an account after 3 failed login attempts. Some systems require users to use both a password and a secret generated by the user's portable cryptographic device (e.g., an authentication device with a cryptographic coprocessor). In addition, system designers often try to make it difficult for adversaries to compile a list of all users on a service and limit access to the file with cryptographic hashes of passwords.

Since the verification of identity depends solely on the password, it is prudent to make sure that the password is never disclosed in insecure areas. For example, when a user logs on to a remote computer, the system should avoid sending the password unprotected over an untrusted network. That is easier said than done. For example, sending the cryptographic hash of the password is not good enough because if the adversary can capture the hash by eavesdropping, the adversary might be able to replay the hash in a later message and impersonate a principal or determine the secret using a dictionary attack.

In general, it is advisable to minimize repeated use of a secret because each exposure increases the chance that the adversary may discover the secret. To minimize exposure, any security scheme based on passwords should use them only once per session with a particular service: to verify the identity of a person at the first access. After the first access, one should use a newly-generated, strong secret for further accesses. More generally, what we need is a protocol between the user and the service that has the following properties:

1. it authenticates the principal to the guard;
2. it authenticates the service to the principal;
3. the password never travels over the network so that adversaries cannot learn the password by eavesdropping on network traffic;
4. the password is used only once per session so that the protocol exposes this secret as few times as possible. This has the additional advantage that the user must type the password only once per session.

The challenge in designing such a protocol is that the designer must assume that one or more of the parties involved in the protocol may be under the control of an adversary. An adversary should not be able to impersonate a principal, for example, by recording all network messages between the principal and the service, and replaying it later. To withstand such attacks we need a **security protocol**, a protocol designed to achieve some security objective. Before we can discuss such protocols, however, we need some other security mechanisms. For example, since any message in a security protocol might be forged by an adversary, we first need a method to check the authenticity of messages. We discuss message authentication next, the design of confidential communication links in Section 5.5, and the design of security protocols in Section 5.6. With these mechanisms one can design, among many other things, a secure password protocol.

---

* A classic study is by Frederick T. Grampp and Robert H. Morris. UNIX operating system security. *Bell System Technical Journal 63*, 8, Part 2 (October, 1984), pages 1649–1672. The authors made a list of 200 names by selecting 20 common female names and appending to each one a single digit (the system they tested required users to select a password containing at least 6 characters and one digit). At least one entry of this list was in use as a password on each of several dozen UNIX machines they examined.

This page titled 5.3: Authenticating Principals is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

5.4: Authenticating Messages

Returning to Figure 5.3.1, when receiving a message, the guard needs an ensured way of determining *what* the sender said in the message and *who* sent the message. Answering these two questions is the province of **message authentication**. Message authentication techniques prevent an adversary from forging messages that pretend to be from someone else, and allow the guard to determine if an adversary has modified a legitimate message while it was en route.

In practice, the ability to establish who sent a message is limited; all that the guard can establish is that the message came from the same origin as some previous message. For this reason, what the guard really does is to establish that a message is a member of a chain of messages identified with some principal. The chain may begin in a message that was communicated by a physical rendezvous. That physical rendezvous securely binds the identity of a real-world person with the name of a principal, and both the real-world person and that principal can now be identified as the origin of the current message. For some applications it is unimportant to establish the real-world person that is associated with the origin of the message. It may be sufficient to know that the message originated from the same source as earlier messages and that the message is unaltered. Once the guard has identified the principal (and perhaps the real-world identity associated with the principal), then we may be able to use psychological means to establish trust in the principal, as explained in Section 5.3.

To establish that a message belongs to a chain of messages, a guard must be able to verify the authenticity of the message. Message authenticity requires *both*:

- **data integrity**: the message has not been changed since it was sent;
- **origin authenticity**: the claimed origin of the message, as learned by the receiver from the message content or from other information, is the actual origin.

The issues of data integrity and origin authenticity are closely related. Messages that have been altered effectively have a new origin. If an origin cannot be determined, the very concept of message integrity becomes questionable (the message is unchanged with respect to what?). Thus, integrity of message data has to include message origin, and vice versa. The reason for distinguishing them is that designers using different techniques to tackle the two.

In the context of authentication, we mostly talk about authenticating messages. However, the concept also applies to communication streams, files, and other objects containing data. A stream is authenticated by authenticating successive segments of the stream. We can think of each segment as a message from the point of view of authentication.

5.4.1: Message Authentication is Different from Confidentiality

The goal of message confidentiality (keeping the content of messages private) and the goal of message authentication are related but different, and separate techniques are usually used for each objective, similar to the physical world. With paper mail, signatures authenticate the author and sealed envelopes protect the letter from being read by others.

Authentication and confidentiality can be combined in four ways, three of which have practical value:

- Authentication and confidentiality. An application (e.g., electronic banking) might require both authentication and confidentiality of messages. This case is like a signed letter in a sealed envelope, which is appropriate if the content of the message (e.g., it contains personal financial information) must be protected and the origin of the message must be established (e.g., the user who owns the bank account).
- Authentication only. An application, like DNS, might require just authentication for its announcements. This case is like a signed letter in an unsealed envelope. It is appropriate, for example, for a public announcement from the president of a company to its employees.
- Confidentiality only. Requiring confidentially without authentication is uncommon. The value of a confidential message with an unverified origin is not great. This case is like a letter in a sealed envelope, but without a signature. If the guard has no idea who sent the letter, what level of confidence can the guard have in the content of the letter? Moreover, if the receiver doesn't know who the sender is, the receiver has no basis to trust the sender to keep the content of the message confidential; for all the receiver knows, the sender may have released the content of the letter to someone else too. For these reasons confidentiality only is uncommon in practice.
- Neither authentication or confidentiality. This combination is appropriate if there are no intentionally malicious users or there is a separate code of ethics.

To illustrate the difference between authentication and confidentiality, consider a user who browses a Web service that publishes data about company stocks (e.g., the company name, the current trading price, recent news announcements about the company, and background information about the company). This information travels from the Web service over the Internet, an untrusted network, to the user's Web browser. We can think of this action as a message that is being sent from the Web service to the user's browser:

From: stock.com
To: John's browser
Body: At 10 a.m. Generic Moneymaking, Inc. was trading at $1

The user is not interested in confidentiality of the data; the stock data is public anyway. The user, however, is interested in the authenticity of the stock data, since the user might decide to trade a particular stock based on that data. The user wants to be assured that the data is coming from "stock.com" (and not from a site that is pretending to be stock.com) and that the data was not altered when it crossed the Internet. For example, the user wants to be assured that an adversary hasn't changed "Generic Moneymaking, Inc.", the price, or the time. We need a scheme that allows the user to verify the authenticity of the publicly readable content of the message. The next section introduces cryptography for this purpose. When cryptography is used, content that is publicly readable is known as **plaintext** or **cleartext**.

5.4.2: Closed versus Open Designs and Cryptography

In the authentication model there are two **secure areas** (a physical space or a virtual address space in which information can be safely confined) separated by an insecure communication path (as shown in Figure 5.4.1) and two boxes: `SIGN` and `VERIFY`. Our goal is to set up a secure channel between the two secure areas that provides authenticity for messages sent between the two secure areas. (Section 5.5 shows how one can implement a secure channel that also provides confidentiality.)

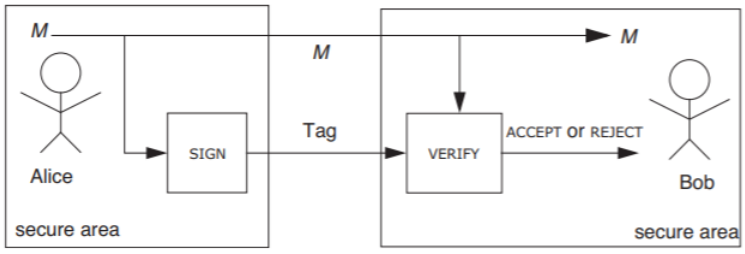


Figure 5.4.1: A closed design for authentication relies on the secrecy of an algorithm.

Before diving in the details of how to implement `SIGN` and `VERIFY`, let's consider how we might use them. In a secure area, the sender Alice creates an authentication tag for a message by invoking `SIGN` with the message as an argument. The tag and message are communicated through the insecure area to the receiver Bob. The insecure communication path might be a physical wire running down the street or a connection across the Internet. In both cases, we must assume that a wire-tapper can easily and surreptitiously gain access to the message and authentication tag. Bob verifies the authenticity of the message by a computation based on the tag and the message. If the received message is authentic, `VERIFY` returns `ACCEPT`; otherwise it returns `REJECT`.

Cryptographic transformations can be used protect against a wide range of attacks on messages, including ones on the authenticity of messages. Our interest in cryptographic transformations is not the underlying mathematics (which is fascinating by itself, as can been seen in Section 5.9), but that these transformations can be used to implement security primitives such as `SIGN` and `VERIFY`.

One approach to implementing a cryptographic system, called a closed design, is to keep the construction of cryptographic primitives, such as `VERIFY` and `SIGN`, secret through that idea that if the adversary doesn't understand how `SIGN` and `VERIFY` work, it will be difficult to break the tag. Auguste Kerchkoffs observed more than a century ago* that this closed approach is typically bad, since it violates the basic design principles for secure systems in a number of ways. It doesn't minimize what needs to be secret. If the design is compromised, the whole system needs to be replaced. A review to certify the design must be limited, since it requires revealing the secret design to the reviewers. Finally, it is unrealistic to attempt to maintain secrecy of any system that receives wide distribution.

These problems with closed designs led Kerchkoffs to propose a design rule, now known as Kerchkoffs' criterion, which is a particular application of the principles of open design and least privilege: minimize secrets. For a cryptographic system, open design means that we concentrate the secret in a corner of a cryptographic transformation, and make the secret removable and easily changeable. An effective way of doing this is to reduce the secret to a string of bits; this secret bit string is known as a cryptographic key, or key for short. By choosing a longer key, one can generally increase the time for the adversary to compromise the transformation.

Figure 5.4.2 shows an open design for SIGN and VERIFY. In this design the algorithms for SIGN and VERIFY are public and the only secrets are two keys, K_1 and K_2. What distinguishes this open design from a closed design is (1) that public analysis of SIGN and VERIFY can provide verification of their strength without compromising their security; and (2) it is easy to change the secret parts (i.e., the two keys) without having to reanalyze the system's strength.

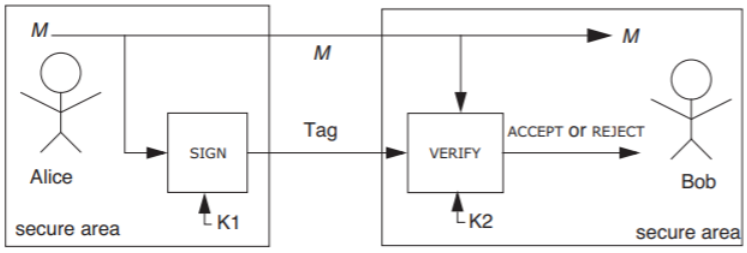


Figure 5.4.2: An open design for authentication relies on the secrecy of keys.

Depending on the relation between K_1 and K_2, there are two basic approaches to key-based transformations of a message: **shared-secret cryptography** and **public-key cryptography**. In shared-secret cryptography K_1 is easily computed from K_2 and vice versa. Usually in shared-secret cryptography $K_1 = K_2$, and we make that assumption in the text that follows.

In public-key cryptography K_1 cannot be derived easily from K_2 (and vice versa). In public-key cryptography, only one of the two keys must be kept secret; the other one can be made public. (A better label for public-key cryptography might be "cryptography without shared secrets", or even "non-secret encryption", which is the label adopted by the intelligence community. Either of those labels would better contrast it with shared-secret cryptography, but the label "public-key cryptography" has become too widely used to try to change it.)

Public-key cryptography allows Alice and Bob to perform cryptographic operations without having to share a secret. Before public-key systems were invented, cryptographers worked under the assumption that Alice and Bob needed to have a shared secret to create, for example, SIGN and VERIFY primitives. Because sharing a secret can be awkward and maintaining its secrecy can be problematic, this assumption made certain applications of cryptography complicated. Because public-key cryptography removes this assumption, it resulted in a change in the way cryptographers thought, and has led to interesting applications, as we will see in this chapter.

To distinguish the keys in shared-secret cryptography from the ones in public-key cryptography, we refer to the key in shared-secret cryptography as the **shared-secret key**. We refer to the key that can be made public in public-key cryptography as the **public key** and to the key that is kept secret in public-key cryptography as the **private key**. Since shared-secret keys must also be kept secret, the unqualified term "secret key," which is sometimes used in the literature, can be ambiguous, so we avoid using it.

We can now see more specifically the two ways in which SIGN and VERIFY can benefit if they are an open design. First, if K_1 or K_2 is compromised, we can select a new key for future communication, without having to replace SIGN and VERIFY. Second, we can now publish the overall design of the system and how SIGN and VERIFY work. Anyone can review the design and offer opinions about its correctness.

Because most cryptographic techniques use open design and reduce any secrets to keys, a system may have several keys that are used for different purposes. To keep the keys apart, we refer to the keys for authentication as **authentication keys**.

---

* "Il faut un systeme remplissant certaines conditions exceptionelles ... il faut qu'il n'exige pas le secret, et qu'il puisse sans inconvenient tomber entre les mains de l'ennemi." (Compromise of the system should not disadvantage the participants.) Auguste Kerchkoffs, *La cryptographie Militaire*, Chapter II (1883).

5.4.3: Key-Based Authentication Model

Returning to Figure 5.4.2, to authenticate a message, the sender **signs** the messages using a key K_1. Signing produces as output an **authentication tag**: a **key-based cryptographic transformation** (usually shorter than the message). We can write the operation of signing as follows:

$T \leftarrow \text{SIGN}\ (M, K_1)$

where T is the authentication tag.

The tag may be sent to the receiver separately from the message or it may be appended to the message. The message and tag may be stored in separate files or attachments. The details don't matter.

Let's assume that the sender sends a message $\{M, T\}$. The receiver receives a message $\{M', T'\}$, which may be the same as $\{M, T\}$ or may not. The purpose of message authentication is to decide which. The receiver unmarshals $\{M', T'\}$ into its components M' and T', and verifies the authenticity of the received message, by performing the computation:

$result \leftarrow \text{VERIFY}\ (M', T', K_2)$

This computation returns ACCEPT if M' and T' match; otherwise, it returns REJECT.

The design of SIGN and VERIFY should be such that if an adversary forges a tag, re-uses a tag from a previous message on a message fabricated by the adversary, etc. the adversary won't succeed. Of course, if the adversary replays a message $\{M, T\}$ without modifying it, then VERIFY will again return ACCEPT; we need a more elaborate security protocol, the topic of Section 5.6, to protect against replayed messages.

If M is a long message, a user might sign and verify the cryptographic hash of M, which is typically less expensive than signing M because the cryptographic hash is shorter than M. This approach complicates the protocol between sender and receiver a bit because the receiver must accurately match up M, its cryptographic hash, and its tag. Some implementations of SIGN and VERIFY implement this performance optimization themselves.

5.4.4: Properties of SIGN and VERIFY

To get a sense of the challenges of implementing SIGN and VERIFY, we outline some of the basic requirements for SIGN and VERIFY, and some attacks that a designer must consider. The sender sends $\{M, T\}$ and the receiver receives $\{M', T'\}$. The requirements for an authentication system with shared-secret key K are as follows:

1. $\text{VERIFY}\ (M', T', K)$ returns ACCEPT if $M' = M$, $T' = \text{SIGN}\ (M, K)$
2. Without knowing K, it is difficult for an adversary to compute an M' and T' such that $\text{VERIFY}\ (M', T', K)$ returns ACCEPT
3. Knowing M, T, and the algorithms for SIGN and VERIFY doesn't allow an adversary to compute K

In short, T should be dependent on the message content M and the key K. For an adversary who doesn't know key K, it should be impossible to construct a message M' and a T' different from M and T that verifies correctly using key K.

A corresponding set of properties must hold for public-key authentication systems:

1. $\text{VERIFY}\ (M', T', K_2)$ returns ACCEPT if $M' = M$, $T' = \text{SIGN}\ (M, K_1)$
2. Without knowing K_1, it is difficult for an adversary to compute an M' and T' such that $\text{VERIFY}\ (M', T', K_2)$ returns ACCEPT
3. Knowing M, T, K_2, and the algorithms for VERIFY and SIGN doesn't allow an adversary to compute K_1

The requirements for SIGN and VERIFY are formulated in absolute terms. Many good implementations of VERIFY and SIGN, however, don't meet these requirements perfectly. Instead, they might guarantee property 2 with very high probability. If the probability is high enough, then as a practical matter we can treat such an implementation as being acceptable. What we require is that the probability of *not* meeting property 2 be much lower than the likelihood of a human error that leads to a security breach.

The work factor involved in compromising SIGN and VERIFY is dependent on the key length; a common way to increase the work factor for the adversary is use a longer key. A typical key length used in the field for the popular RSA public-key cipher (see Section 5.9) is 1,024 or 2,048 bits. SIGN and VERIFY implemented with shared-secret ciphers often use shorter keys (in the range of 128 to 256 bits) because existing shared-secret ciphers have higher work factors than existing public-key ciphers. It is also

advisable to change keys periodically to limit the damage in case a key is compromised, and cryptographic protocols often do so (see Section 5.6).

Broadly speaking, the attacks on authentication systems fall in five categories:

1. Modifications to M and T. An adversary may attempt to change M and the corresponding T. The VERIFY function should return REJECT even if the adversary deletes or flips only a single bit in M and tries to make corresponding change to T. Returning to our trading example, VERIFY should return REJECT if the adversary changes M from "At 10 a.m. Generic Moneymaking, Inc. was trading at $1" to "At 10 a.m. Generic Moneymaking, Inc. was trading at $200" and tries to make the corresponding changes to T.
2. Reordering M. An adversary may not change any bits, but just reorder the existing content of M. For example, VERIFY should return REJECT if the adversary changes M to "At 1 a.m. Generic Moneymaking, Inc. was trading at $10" (The adversary has moved "0" from "10 a.m." to "$10").
3. Extending M by prepending or appending information to M. An adversary may not change the content of M, but just prepend or append some information to the existing content of M. For example, an adversary may change M to "At 10 a.m. Generic Moneymaking, Inc. was trading at $10" (The adversary has appended "0" to the end of the message).
4. Splicing several messages and tags. An adversary may have recorded two messages and their tags, and tried to combine them into a new message and tag. For example, an adversary might take "At 10 a.m. Generic Moneymaking, Inc." from one transmitted message and combine it with "was trading at $9" from another transmitted message, and splice the two tags that go along with those messages by taking the first several bytes from the first tag and the remainder from the second tag.
5. Since SIGN and VERIFY are based on cryptographic transformations, it may also be possible to directly attack those transformations. Some mathematicians, known as cryptanalysts, are specialists in devising such attacks.

These requirements and the possible attacks make clear that the construction of SIGN and VERIFY primitives is a difficult task. To protect messages against the attacks listed above requires a cryptographer who can design the appropriate cryptographic transformations on the messages. These transformations are based on sophisticated mathematics. Thus, we have the worst of two possible worlds: we must achieve a negative goal using complex tools. As a result, even experts have come up with transformations that failed spectacularly. Thus, a non-expert certainly should not attempt to implement SIGN and VERIFY, and their implementation falls outside the scope of this book. (The interested reader can consult Section 5.9 to get a flavor of the complexities.)

The window of validity for SIGN and VERIFY is the minimum of the time to compromise the signing algorithm, the time to compromise the hash algorithm used in the signature (if one is used), the time to try out all keys, and the time to compromise the signing key.

As an example of the importance of keeping track of the window of validity, a team of researchers in 2008 was able to create forged signatures that many Web browsers accepted as valid.* The team used a large array of processors found in game consoles to perform a collision attack on a hash function designed in 1994 called MD5. MD5 had been identified as potentially weak as early as 1996 and a collision attack was demonstrated in 2004. Continued research revealed ways of rapidly creating collisions, thus allowing a search for helpful collisions. The 2008 team was able to find a helpful collision with which they could forge a trusted signature on an authentication message. Because some authentication systems that Web browsers trust had not yet abandoned their use of MD5, many browsers accepted the signature as valid and the team was able to trick these browsers into making what appeared to be authenticated connections to well-known Web sites. The connections actually led to impersonation Web sites that were under the control of the research team. (The forged signatures were on certificates for the transport layer security (TLS) protocol. Certificates are discussed in Sections 5.6.1 and 5.8.4, and Section 5.11 is a case study of TLS.)

---

* A. Sotirov et al. MD5 considered harmful: creating a rogue CA certificate. *25th Annual Chaos Communication Congress*, Berlin, December 2008.

5.4.5: Public-key versus Shared-Secret Authentication

If Alice signs the message using a shared-secret key, then Bob verifies the tag using the *same* shared-secret key. That is, VERIFY checks the received authentication tag from the message and the shared-secret key. An authentication tag computed with a shared-secret key is called a **message authentication code** (MAC). (The verb "to MAC" is the common jargon for "to compute an authentication tag using shared-secret cryptography".)

In the literature, the word "sign" is usually reserved for generating authentication tags with public-key cryptography. If Alice signs the message using public-key cryptography, then Bob verifies the message using a *different* key from the one that Alice used to compute the tag. Alice uses her private key to compute the authentication tag. Bob uses Alice's corresponding public key to verify the authentication tag. An authentication tag computed with a public-key system is called a **digital signature**. The digital signature is analogous to a conventional signature because only one person, the holder of the private key, could have applied it.

Alice's digital signatures can be checked by anyone who knows Alice's public key, while checking her MACs requires knowledge of the shared-secret key that she used to create the MAC. Thus, Alice might be able to successfully **repudiate** (disown) a message authenticated with a MAC by arguing that Bob (who also knows the shared-secret key) forged the message and the corresponding MAC.

In contrast, the only way to repudiate a digital signature is for Alice to claim that someone else has discovered her private key. Digital signatures are thus more appropriate for electronic checks and contracts. Bob can verify Alice's signature on an electronic check she gives him, and later when Bob deposits the check at the bank, the bank can also verify her signature. When Alice uses digital signatures, neither Bob nor the bank can forge a message purporting to be from Alice, in contrast to the situation in which Alice uses only MACs.

Of course, non-repudiation depends on not losing one's private key. If one loses one's private key, a reliable mechanism is needed for broadcasting the fact that the private key is no longer secret so that one can repudiate later forged signatures with the lost private key. Methods for revoking compromised private keys are the subject of considerable debate.

`SIGN` and `VERIFY` are two powerful primitives, but they must be used with care. Consider the following attack. Alice and Bob want to sign a contract saying that Alice will pay Bob $100. Alice types it up as a document using a word-processing application and both digitally sign it. In a few days Bob comes to Alice to collect his money. To his surprise, Alice presents him with a Word document that states he owes her $100. Alice also has a valid signature from Bob for the new document. In fact, it is the exact same signature as for the contract Bob remembers signing and, to Bob's great amazement, the two documents are actually bit-for-bit identical. What Alice did was create a document that included an `if` statement that changed the displayed content of the document by referring to an external input such as the current date or filename. Thus, even though the signed contents remained the same, the displayed contents changed because they were partially dependent on unsigned inputs. The problem here is that Bob's mental model doesn't correspond to what he has signed. As always with security, all aspects must be thought through! Bob is much better off signing only documents that he himself created.

5.4.6: Key Distribution

We assumed that if Bob successfully verified the authentication tag of a message, that Alice is the message's originator. This assumption, in fact, has a serious flaw. What Bob really knows is that the message originated from a principal that knows key K_1. The assumption that the key K_1 belongs to Alice may not be true. An adversary may have stolen Alice's key or may have tricked Bob into believing that K_1 is Alice's key. Thus, the way in which keys are bound to principals is an important problem to address.

The problem of securely distributing keys is also sometimes called the **name-to-key binding problem**; in the real world, principals are named by descriptive names rather than keys. So, when we know the name of a principal, we need a method for securely finding the key that goes along with the named principal. The trust that we put in a key is directly related to how secure the key distribution system is.

Secure key distribution is based on a name discovery protocol, which starts, perhaps unsurprisingly, with trusted physical delivery. When Alice and Bob meet, Alice can give Bob a cryptographic key. This key is authenticated because Bob knows he received it exactly as Alice gave it to him. If necessary, Alice can give Bob this key secretly (in an envelope or on a portable storage card), so others don't see or overhear it. Alice could also use a mutually trusted courier to deliver a key to Bob in a secret and authenticated manner.

Cryptographic keys can also be delivered over a network. However, an adversary might add, delete, or modify messages on the network. A good cryptographic system is needed to ensure that the network communication is authenticated (and confidential, if necessary). In fact, in the early days of cryptography, the doctrine was never to send keys over a network; a compromised key will result in more damage than one compromised message. However, nowadays cryptographic systems are believed to be strong enough to take that risk. Furthermore, with a key-distribution protocol in place it is possible to periodically generate new keys, which is important to limit the damage in case a key is compromised.

The catch is that one needs cryptographic keys already in place in order to distribute new cryptographic keys over the network! This approach works if the recursion "bottoms out" with physical key delivery. Suppose two principals Alice and Bob wish to communicate, but they have no shared (shared-secret or public) key. How can they establish keys to use?

One common approach is to use a mutually-trusted third party (Charles) with whom Alice and Bob already each share key information. For example, Charles might be a mutual friend of Alice and Bob. Charles and Alice might have met physically at some point in time and exchanged keys, and similarly, Charles and Bob might have met and also exchanged keys. If Alice and Bob both trust Charles, then Alice and Bob can exchange keys through Charles.

How Charles can assist Alice and Bob depends on whether they are using shared-secret or public-key cryptography. Shared-secret keys need to be distributed in a way that is both confidential and authenticated. Public keys do not need to be kept secret, but need to be distributed in an authenticated manner. What we see developing here is a need for another security protocol, which we will study in Section 5.6.

In some applications it is difficult to arrange for a common third party. Consider a person who buys a personal electronic device that communicates over a wireless network. The owner installs the new gadget (e.g., digital surveillance camera) in the owner's house and would like to make sure that burglars cannot control the device over the wireless network. But, how does the device authenticate the owner, so that it can distinguish the owner from other principals (e.g., burglars)? One option is that the manufacturer or distributor of the device plays the role of Charles. When purchasing a device, the manufacturer records the buyer's public key. The device has burned into it the public key of the manufacturer; when the buyer turns on the device, the device establishes a secure communication link using the manufacturer's public key and asks the manufacturer for the public key of its owner. This solution is impractical, unfortunately: what if the device is not connected to a global network and thus cannot reach the manufacturer? This solution might also have privacy objections: should manufacturers be able to track when consumers use devices? Sidebar 5.4.1, about the **resurrecting duckling**, provides a solution that allows key distribution to be performed locally, without a central principal involved.

Sidebar 5.4.1

Authenticating personal devices: the resurrecting duckling policy

Inexpensive consumer devices have (or will soon have) embedded microprocessors in them that are able to communicate with other devices over inexpensive wireless networks. If household devices such as the home theatre, the heating system, the lights, and the surveillance cameras are controlled by, say, a universal remote control, an owner must ensure that these devices (and new ones) obey the owner's commands and not the neighbor's or, worse, a burglar's.This situation requires that a device and the remote control be able to establish a secure relationship. The relationship may be transient, however; the owner may want to resell one of the devices, or replace the remote control.

In *The resurrecting duckling: security issues for ad-hoc wireless networks* [Suggestions for Further Reading 5.4.2], Stajano and Anderson provide a solution based on the vivid analogy of how ducklings authenticate their mother. When a duckling emerges from its egg, it will recognize as its mother the first moving object that makes a sound. In the Stajano and Anderson proposal, a device will recognize as its owner the first principal that sends it an authentication key. As soon as the device receives a key, its status changes from newborn to imprinted, and it stays faithful to that key until its death. Only an owner can force a device to die and thereby reverse its status to newborn. In this way, an owner can transfer ownership.

A widely used example of the resurrecting duckling is purchasing wireless routers. These routers often come with the default user name "Admin" and password "password". When the buyer plugs the router in for the first time, it is waiting to be imprinted with a better password; the first principal to change the password gets control of the router. The router has a resurrection button that restores the defaults, thus again making it imprintable (and allowing the buyer to recover if an adversary did grab control).

Not all applications deploy a sophisticated key-distribution protocol. For example, the **secure shell** (SSH), a popular Internet protocol used to log onto a remote computer, has a simple key distribution protocol. The first time that a user logs onto a server named "athena.Scholarly.edu", SSH sends a message in the clear to the machine with DNS name athena.Scholarly.edu asking it for its public key. SSH uses that public key to set up an authenticated and confidential communication link with the remote computer. SSH also caches this key and remembers that the key is associated with the DNS name "athena.Scholarly.edu". The next time the user logs onto athena.Scholarly.edu, SSH uses the cached key to set up the communication link.

Because the DNS protocol does not include message authentication, the security risk in SSH's approach is a masquerading attack: an adversary might be able to intercept the DNS lookup for "athena.Scholarly.edu" and return an IP address for a computer controlled by the adversary. When the user connects to that IP address, the adversary replies with a key that the adversary has generated. When the user makes an SSH connection using that public key, the adversary's computer masquerades as athena.Scholarly.edu. To counter this attack, the SSH client asks a question to the user on the first connection to a remote computer: "I don't recognize the key of this remote computer, should I trust it?", and a wary user should compare the displayed key with one that it received from the remote computer's system administrator over an out-of-band secure communication link (e.g., a piece of paper). Many users aren't wary and just answer "yes" to the question.

The advantage of the SSH approach is that no key distribution protocol is necessary (beyond obtaining the fingerprint). This has simplified the deployment of SSH and has made it a success. As we will see in Section 5.6, securely distributing keys such that a masquerading attack is impossible is a challenging problem.

5.4.7: Long-Term Data Integrity with Witnesses

Careful use of `SIGN` and `VERIFY` can provide both data integrity and authenticity guarantees. Some applications have requirements for which it is better to use different techniques for integrity and authenticity. Sidebar 1.2.1 mentions a digital archive, which requires protection against an adversary who tries to change the content of a file stored in the archive. To protect a file, a designer wants to make many separate replicas of the file, following the durability mantra, preferably in independently administered and thus separately protected domains. If the replicas are separately protected, it is more difficult for an adversary to change all of them.

Since maintaining widely-separated copies of large files consumes time, space, and communication bandwidth, one can reduce the resource expenditure by replacing some (but not all) copies of the file with a smaller witness, with which users can periodically check the validity of replicas (as explained in Section 4.4.4). If the replica disagrees with the witness, then one repairs the replica by finding a replica that matches the witness. Because the witness is small, it is easy to protect it against tampering. For example, one can publish the witness in a widely-read newspaper, which is likely to be preserved either on microfilm or digitally in many public libraries.

This scheme requires that a witness be cryptographically secure. One way of constructing a secure witness is using `SIGN` and `VERIFY` . The digital archiver uses a cryptographic hash function to create a secure fingerprint of the file, signs the fingerprint with its private key, and then distributes copies of the file widely. Anyone can verify the integrity of a replica by computing the fingerprint of the replica, verifying the witness using the public key of the archiver, and then comparing the fingerprint of the witness against the fingerprint of the replica.

This scheme works well in general, but is less suitable for long-term data integrity. The window of validity of this scheme is determined by the minimum time to compromise the private key used for signing, the signing algorithm, the hashing algorithm, and the validity of the name-to-public key binding. If the goal of the archiver is to protect the data for many decades (or forever), it is likely that the digital signature will be invalid before the data.

In such cases, it is better to protect the witness by widely publishing just the cryptographic hash instead of using `SIGN` and `VERIFY` . In this approach, the validity of the witness is the time to compromise the cryptographic hash. This window can be made large. One can protect against a compromised cryptographic hash algorithm by occasionally computing and publishing a new witness with the latest, best hash algorithm. The new witness is a hash of the original data, the original witness, and a timestamp, thereby demonstrating the integrity of the original data at the time of the new witness calculation.

The confidence a user has in the authenticity of a witness is determined by how easily the user can verify that the witness was indeed produced by the archiver. If the newspaper or the library physically received the witnesses directly from the archiver, then this confidence may be high.

This page titled 5.4: Authenticating Messages is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

5.5: Message Confidentiality

Some applications may require message confidentiality in addition to message authentication. Two principals may want to communicate *privately* without adversaries having access to the communicated information. If the principals are running on a shared physical computer, this goal is easily accomplished using the kernel. For example, when sending a message to a port, it is safe to ask the kernel to copy the message to the recipient's address space, since the kernel is already trusted; the kernel can read the sender's and receiver's address space anyway.

If the principals are on different physical processors, and can communicate with each other only over an untrusted network, ensuring confidentiality of messages is more challenging. By definition, we cannot trust the untrusted network to not disclose the bits that are being communicated. The solution to this problem is to introduce encryption and decryption to allow two parties to communicate without anyone else being able to tell what is being communicated.

5.5.1: Message Confidentiality Using Encryption

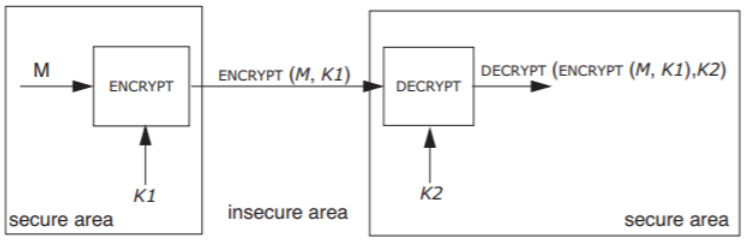


Figure 5.5.1: Providing confidentiality using `ENCRYPT` and `DECRYPT` over untrusted networks.

The setup for providing confidentiality over untrusted networks is shown in Figure 5.5.1. Two secure areas are separated by an insecure communication path. Our goal is to provide a secure channel between the two secure areas that provides confidentiality.

Encryption transforms a **plaintext** message into **ciphertext** in such a way that an observer cannot construct the original message from the ciphertext version, yet the intended receiver can. **Decryption** transforms the received ciphertext into plaintext. Thus, one challenge in the implementation of channels that provide confidentiality is to use an encrypting scheme that is difficult to reverse for an adversary. That is, even if an observer could copy a message that is in transit and has an enormous amount of time and computing power available, the observer should not be able to transform the encrypted message into the plaintext message. (As with "signing", we use the term "messages" conceptually; one can also encrypt and decrypt files, email attachments, streams, or other data objects.)

The `ENCRYPT` and `DECRYPT` primitives can be implemented using cryptographic transformations. `ENCRYPT` and `DECRYPT` can use either shared-secret cryptography or public-key cryptography. We refer to the keys used for encryption as **encryption keys**.

With shared-secret cryptography, Alice and Bob share a key K that only they know. To keep a message M confidential, Alice computes `ENCRYPT (M, K)` and sends the resulting ciphertext C to Bob. If the encrypting box is good, an adversary will not to be able to get any use out of the ciphertext. Bob computes `DECRYPT (C, K)` , which will recover the plaintext form of M . Bob can send a reply to Alice using exactly the same system with the same key. (Of course, Bob could also send the reply with a different key, as long as that different key is also shared with Alice.)

With public-key cryptography, Alice and Bob do not have to share a secret to achieve confidentiality for communication. Suppose Bob has a private and public key pair (K_{Bpriv} , K_{Bpub}), where K_{Bpriv} is Bob's private key and K_{Bpub} is Bob's public key. Bob gives his public key to Alice through an existing channel; this channel does not have to be secure, but it does have to provide authentication: Alice needs to know for sure that this key is really Bob's key.

Given Bob's public key (K_{Bpub}), Alice can compute `ENCRYPT (M,` K_{Bpub}`)` and send the encrypted message over an insecure network. Only Bob can read this message, since he is the only person who has the secret key that can decrypt her ciphertext message. Thus, using encryption, Alice can ensure that her communication with Bob stays confidential. To achieve confidential communication in the opposite direction (from Bob to Alice), we need an additional set of keys, a K_{Apub} and K_{Apriv} for Alice, and Bob needs to learn Alice's public key.

5.5.2: Properties of ENCRYPT and DECRYPT

For both the shared-key and public-key encryption systems, the procedures ENCRYPT and DECRYPT should have the following properties. It should be easy to compute:

- $C \leftarrow$ ENCRYPT (M, K_1)
- $M' \leftarrow$ DECRYPT (C, K_2)

and the result should be that $M = M'$.

The implementation of ENCRYPT and DECRYPT should withstand the following attacks:

1. Ciphertext-only attack. In this attack, the primary information available to the adversary is examples of ciphertext and the algorithms for ENCRYPT and DECRYPT . Redundancy or repeated patterns in the original message may show through even in the ciphertext, allowing an adversary to reconstruct the plaintext. In an open design the adversary knows the algorithms for ENCRYPT and DECRYPT , and thus the adversary may also be able to mount a brute-force attack by trying all possible keys.

   More precisely, when using shared-secret cryptography, the following property must hold:
   • Given ENCRYPT and DECRYPT , and some examples of C , it should be difficult for an adversary to reconstruct K or compute M .

   When using public-key cryptography, the corresponding property holds:
   • Given ENCRYPT and DECRYPT , some examples of C , and assuming an adversary knows K_1 (which is public), it should be difficult for the adversary to compute either the secret key K_2 or M .

2. Known-plaintext attack. The adversary has access to the ciphertext C and also to the plaintext M corresponding to at least some of the ciphertext C . For instance, a message may contain standard headers or a piece of predictable plaintext, which may help an adversary figure out the key and then recover the rest of the plaintext.

3. Chosen-plaintext attack. The adversary has access to ciphertext C that corresponds to plaintext M that the adversary has chosen. For instance, the adversary may convince you to send an encrypted message containing some data chosen by the adversary, with the goal of learning information about your transforming system, which may allow the adversary to more easily discover the key. As a special case, the adversary may be able in real time to choose the plaintext M based on ciphertext C just transmitted. This variant is known as an adaptive attack.

   A common design mistake is to unintentionally admit an adaptive attack by providing a service that happily encrypts any input it receives. This service is known as an **oracle**, and it may greatly simplify the effort required by an adversary to crack the cryptographic transformation. For example, consider the following adaptive chosen-plaintext attack on the encryption of packets in WiFi wireless networks. The adversary sends a carefully-crafted packet from the Internet addressed to some node on the WiFi network. The network will encrypt and broadcast that packet over the air, where the adversary can intercept the ciphertext, study it, and immediately choose more plaintext to send in another packet. Researchers used this attack as one way of breaking the design of the security of WiFi Wired Equivalent Privacy (WEP)*.

4. Chosen-ciphertext attack. The adversary might be able to select a ciphertext C and then observe the M' that results when the recipient decrypts C . Again, an adversary may be able to mount an adaptive chosen-ciphertext attack.

Section 5.9 describes cryptographic implementations of ENCRYPT and DECRYPT that provide protection against these attacks. A designer can increase the work factor for an adversary by increasing the key length. A typical key length used in the field is 1,024 bits.

The window of validity of ENCRYPT and DECRYPT is the minimum of the time to compromise of the underlying cryptographic transformation, the time to try all keys, and the time to compromise the key itself. When considering what implementation of ENCRYPT and DECRYPT to use, it is important to understand the required window of validity. It is likely that the window of validity required for encrypting protocol messages between a client and a server is smaller than the window of validity required for encrypting long-term file storage. A protocol message that must be private just for the duration of a conversation might be adequately protected by an cryptographic transformation that can be compromised with, say, one year of effort. On the other hand,

if the period of time for which a file must be protected is greater than the window of validity of a particular cryptographic system, the designer may have to consider additional mechanisms, such as multiple encryptions with different keys.

---

* N. Borisov, I. Goldberg, and D. Wagner, *Intercepting mobile communications: the insecurity of 802.11*, MOBICOM '01, Rome, Italy, July 2001.

5.5.3: Achieving both Confidentiality and Authentication

Confidentiality and message authentication can be combined in several ways:

- For confidentiality only, Alice just encrypts the message.
- For authentication only, Alice just signs the message.
- For both confidentiality and authentication, Alice first encrypts and then signs the encrypted message (i.e., `SIGN (ENCRYPT (`M`,` $K_{\texttt{encrypt}}$`), `$K_{\texttt{sign}}$`)`), or, the other way around. (If good implementations of `SIGN` and `VERIFY` are used, it doesn't matter for correctness in which order the operations are applied.)

The first option, confidentiality without authentication, is unusual. After all, what is the purpose of keeping information confidential if the receiver cannot tell if the message has been changed? Therefore, if confidentiality is required, one also provides authentication.

The second option is common. Much data is public (e.g., routing updates, stock updates, etc.), but it is important to know its origin and integrity. In fact, it is easy to argue the default should be that all messages are at least authenticated.

For the third option, the keys used for authentication and confidentiality are typically different. The sender authenticates with an authentication key, and encrypts with a encryption key. The receiver would use the appropriate corresponding keys to decrypt and to verify the received message. The reason to use different keys is that the key is a bit pattern, and using the same bit pattern as input to two cryptographic operations on the same message is risky because a clever cryptanalyst may be able to discover a way of exploiting the repetition. Section 5.9 gives an example of exploitation of repetition in an otherwise unbreakable encryption system known as the one-time pad. Problem sets 44 and 46 also explore one-time pads to setup a secure communication channel.

In addition to using the appropriate keys, there are other security hazards. For example, M should have identified explicitly the communicating parties. When Alice sends a message to Bob, she should include in the message the names of Alice and Bob to avoid impersonation attacks. Failure to follow this *explicitness principle* can create security problems, as we will see in Section 5.6.

5.5.4: Can Encryption be Used for Authentication?

As specified, `ENCRYPT` and `DECRYPT` don't protect against an adversary modifying M and one must `SIGN` and `VERIFY` for integrity. With some implementations, however, a recipient of an encrypted message can be confident not only of its confidentiality, but also of its authenticity. From this observation arose the misleading intuition that decrypting a message and finding something recognizable inside is an effective way of establishing the authenticity of that message. The intuition is based on the claim that if only the sender is able to encrypt the message, and the message contains at least one component that the recipient expected the sender to include, then the sender must have been the source of the message.

The problem with this intuition is that as a general rule, the claim is wrong. It depends on using a cryptographic system that links all of the ciphertext of the message in such a way that it cannot be sliced apart and re-spliced, perhaps with components from other messages between the same two parties and using the same cryptographic key. As a result, it is non-trivial to establish that a system based on the claim is secure even in the cases in which it is. Many protocols that have been published and later found to be defective were designed using that incorrect intuition. Those protocols using this approach that are secure require much effort to establish the necessary conditions, and it is remarkably hard to make a compelling argument that they are secure; the argument typically depends on the exact order of fields in messages, combined with some particular properties of the underlying cryptographic operations.

Therefore, in this book we treat message confidentiality and authenticity as two separate goals that are implemented independently of each other. Although both confidentiality and authenticity rely in their implementation on cryptography, they use the cryptographic operations in different ways. As explained in Section 5.9, the shared-secret AES cryptographic transformation, for example, isn't by itself suitable for *either* signing or encrypting; it needs to be surrounded by various cipher-feedback mechanisms,

and the mechanisms that are good for encrypting are generally somewhat different from those that are good for signing. Similarly, when RSA, a public-key cryptographic transformation, is used for signing, it is usually preceded by hashing the message to be signed, rather than applying RSA directly to the message; a failure to hash can lead to a security blunder.

A relatively recent paper* on the topic on the order of authentication and encrypting suggests that first encrypting and then computing an authentication tag may cover up certain weaknesses in some implementations of the encrypting primitives. Also, cryptographic transformations have been proposed that perform the transformation for encrypting and computing an authentication tag in a single pass over the message, saving time compared to first encrypting and then computing an authentication tag. Cryptography is a developing area, and the last word on this topic has not been said; interested readers should check out the proceedings of the conferences on cryptography. For the rest of the book, however, the reader can think of message authentication and confidentiality as two separate, orthogonal concepts.

---

* Hugo Krawczyk, *The Order of Encryption and Authentication for Protecting Communications (or: How Secure is SSL?)*, Advances in Cryptology (Springer LNCS 2139), 2001, pages 310–331.

This page titled 5.5: Message Confidentiality is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

5.6: Security Protocols

In the previous sections we discovered a need for protecting a principal's password when authenticating to a remote service, a need for distributing keys securely, etc. Security protocols can achieve those objectives. A **security protocol** is an exchange of messages designed to allow mutually-distrustful parties to achieve an objective. Security protocols often use cryptographic techniques to achieve the objective. Other example objectives include electronic voting, postage stamps for email, anonymous email, and electronic cash for micropayments.

In a security protocol with two parties, the pattern is generally a back-and-forth pattern. Some security protocols involve more than two parties in which case the pattern may be more complicated. For example, key distribution usually involves at least three parties (two principals and a trusted third party). A credit-purchase on the Internet is likely to involve many more principals than three (a client, an Internet shop, a credit card company, and one or more trusted third parties) and thus require four or more messages.

The difference between the network protocols discussed in Chapter 1 and the security ones is that standard networking protocols assume that the communicating parties cooperate and trust each other. In designing security protocols, we instead assume that some parties in the protocol may be adversaries and also that there may be an outside party attacking the protocol.

5.6.1: Example: Key Distribution

To illustrate the need for security protocols, let's study two protocols for key distribution. In Section 5.4.6, we have already seen that distributing keys is based on a name discovery protocol, which starts with trusted physical delivery. So, let's assume that Alice has met Charles in person, and Charles has met Bob in person. The question then is: is there a protocol such that Alice and Bob, who have never met, can exchange keys securely over an untrusted network? This section introduces the basic approach and subsequent sections work out the approach in detail.

The public-key case is simpler, so we treat it first. Alice and Bob already know Charles's public key (since they have met in person), and Charles knows each of Alice and Bob's public keys. If Alice and Bob both trust Charles, then Alice and Bob can exchange keys through Charles.

Alice sends a message to Charles (it does not need to be either encrypted or signed), asking:

1. Alice ⇒ Charles: {"Please give me keys for Bob"}

The message content is the string "Please give me keys for Bob". The source address is "Alice" and the destination address is "Charles." When Charles receives this message from Alice, he cannot be certain that if the message came from Alice, since the source and destination fields of Chapter 1 are not authenticated.

For this message, Charles doesn't really care who sent it, so he replies:

2. Charles ⇒ Alice: {"To communicate with Bob, use public key K_{Bpub}."}$_{Cpriv}$

The notation $\{M\}_k$ denotes signing a message M with key k . In this example, the message is signed with Charles's private authentication key. This signed message to Alice includes the content of the message as well as the authentication tag. When Alice receives this message, she can tell from the fact that this message verifies with Charles's public key that the message actually came from Charles.

Of course, these messages would normally not be written in English, but in some machine-readable semantically equivalent format. For expository and design purposes, however, it is useful to write down the meaning of each message in English. Writing down the meaning of a message in English helps make apparent oversights, such as omitting the name of the intended recipient. This method is an example of the design principle *be explicit*.

To illustrate that problems can be caused by of lack of explicitness, suppose that the previous message 2 were:

2'. Charles ⇒ Alice: {"Use public key K_{Bpub}."}$_{Cpriv}$

If Alice receives this message, she can verify with Charles's public key that Charles sent the message, but Alice is unable to tell whose public key K_{Bpub} is. An adversary Lucifer, whom Charles has met but doesn't know is bad, might use this lack of explicitness as follows. First, Lucifer asks Charles for Lucifer's public key, and Charles replies:

2'. Charles ⇒ Lucifer: {"Use public key K_{Lpub}."}$_{Cpriv}$

Lucifer saves the reply, which is signed by Charles. Later when Alice asks Charles for Bob's public key, Lucifer replaces Charles's response with the saved reply. Alice receives the message:

2'. Someone ⇒ Alice: {"Use public key K_{Lpub}."}$_{Cpriv}$

From looking at the source address (Someone), she *cannot* be certain where message 2' came from. The source and destination fields of Chapter 1 are not authenticated, so Lucifer can replace the source address with Charles's source address. This change won't affect the routing of the message, since the destination address is the only address needed to route the message to Alice. Since the source address cannot be trusted, the message itself has to tell her where it came from, and message 2' says that it came from Charles because it is signed by Charles.

Believing that this message came from Charles, Alice will think that this message is Charles's response to her request for Bob's key. Thus, Alice will incorrectly conclude that K_{Lpub} is Bob's public key. If Lucifer can intercept Alice's subsequent messages to Bob, Lucifer can pretend to be Bob, since Alice believes that Bob's public key is K_{Lpub} and Lucifer has K_{Lpriv}. This attack would be impossible with message 2 because Alice would notice that it was Lucifer's key, rather than Bob's.

Returning to the correct protocol using message 2 rather than message 2', after receiving Charles's reply, Alice can then sign (with her own private key, which she already knows) and encrypt (with Bob's public key, which she just learned from Charles) any message that she wishes to send to Bob. The reply can be handled symmetrically, after Bob obtains Alice's public key from Charles in a similar manner.

Alice and Bob are trusting Charles to correctly distribute their public keys for them. Charles's message (2) must be signed, so that Alice knows that it really came from Charles, instead of being forged by an adversary. Since we presumed that Alice already had Charles's public key, she can verify Charles's signature on message 2 .

Bob cannot send Alice his public key over an insecure channel, even if he signs it. The reason is that she cannot believe a message signed by an unknown key asserting its own identity. But a message like 2 that is signed by Charles can be believed by Alice, if she trusts Charles to be careful about such things. Such a message is called a **certificate**: it contains Bob's name and public key, certifying the binding between Bob and his key. Bob himself could have sent Alice the certificate Charles signed, if he had the foresight to have already obtained a copy of that certificate from Charles. In this protocol Charles plays the role of a **certificate authority** (CA). The idea of using the signature of a trusted authority to bind a public key to a principal identifier and calling the result a certificate was invented in Loren Kohnfelder's 1978 M.I.T. bachelor's thesis.

When shared-secret instead of public-key cryptography is being used, we assume that Alice and Charles have pre-established a shared-secret authentication key Ak_{AC} and a shared-secret encryption key Ek_{AC}, and that Bob and Charles have similarly pre-established a shared-secret authentication key Ak_{BC} and a shared-secret encryption key Ek_{BC}. Alice begins by sending a message to Charles (again, it does not need to be encrypted or signed):

1. Alice ⇒ Charles: {"Please give me keys for Bob"}

Since shared-secret keys must be kept confidential, Charles must both sign and encrypt the response, using the two shared-secret keys Ak_{AC} and Ek_{AC}. Charles would reply to Alice:

2. Charles ⇒ Alice: {"Use temporary authentication key Ak_{AB} and temporary encryption key Ek_{AB} to talk to Bob."} $_{AkAC}$ $^{Ek_{AC}}$

The notation $\{M\}^k$ denotes encrypting message M with encryption key k. In this example, the message from Charles to Alice is signed by the shared-secret authentication key Ak_{AC} and encrypted with the shared-secret encryption key Ek_{AC}.

The keys Ak_{AB} and Ek_{AB} in Charles's reply are newly-generated random shared-secret keys. If Charles would have replied with Ak_{BC} and Ek_{BC} instead of newly-generated keys, then Alice would be able to impersonate Bob to Charles, or Charles to Bob.

It is also important that message 2 is both authenticated with Charles's and Alice's shared key Ak_{AC} and encrypted with their shared Ek_{AC}. The k_{AC}'s are known only to Alice and Charles, so Alice can be confident that the message came from Charles and that only she and Charles know the k_{AB}'s. The next step is for Charles to tell Bob the keys:

3. Charles ⇒ Bob: {"Use the temporary keys Ak_{AB} and Ek_{AB} to talk to Alice."} $_{AkBC}$ $^{Ek_{BC}}$

This message is both authenticated with key Ak_{BC} and encrypted with key Ek_{BC}, which are known only to Charles and Bob, so Bob can be confident that the message came from Charles and that no one else but Alice and Charles know the two k_{AB}'s.

From then on, Alice and Bob can communicate using the temporary key Ak_{AB} to authenticate and the temporary key Ek_{AB} to encrypt their messages. Charles should immediately erase any memory he has of the two temporary k_{AB} keys. In such an arrangement, Charles is usually said to be acting as a **key distribution center** (or KDC). The idea of a shared-secret key distribution center was developed in classified military circles and first revealed to the public in a 1973 paper by Dennis Branstad.* In the academic community it first showed up in a paper by Needham and Schroeder.**

A common variation is for Charles to include message 3 to Bob as an attachment to his reply 2 to Alice; Alice can then forward this attachment to Bob along with her first real message to him. Since message 3 is both authenticated and encrypted, Alice is simply acting as an additional, more convenient forwarding point so that Bob does not have to match up messages arriving from different places.

Not all key distribution and authentication protocols separate authentication and encryption (e.g., see Sidebar 5.6.1 about Kerberos); they instead accomplish authentication by using carefully-crafted encrypting, with just one shared key per participant. Although having fewer keys seems superficially simpler, it is then harder to establish the correctness of the protocols. It is simpler to use the divide-and-conquer strategy: the additional overhead of having two separate keys for authentication and encrypting is well worth the simplicity and ease of establishing correctness of the overall design.

Sidebar 5.6.1

The Kerberos authentication system

Kerberos[1] was developed in the late 1980s for project Athena, a network of engineering workstations and servers designed to support undergraduate education at M.I.T.[2] The first version in widespread use was Version 4, which is described here in simplified form; newer versions of Kerberos improve and extend Version 4 in various ways, but the general approach hasn't changed much.

A Kerberos service implements a unique identifier name space, called a **realm**, in which each name of the name space is the principal identifier of either a network service or an individual user. Kerberos also allows a confederation of Kerberos services belonging to different organizations to implement a name space of realms. Principal names are of the form "alice@Scholarly.edu", a principal identifier followed by the name of the realm to which that principal belongs. Kerberos principal identifiers are case-sensitive. Users and services are connected by an open, untrusted network. The goal of Kerberos is to provide two-way authentication between a user and a network service securely under the threat of adversaries.

A user authenticates the user's identity and logs on to a realm using a shared-secret protocol with the realm's Kerberos Key Distribution Service (KKDS). Kerberos derives the shared-secret key by cryptographically hashing a user-chosen password. During the name-discovery step (e.g., a physical rendezvous with its administrator), the Kerberos service learns the principal identifier for the user and the shared secret. When logging on, the user sends its principal identifier to KKDS and asks it for authentication information to talk to service S :

$$\text{Alice} \Rightarrow \text{KDDS: } \{\text{"alice@Scholarly.edu"}, S, T_{current}\}$$

and the service responds with a **ticket** identifying the user:

$$\text{KKDS} \Rightarrow \text{Alice: } \{K_{tmp}, S, Lifetime, T_{current}, ticket\}^{K_{alice}}$$

The service encrypts this response with the user's shared secret. The verification step occurs when the user decrypts the encrypted response. If $T_{current}$ and S in the response match with the values in the request, then Kerberos considers the response authentic, and uses the information in the decrypted response to authenticate the user to S . If the user does not posses the key (the hashed password) that decrypts the response, the information inside the response is worthless.

The ticket is a kind of certificate; it binds the user name to a temporary key for use during one session with service S . Kerberos includes the following information in the ticket:

$$ticket = \{K_{tmp}, \text{"alice@Scholarly.edu"}, S, T_{current}, Lifetime\}^{K_S}$$

The temporary key K_{tmp} is to allow a user to establish a continued chain of authentication without having to go back to KKDS for each message exchange. The ticket contains a time stamp, the principal identifier of the user, the principal identifier of the service, and a second copy of the temporary key, all encrypted in the key shared between the KDDS and the service S (e.g., a network file service).[3]

Kerberos includes in a request to a Kerberos-mediated network service the ticket identifying the user. When the service receives a request, it authenticates the ticket using the information in the ticket. It decrypts the ticket, checks that the timestamp inside is recent and that its own principal identifier is accurate. If the ticket passes these tests, the service believes that it has the authentic principal identifier of the requesting user and the Kerberos protocol is complete. Knowing the user's principal identifier, the service can then apply its own authorization system to establish that the user has permission to perform the requested operation.

A user can perform cross-realm authentication by applying the basic Kerberos protocol twice: first obtain a ticket from a local KDC for the other realm's KDC, and then using that ticket obtain a second ticket from the remote realm's KDC for a service in the remote realm. For cross-realm authentication to work, there are two prerequisites: (1) initialization: the two realms must have previously agreed upon a shared-secret key between the realms and (2) name discovery: the user and service must each know the other's principal identifier and realm name.

Versions 4 and 5 of Kerberos are in widespread use outside of M.I.T. (e.g., they were adopted by Microsoft). They are based on formerly classified key distribution principles first publicly described in a paper by Branstad and are strengthened versions of a protocol described by Needham and Schroeder (mentioned in the footnotes for Section 5.6.1). These protocols don't separate authentication from confidentiality. They instead rely on clever use of cryptographic operations to achieve both goals. As explained in Section 5.5.4, this property makes the protocols difficult to analyze.

---

[1] S[teven] P. Miller, B. C[lifford] Neuman, J[effrey] I. Schiller, and J[erome] H. Saltzer. Kerberos authentication and authorization system. Section E.2.1 of Athena Technical Plan, M.I.T. Project Athena, October 27, 1988.

[2] George A. Champine. M.I.T. Project Athena: A Model for Distributed Campus Computing. Digital Press, Bedford, Massachusetts, 1991. ISBN 1–55558–072–6. 282 pages.

[3] This description is a simplified version of the Kerberos protocol. One important omission is that the ticket a user receives as a result of successfully logging in is actually one for a ticket-granting service (TGS), from which the user can obtain tickets for other services. TGS provides what is sometimes called a **single login** or **single sign-on** system, meaning that a user needs to present a password only once to use several different network services.

For performance reasons, computer systems typically use public-key systems for distributing and authenticating keys and shared-secret systems for sending messages in an authenticated and confidential manner. The operations in public-key systems (e.g., raising to an exponent) are more expensive to compute than the operations in shared-secret cryptography (e.g., table lookups and computing several `XOR` 's). Thus, a session between two parties typically follows two steps:

1. At the start of the session use public-key cryptography to authenticate each party to the other and to exchange new, temporary, shared-secret keys;
2. Authenticate and encrypt subsequent messages in the session using the temporary shared-secret keys exchanged in step 1.

Using this approach, only the first few messages require computationally expensive operations, while all subsequent messages require only inexpensive operations.

One might wonder why it is not possible to the design the ultimate key distribution protocol once, get it right, and be done with it. In practice, there is no single protocol that will do. Some protocols are optimized to minimize the number of messages, others are optimized to minimize the cost of cryptographic operations, or to avoid the need to trust a third party. Yet others must work when the communicating parties are not both online at the same time (e.g., email), provide only one-way authentication, or require client anonymity. Some protocols, such as protocols for authenticating principals using passwords, require other properties than basic confidentiality and authentication: for example, such a protocol must ensure that the password is sent only once per session (see Section 5.3).

---

* Dennis K. Branstad. Security aspects of computer networks.American Institute of Aeronautics and Astronautics Computer Network Systems Conference, paper 73–427 (April, 1973).

** Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large net works of computers. Communications of the ACM 21, 12 (December, 1978), pages 993–999.

5.6.2: Designing Security Protocols

Security protocols are vulnerable to several attacks in addition to the ones described in Section 5.4.4 and Section 5.5.2 on the underlying cryptographic transformations. The new attacks to protect against fall in the following categories:

- **Known-key attacks**. An adversary obtains some key used previously and then uses this information to determine new keys.
- **Replay attacks**. An adversary records parts of a session and replays them later, hoping that the recipient treats the replayed messages as new messages. These replayed messages might trick the recipient into taking an unintended action or divulging useful information to the adversary.
- **Impersonation attacks**. An adversary impersonates one of the other principals in the protocol. A common version of this attack is the person-in-the-middle attack, where an adversary relays messages between two principals, impersonating each principal to the other, reading the messages as they go by.
- **Reflection attacks**. An adversary records parts of a session and replays it to the party that originally sent it. Protocols that use shared-secret keys are sometimes vulnerable to this special kind of replay attack.

The security requirements for a security protocol go beyond simple confidentiality and authentication. Consider a replay attack. Even though the adversary may not know what the replayed messages say (because they are encrypted), and even though the adversary may not be able to forge new legitimate messages (because the adversary doesn't have the keys used to compute authentication tags), the adversary may be able to cause mischief or damage by replaying old messages. The (duplicate) replayed messages may well be accepted as genuine by the legitimate participants, since the authentication tag will verify correctly.

The participants are thus interested not only in confidentiality and authentication, but also in the three following properties:

- **Freshness**. Does this message belong to this instance of this protocol, or is it a replay from a previous run of this protocol?
- **Explicitness**. Is this message really a member of this run of the protocol, or is it copied from an run of another protocol with an entirely different function and different participants?
- **Forward secrecy**. Does this protocol guarantee that if a key is compromised that confidential information communicated in the past stays confidential? A protocol has forward secrecy if it doesn't reveal, even to its participants, any information from previous uses of that protocol.

We study techniques to ensure freshness and explicitness; forward secrecy can be accomplished by using different temporary keys in each protocol instance and changing keys periodically. A brief summary of standard approaches to ensure freshness and explicitness include:

- Ensure that each message contains a nonce (a value, perhaps a counter value, serial number, or a timestamp, that will never again be used for any other message in this protocol), and require that a reply to a message include the nonce of the message being replied to, as well as its own new nonce value. The receiver and sender of course have to remember previously used nonces to detect duplicates. The nonce technique provides freshness and helps foil replay attacks.
- Ensure that each message explicitly contain the name of the sender of the message and of the intended recipient of the message. Protocols that omit this information, and that use shared-secret keys for authentication, are sometimes vulnerable to reflection attacks, as we saw in the example protocol in Section 5.6.1 above. Including names provides explicitness and helps foil impersonation and reflection attacks.
- Ensure that each message specifies the security protocol being followed, the version number of that protocol, and the message number within this instance of that protocol. If such information is omitted, a message from one protocol may be replayed during another protocol and, if accepted as legitimate there, cause damage. Including all protocol context in the message provides explicitness and helps foil replay attacks.

The explicitness property is an example of the *be explicit* design principle: ensure that each message be totally explicit about what it means. If the content of a message is not completely explicit, but instead its interpretation depends on its context, an adversary might be able to trick a receiver into interpreting the message in a different context and break the protocol. Leaving the names of the participants out of the message is a violation of this principle.

When a protocol designer applies these techniques, the key-distribution protocol of Section 5.6.1 might look more like:

$$1.\ \text{Alice} \Rightarrow \text{Charles: } \{\text{"This is message number one of the "Get Public Key" protocol, version 1.0. This message is sent by Alice and intended for Charles. This message was sent at 11:03:04.114 on 3 March 1999. The nonce for this message is 1456255797824510. What is the public key of Bob?"}\}_{A\text{priv}}$$

2. Charles ⇒ Alice: {"This is message number two of the "Get Public Key" protocol, version 1.0. This message is sent by Charles and intended for Alice. This message was sent at 11:03:33.004 on 3 March 1999. This is a reply to the message with nonce 1456255797824510. The nonce for this message is 5762334091147624. Bob's public key is (…)."}$_{Cpriv}$

In addition, the protocol would specify how to marshal and unmarshal the different fields of the messages so that an adversary cannot trick the receiver into unmarshaling the message incorrectly.

In contrast to the public-key protocol described above, the first message in this protocol is signed. Charles can now verify that the information included in the message came indeed from Alice and hasn't been tampered with. Now Charles can, for example, log who is asking for Bob's public key.

This protocol is almost certainly overdesigned, but it is hard to be confident about what can safely be dropped from a protocol. It is surprisingly easy to underdesign a protocol and leave security loopholes. The protocol may still seem to "work OK" in the field, until the loophole is exploited by an adversary. Whether a protocol "seems to work OK" for the legitimate participants following the protocol is an altogether different question from whether an adversary can successfully attack the protocol. Testing the security of a protocol involves trying to attack it or trying to prove it secure, not just implementing it and seeing if the legitimate participants can successfully communicate with it. Applying the safety net approach to security protocols tells us to overdesign protocols instead of underdesign.

Some applications require properties beyond freshness, explicitness, and forward secrecy. For example, a service way want to make sure that a single client cannot flood the service with messages, overloading the service and making it unresponsive to legitimate clients. One approach to provide this property is for the service to make it expensive for the client to generate legitimate protocol messages. A service could achieve this by challenging the client to perform an expensive computation (e.g., computing the inverse of a cryptographic function) before accepting any messages from the client. Yet other applications may require that more than one party be involved (e.g., a voting application). As in designing cryptographic primitives, designing security protocols is difficult and should be left to experts. The rest of this section presents some common security protocol problems that appear in computer systems and shows how one can reason about them. Problem set 43 explores how to use the signing and encryption primitives to achieve some simple security objectives.

5.6.3: Authentication Protocols

To illustrate the issues in designing security protocols, we will look at two simple authentication protocols. The second protocol uses a challenge and a response, which is an idea found in many security protocols. These protocols also provide the motivation for other protocols that we will discuss in subsequent sections.

A simple example of an authentication protocol is the one for opening a garage door remotely while driving up to the garage. This application doesn't require strong security properties (the adversary can always open the garage with a crowbar) but must be low cost. We want a protocol that can be implemented inexpensively so that the remote can be small, cheap, and battery-powered. For example, we want a protocol that involves only one-way communication, so that the remote control needs only a transmitter. In addition, the protocol should avoid complex operations so that the remote control can use an inexpensive processor.

The parties in the protocol are the remote control, a receiving device (the receiver), and an adversary. The remote control uses a wireless radio to transmit "open" messages to a receiver, which opens the garage door if an authorized remote control sends the message. The goal of the adversary is to open the garage without the permission of the owner of the garage.

The adversary is able to listen, replay, and modify the messages that the remote control sends to the receiver over the wireless medium. Of course, the adversary can also try to modify the remote control, but we assume that stealing the remote control is at least as hard as breaking into the garage physically, in which case there isn't much need to also subvert the remote control protocol.

The basic idea behind the protocol is for the receiver and the remote control to share a secret. The remote control sends the secret to the receiver and if it matches the receiver's secret, then the receiver opens the garage. If the adversary doesn't know the secret, then the adversary cannot open the garage. Of course, if the secret is transmitted over the air in clear text, the adversary can easily learn the secret, so we need to refine this basic idea.

A lightweight but correct protocol is as follows. At initialization, the remote control and receiver agree on some random number, which functions as a shared-secret key, and a random number, which is an initial counter value. When the remote control is pressed, it sends the following message:

remote ⇒ receiver: {*counter*, HASH(*key*, *counter*)}

and increments the counter.

When receiving the message, the receiver performs the following operations:

1. verify hash: compute HASH(*key*, *counter*) and compare result with the one in message
2. if hash verifies, then increment counter and open garage. If not, do nothing.

Because the holder of the remote control may have pressed the remote while out of radio range of the receiver, the receiver generally tries successive values of counter between its previous values N and, e.g., $N+100$ in step 1. If it finds that one of the values works, it resets the counter to that value and opens the garage.

This protocol meets our basic requirements. It doesn't involve two-way communication. It does involve computing a hash, but strong, inexpensive-to-compute hashes are readily available in the literature. Most important, the protocol is likely to provide a good enough level of security for this application.

The adversary cannot easily construct a message with the appropriate hash because the adversary doesn't know the shared-secret key. The adversary could try all possible values for the hash output (or all possible keys, if the keys are shorter than the hash output). If the hash output and key are sufficiently long, then this brute-force attack would take a long time. In addition, if necessary, the protocol could periodically re-initialize the key and counter.

The protocol is not perfect. For example, it has a replay attack. Suppose an impatient user presses the button on the remote control twice in close succession; the receiver responds to the first signal and doesn't hear the second signal. An adversary who happens to be recording the signals at the time can notice the two signals and guess that replaying the recording of the second signal may open the garage door, at least until the next time that legitimate user again uses the remote control. This weakness is probably acceptable.

The adversary can also launch a denial-of-service attack on the protocol (e.g., by jamming the radio signal remotely). The adversary, however, could also wreck the garage's door physically, which is simpler. The owner can also always get out of the car, walk to the garage, and use a physical key, so there is little motivation to deny access to the remote control.

Protocols such as the one described above are used in practice. For example, the Chamberlain garage door opener[1] uses a similar protocol with an extremely simple hash function (multiplication by 3 in a finite field) and it computes the hash over the previous hash, instead of over the counter and key. The simple hash probably provides a little less security, but it has the advantage that is cheap to implement. Other vendors seem to use similar protocols, but it is difficult to confirm because this industry has a practice of keeping its proprietary protocols secret, perhaps hoping to increase security through obscurity, which violates the *open design principle* and historically hasn't worked.

A version that is more secure than the garage-door protocol is used for authentication of users who want to download their email from an email service. Protocols for this application can assume two-way communication and exploit the idea of a challenge and a response. One widely used **challenge-response protocol** is the following[2]:

1. *Initialization*. M_1: Client ⇒ Server: (Opens a TCP connection)
2. *Challenge*. M_2: Server ⇒ Client: {"This is server *S* at 9:35:20.00165 EDT, 22 September 2006."}
3. *Response*. M_3: Client ⇒ Server: {"This is user U and the hash of M_2 and U's password is:", HASH{M_2, U's password}}

The server, which has its own copy of the secret password associated with user U, does its own calculation of HASH{M_2, U's password}, and compares the result with the second field of M_3. If they match, it considers the authentication successful and it proceeds to download the email messages.

The protocol isn't vulnerable to the person-in-the-middle attack of the garage protocol because the date and time in M_2 functions as a nonce, which is included in the hash of M_3. But addressing the person-in-the-middle attack requires two-way communication, which couldn't be used by the garage door opener.

Although this protocol is a step up over the garage door protocol, it has weaknesses too. It is vulnerable to brute-force attacks. The adversary can learn the username U from M_3. Then, later the adversary can connect to the mail server, receive M_2, guess a password for U, and see if the attempt is successful. Although each guess takes one round of the protocol and leaves an audit trail on the server, this might not stop a determined adversary.

A related weakness is that the protocol doesn't authenticate the server S, so the adversary can impersonate the server. The adversary tricks the client in connecting to a machine that the adversary controls (e.g., by spoofing a DNS response for the name S). When the client connects, the adversary sends M_2, and receives a correct M_3. Now the adversary can do an off-line brute-force attack on the user's password, without leaving an audit trail. The adversary can also provide the client with bogus email.

These weaknesses can be addressed. For example, instead of sending messages in the clear over a TCP connection, the protocol could set up a confidential, authenticated connection to the server using SSL/TLS (see Section 5.11). Then, the client and server can run the challenge-response protocol over this connection. The server can also send the email messages over the connection so that they are protected too. SSL/TLS authenticates all messages between a client and server and sends them encrypted. In addition, the client can require that the server provides a certificate with which the client can verify that the server is authentic. This approach could be further improved by using a client certificate instead of using U's password, which is a weak secret and vulnerable to dictionary attacks. Using SSL/TLS (either with or without client certificate) is common practice today.

A challenge-response protocol is a valuable tool only if it is implemented correctly. For example, a version of the UW IMAP server (a mail server that speaks the IMAP protocol and developed by the University of Washington) contained an implementation error that incorrectly specifies the conditions of successful authentication when using the challenge-response protocol described above. [3] After authenticating three times unsuccessfully using the challenge-response protocol, the server allowed the fourth attempt to succeed; the intention was to fail the fourth attempt immediately, but the implementers got the condition wrong. This error allowed an adversary to successfully authenticate as any user on the server after three attempts. Such programming errors are all too often the reason why the security of a system can be broken.

---

[1] Chamberlain Group, Inc. v. Skylink Techs., Inc., 292 F. Supp. 2d 1040 (N.D. Ill. 2003); aff'd 381 F.3d 1178 (U.S. App. 2004)

[2] Myers and M. Rose, *Post Office Protocol Version 3*, Internet Engineering Task Force Request For Comments (RFC) 1939, May 1996.

[3] United States Computer Emergency Readiness Team (US-CERT), *UW-imapd fails to properly authenticate users when using CRAM-MD5*, Vulnerability Note VU #702777, January 2005.

5.6.4: An Incorrect Key Exchange Protocol

The challenge-response protocol over SSL/TLS assumes SSL/TLS can set up a confidential and authenticated channel, which requires that the sender and receiver exchange keys securely over an untrusted network. It is possible to do such an exchange, but it must be done with care. We consider two different protocols for key exchange. The first protocol is incorrect, the second is (as far as anyone knows) correct. Both protocols attempt to achieve the same goal, namely for two parties to use a public-key system to negotiate a shared-secret key that can be used for encrypting. Both protocols have been published in the computer science literature and systems incorporating them have been built.

In the first protocol, there are three parties: Alice, Bob, and a certificate authority (CA). The protocol is as follows:

1. Alice ⇒ CA: {"Give me certificates for Alice and Bob"}
2. CA ⇒ Alice: {"Here are the certificates:", {Alice, A_{pub}, T}$_{CApriv}$, {Bob, B_{pub}, T}$_{CApriv}$}

In the protocol, the CA returns certificates for Alice and Bob. The certificates bind the names to public keys. Each certificate contains a timestamp T for determining if the certificate is fresh. The certificates are signed by the CA.

Equipped with the certificates from the CA, Alice constructs an encrypted message for Bob:

3. Alice ⇒ Bob: {"Here is my certificate and a proposed key:", {Alice, A_{pub}, T}$_{CApriv}$, {K_{AB}, T}$_{Apriv}$ }$^{B_{pub}}$

The message contains Alice's certificate and her proposal for a shared-secret key (K_{AB}). Bob can verify that A_{pub} belongs to Alice by checking the validity of the certificate using the CA's public key. The time-stamped shared-secret key proposed by Alice is signed by Alice, which Bob can verify using A_{pub} . The complete message is encrypted with Bob's public key. Thus, only Bob should be able to read K_{AB} .

Now Alice sends a message to Bob encrypted with K_{AB} :

$$4.\ \text{Alice} \Rightarrow \text{Bob: } \{\text{"Here is my message:", ... } T\}^{K_{AB}}$$

Bob should be able to decrypt this message, once he has read message 3 . So, what is the problem with this protocol? We suggest the reader pause for some time and try to discover the problem before continuing to read further. As a hint, note that Alice has signed only part of message 3 instead of the complete message. Recall that we should assume that some of the parties to the protocol may be adversaries.

The fact that there is a potential problem should be clear because the protocol fails the be explicit design principle. The essence of the protocol is part of message 3 , which contains her proposal for a shared-secret key:

$$\text{Alice} \Rightarrow \text{Bob: } \{K_{AB},\ T\}_{Apriv}$$

Alice tells Bob that K_{AB} is a good key for Alice and Bob at time T , but the names of Alice and Bob are missing from this part of message 3 . The interpretation of this segment of the message is dependent on the context of the conversation. As a result, Bob can use this part of message 3 to masquerade as Alice. Bob can, for example, send Charles a claim that he is Alice and a proposal to use K_{AB} for encrypting messages.

Suppose Bob wants to impersonate Alice to Charles. Here is what Bob does:

$$1.\ \text{Bob} \Rightarrow \text{CA: } \{\text{"Give me the certificates for Bob and Charles"}\}$$

$$2.\ \text{CA} \Rightarrow \text{Bob: } \{\text{"Here are the certificates:", } \{\text{Bob, } B_{pub},\ T'\}_{CApriv},\ \{\text{Charles, } C_{pub},\ T'\}_{CApriv}\}$$

$$3.\ \text{Bob} \Rightarrow \text{Charles: } \{\text{"Here is my certificate and a proposed key":, } \{\text{Alice, } A_{pub},\ T\}_{CApriv},\ \{K_{AB},\ T\}_{Apriv}\ \}^{C_{pub}}$$

Bob's message 3 is carefully crafted: he has placed Alice's certificate in the message (which he has from the conversation with Alice), and rather than proposing a new key, he has inserted the proposal, signed by Alice, to use K_{AB} , in the third component of the message.

Charles has no way of telling that Bob's message 3 didn't come from Alice. In fact, he thinks this message comes from Alice, since $\{K_{AB},\ T\}$ is signed with Alice's private key. So he (erroneously) believes he has key that is shared with only Alice, but Bob has it too. Now Bob can send a message to Charles:

$$1.\ \text{Bob} \Rightarrow \text{Charles: } \{\text{"Please send me the secret business plan. Yours truly, Alice."}\}^{K_{AB}}$$

Charles believes that Alice sent this message because he thinks he received K_{AB} from Alice, so he will respond. Designing security protocols is tricky! It is not surprising that Denning and Sacco*, the designers of this protocol, overlooked this problem when they originally proposed this protocol.

An essential assumption of this attack is that the adversary (Bob) is trusted for something because Alice first has to have a conversation with Bob before Bob can masquerade as Alice. Once Alice has this conversation, Bob can use this trust as a toehold to obtain information he isn't supposed to know.

The problem arose because of lack of explicitness. In this protocol, the recipient can determine the intended use of K_{AB} (for communication between Alice and Bob) only by examining the context in which it appears, and Bob was able to undetectably change that context in a message to Charles.

Another problem with the protocol is its lack of integrity verification. An adversary can replace the string "Here is my certificate and a proposed key" with any other string (e.g., "Here are the President's certificates") and the recipient would have no way of determining that this message is not part of the conversation. Although Bob didn't exploit this problem in his attack on Charles, it is a weakness in the protocol.

One way of repairing the protocol is to make sure that the recipient can always detect a change in context; that is, the recipient can always determine that the context is authentic. If Alice had signed the entire message 3 , and Charles had verified that message 3 was properly signed, that would ensure that the context is authentic, and Bob would not have been able to masquerade as

Alice. If we follow the explicitness principle, we should also change the protocol to make the key proposal itself explicit, by including the name of Alice and Bob with the key and timestamp and signing that entire block of data (i.e., $\{Alice, Bob, K_{AB}, T\}_{Apriv}$).

Making Alice and Bob explicit in the proposal for the key addresses the lack of explicitness, but doesn't address the lack of verifying the integrity of the explicit information. Only signing the entire message 3 addresses that problem.

You might wonder how it is possible that many people missed these seemingly obvious problems. The original protocol was designed in an era before the modular distinction between encrypting and signing was widely understood. It used encrypting of the entire message as an inexpensive way of authenticating the content; there are some cases where that trick works, but this is one where the trick failed. This example is another one of why the idea of obtaining authentication by encrypting is now considered to be a fundamentally bad practice.

---

* D. Denning and G. Sacco. Timestamps in key distribution protocols. *Communication of the ACM 24*, 8, pages 533–535, 1981.

5.6.5: Diffie-Hellman Key Exchange Protocol

The second protocol uses public-key cryptography to negotiate a shared-secret key. Before describing that protocol, it is important to understand the Diffie-Hellman key agreement protocol first. In 1976 Diffie and Hellman published the groundbreaking paper *New Directions in cryptography* *, which proposed the first protocol that allows two users to exchange a shared-secret key over an untrusted network without any prior secrets. This paper opened the floodgates for new papers in cryptography. Although there was much work behind closed doors, between 1930 and 1975 few papers with significant technical contributions regarding cryptography were published in the open literature. Now there are several conferences on cryptography every year.

The Diffie-Hellman protocol has two public system parameters: p, a prime number, and g, the generator. The generator g is an integer less than p, with the property that for every number n between 1 and $p-1$ inclusive, there is a power k of g such that $n = g^k (\text{modulo } p)$.

If Alice and Bob want to agree on a shared-secret key, they use p and g as follows. First, Alice generates a random value a and Bob generates a random value b. Both a and b are drawn from the set of integers $1, \ldots, p-2$. Alice sends to Bob: $g^a (\text{modulo } p)$, and Bob sends to Alice: $g^b (\text{modulo } p)$.

On receiving these messages, Alice computes $g^{ab} = (g^b)^a (\text{modulo } p)$, and Bob computes $g^{ba} = (g^a)^b (\text{modulo } p)$. Since $g^{ab} = g^{ba} = k$, Alice and Bob now have a shared-secret key k. An adversary hearing the messages exchanged between Alice and Bob cannot compute that value because the adversary doesn't know a and b; the adversary hears only p, g, g^a and g^b.

The protocol depends on the difficulty of calculating discrete logarithms in a finite field. It assumes that if p is sufficiently large, it is computationally infeasible to calculate the shared-secret key $k = g^{ab} (\text{modulo } p)$ given the two public values $g^a (\text{modulo } p)$ and $g^b (\text{modulo } p)$. It has been shown that breaking the Diffie-Hellman protocol is equivalent to computing discrete logarithms under certain assumptions.

Because the participants are not authenticated, the Diffie-Hellman protocol is vulnerable to a person-in-the-middle attack, similar to the one in Section 5.6.4. The importance of the Diffie-Hellman protocol is that it is the first example of a much more general cryptographic approach, namely the derivation of a shared-secret key from one party's public key and another party's private key. The second protocol is a specific instance of this approach, and addresses the weaknesses of the Denning-Sacco protocol.

---

* Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory IT–22*, 6 (November 1976), pages 644–654.

> Diffie and Hellman were the second inventors of public key cryptography (the first inventor, James H. Ellis, was working on classified projects for the British Government Communications Headquarters at the time, in 1970, and was not able to publish his work until 1987). This is the paper that introduced the idea to the unclassified world.

5.6.6: A Key Exchange Protocol Using a Public-Key System

The second protocol uses a Diffie-Hellman-like exchange to set up keys for encrypting and authentication. The protocol is designed to set up a secure channel from a client to a service in the SFS self-certifying file system [Suggestions for Further Reading 5.4.3]; a

similar protocol is also used in the Taos distributed operating system [Suggestions for Further Reading 5.3.2 11.3.2]. Web clients and servers use the more complex SSL/TLS protocol, which is described in Section 5.11.

The goal of the SFS protocol is to create a secure (authenticated and encrypted) connection between a client and a server that has a well-known public key. The client wants to be certain that it can authenticate the server and that all communication is confidential, but at the end of this protocol, the client will still be unauthenticated; an additional protocol will be required to identify and authenticate the client.

The general plan is to create two shared-secret nonce keys for each connection between a client and a server. One nonce key (K_{cs}) will be used for authentication and encryption of messages from client to server, the other (K_{sc}) for authentication and encryption of messages from server to client. Each of these nonce keys will be constructed using a Diffie-Hellman-like exchange in which the client and the server each contribute half of the key.

To start, the client fabricates two nonce half-keys, named $K_{c\text{-}cs}$ and $K_{c\text{-}sc}$, and also a nonce private and public key pair: T_{priv} and T_{pub} . T_{pub} is, in effect, a temporary name for this connection with this anonymous client.

The client sends to the service a request message to open a connection, containing T_{pub} , $K_{c\text{-}cs}$, and $K_{c\text{-}sc}$. The client encrypts the latter two with S_{pub} , the public key of the service:

$$\text{Client} \Rightarrow \text{service: } \{\text{"Here is a temporary public key } T_{pub} \text{ and two key halves encrypted with your public key:", } \{K_{c\text{-}cs}, K_{c\text{-}sc}\}^{S_{pub}}\}$$

The protocol encrypts $K_{c\text{-}cs}$, and $K_{c\text{-}sc}$ to protect against eavesdroppers. Since T_{pub} is a public key, there is no need to encrypt it.

The service can decrypt the keys proposed by the client with its private key, thus obtaining the three keys. At this point, the service has no idea who the client may be, and because the message may have been modified by an adversary, all it knows is that it has received three keys, which it calls T_{pub}' , $K_{c\text{-}cs}'$ and $K_{c\text{-}sc}'$, and which may or may not be the same as the corresponding keys fabricated by the client. If they are the same, then K_{c}' and $K_{c\text{-}sc}'$ are shared secrets known only to the client and the server.

The service now fabricates two more nonce half-keys, named $K_{s\text{-}cs}$ and $K_{s\text{-}sc}$. It sends a response to the client, consisting of these two half-keys encrypted with T_{pub}' :

$$\text{Service} \Rightarrow \text{client: } \{\text{"Here are two key halves encrypted with your temporary public key:", } \{K_{s\text{-}cs}, K_{s\text{-}sc}\}^{T_{pub}}\}$$

Unfortunately, even if $T_{pub}' = T_{pub}$, T_{pub} is public, so the client has no assurance that the response message came from the service; an adversary could have sent it or modified it. The client decrypts the message using T_{priv} , to obtain $K_{s\text{-}cs}'$ and $K_{s\text{-}sc}'$.

At this point in the protocol, the two parties have the following components in hand:

- Client: S_{pub} , T_{pub} , $K_{c\text{-}cs}$, $K_{c\text{-}sc}$, $K_{s\text{-}cs}'$, $K_{s\text{-}sc}'$
- Server: S_{pub} , T_{pub}' , $K_{c\text{-}cs}'$, $K_{c\text{-}sc}'$, $K_{s\text{-}cs}$, $K_{s\text{-}sc}$

Now the client calculates

- $K_{cs} \leftarrow \text{HASH } (\text{"client to server"}, S_{pub}, T_{pub}, K_{s\text{-}cs}', K_{c\text{-}cs})$
- $K_{sc} \leftarrow \text{HASH } (\text{"server to client"}, S_{pub}, T_{pub}, K_{s\text{-}sc}', K_{c\text{-}sc})$

and the server calculates

- $K_{cs}' \leftarrow \text{HASH } (\text{"client to server"}, S_{pub}, T_{pub}', K_{s\text{-}cs}, K_{c\text{-}cs}')$
- $K_{sc}' \leftarrow \text{HASH } (\text{"server to client"}, S_{pub}, T_{pub}', K_{s\text{-}sc}, K_{c\text{-}sc}')$

If all has gone well (that is, there have been no attacks), $K_{cs} = K_{cs}'$ and $K_{sc} = K_{sc}'$.

At this point there are three concerns:

1. An adversary may have replaced one or more components in such a way that the two parties do not have matching sets. If so, and assuming that the hash function is cryptographically secure, about half the bits of K_{cs} will not match K_{cs}' ; the same will be true for K_{sc} and K_{sc}' . K_{sc} and K_{cs} are about to be used as keys, so the parties will quickly discover any such mismatch.

2. An adversary may have replaced a component in such a way that both parties still have matching sets. But if we compare the components of K_{cs} and K_{cs}', we notice that at least one of the parties uses a personally chosen (unprimed) version of every component, and the adversary could not have changed that version, so there is no way for an adversary to make a matching change for both parties.
3. An adversary may have been able to discover all of the components and thus be able to calculate K_{sc}, K_{cs}, or both. But the values of $K_{c\text{-}cs}$ and $K_{c\text{-}sc}$ were created by the client and encrypted under S_{pub} before sending them to the service, so only the client and the service know those two components.

If $K_{cs} = K_{cs}'$ and $K_{sc} = K_{sc}'$, the two parties have two keys that only they know, and only the service and this client could have calculated them. In addition, because they are calculated using $K_{s\text{-}sc}$, $K_{c\text{-}sc}$, $K_{s\text{-}cs}$, and $K_{c\text{-}cs}$, which are nonces created just for this exchange, both parties are ensured that K_{cs} and K_{sc} are fresh. In summary, K_{cs} and K_{sc} are newly generated shared secrets.

The protocol proceeds with the client generating a shared-secret authentication key $K_{ssa\text{-}cs}$ and a shared-secret encryption key $K_{sse\text{-}cs}$ from K_{cs}, perhaps by simply using the first half of K_{cs} as $K_{ssa\text{-}cs}$ and the second half as $K_{sse\text{-}cs}$. The client can now prepare and send an encrypted and authenticated request to the server:

$$\{M\}^{K_{sse\text{-}cs}}_{K_{ssa\text{-}cs}}$$

The server generates the same shared-secret authentication key $K_{ssa\text{-}cs}$ and a shared-secret encryption key $K_{sse\text{-}cs}$ from K_{cs}' and it can now try to decrypt and authenticate M. If the authentication succeeds, the server knows that $K_{cs} = K_{cs}'$.

The server performs a similar procedure based on K_{sc} for its response. If the client successfully authenticates the response the client knows $K_{sc} = K_{sc}'$. The fact that it received a response tells it that the server successfully verified that $K_{cs} = K_{cs}'$.

From now on, the client knows that it is talking to the server associated with S_{pub}, and the connection is confidential. The server knows that the connection is confidential and that all messages are coming from the same source, but it does not know what that source is. If the server wants to know the source, it can ask and, for example, demand a password to authenticate the identity that the source claims.

To ensure forward secrecy, the client periodically repeats the whole protocol periodically. At regular intervals (e.g., every hour), the client discards the temporary keys T_{pub} and T_{priv}, generates a new public key T_{pub} and private key T_{priv}, and runs the protocol again.

5.6.7: Summary

This section described several security protocols to obtain different objectives. We studied a challenge-response protocol to open garage doors. We studied an incorrect protocol to set up a secure communication channel between two parties. Then, we studied a correct protocol for that same purpose that provides confidentiality but doesn't authenticate the participants. Finally, we studied a protocol for setting up a secure communication channel that provides both confidentiality and authenticity. Protocols for setting up secure channels become important whenever the participants are separated by a network. Section 5.11 describes a protocol for setting up secure channels in the World-Wide Web.

Many systems have additional security requirements, and therefore may need protocols with different features. For example, a system that provides anonymous email must provide an authenticated and confidential communication channel between two parties with the property that the receiver knows that a message came from the same source as previous messages and that nobody else has read the message, but must also hide the identity of the sender from the receiver. Such a system requires a more sophisticated design and protocols because hiding the identity of the sender is a difficult problem. The receiver may be able to learn the Internet address from which some of the messages were sent or may be able to observe traffic on certain communication links; to make anonymous email resist such analysis requires elaborate protocols that are beyond the scope of this text, but see, for example, Chaum's paper for a solution [Suggestions for Further Reading 5.5.8]. Security protocols are also an active area of research and researchers continuously develop novel systems and protocols for new scenarios or for particular challenging problems such as electronic voting, which may require keeping the identity of the voter secret, preventing a voter from voting more than once, allowing the voter to verify that the vote was correctly recorded, and permitting recounts. The interested reader is encouraged to consult the professional literature for developments.

This page titled 5.6: Security Protocols is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

5.7: Authorization-Controlled Sharing

Some data must stay confidential. For example, users require that their private authentication key stay confidential. Users wish to keep their passwords and credit card numbers confidential. Companies wish to keep the specifics of their upcoming products confidential. Military organizations wish to keep attack plans confidential.

The simplest way of providing confidentiality of digital data is to separate the programs that manipulate the data. One way of achieving that is to run each program and its associated data on a separate computer and require that the computers cannot communicate with each other.

The latter requirement is usually too stringent: different programs typically need to share data and strict separation makes this sharing impossible. A slight variation, however, of the strict separation approach is used by military organizations and some businesses. In this variation, there is a trusted network and an untrusted network. The trusted network connects trusted computers with sensitive data, and perhaps uses encryption to protect data as it travels over the network. By policy, the computers on the untrusted network don't store sensitive data, but might be connected to public networks such as the Internet. The only way to move data between the trusted and untrusted network is manual transfer by security personnel who can deny or authorize the transfer after a careful inspection of the data.

For many services, however, this slightly more relaxed version of strict isolation is still inconvenient because users need to have the ability to share more easily but keep control over what is shared and with whom. For example, users may want share files on a file server, but have control over whom they authorize to have access to what files. As another example, many users acquire programs created by third parties, run them on their computer, but want to be assured that their confidential data cannot be read by these untrusted programs. This section introduces authorization systems that can support these requirements.

5.7.1: Authorization Operations

We can distinguish three primary operations in authorization systems:

- **authorization**. This operation grants a principal permission to perform an operation on an object.
- **mediation**. This operation checks whether or not a principal has permission to perform an operation on a particular object.
- **revocation**. This decision removes a previously-granted permission from a principal.

The agent that makes authorization and revocation decisions is known as an **authority**. The authority is the principal that can increase or decrease the set of principals that have access to a particular object by granting or revoking respectively their permissions. In this chapter we will see different ways how a principal can become an authority.

5.7.2: The Simple Guard Model

The simple guard model is based on an **authorization matrix**, in which principals are the rows and objects are the columns. Each entry in the matrix contains the permissions that a principal has for the given object. Typical permissions are read access and write access. When the service receives a request for an object, the guard verifies that the requesting principal has the appropriate permissions in the authorization matrix to perform the requested operation on the object, and if so, allows the request.

The authority of an object is the principal who can set the permissions for each principal, which raises the question how a principal can become an authority. One common design is that the principal who creates an object is automatically the authority for that object. Another option is to have an additional permission in each entry of the authorization matrix that grants a principal permission to change the permissions. That is, the permissions of an object may also include a permission that grants a principal authority to change the permissions for the object.

When a principal creates a new object, the access-control system must determine which is the appropriate authority for the new object and also what initial permissions it should set. Discretionary access-control systems make the creator of the object the authority and allow the creator to change the permission entries at the creator’s discretion. The creator can specify the initial permission entries as an argument to the create operation or, more commonly, use the system’s default values. Non-discretionary access-control systems don't make the creator the authority but chose an authority and set the permission entries in some other way, which the creator cannot change at the creator's discretion. In the simple guard model, access control is usually discretionary. We will return to nondiscretionary access control in Section 5.7.5, below.

There are two primary instances of the simple guard model: list systems, which are organized by column, and ticket systems, which are organized by row. The primary way these two systems differ is who stores the authorization matrix: the list system stores columns in a

place that the guard can refer to, while the ticket system stores rows in a place that principals have access to. This difference has implications on the ease of revocation. We will discuss ticket systems, list systems, and systems that combine them, in turn.

5.7.2.1: The Ticket System

In the **ticket system**, each guard holds a ticket for each object it is guarding. A principal holds a separate ticket for each different object the principal is authorized to use. One can compare the set of tickets that the principal holds to a ring with keys. The set of tickets that principal holds determines exactly which objects the principal can obtain access to. A ticket in a ticket-oriented system is usually called a **capability**.

To authorize a principal to have access to an object, the authority gives the principal a matching ticket for the object. If the principal wishes, the principal can simply pass this ticket to other principals, giving them access to the object.

To revoke a principal's permissions, the authority has to either hunt down the principal and take the ticket back, or change the guard's ticket and reissue tickets to any other principals who should still be authorized. The first choice may be hard to implement; the second may be disruptive.

5.7.2.2: The List System

In the **list system**, revocation is less disruptive. In the list system, each principal has a token identifying the principal (e.g., the principal's name) and the guard holds a list of tokens that correspond to the set of principals that the authority has authorized. To mediate, a guard must search its list of tokens to see if the principal's token is present. If the search for a match succeeds, the guard allows the principal access; if not, the guard denies that principal access. To revoke access, the authority removes the principal's token from the guard's list. In the list system, it is also easy to perform audits of which principals have permission for a particular object because the guard has access to the list of tokens for each object. The list of tokens is usually called an **access-control list (ACL)**.

5.7.2.3: Tickets versus Lists, and Agencies

Table 5.7.1: Comparison of access control systems

System	Advantage	Disadvantage
Ticket	• Quick access check • Tickets can be passed around	•

System	Advantage	Disadvantage
		•

System	Advantage	Disadvantage
List	• Revocation is easy • Audit possible	•

System	Advantage	Disadvantage
Agency	• List available	•

Ticket and list systems each have advantages over the other. Table 5.7.1 summarizes the advantages and disadvantages. The differences in the ticket and list system stem primarily from who gathers, stores, and searches the authorization information. In the ticket system, the responsibility for gathering, storing, and searching the tickets rests with the principal. In the list system, responsibility for gathering, storing, and searching the tokens on a list rests with the guard. In most ticket systems, the principals store the tickets and they can pass tickets to other principals without involving the guard. This property makes sharing easy (no interaction with the authority required), but makes it hard for an authority to revoke access and for the guard to prepare audit trails. In the list system, the guard stores the tokens and they identify principals, which makes audit trails possible; on the other hand, to grant another principal access to an object requires an interaction between the authority and the guard.

The tokens in the ticket and list systems must be protected against forgery. In the ticket system, tickets must be protected against forgery. If an adversary can cook up valid tickets, then the adversary can obtain access to any object. In the list system, the token identifying the principal and the access control list must be protected. If an adversary can cook up valid principal identifiers and change the access control list at will, then the adversary can have access to any object. Since the principal identifier tokens and access control lists are in the storage of the system, protecting them isn't too hard. Ticket storage, on the other hand, may be managed by the user, and in that case protecting the tickets requires extra machinery.

A natural question to ask is if it is possible to get the best of both ticket and list systems. An **agency** can combine list and ticket systems by allowing one to switch from a ticket system to a list system, or vice versa. For example, at a by-invitation-only conference, upon your arrival, the organizers may check your name against the list of invited people (a list system) and then hand you a batch of coupons for lunches, dinners, etc. (a ticket system).

5.7.2.4: Protection Groups

Cases often arise where it would be inconvenient to list by name every principal who is to have access to each of a large number of objects that have identical permissions, either because the list would be awkwardly long, or because the list would change frequently, or to ensure that several objects have the same list. To handle this situation, most access control list systems implement **protection groups**, which are principals that may be used by more than one user. If the name of a protection group appears in an access control list for an object, all principals who are members of that protection group share the permissions for that object.

A simple way to implement protection groups is to create an access control list for each group, consisting of a list of tokens representing the individual principals who are authorized to use the protection group's principal identifier. When a user logs in, the system authenticates the user, for example, by a password, and identifies the user's token. Then, the system looks up the user's token on each group's access control list and gives the user the group token for each protection group the user belongs to. The guard can then mediate access based on the user and group tokens.

5.7.3: Example: Access Control in UNIX

The previous section described access control based on a simple guard model in the abstract. This section describes a concrete access control system, namely the one used by UNIX. UNIX was originally designed for a computer shared among multiple users, and therefore had to support access control. The Network File System (NFS) extends the UNIX file system to shared file servers, reinforcing the importance of access control, since without access control any user has access to all files. Early versions of the UNIX system didn't provide networking and didn't support servers well; modern UNIX systems, however, do, which further reinforces the need of security. For this reason, this section mostly describes the core access control features that one can find in a modern UNIX system, which are based on the features found in early UNIX systems. For the more advanced and latest features the reader is encouraged to consult the professional literature.

One of the benefits of studying a concrete example is that it makes the clear the importance of the dynamics of use in an access control system. How are running programs associated with principals? How are access control lists changed? Who can create new principals? How does a system get initialized? How is revocation done? From these questions it should be clear that the overall security of a computer system is to a large part based on how carefully the dynamics of use have been thought through.

5.7.3.1: Principals in UNIX

The principals in UNIX are users and groups. Users are named by a string of characters. A user name with some auxiliary information is stored in a file that is historically called the password file. Because it is inconvenient for the kernel to use character strings for user names, it uses fixed-length integer names (called UIDs). The UID of each user is stored along with the user name in a file called colloquially the password file (/etc/passwd). The password file usually contains other information for each user too; for example, it contains the name of the program that a users wants the system to run when the user logs in.

A group is a protection group of users. Like users, groups are named by a string of characters. The group file ("/etc/group") stores all groups. For each group it stores the group name, a fixed-length integer name for the group (called the GID), and the user names (or UIDs depending on which version of UNIX) of the users who are a member of the group. A user can be in multiple groups; one of these group is the user's default group. The name of the default group is stored in the user's entry in the password file.

The principal **superuser** is the one used by system administrators and has full authority; the kernel allows the superuser to change any permissions. The superuser is also called **root**, and has the UID 0.

A system administrator usually creates several service principals to run services instead of for running them with superuser authority. For example, the principal named "www" runs the Web server in a typical UNIX configuration. The reason to do so is that if the server is compromised (e.g., through a buffer overrun attack), then the adversary acquires only the privileges of the principal www, and not those of the superuser.

5.7.3.2: ACLs in UNIX

UNIX represents all shared objects (files, devices, etc.) as files, which are protected by the UNIX kernel (the guard). All files are manipulated by programs, which act on behalf of some principal. To isolate programs from one another, UNIX runs each program in its own address space with one or more threads (called a process in UNIX). All mediation decisions can be viewed as whether or not a particular process (and thus principal) should be allowed to have access to a particular file. UNIX implements this mediation using ACLs.

Each file has an owner, a principal that is the authority for the file. The UID of the owner of a file is stored in a file's inode (index node). Each file also has an owning group, designated by a GID stored in the file's inode. When a file is created its UID is the UID of the principal who created the file and its GID is the GID of principal's default group. The owner of a file can change the owner and group of the file.

The inode for each file also stores an ACL. To avoid long ACLs, UNIX ACLs contain only 3 entries: the UID of the owner of the file, a group identifier (GID), and other. "Other" designates all users with UIDs and GIDs different from the ones on the ACL.

This design is sufficient for a time-sharing system for a small community, where all one needs is some privacy between groups. But when such a system is attached to the Internet, it may run services such as a Web service that provide access to certain files to any user on the Internet. The Web server runs under some principal (e.g., "www"). The UID associated with that principal is included in the "other" category, which means that "other" can mean anyone in the entire Internet. Because allowing access to the entire world may be problematic, Web servers running under UNIX usually implement their own access restrictions in addition to those enforced by the ACL. (But recall the discussion of the TCB in Section 5.2.7. This design drags the Web server inside the TCB.) For reasons such as these, file servers that are designed for a larger community or to be attached to the Internet, such as the Andrew File System *, support full-blown ACLs.

Per ACL entry, UNIX keeps several permissions: `READ` (if set, read operations are allowed), `WRITE` (if set, write operations are allowed), and `EXECUTE` (if set, the file is allowed to be executed as a program). So, for example, the file "y" might have an ACL with UID 18, GID 20, and permissions "rwxr-xr--". This information says the owner (UID 18) is allowed to read, write, and execute file "y", users belonging to group 20 are allowed to read and execute file "y", and all other users are allowed only read access. The owner of a file has the authority to change the permission on the file.

The initial owner and permission entries of a new file are set to the corresponding values of the process that created the file. What the default principal and permissions are of a process is explained next.

---

* John H. Howard et al. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6*, 1 (February 1988), pages 51–81.

> This paper describes experience with a prototype of the Andrew network file system for a campus network and shows how the experience motivated changes in the design. The Andrew file system had strong influence on version 4 of NFS.

5.7.3.3: The Default Principal and Permissions of a Process

The kernel stores for a process the UID and the GIDs of the principal on whose behalf the process is running. The kernel also stores for a process the default permissions for files that that process may create. A common default permission is write permission for the owner, and read permission for the owner, group, and other. A process can change its default permissions with a special command (called `UMASK`).

By default, a process inherits the UID, GIDs, and default permissions of the process that created it. However, if the `SETUID` permission of a file is set on—a bit in a file's inode—the process that runs the program acquires the UID of the principal that owns the file storing the program. Once a process is running, a process can invoke the `SETUID` supervisor call to change its UID to one with fewer permissions.

The `SETUID` permission of a file is useful for programs that need to increase their privileges to perform privileged operations. For example, an email delivery program that receives an email for a particular user must be able to append the mail to the user's mailbox. Making the target mailbox writable for anyone would allow any user to destroy another user's mailbox. If a system administrator sets the `SETUID` permission on the mail delivery program and makes the program owned by the superuser, then the mail program will run with superuser privileges. When the program receives an email for a user, the program changes its UID to the target user's, and can append the mail to the user's mailbox. (In principle the delivery program doesn't have to change to the target's UID, but changing the UID is better practice than running the complete program with superuser privileges. It is another example of the *principle of least privilege*.)

Another design option would be for UNIX to set the ACL on the mailbox to include the principal of the e-mail deliver program. Unfortunately, because UNIX ACLs are limited to the user, group, and other entries, they are not flexible enough to have an entry for a specific principal, and thus the `SETUID` plan is necessary. The `SETUID` plan is not ideal either, however, because there is a temptation for application designers to run applications with superuser privileges and never drop them, violating the *principle of least privilege*. In retrospect, UNIX's plan for security is weak, and the combination of buffer-overrun attacks and applications running with too much privilege has led to many security breaches. To design an application to run securely on UNIX requires much careful thought and sophisticated use of UNIX.

With the exception of the superuser, only the principal on whose behalf a process is running can control a process (e.g., stop it). This design makes it difficult for an adversary who successfully compromised one principal to damage other processes that act on behalf of a different principal.

5.7.3.4: Authenticating Users

When a UNIX computer starts, it boots the kernel. The kernel starts the first user program (called init in UNIX) and runs it with the superuser authority. The init program starts, among other things, a login program, which also executes with the superuser authority. Users type in their user name and a password to a login program. When a person types in a name and password, the login program hashes the password using a cryptographic hash (as was explained in Section 5.3.3) and compares it with the hash of the password that it has on file that corresponds to the user name the person has claimed. If they match, the login program looks up the UID, GIDs, and the starting program for that user, uses `SETUID` to change the UID of the login program to the user's UID, and runs the user's starting program. If hashes don't match, the login program denies access.

As mentioned earlier, the user name, UID, default GID, and other information are stored in the password file (named "/etc/passwd"). At one time, hashed passwords were also stored in the password file. But because the other information is needed by many programs, including programs run by other users, most systems now store the hashed password in a separate file called the "shadow file" that is accessible only to the superuser. Storing the passwords in a limited access file makes it harder for an adversary to mount a dictionary attack against the passwords. Users can change their password by invoking a `SETUID` program that can write the shadow file. Storing public user information in the password file and sensitive hashed passwords in the shadow file with more restrictive permissions is another example of applying the *principle of least privilege*.

5.7.3.5: Access Control Check

Once a user is logged in, subsequent access control is performed by the kernel based on UIDs and GIDs of processes, using a list system. When a process invokes `OPEN` to use a file, the process performs a system call to enter the kernel. The kernel looks up the UID and GIDs for the process in its tables. Then, the kernel performs the access check as follows:

1. If the UID of the process is 0 (superuser), the process has the necessary permissions by default.
2. If the UID of the process matches the UID of the owner of the file, the kernel checks the permissions in the ACL entry for owner.
3. If UIDs do not match, but if one of the process's GIDs match the GID of the file, the kernel checks the permissions in the ACL entry for group.
4. If the UID and GIDs do not match, the kernel checks the permissions in the ACL entry for "other" users.

If the process has the appropriate permission, the kernel performs the operation; otherwise, it returns a permission error.

5.7.3.6: Running Services

In addition to starting the login program, the init program usually starts several services (e.g., a Web server, an email server, a X Windows System server, etc.). The services often start run with the privileges of the superuser principal, but switch to a service principal using `SETUID` . For example, a well-designed Web server changes its UID from the superuser principal to the www principal after it has done the few operations that require superuser privileges. To ensure that these services have limited access if an adversary compromises one of them, the system administrator sets file permissions so that, for example, the principal named www has permission to access only the files it needs. In addition, a Web server designed with security in mind will also use the `CHROOT` call so that it can name only the files in its corner of the file system. These measures ensure that an adversary can do only restricted harm when compromising a service. These measures are examples of both the paranoid design attitude and of the *principle of least privilege*.

5.7.3.7: Summary of UNIX Access Control

The UNIX login program can be viewed as an access control system following the pure guard model that combines authentication of users with mediating access to the computer to which the user logs in. The guard is the login program. The object is the UNIX system. The principal is the user. The ticket is the password, which is protected using a cryptographic hash function. If the tickets match, access is allowed; otherwise, access is denied. We can view the whole UNIX system as an agent system. It switches from a simple ticket-based guard system (the login program) to a list-oriented system (the kernel and file system). UNIX thus provides a comprehensive example of the simple guard model. In the next two sections we investigate two other models for access control.

5.7.4: The Caretaker Model

The caretaker model generalizes the simple guard model. It is the object-oriented version of the simple guard model. The simple guard model checks permissions for simple methods such as read, write, and execute. The caretaker model verifies permissions for arbitrary

methods. The caretaker can enforce arbitrary constraints on access to an object, and it may interpret the data stored in the object to decide what to do with a given request.

Example access-control systems that follow the caretaker model are:

- A bank vault that can be opened at 5:30 p.m., but not at any other time.
- A box that can be opened only when two principals agree.
- Releasing salary information only to principals who have a higher salary.
- Allowing the purchase of a book with a credit card only after the bank approves the credit card transaction.

The hazard in the caretaker model is that the program for the caretaker is more complex than the program for the guard, which makes it easy to make mistakes and leave loopholes to be exploited by adversaries. Furthermore, the specification of what the caretaker's methods do and how they interact with respect to security may be difficult to understand, which may lead to configuration errors. Despite these challenges, database systems typically support the caretaker model to control access to rows and columns in tables.

5.7.5: Non-Discretionary Access and Information Flow Control

The description of authorization has so far rested on the assumption that the principal that creates an object is the authority. In the UNIX example, the owner of a file is the authority for that file; the owner can give all permissions including the ability to change the ACL, to another user.

This authority model is **discretionary**: an individual user may, at the user's own discretion, authorize other principals to obtain access to the objects the user creates. In certain situations, discretionary control may not be acceptable and must be limited or prohibited. In this case, the authority is not the principal who created the object, but some other principal. For example, the manager of a department developing a new product line may want to **compartmentalize** the department's use of the company computer system to ensure that only those employees with a need to know have access to information about the new product. The manager thus desires to apply the *least privilege principle*. Similarly, the marketing manager may wish to compartmentalize all use of the company computer for calculating product prices, since pricing policy may be sensitive.

Either manager may consider it unacceptable that any individual employee within the department can abridge the compartments merely by changing an access control list on an object that the employee creates. The manager has a need to limit the use of discretionary controls by the employees. Any limits the manager imposes on authorization are controls that are out of the hands of the employees, and are viewed by them as **non-discretionary.**

Similar constraints are imposed in military security applications, in which not only isolated compartments are required, but also nested sensitivity levels (e.g., unclassified, confidential, secret, and top secret) that must be modeled in the authorization mechanics of the computer system. Commercial enterprises also use non-discretionary controls. For example, a non-disclosure agreement may require a person for the rest of the person’s life not to disclose the information that the agreement gave the person access to.

Non-discretionary controls may need to be imposed in addition to or instead of discretionary controls. For example, the department manager may be prepared to allow the employees to adjust their access control lists any way they wish, within the constraint that no one outside the compartment is ever given access. In that case, both non-discretionary and discretionary controls apply.

The reason for interest in non-discretionary controls is not so much the threat of malicious insubordination as the need to safely use complex and sophisticated programs created by programmers who are not under the authority's control. A user may obtain some code from a third party (e.g., a Web browser extension, a software upgrade, a new application) and if the supplied program is to be useful, it must be given access to the data it is to manipulate or interpret (see Figure 5.7.1). But unless the downloaded program has been completely audited, there is no way to be sure that it does not misuse the data (for example, by making an illicit copy and sending it somewhere) or expose the data either accidentally or intentionally. One way to prevent this kind of security violation would be to forbid the use of untrusted third-party programs, but for most organizations the requirement that all programs be locally written (or even thoroughly audited) would be an unbearable economic burden. The alternative is confinement of the untrusted program. That is, the untrusted program should run on behalf of some principal in a compartment containing the necessary data, but should be constrained so that it cannot authorize sharing of anything found or created in that compartment with other compartments.

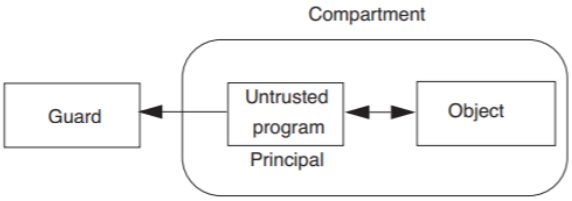


Figure 5.7.1: Confining a program within a compartment.

Complete elimination of discretionary controls is easy to accomplish. For example, one could arrange that the initial value for the access control list of all newly created objects not give "ACL-modification" permission to the creating principal (under which the downloaded program is running). Then the downloaded program could not release information by copying it into an object that it creates and then adjusting the access control list on that object. If, in addition, all previously existing objects in the compartment of the downloaded program do not permit that principal to modify the access control list, the downloaded program would have no discretionary control at all.

An interesting requirement for a non-discretionary control system that implements isolated compartments arises whenever a principal is authorized to have access to two or more compartments simultaneously, and some data objects may be labeled as being simultaneously in two or more compartments (e.g., pricing data for a new product may be labeled as requiring access to the "pricing policy" compartment as well as the "new product line" compartment). In such a case it would seem reasonable that, before permitting reading of data from an object, the control mechanics should require that the set of compartments of the object being referenced be a subset of the compartments to which the accessor is authorized.

A more stringent interpretation, however, is required for permission to write, if downloaded programs are to be confined. Confinement requires that the program be constrained to write only into objects that have a compartment set that is a subset of that of the program itself. If such a restriction were not enforced, a malicious downloaded program could, upon reading data labeled for both the "pricing policy" and the "new product line" compartments, make a copy of part of it in an object labeled only "pricing policy," thereby compromising the “new product line’’ compartment boundary. A similar set of restrictions on writing can be expressed for sensitivity levels. A set of such restrictions is known as rules for information flow control.

5.7.5.1: Information Flow Control Example

To make information flow control more concrete, consider a company that has information divided in two compartments:

1. financial (e.g., product pricing)
2. product (e.g., product designs)

Each file in the computer system is labeled to belong to one of these compartments. Every principal is given a clearance for one or both compartments. For example, the company's policy might be as follows: the company's accounts have clearance for reading and writing files in the financial compartment, the company's engineers have clearance for reading and writing files in the product compartment, and the company's product managers have clearance for reading and writing files in both compartments.

The principals of the system interact with the files through programs, which are untrusted. We want ensure that information flows only to the company's policy. To achieve this goal, every thread records the labels of the compartments for which the principal is cleared; this clearance is stored in $T_{\texttt{labelsseen}}$. Furthermore, the system remembers the maximum compartment label of data the thread has seen, $T_{\texttt{maxlabels}}$. Now the information flow control rules can be implemented as follows. The read rule is:

- Before reading an object with labels $O_{\texttt{labels}}$, check that $O_{\texttt{labels}} \subseteq T_{\texttt{maxlabels}}$.
- If so, set $T_{\texttt{labelsseen}} \leftarrow T_{\texttt{labelsseen}} \cup C_{\texttt{labels}}$, and allow access.

This rule can be summarized by "no read up." The thread is not allowed to have access to information in compartments for which it has no clearance.

The corresponding write rule is:

- Allow a write to an object with clearance $O_{\texttt{labels}}$ only if $T_{\texttt{labelsseen}} \subseteq O_{\texttt{labels}}$

This rule could be called "no write down." Every object written by a thread that read data in compartments L must be labeled with L's labels. This rule ensures that if a thread $\top$ has read information in a compartment other than the ones listed in L, then that information doesn't leak into the object O.

These information rules can be used to implement a wide range of policies. For example, the company can create more compartments, more principals, or modify the list of compartments a principal has clearance for. These changes in policy don't require changes in the information flow rules. This design is another example of the principle *separate mechanism from policy*.

Sometimes there is a need to move an object from one compartment to another because, for example, the information in the object isn't confidential anymore. Typically downgrading of information (**declassification** in the security jargon) must be done by a person who inspects the information in the object, since a program cannot exercise judgment. Only a human can establish that information to be declassified is not sensitive.

This example sketches a set of simple information flow control rules. In real system systems more complex information flow rules are needed, but they have a similar flavor. The United States National Security Agency has a strong interest in computer systems with information flow control, as do companies that have sensitive data to protect. The Department of Defense has a specification for what these computer systems should provide (this specification is part of a publication known as the Orange Book*, which classifies systems according to their security guarantees). It is possible that information flow control will find other usages than in high-security systems, as the problems with untrusted programs become more prevalent in the Internet, and sophisticated confinement is required.

---

* U.S.A. Department of Defense, *Department of Defense trusted computer system evaluation criteria*, Department of Defense standard 5200, December 1985.

5.7.5.2: Covert Channels

Complete confinement of a program in a system with shared resources is difficult, or perhaps impossible, to accomplish, since the program may be able to signal to other users by strategies more subtle than writing into shared objects. Computer systems with shared resources always contain covert channels, which are hidden communication channels through which information can flow unchecked. For example, two threads might conspire to send bits by the logical equivalent of "banging on the wall." See Section 5.12.10.1 for a concrete example and see Problem Set 26 for an example that literally involves banging. In practice, just finding covert channels is difficult. Blocking covert channels is an even harder problem: there are no generic solutions.

This page titled 5.7: Authorization-Controlled Sharing is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

5.8: Reasoning About Authentication (Advanced Topic)

The security model has three key steps that are executed by the guard on each request: authenticating the user, verifying the integrity of the request, and determining if the user is authorized. Authenticating the user is typically the most difficult of the three steps because the guard can establish only that the message came from the same origin as some previous message. To determine the principal that is associated with a message, the guard must establish that it is part of a chain of messages that often originated in a message that was communicated by physical rendezvous. That physical rendezvous securely binds the identity of a real-world person with a principal.

The authentication step is further complicated because the messages in the chain might even come from different principals, as we have seen in some of the security protocols in Section 5.6. If a message in the chain comes from a different principal and makes a statement about another principal, we can view the message as one principal speaking for another principal. To establish that the chain of messages originated from a particular real-world user, the guard must follow a chain of principals.

Consider a simple security protocol, in which a certificate authority signs certificates, associating authentication keys with names (e.g., "key K_{pub} belongs to the user named X"). If a service receives this certificate together with a message M for which `VERIFY (`M`,` K_{pub}`)` returns `ACCEPT` , then the question is if the guard should believe this message originated with "X". The answer is no until the guard can establish the following facts:

1. The guard knows that a message originated from a principal who knows a private authentication key K_{priv} because the message verified with K_{pub} .
2. The certificate is a message from the certification authority telling the guard that the authentication key K_{pub} is associated with user "X". (The guard can tell that the certificate came from the certificate authority because the certificate was signed with the private authentication key of the authority and the guard has obtained the public authentication key of the authority through some other chain of messages that originated in physical rendezvous.)
3. The certification authority speaks for user "X". The guard may believe this assumption, if the guard can establish two facts:
   - User "X" says the certificate authority speaks for "X". That is, user "X" delegated authority to the certificate authority to speak on behalf of "X". If the guard believes the certificate authority carefully minted a key for "X" that speaks for only "X" and verified the identity of "X", then the guard may consider this belief a fact.
   - The certificate authority says K_{pub} speaks for user "X". If the guard believes that the certificate authority carefully minted a key for "X" that speaks for only "X" and verified the identity of "X", then the guard may consider this belief a fact.

With these facts, the guard can deduce that the origin of the first message is user "X" as follows:

1. If user "X" says that the certificate authority speaks on behalf of "X", then the guard can conclude that the certificate authority speaks for "X" because "X" said it.
2. If we combine the first conclusion with the statement that the certificate authority says that "X" says that K_{pub} speaks for "X", then the guard can conclude that "X" says that K_{pub} speaks for "X".
3. If "X" says that K_{pub} speaks for "X", then the guard can conclude that K_{pub} speaks for "X" because "X" said it.
4. Because the first message verified with K_{pub} , the guard can conclude that the message must have originated with user “X”.

In this section, we will formalize this type of reasoning using a simple form of what is called **authentication logic**, which defines more precisely what "speaks for" means. Using that logic we can establish the assumptions under which a guard is willing to believe that a message came from a particular person. Once the assumptions are identified, we can decide if the assumptions are acceptable, and, if the assumptions are acceptable, the guard can accept the authentication as valid and go on to determine if the principal is authorized.

5.8.1: Authentication Logic

Burrows-Abadi-Needham (BAN) authentication logic is a particular logic to reason about authentication systems. We give an informal and simplified description of the logic and its usage. If you want to use it to reason about a complete protocol, read *Authentication in Distributed Systems: Theory and Practice* [Suggestions for Further Reading 5.3.1].

Consider the following example. Alice types at her workstation "Send me the quiz" (see Figure 5.8.1). Her workstation A sends a message over the wire from network interface 14 to network interface 5, which is attached to the file service machine F, which runs the file service. The file service stores the object "quiz".

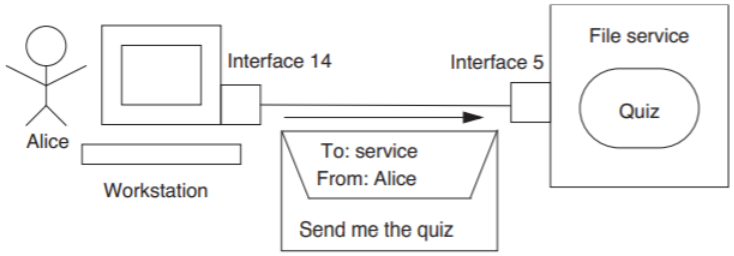


Figure 5.8.1: Authentication example.

What the file service needs to know is that "Alice **says** send quiz". This phrase is a statement in the BAN authentication logic. This statement "A **says** B" means that agent A originated the request B. Informally, "A **says** B" means we have determined somehow that A actually said B. If we were within earshot, "A **says** B" is an axiom (we directly heard A say it!), but if we only know that "A **says** B" indirectly ("through hearsay"), we need to use additional reasoning, and perhaps make some other assumptions before we believe it. Unfortunately, the file system knows only that network interface F.5 (that is, network interface 5 on machine F) said Alice wants the quiz sent to her. That is, the file system knows "network interface F.5 **says** (Alice **says** send the quiz)". So "Alice **says** send the quiz" is only hearsay at the moment. The question is, can we trust network interface F.5 to tell the truth about what Alice did or did not say? If we do trust F.5 to speak for Alice, we write "network interface F.5 **speaks for** Alice" in BAN authentication logic. In this example, then, if we believe that "network interface F.5 **speaks for** Alice", we can deduce that "Alice **says** send the quiz."

To make reasoning with this logic work, we need three rules:

- Rule 1: Delegating authority:
  If A **says** (B **speaks for** A)
  then B **speaks for** A
  This rule allows Alice to delegate authority to Bob, which allows Bob to speak for Alice.

- Rule 2: Use of delegated authority.
  If A **speaks for** B
  and A **says** (B **says** X)
  then B **says** X
  This rule says that if Bob delegated authority to Alice, and Alice says that Bob said something, then we can believe that Bob actually said it.

- Rule 3: Chaining of delegation.
  If A **speaks for** B
  and B **speaks for** C
  then A **speaks for** C
  This rule says that delegation of authority is transitive: if Bob has delegated authority to Alice and Charles has delegated authority to Bob, then Charles also delegated authority to Alice.

To capture real-world situations better, the full-bore BAN logic uses more refined rules then these. However, as we will see in the rest of this chapter, even these three simple rules are useful enough to help flush out fuzzy thinking.

5.8.1.1: Hard-wired Approach

How can the file service decide that "network interface F.5 **speaks for** Alice"? The first approach would be to hard-wire our installation. If we hard-wire Alice to her workstation, her workstation to network interface A.14, and network interface A.14 through the wire to network interface F.5, then we have:

- network interface F.5 **speaks for** the wire: we must assume no one rewired it.
- the wire **speaks for** network interface A.14: we must assume no one tampered with the channel.
- network interface A.14 **speaks for** workstation A: we must assume the workstation was wired correctly.
- workstation A **speaks for** Alice: we assume the operating system on Alice's workstation can be trusted.

In short, we assume that the network interface, the wiring, and Alice's workstation are part of the trusted computing base. With this assumption we can apply the chaining of delegation rule repeatedly to obtain "network interface F.5 **speaks for** Alice". Then, we can apply the use of delegated authority rule and obtain "Alice **says** send the quiz". Authentication of message origin is now complete, and the file system can look for Alice's token on its access control list.

The logic forced us to state our assumptions explicitly. Having made the list of assumptions, we can inspect them and see if we believe each is reasonable. We might even hire an outside auditor to offer an independent opinion.

5.8.1.2: Internet Approach

Now, suppose we instead connect the workstation's interface 14 to the file service's interface 5 using the Internet. Then, following the previous pattern, we get:

- network interface F.5 speaks for the Internet: we must assume no one rewired it.
- the Internet speaks for network interface A.14: we must assume the Internet is trusted!

The latter assumption is clearly problematic; we are dead in the water.

What can we do? Suppose the message is sent with some authentication tag—Alice actually sends the message with a MAC (reminder: $\{M\}_k$ denotes a plaintext message signed with a key k):

$$\text{Alice} \Rightarrow \text{file service: } \{\text{From: Alice; To: file service; "send the quiz"}\}_T$$

Then, we have:

- key T **says** (Alice **says** send the quiz).

If we know that Alice was the only person in the world who knows the key T , then we would be able to say:

- key T **speaks for** Alice.

With the use of delegated authority rule we could conclude "Alice **says** send the quiz". But is Alice really the only person in the world who knows key T ? We are using a shared-secret key system, so the file service must also know the key, and somehow the key must have been securely exchanged between Alice and the file service. So we must add to our list of assumptions:

- the file service is not trying to trick itself;
- the exchange of the shared-secret key was secure;
- neither Alice nor the file service have revealed the key.

With these assumptions we really can believe that "key T **speaks for** Alice", and we are home free. This reasoning is not a proof, but it is a method that helps us to discover and state our assumptions clearly.

The logic as presented doesn't deal with freshness. In fact, in the example, we can conclude only that "Alice **said** send the quiz", but not that Alice said it recently. Someone else might be replaying the message. Extensions to the basic logic can deal with freshness by introducing additional rules for freshness that relate **says** and **said**.

5.8.2: Authentication in Distributed Systems

All of the authentication examples we have discussed so far have involved one service. Using the techniques from Section 5.7, it is easy to see how we can build a single-service authentication and authorization system. A user sets up a confidential and authenticated communication channel to a particular service. The user authenticates itself over the secure channel and receives from the service a token to be used for access control. The user sends requests over the secure channel. The service then makes its access control decisions based on the token that accompanies the request.

Authentication in the World-Wide Web is an example of this approach. The browser sets up a secure channel using the SSL/TLS protocol described in Section 5.11. Then, the browser asks the user for a password and sends this password over the secure channel to the service. If the service identifies the user successfully with the received password, the service returns a token (a cookie in Web terminology), which the browser stores. The browser sends subsequent Web requests over the secure channel and includes the cookie with each request so that the user doesn't have to retype the password for each request. The service authenticates the principal and authorizes the request based on the cookie. (In practice, many Web applications don't set up a secure channel, but just communicate the password and cookie without any protection. These applications are vulnerable to most of the attacks discussed in previous sections.)

The disadvantage of this approach to authentication is that services cannot share information about clients. The user has to log in to each service separately and each service has to implement its own authentication scheme. If the user uses only a few services, these shortcomings are not a serious inconvenience. However, in a realm (say a large company or a university) where there are many services and where information needs to be shared between services, a better plan is needed.

In such an environment we would like to have the following properties:

1. the user logs in once;
2. the tokens the user obtains after login in should be usable by all services for authentication and to make authorization decisions;
3. users are named in a uniform way so that their names can be put on and removed from access control lists;
4. users and services don't have to trust the network.

These goals are sometimes summarized as single login or single sign-on. Few system designs or implementations meet these requirements. One system that comes close is Kerberos (see Sidebar 5.6.1). Another system that is gaining momentum for single sign-on to Web sites is openID; its goal is to allow users to have one ID for different Internet stores. The openID protocols are driven by a public benefit organization called the OpenID Foundation. Many major companies have joined the openID Foundation and are providing support in their services for openID.

5.8.3: Authentication across Administrative Realms

Extending authentication across realms that are administrated by independent authorities is a challenge. Consider a student who is running a service on a personal computer in his dorm room. The personal computer is not under the administrative authority of the university; yet, the student might want to obtain access to his service from a computer in a laboratory, which is administered by central campus authority. Furthermore, the student might want to provide access to his service to family and friends, who are in yet other administrative realms. It is unlikely that the campus administration will delegate authority to the personal computer and set up secure channels from the campus authentication service to each student's authentication service.

Sharing information with many users across many different administrative realms raises a number of questions:

1. How can we authenticate services securely? The Domain Name System (DNS) doesn't provide authenticated bindings of name to IP addresses, and so we cannot use DNS names to authenticate services.
2. How can we name users securely? We could use email addresses, such as bob@Scholarly.edu, to identify principals but email addresses can be spoofed.
3. How do we manage many users? If Pedantic University is willing to share course software with all students at The Institute of Scholar Studies, Pedantic University shouldn't have to list individually every student of The Institute of Scholar Studies on the access control list for the files. Clearly, protection groups are needed. But, how does a student at The Institute of Scholar Studies prove to Pedantic University's service that the student is part of the group students@Scholarly.edu?

These three problems are naming problems: how do we name a service, a user, a group, and a member of a protection group **securely**? A promising approach is to split the problem into two parts: (1) name all principals (e.g., services, users, and groups) by **public keys** and (2) securely distribute symbolic names for the public keys separately. We discuss this approach in more detail.

By naming principals by a public key we eliminate the distinction of realms. For example, a user Alice at Pedantic University might be named by a public key K_{Apub} and a user Bob at The Institute of Scholar Studies is named by a public K_{Bpub} ; from the public key we cannot tell whether the Alice is at Pedantic University or The Institute of Scholar Studies. From the public key alone we cannot tell if the public key is Alice's, but we will solve the binding from public key to symbolic name separately in the next sections, 5.8.4 through 5.8.6.

If the Alice wants to authorize Bob to have access to her files, Alice adds K_{Bpub} to her access control list. If Bob wants to use Alice's files, Bob sends a request to Alice's service including his public key K_{Bpub} . Alice checks if K_{Bpub} appears on her access control list. If not, she denies the request. Otherwise, Alice's service challenges Bob to prove that he has the private key corresponding to K_{Bpub} . If Bob can prove that he has K_{Bpriv} (for example, by signing a challenge that Alice's service verifies with Bob's public key K_{Bpub}), then Alice's service allows access.

When Alice approves the request, she doesn't know for sure if the request came from the principal named "Bob"; she just knows the request came from a principal holding the private key K_{Bpriv} . The symbolic name "Bob" doesn't play a role in the mediation decision. Instead, the crucial step was the authorization decision when Alice added K_{Bpub} to her access control; as part of that

authorization decision Alice must assure herself that K_{Bpub} speaks for Bob before adding K_{Bpub} to her access control list. That assurance relies on securely distributing bindings from name to public key, which we separated out as an independent problem and will discuss in the sections 5.8.4 through 5.8.6.

We can name protection groups also by a public key. Suppose that Alice knew for sure that $K_{ISSstudentspub}$ is a public key representing students of The Institute of Scholarly Studies. If Alice wanted to grant all students at The Institute of Scholarly Studies access to her files, she could add $K_{ISSstudentspub}$ to her access control list. Then, if Charles, a student at The Institute of Scholar Studies, wanted to have access to one of Alice's files, he would have to present a proof that he is a member of that group, for example, by providing a statement to Alice signed by $K_{ISSstudentspriv}$ to Alice saying:

$$\{K_{Charlespub} \text{ is a member of the group } K_{ISSstudentspub}\}\ K_{ISSstudentspriv}$$

which in the BAN logic translates to:

$$K_{Charlespub} \textbf{ speaks for } K_{ISSstudentspub}$$

That is, Alice has delegated authority to the member Charles to speak on behalf of the group of students at The Institute of Scholarly Studies.

Alice's service can verify this statement using $K_{ISSstudentspub}$, which is on Alice's access control list. After Alice's service successfully verifies the statement, then the service can challenge Charles to prove that he is the holder of the private key $K_{Charlespriv}$. Once Charles can prove he is the holder of that private key, then Alice's service can grant access to Charles.

In this setup, Alice must trust the holder of $K_{ISSstudentspriv}$ to be a responsible person who carefully verifies that Charles is a student at The Institute of Scholarly Studies. If she trusts the holder of that key to do so, then Alice doesn't have to maintain her own list of who is a student at The Institute of Scholar Studies; in fact, she doesn't need to know at all which particular principals are students at The Institute of Scholarly Studies.

If services are also named by public keys, then Bob and Charles can easily authenticate Alice's service. When Bob wants to connect to Alice's service, he specifies the public key of the service. If the service can prove that it possesses the corresponding private key, then Bob can have confidence that he is talking to the right service.

By naming all principals with public keys we can construct distributed authentication systems. Unfortunately, public keys are long, unintelligible bit strings, which are awkward and unfriendly for users to remember or type. When Alice adds K_{Bobpub} and $K_{ISSstudentspub}$ to her access control list, she shouldn't be required to type in a 1,024-bit number. Similarly when Bob and Charles refer to Alice's service, they shouldn't be required to know the bit representation of the public key of Alice's service. What is necessary is a way of naming public keys with symbolic names and authenticating the binding between name and key, which we will discuss next.

5.8.4: Authenticating Public Keys

How do we authenticate that K_{Bpub} is Bob's public key? As we have seen before, that authentication can be based on a key-distribution protocol, which start with a rendezvous step. For example, Bob and Alice meet face-to-face and Alice hands Bob a signed piece of paper with her public key and name. This piece of paper constitutes a **self-signed certificate**. Bob can have reasonable confidence in this certificate because Bob can verify that the certificate is valid and is Alice's. (Bob can ask Alice to sign again and compare it with the signature on the certificate and ask Alice for her driver license to prove her identity.)

If Bob receives a self-signed certificate over an untrusted network, however, we are out of luck. The certificate says "Hi, I am Alice and here is my public key" and it is signed with Alice's digital signature, but Bob does not know Alice's public key yet. In this case, anybody could impersonate Alice to Bob because Bob cannot verify whether or not Alice produced this certificate. An adversary can generate a public/private key pair, create a certificate for Alice listing the public key as Alice's public key, and sign it with the private key, and send this self-signed certificate to Bob.

Bob needs a way to find out securely what Alice's public key is. Most systems rely on a separate infrastructure for naming and distributing public keys securely. Such an infrastructure is called a **public key infrastructure**, PKI for short. There is a wide range of designs for such infrastructures, but their basic functions can be described well with the authentication logic. We start with a simple example using physical rendezvous and then later use certificate authorities to introduce principals to each other who haven't met through physical rendezvous.

Consider the following example where Alice receives a message from Bob, asking Alice to send a private file, and Alice wants to decide whether or not to send it. The first step in this decision is for Alice to establish if the message really came from Bob.

Suppose that Bob previously handed Alice a piece of paper on which Bob has written her public key, K_{pubBob}. We can describe Alice's take on this event in authentication logic as

$$\text{Bob says } (K_{\text{pubBob}} \text{ speaks for Bob}) \qquad \text{(belief \#1)}$$

and by applying the delegation of authority rule, Alice can immediately conclude that she is safe in believing

$$K_{\text{pubBob}} \text{ speaks for Bob} \qquad \text{(belief \#2),}$$

assuming that the information on the piece of paper is accurate. Alice realizes that she should should start making a list of assumptions for review later. (She ignores freshness for now because our stripped-down authentication logic has no **said** operation for capturing that.)

Next, Bob prepares a message, M_1 :

$$\text{Bob says } M_1$$

signs it with his private key:

$$\{M_1\}_{K\text{privBob}}$$

which, in authentication logic, can be described as

$$K_{\text{privBob}} \text{ says } (\text{Bob says } M_1)$$

and sends it to Alice. Since the message arrived via the Internet, Alice now wonders if she should believe

$$\text{Bob says } M_1 \qquad \text{(?)}$$

Fortunately, M_1 is signed, so Alice doesn't need to invoke any beliefs about the Internet. But the only beliefs she has established so far are (#1) and (#2), and those are not sufficient to draw any conclusions. So the first thing Alice does is check the signature:

$$result \leftarrow \text{VERIFY } (\{M_1\}_{K\text{privBob}},\ K_{\text{pubBob}})$$

If $result$ is ACCEPT then one might think that Alice is entitled to believe:

$$K_{\text{privBob}} \text{ says } (\text{Bob says } M_1) \qquad \text{(belief \#3?)}$$

but that belief actually requires a leap of faith that the cryptographic system is secure. Alice decides that it probably is, adds that assumption to her list, and removes the question mark on belief #3. But she still hasn't collected enough beliefs to answer the question. In order to apply the chaining and use of authority rules, Alice needs to believe that

$$(K_{\text{privBob}} \text{ speaks for } K_{\text{pubBob}}) \qquad \text{(belief \#4?)}$$

which sounds plausible, but for her to accept that belief requires another leap of faith: that Bob is the only person who knows K_{privBob} . Alice decides that Bob is probably careful enough to be trusted to keep his private key private, so she adds that assumption to her list and removes the question mark from belief #4.

Now, Alice can apply chaining of delegation rule to beliefs #4 and #2 to conclude

$$K_{\text{privBob}} \text{ speaks for Bob} \qquad \text{(belief \#5)}$$

and she can now use the use of delegated authority rule to beliefs #5 and #3 to conclude that

$$\text{Bob says } M_1 \qquad \text{(belief \#6)}$$

Alice decides to accepts the message as a genuine utterance of Bob. The assumptions that emerged during this reasoning were:

- K_{pubBob} is a true copy of Bob's public key.
- The cryptographic system used for signing is computationally secure.
- Bob has kept K_{privBob} secret.

5.8.5: Authenticating Certificates

One of the prime usages of a public key infrastructure is to introduce principals that haven't met through a physical rendezvous. To do so, a public key infrastructure provides certificates and one or more certificate authorities.

Continuing our example, suppose that Charles, whom Alice does not know, sends Alice the message

$\{M_2\}_{KprivCharles}$

This situation resembles the previous one, except that several things are missing: Alice does not know $K_{pubCharles}$, so she can't verify the signature, and in addition, Alice does not know who Charles is. Even if Alice finds a scrap of paper that has written on it Charles's name and what purports to be Charles's public key, $K_{pubCharles}$, and

$result \leftarrow VERIFY\ (M_2,\ SIGN\ (M_2,\ K_{privCharles}),\ K_{pubCharles})$

is ACCEPT, all she believes (again assuming that the cryptographic system is secure) is that

$K_{privCharles}$ **says** (Charles **says** M_2)

Without something corresponding to the previous beliefs #2 and #4, Alice still does not know what to make of this message. Specifically, Alice doesn't yet know whether or not to believe

$K_{privCharles}$ **speaks for** Charles (?)

Knowing that this might be a problem, Charles went to a well-known certificate authority, TrustUs.com, purchased the digital certificate:

$\{\text{"Charles's public key is } K_{pubCharles}\text{"}\}_{KprivTrustUs}$

and posted this certificate on his Web site. Alice discovers the certificate and wonders if it is any more useful than the scrap of paper she previously found. She knows that where she found the certificate has little bearing on its trustworthiness; a copy of the same certificate found on Lucifer's Web site would be equally trustworthy (or worthless, as the case may be).

Expressing this certificate in authentication logic requires two steps. The first thing we note is that the certificate is just another signed message, M_3, so Alice can interpret it in the same way that she interpreted the message from Bob:

$K_{privTrustUs}$ **says** M_3

Following the same reasoning that she used for the message from Bob, if Alice believes that she has a true copy of $K_{pubTrustUs}$ she can conclude that

TrustUs says M_3

subject to the assumptions (exactly parallel to the assumptions she used for the message from Bob)

- $K_{pubTrustUs}$ is a true copy of the TrustUs.com public key.
- The cryptographic system used for signing is computationally secure.
- TrustUs.com has kept $K_{privTrustUs}$ secret.

Alice decides that she is willing to accept those assumptions, so she turns her attention to M_3, which was the statement "Charles's public key is $K_{pubCharles}$". Since TrustUs.com is taking Charles's word on this, that statement can be expressed in authentication logic as

Charles **says** ($K_{pubCharles}$ **speaks for** Charles)

Combining, we have:

TrustUs **says** (Charles **says** ($K_{pubCharles}$ **speaks for** Charles))

To make progress, Alice needs to a further leap of faith. If Alice knew that

TrustUs **speaks for** Charles (?)

then she could apply the delegated authority rule to conclude that

Charles **says** (KpubCharles **speaks for** Charles)

and she could then follow an analysis just like the one she used for the earlier message from Bob. Since Alice doesn't know Charles, she has no way of knowing the truth of the questioned belief (TrustUs **speaks for** Charles), so she ponders

what it really means:

1. TrustUs.com has been authorized by Charles to create certificates for her. Alice might think that finding the certificate on Charles's Web site gives her some assurance on this point, but Alice has no way to verify that Charles's Web site is secure, so she has to depend on TrustUs.com being a reputable outfit.
2. TrustUs.com was careful in checking the credentials—perhaps a driver's license—that Charles presented for identification. If TrustUs.com was not careful, it might, without realizing it, be speaking for Lucifer rather than Charles. (Unfortunately, certificate authorities have been known to make exactly that mistake.) Of course, TrustUs.com is assuming that the credentials Charles presented were legitimate; it is possible that Charles has stolen someone else's identity. As usual, authentication of origin is never absolute; at best it can provide no more than a secure tie to some previous authentication of origin.

Alice decides to review the complete list of the assumptions she needs to make in order to accept Charles's original message M_2 as genuine:

- $K_{pubTrustUs}$ is a true copy of the TrustUs.com public key.
- The cryptographic system used for signing is computationally secure.
- TrustUs.com has kept $K_{privTrustUs}$ secret.
- TrustUs.com has been authorized by Charles.
- TrustUs.com carefully checked Charles's credentials.
- TrustUs.com has signed the right public key (that is $K_{pubCharles}$).
- Charles has kept $K_{privCharles}$ secret.

She notices that in addition to relying heavily on the trustworthiness of TrustUs.com, she doesn't know Charles, so the last assumption may be a weakness. For this reason, she would be well-advised to accept message M_2 with a certain amount of caution. In addition, Alice should keep in mind that since Charles's public key was not obtained by a physical rendezvous, she knows only that the message came from someone named "Charles"; she as yet has no way to connect that name with a real person.

As in the previous examples, the stripped-down authentication logic we have been using for illustration has no provision for checking freshness, so it hasn't alerted Alice that she is also assuming that the two public keys are fresh and that the message itself is recent.

The above example is a distributed authorization system that is ticket-oriented. TrustUs.com has generated a ticket (the certificate) that Alice uses to authenticate Charles's request. Given this observation, this immediately raises the question of how Charles revokes the certificate that he bought from TrustUs.com. If Charles, for example, accidently discloses his private key, the certificate from TrustUS.com becomes worthless and he should revoke it so that Alice cannot be tricked into believing that M_2 came from Charles. One way to address this problem is to make a certificate valid for only a limited length of time. Another approach is for TrustUs.com to maintain a list of revoked certificates and for Alice to first check with TrustUS.com before accepting an certificate as valid.

Neither solution is quite satisfactory. The first solution has the disadvantage that if Charles loses his private key, the certificate will remain valid until it expires. The second solution has the disadvantage that TrustUs.com has to be available at the instant that Alice tries to check the validity of the certificate.

5.8.6: Certificate Chains

The public key infrastructure developed so far has one certificate authority, TrustUs.com. How do we certify the public key of TrustUs.com? There might be many certificate authorities, some of which Alice doesn't know about. However, Alice might possess a certificate for another certificate authority that certifies TrustUs.com, creating a chain of certification. Public key infrastructures organize such chains in two primary ways; we discuss them in turn.

5.8.6.1: Hierarchy of Central Certificate Authorities

In the central-authority approach, key certificate authorities record public keys and are managed by central authorities. For example, in the Word Wide Web, certificates authenticating Web sites are usually signed by one of several well-known root certificate authorities. Commercial Web sites, such as amazon.com, for instance, present a certificate signed by Verisign to a client when it connects. All Web browsers embed the public key of the root certificates in their programs. When the browser receives a certificate from amazon.com, it uses the embedded public key for Verisign to verify the certificate.

Some Web sites, such as a company's internal Web site, generate a self-signed certificate and send that to a client when it connects. To be able to verify a self-signed certificate, the client must have obtained the key of the Web site securely in advance.

The Web approach to certifying keys has a shallow hierarchy. In DNSSEC*, a secure version of DNS, CAs can be arranged in a deeper hierarchy. If Alice types in the name "athena.Scholarly.edu", her resolver will contact one of the root servers and obtain an address and certificate for "edu". In authentication logic, the meaning of this certificate is " $K_{privroot}$ says that K_{pubedu} speaks for edu ". To be able to verify this certificate she must have obtained the public key of the root servers in some earlier rendezvous step. If the certificate for "edu" verifies, she contacts the server for the "edu" domain, and asks for the server's address and certificate for "Scholarly", and so on.

One problem with the hierarchical approach is that one must trust a central authority, such as the DNS root service. The central authority may ask an unreasonable price for the service, enforce policies that you don't like, or considered untrustworthy by some. For example, in DNS and DNSSEC, there is a lot of politics around which institution should run the root servers and the policies of that institution. Since the Internet and DNS originated in the U.S.A., it is currently run by an U.S.A. organization. Unhappiness with this organization has led the Chinese to start their own root service.

Another problem with the hierarchical approach is that certificate authorities determine to whom they delegate authority for a particular domain name. You might be happy with the Institute of Scholarly Studies managing the "Scholarly" domain, but have less trust in a rogue government managing the top-level domain for all DNS names in that country.

Because of problems like these, it is difficult in practice to agree and manage a single PKI that allows for strong authentication world wide. Currently, no global PKI exist.

---

* D. Eastlake, *Domain Name System Security Extensions*, Internet Engineering Task Force Request For Comments (RFC 2535), Mach 1999.

5.8.6.2: Web of Trust

The web-of-trust approach avoids using a chain of central authorities. Instead, Bob can decide himself whom he trusts. In this approach, Alice obtains certificates from her friends Charles, Dawn, and Ella and posts these on her Web page: $\{Alice, K_{Apub}\}_{KCpriv}$, $\{Alice, K_{Apub}\}_{KDpriv}$, $\{Alice, K_{Apub}\}_{KEpriv}$. If Bob knows the public key of any one of Charles, Dawn, or Ella, he can verify one of the certificates by verifying the certificate that person signed. To the extent that he trusts that person to be careful in what he or she signs, he has confidence that he now has Alice's true public key.

On the other hand, if Bob doesn't know Charles, Dawn, or Ella, he might know someone (say Felipe) who knows one of them. Bob may learn that Felipe knows Ella because he checks Ella's Web site and finds a certificate signed by Felipe. If he trusts Felipe, he can get a certificate from Felipe, certifying one of the public keys K_{Cpub} , K_{Dpub} , or K_{Epub} , which he can then use to certify Alice's public key. Another possibility is that Alice offers a few certificate chains in the hope that Bob trusts one of the signers in one of the chains, and has the signer's public key in his set of keys. Independent of how Bob learned Alice's public key, he can inspect the chain of trust by which he learned and verified Alice's public key and see whether he likes it or not. The important point here is that Bob must trust *every* link in the chain. If any link untrustworthy, he will have no guarantees.

The web of trust scheme relies on the observation that it usually takes only a few acquaintance steps to connect anyone in the world to anyone else. For example, it has been claimed that everyone is separated by no more than 6 steps from the President of the United States. (There may be some hermits in Tibet that require more steps.) With luck, there will be many chains connecting Bob with Alice, and one of them may consist entirely of links that Bob trusts.

The central idea in the web-of-trust approach is that Bob can decide whom he trusts instead of having to trust a central authority. PGP (Pretty Good Privacy) * and a number of other systems use the web of trust approach.

* Simson Garfinkel. *PGP: Pretty Good Privacy*. O'Reilly & Associates, Sebastopol, California, 1995. ISBN: 978–1–56592–098–9 (paperback). 430 pages.

> Nominally a user's guide to the PGP encryption package developed by Phil Zimmermann, this book starts out with six readable overview chapters on the subject of encryption, its history, and the political and licensing environment that surrounds encryption systems. Even the later chapters, which give details on how to use PGP, are filled with interesting tidbits and advice applicable to all encryption uses.

This page titled 5.8: Reasoning About Authentication (Advanced Topic) is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

5.9: Cryptography as a Building Block (Advanced Topic)

This section sketches how primitives such as `ENCRYPT`, `DECRYPT`, pseudorandom number generators, `SIGN`, `VERIFY`, and cryptographic hashes can be implemented using **cryptographic transformations** (also called **ciphers**). Readers who wish to understand the implementations in detail should consult books such as *Applied Cryptography by Bruce Schneier* [Suggestions for Further Reading 5.5.1], or *Handbook of Applied Cryptography* by Menezes, van Oorschot, and Vanstore [Suggestions for Further Reading 5.5.2]. *Introduction to cryptography* by Buchmann provides a concise description of the number theory that underlies cryptography [Suggestions for Further Reading 5.5.9]. There are many subtle issues in designing secure implementations of the primitives, which are beyond the scope of this text.

5.9.1: Unbreakable Cipher for Confidentiality (One-Time Pad)

Making an unbreakable cipher for *only* confidentiality is easy, but there's a catch. The recipe is as follows. First, find a process that can generate a truly random unlimited string of bits, which we call the **key string**, and transmit this key string through **secure** (i.e., providing confidentiality and authentication) channels to both the sender and receiver before they transmit any data through an insecure network.

Once the key string is securely in the hands of the sender, the sender converts the plaintext into a bit string and computes bit-for-bit the exclusive `OR` (`XOR`) of the plaintext and the key string. The sender can send the resulting ciphertext over an insecure network to a receiver. Using the previously communicated key string, the receiver can recover the plaintext by computing the `XOR` of the ciphertext and key string.

To be more precise, this transforming scheme is a stream cipher. In a **stream cipher**, the conversion from plaintext to ciphertext is performed one bit or one byte at a time, and the input can be of any length. In our example, a sequence of message (plaintext) bits $m_1, m_2, \ldots, m_n$ is transformed using an equal-length sequence of secret key bits $k_1, k_2, \ldots, k_n$ that is known to both the sender and the receiver. The i-th bit c_i of the ciphertext is defined to be the `XOR` (modulo-2 sum) of m_i and k_i, for $i = 1, \ldots, n$:

$$c_i = m_i \oplus k_i \tag{5.9.1}$$

Untransforming is just as simple, because:

$$m_i = c_i \oplus k_i = m_i \oplus k_i \oplus k_i = m_i \tag{5.9.2}$$

This scheme, under the name "one-time pad," was patented by Vernam in 1919 (U.S. patent number 1,310,719). In his version of the scheme, the "pad" (that is, the one-time key) was stored on paper tape.

The key string is generated by a **random number generator**, which produces as output a "random" bit string. That is, from the bits generated so far, it is impossible to predict the next bit. True random-number generators are difficult to construct; in fact, true sources of random sequences come only from physical processes, not from deterministic computer programs.

If we flip a single message bit, the corresponding ciphertext bit flips. Similarly, if a single ciphertext bit is flipped by a network error (or an adversary), the receiver will untransform the ciphertext to obtain a message with a single bit error in the corresponding position. Thus, the one-time pad (both transforming and untransforming) has **limited change propagation**: changing a single bit in the input causes only a single bit in the output to change.

Unless additional measures are taken, an adversary can add, flip, or replace bits in the stream without the recipient realizing it. The adversary may have no way to know exactly how these changes will be interpreted at the receiving end, but the adversary can probably create quite a bit of confusion. This cipher provides another example of the fact that message confidentiality and integrity are separate goals.

The catch with a one-time pad is the key string. We must have a secure channel for sending the key string and the key string must be at least as long as the message. One approach to sending the key string is for the sender to generate a large key string in advance. For example, the sender can generate 10 CDs full of random bits and truck them over to the receiver by armored car. Although this scheme may have high bandwidth (6.4 Gigabytes per truckload), it probably has latency too large to be satisfactory.

The key string must be at least as long as the message. It is not hard to see that if the sender re-uses the one-time pad, an adversary can determine quickly a bit (if not everything) about the plaintext by examining the XOR of the corresponding ciphertext (if the bits are aligned properly, the pads cancel). The National Security Agency (NSA) once caught the Russians in such a mistake* in Project VENONA**.

* R. L. Benson, *The Venona Story, National Security Agency, Center for logic History*, 2001. http://www.nsa.gov/publications/publi00039.cfm

** D. P. Moynihan (chair), Secrecy: Report of the commision on protecting and reducing government secrecy, *Senate document 105-2, 103rd congress*, United States government printing office,1997.

5.9.2: Pseudorandom Number Generators

One shortcut to avoid having to send a long key string over a secure channel is to use a **pseudorandom number generator**. A pseudorandom number generator produces deterministically a random-appearing bit stream from a short bit string, called the **seed**. Starting from the same seed, the pseudorandom generator will always produce the same bit stream. Thus, if both the sender and the receiver have the secret short key, using the key as a seed for the pseudorandom generator they can generate the same, long key string from the short key and use the long key string for the transformation.

Unlike the one-time pad, this scheme can in principle be broken by someone who knows enough about the pseudorandom generator. The design requirement on a pseudorandom number generator is that it is difficult for an opponent to predict the next bit in the sequence, even with full knowledge of the generating algorithm and the sequence so far. More precisely:

1. Given the seed and algorithm, it is easy to compute the next bit of the output of the pseudorandom generator.
2. Given the algorithm and some output, it is difficult (or impossible) to predict the next bit.
3. Given the algorithm and some output, it is difficult (or impossible) to compute what the seed is.

Analogous to ciphers, the design is usually open: the algorithm for the pseudorandom generator is open. Only the seed is secret, and it must be produced from a truly random source.

5.9.2.1: RC4: A Pseudorandom Generator and its Use

RC4 was designed by Ron Rivest for RSA Data Security, Inc. RC4 stands for Ron's Code number 4. RSA tried to keep this cipher secret, but someone published a description anonymously on the Internet. (This incident illustrates how difficult it is to keep something secret, even for a security company!) Because RSA never confirmed whether the description is indeed RC4, people usually refer to the published version as ARC4, or alleged RC4.

The core of the RC4 cipher is a pseudorandom generator, which is surprisingly simple. It maintains a fixed array S of 256 entries, which contains a permutation of the numbers 0 through 255 (each array entry is 8 bits). It has two counters i and j , which are used as follows to generate a pseudorandom byte k :

```
1 procedure RC4_GENERATE ()
2     i ← (i + 1) modulo 256
3     j ← (j + S[i]) modulo 256
4     SWAP (S[i], S[j])
5     t ← (S[i] + S[j]) modulo 256
6     k ← S[t]
7     return k
```

The initialization procedure takes as input a seed, typically a truly-random number, which is used as follows:

```
1 procedure RC4_INIT (seed)
2     for i from 0 to 255 do
3         S[i] ← i
4         K[i] ← seed[i]
5     j ← 0
6     for i from 0 to 255 do
7         j ← (j + S[i] + K[i]) modulo 256
8         SWAP(S[i], S[j])
9     i ← j ← 0
```

The procedure `RC4_INIT` fills each entry of S with its index: $S[0] \leftarrow 0$, $S[1] \leftarrow 1$, etc. (see lines 2 through 4). It also allocates another 256-entry array (K) with each 8-bit entries. It fills K with the seed, repeating the seed as necessary to fill the

array. Thus, `K[0]` contains the first 8 bits of the key string, `K[1]` the second 8 bits, etc. Then, it runs a loop (lines 6 through 8) that puts `S` in a pseudorandom state based on `K` (and thus the seed).

5.9.2.2: Confidentiality using RC4

Given the RC4 pseudorandom generator, `ENCRYPT` and `DECRYPT` can be implemented as in the one-time pad, except instead of using a truly-random key string, we use the output of the pseudorandom generator. To initialize, the sender and receiver invoke on their respective computers `RC4_INIT` , supplying the shared-secret key for the stream as the seed. Because the sender and receiver supply the same key to the initialization procedure, `RC4_GENERATE` on the sender and receiver computer will produce identical streams of key bytes, which `ENCRYPT` and `DECRYPT` use as a one-time pad.

In more detail, to send a byte b, the sender invokes `RC4_GENERATE` to generate a pseudorandom byte k and encrypts byte b by computing $c = b \oplus k$. When the receiver receives byte c, it invokes `RC4_GENERATE` on its computer to generate a pseudorandom byte k_1 and decrypts the byte c by computing $b \oplus k_1$. Because the sender and receiver initialized the generator with the same seed, k and k_1 are identical, and $c \oplus k_1$ gives b.

RC4 is simple enough that it can be coded from memory, yet it appears it is computationally secure and a moderately strong stream cipher for confidentiality, though it has been noticed that the first few bytes of its output leak information about the shared-secret key, so it is important to discard them. Like any stream cipher, it cannot be used for authentication without additional mechanism. When using it to encrypt a long stream, it doesn't seem to have any small cycles and its output values vary highly (RC4 can be in about $256! \times 256^2$ possible states). The key space contains 2^{256} values, so it is also difficult to attack RC4 by brute force. RC4 must be used with care to achieve a system's overall security goal. For example, the Wired Equivalent Privacy scheme for WiFi wireless networks (see page 11–50) uses the RC4 output stream without discarding the beginning of the stream. As a result, using the leaked key information mentioned above it is relatively easy to crack WEP wireless encryption*.

The story of flawed confidentiality in WiFi's use of RC4 illustrates that it is difficult to create a really good pseudorandom number generator. Here is another example of that difficulty: during World War II, the Lorenz SZ 40 and SZ 42 cipher machines, used by the German Army, were similarly based on a (mechanical) pseudorandom number generator, but a British code-breaking team was able, by analyzing intercepted messages, to reconstruct the internal structure of the generator, build a special-purpose computer to search for the seed, and thereby decipher many of the intercepted messages of the German Army**.

---

* A. Stubblefield, J. Ioannidis, and A. Rubin, *Using the Fluhrer, Mantin, and Shamir attack to break WEP*, Symposium on Network and Distributed System Security, 2002.

** F. H. Hinsley and Alan Stripp, *Code Breakers: The Inside Story of Bletchley Park* (Oxford University Press, 1993), page 161.

5.9.3: Block Ciphers

Depending on the constraints on their inputs, ciphers are either stream ciphers or block ciphers. In a block cipher, the cipher performs the transformation from plaintext to ciphertext on fixed-size blocks. If the input is shorter than a block, `ENCRYPT` must pad the input to make it a full block in length. If the input is longer than a block, `ENCRYPT` breaks the input into several blocks, padding the last block is padded, if necessary, and then transforms the individual blocks. Because a given plaintext block always produces the same output with a block cipher, `ENCRYPT` must use a block cipher with care. We outline one widely used block cipher and how it can be used to implement `ENCRYPT` and `DECRYPT` .

5.9.3.1: Advanced Encryption Standard (AES)

Advanced Encryption Standard (AES)[1] has 128-bit (or longer) keys and 128-bit plaintext and ciphertext blocks. AES replaces **Data Encryption Standard (DES)**[2] [3], which is now regarded as too insecure for many applications, as distributed Internet computations or dedicated special-purpose machines can use a brute-force exhaustive search to quickly find a 56-bit DES key given corresponding plaintext and ciphertext [Suggestions for Further Reading 5.5.4].

AES takes a 128-bit input and produces a 128-bit output. If you don't know the 128-bit key, it is hard to reconstruct the input given the output. The algorithm works on a 4×4 array of bytes, called `state` . At the beginning of the cipher the input array `in` is copied to the `state` array as follows:

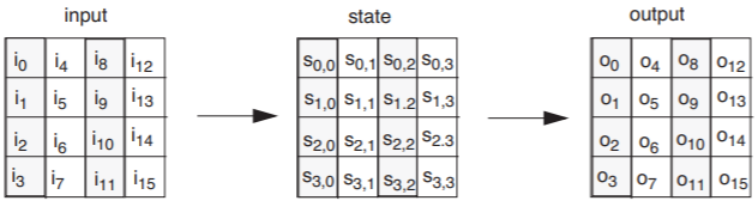


Figure 5.9.1: AES transformation of the *input* array to the *state* array to the *output* array.

At the end of the cipher the *state* array is copied into the output array *out* as depicted. The four bytes in a column form 32-bit words.

The cipher transforms *state* as follows:

```
1 procedure AES (in, out, key)
2     state ← in                                        // copy in into state as
described above
3     ADDROUNDKEY (state, key)                          // mix key into state

4     for r from 1 to 9 do
5         SUBBYTES (state)                              // substitute some bytes in
state
6          SHIFTROWS (state)                            // shift rows of state
cyclically
7         MIXCOLUMNS (state)                            // mix the columns up
8         ADDROUNDKEY (state, key[r×4, (r+1)×4 – 1])    // expand key, mix in

9     SUBBYTES (state)
10    SHIFTROWS (state)
11    ADDROUNDKEY (state, key[10×4, 11×4 – 1])
12    out ← state                                       // copy state into out as
described above
```

The cipher performs 10 rounds (denoted by the variable r), but the last round doesn't invoke MIXCOLUMNS . Each ADDROUNDKEY takes the 4 words from *key* and adds them into the columns of *state* as follows:

$$[s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}, s_{4,c}] \leftarrow [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}, s_{4,c}] \oplus \text{key}_{r\times 4+c}, \text{ for } 0 \le c < 4.$$

That is, each word of *key* is added to the corresponding column in *state* .

For the first invocation (on line 3) of ADDROUNDKEY , r is 0, and in that round ADDROUNDKEY uses the 128-bit key completely. For subsequent rounds, AES generates additional key words using a carefully-designed algorithm. The details and justification are outside of the scope of this textbook, but the flavor of the algorithm is as follows. It takes earlier-generated words of the key and produces a new word, by substituting well-chosen bits, rotating words, and computing the XOR of certain words.

The procedure SUBBYTES applies a substitution to the bytes of *state* according to a well-chosen substitution table. In essence, this mixes the bytes of *state* up.

The procedure SHIFTROWS shifts the last three rows of state cyclically as follows:

$$s_{r,c} \leftarrow s_{r,(c + \text{shift}(r, 4)) \bmod 4}, \text{ for } 0 \le c < 4$$

The value of SHIFT is dependent on the row number as follows:

$$\text{SHIFT}(1,4) = 1, \text{ SHIFT}(2,4) = 2, \text{ and SHIFT}(3,4) = 3$$

The procedure MIXCOLUMNS operates column by column, applying a well-chosen matrix multiplication.

In essence, AES is a complicated transformation of *state* based on *key* . Why this transformation is thought to be computationally secure is beyond the scope of this text. We just note that it has been studied by many cryptographers and it is believed to secure.

[1] Advanced Encryption Standard, *Federal Information Processing Standards Publications (FIPS PUBS) 197*, National Institute of Standards and Technology (NIST), Nov. 2001.

[2] Data Encryption Standard. U.S. Department of Standards, National Bureau of Standards, Federal Information Processing Standard (FIPS) Publication #46, January, 1977 (#46–1 updated 1988; #46–2 updated 1994).

[3] Horst Feistel, William A. Notz, and J. Lynn Smith. *Some cryptographic techniques for machine-to-machine data communications*. Proceedings of the IEEE 63, 11 (November, 1975), pages 1545–1554. An older paper by the designers of the DES providing background on why it works the way it does. One should be aware that the design principles described in this paper are incomplete; the really significant design principles are classified as military secrets.

5.9.3.2: Cipher-Block Chaining

With block ciphers, the same input with the same key generates the same output. Thus, one must be careful in using a block cipher for encryption. For example, if the adversary knows that the plaintext is formatted for a printer and each line starts with 16 blanks, then the line breaks will be apparent in the ciphertext because there will always be an 8-byte block of blanks, enciphered the same way. Knowing the number of lines in the text and the length of each line may be usable for frequency analysis to search for the shared-secret key.

A good approach to constructing ENCRYPT using a block cipher is cipher-block chaining. **Cipher-block chaining (CBC)** randomizes each plaintext block by XOR -ing it with the previous ciphertext block before transforming it (see Figure 5.9.2). A random dummy ciphertext block, called the **initialization vector** or IV, is inserted at the beginning.

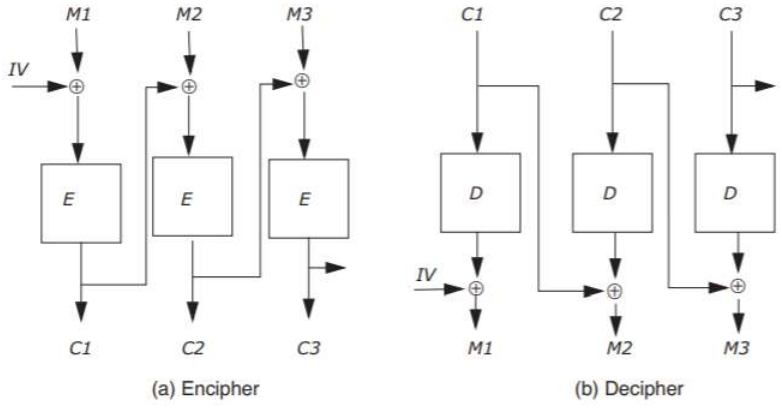


Figure 5.9.2: Cipher-block chaining.

More precisely, if the message has blocks $M_1, M_2, \ldots, M_n$, ENCRYPT produces the ciphertext consisting of blocks $C_0, C_1, \ldots, C_n$ as follows:

$$C_0 = IV \text{ and } C_i \leftarrow BC(M_i \oplus C_{i-1}, key) \text{ for } i = 1, 2, \ldots, n$$

where BC is some block cipher (e.g., AES). To implement DECRYPT, one computes:

$$M_i \leftarrow C_{i-1} \oplus BC(C_i, key)$$

CBC has **cascading change propagation** for the plaintext: changing a single message bit (say in M_i) causes a change in C_i, which causes a change in C_{i+1}, and so on. CBC's cascading change property, together with the use of a random IV as the first ciphertext block, implies that two encryptions of the same message with the same key will result in entirely different-looking ciphertexts. The last ciphertext block Cn is a complicated keydependent function of the IV and of all the message blocks. We will use this property later.

On the other hand, CBC has limited change propagation for the ciphertext: changing a bit in ciphertext block C_i causes the receiver to compute M_i and M_{i+1} incorrectly, but all later message blocks are still computed correctly. Careful study of Figure 5.9.2 should convince you that this property holds.

Ciphers with limited change propagation have important applications, particularly in situations where ciphertext bits may sometimes be changed by random network errors and where, in addition, the receiving application can tolerate a moderate amount of consequently modified plaintext.

5.9.4: Computing a Message Authentication Code

So far we used ciphers for only confidentiality, but we can use ciphers also to compute authentication tags so that the receiver can detect if an adversary has changed any of the bits in the ciphertext. That is, we can use ciphers to implement the SIGN and VERIFY interface, discussed in Section 5.3. Using shared-secret cryptography, there are two different approaches to implementing the interface: 1) using a block or stream cipher or 2) using a cryptographic hash function. We discuss both.

5.9.4.1: MACs Using Block Cipher or Stream Cipher

CBC-MAC is a simple message authentication code scheme based on a block cipher in CBC mode. To produce an authentication tag for a message M with a key k, SIGN pads the message out to an integral number of blocks with zero bits, if necessary, and transforms the message M with cipher-block chaining, using the key k as the initialization vector (IV). (The key k is an authentication key, different from the encryption key that the sender and receiver may also use.) All ciphertext blocks except the last are discarded, and the *last* ciphertext block is returned as the value of the authentication tag (the MAC). As noted earlier, because of cascading change propagation, the last ciphertext block is a complicated function of the secret key and the entire message.

VERIFY recomputes the MAC from M and key k using the same procedure that SIGN used, and compares the result with the received authentication tag. An adversary cannot produce a message M that the receiver will believe is authentic because the adversary doesn't know key k.

One can also build SIGN and VERIFY using stream ciphers by, for example, using the cipher in a mode called **cipher-feedback (CFB)**. CFB works like CBC in the sense that it links the plaintext bytes together so that the ciphertext depends on all the preceding plaintext. For the details consult the literature.

5.9.4.2: MACs Using a Cryptographic Hash Function

The basic idea for computing a MAC with a cryptographic hash function is as follows. If the sender and receiver share an authentication key k, then the sender constructs a MAC for a message M by computing the cryptographic hash of the concatenated message $k + M$: $\text{HASH}(k + M)$. Since the receiver knows k, the receiver can recompute $\text{HASH}(k + M)$ and compare the result with the received MAC. Because an adversary doesn't know k, the adversary cannot forge the MAC for the message M.

This basic idea must be refined to make the MAC secure because without modifications it has problems. For example, Lucifer can add bytes to the end of the message without the receiver noticing. This attack can perhaps be countered with adding the length of the message to the beginning of the message. Cryptographers have given this problem a lot of attention and have come up with a construction, called **HMAC** [Suggestions for Further Reading 5.5.7], which is said to be as secure as the underlying cryptographic hash function. HMAC uses two strings:

- $innerpad$, which is the byte 36_{hex} repeated 64 times
- $outerpad$, which is the byte $5C_{hex}$ repeated 64 times

Using these strings, HMAC computes the MAC for a message M and an authentication key k as follows:

$$\text{HASH}((k \oplus outerpad) + \text{HASH}((k \oplus innerpad) + M))$$

To compute the XOR, HMAC pads k with enough zero bytes to make it of length 64. If k is longer than 64 bytes, HMAC uses HASH (k), padded with enough zero bytes to make the result of length 64 bytes.

HMAC can be used with any good cryptographic hash function. Sidebar 5.9.1 describes SHA-1, a widely used cryptographic hash function. Even though SHA-1 must have collisions, no one has uncovered an example of one so far. Recent findings (February 2005) suggest weaknesses in SHA-1 and National Institute for Standards and Technology is recommending switching to longer versions named SHA-256 and SHA512. Some cryptographers are recommending that research on designing cryptographic hash functions should start over.

Sidebar 5.9.1

Secure Hash Algorithm (SHA)

SHA* is a family of cryptographic hash algorithms. SHA-1 takes as input a message of any length smaller than 264 bits and produces a 160-bit hash. It is cryptographic in the sense that given a hash value, it is computationally infeasible to recover the corresponding message or to find two different messages that produce the same hash.

SHA-1 computes the hash as follows. First, the message being hashed is padded to make it a multiple of 512 bits long. To pad, one appends a 1, then as many 0's as necessary to make it 64 bits short of a multiple of 512 bits, and then a 64-bit big-endian representation of the length (in bits) of the unpadded message. The padded string of bits is turned into a 160-bit value as follows.

The message is split into 512-bit blocks. Each block is expanded from 512 bits (16 32-bit words M) to 80 32-bit words as follows ($W(t)$ is the t -th word):

$$W(t) = \begin{cases} M_t, & \text{for } t = 0 \text{ to } 15 \\ (W(t-3) \oplus (W(t-8) \oplus (W(t-14) \oplus (W(t-16) \lll 1 & \text{for } t = 16 \text{ to } 79 \end{cases}$$

where $\lll$ is a left circular shift.

SHA uses four nonlinear functions and four 32-bit constants. The four functions are

$$F(t, x, y, z) = \begin{cases} (X \,\&\, Y) \mid ((\sim X) \,\&\, Z), & \text{for } t = 0 \text{ to } 19 \\ (X \oplus Y \oplus Z), & \text{for } t = 20 \text{ to } 39 \\ (X \,\&\, Y) \mid (X \,\&\, Z) \mid (Y \,\&\, Z), & \text{for } t = 40 \text{ to } 59 \\ X \oplus Y \oplus Z, & \text{for } t = 60 \text{ to } 79 \end{cases}$$

The constants are

$$K(t) = \begin{cases} 5A827999_{hex}, & \text{for } t = 0 \text{ to } 19 \text{ // 2.5/4 in hex} \\ 6ED9EBA1_{hex}, & \text{for } t = 20 \text{ to } 39 \text{ // 3.5/4 in hex} \\ 8F1BBCDC_{hex}, & \text{for } t = 40 \text{ to } 59 \text{ // 5.5/5 in hex} \\ CA62C1D6_{hex}, & \text{for } t = 60 \text{ to } 79 \text{ // 10.5/4 in hex} \end{cases}$$

SHA uses five 32-bit variables (160 bits) to compute the hash. They are initialized and copied into 5 temporary variables:

$$a \leftarrow A \leftarrow 67452301_{hex}$$
$$b \leftarrow B \leftarrow EFCDAB89_{hex}$$
$$c \leftarrow C \leftarrow 98BADCFE_{hex}$$
$$d \leftarrow D \leftarrow 10325476_{hex}$$
$$e \leftarrow E \leftarrow C3D2E1F0_{hex}$$

The 160-bit hash value for a message is now computed as follows:

```
1  for each 512-bit block of M do
2      for t from 0 to 79 do
3          x ← (a <<< 5) + F(t, b, c, d) + e + W(t) + K(t)
4          e ← d
5          d ← c
6          c ← b <<< 30
7          b ← a
8          a ← x
9      A ← A + a; B ← B + b; C ← C + c; D ← D + d; E ← E + e
10 hash = A + B + C + D + E                          // concatenate A, B, C, D,
and E
```

Other hashes in the SHA family are similar in spirit, but have different constants, word sizes, and produce hash values with more bits. For example, SHA-256 has a different W and F , and produces a 256-bit value.The justification for the SHA family of hashes is outside the scope of this text.

* Secure hash standard, *Federal Information Processing Standards Publications (FIPS PUBS) 180-1*, National Institute of Standards and Technology (NIST), April 1995.

5.9.5: A Public-Key Cipher

The ciphers described so far are shared-secret ciphers. Both the sender and receiver must know the shared secret key. Public-key ciphers remove this requirement, which opens up new kinds of applications, as the main body of the chapter described. The literature contains several public-key ciphers. We explain the first invented one because it is easy to explain, yet is still believed to be secure.

5.9.5.1: Rivest-Shamir-Adleman (RSA) Cipher

The security of the RSA cipher relies on a simple-to-state (but hard to solve) well-known problem in number theory [Suggestions for Further Reading 5.5.3]. RSA was developed at M.I.T. in 1977 (patent number 4,405,829), and is named after its inventors: **Rivest, Shamir, and Adleman (RSA)**. It is based on properties of prime numbers; in particular, it is computationally expensive to factor large numbers (for ages mathematicians have been trying to come up with efficient algorithms with little success), but much cheaper to find large primes.

The basic idea behind RSA is as follows. Initially you choose two large prime numbers (p and q, each larger than 10^{100}). Then compute $n = p \times q$ and $z = (p-1) \times (q-1)$, and find a large number d that is relatively prime to z. Finally, find an e such that $e \times d = 1 (\text{modulo } z)$. After finding these numbers once, you have two keys, (e, n) and (d, n), which are hard to derive from each other, even though n is public.

For now assume that the message to be transformed using RSA has a value P that is greater than or equal to zero and smaller than n. (below, Sections 5.9.5.2 and 5.9.5.3 discuss how to use RSA for signatures and encryption of any message in more detail.) The cipher C is computed by raising P to the power e: $P^e (\text{modulo } n)$. To decipher, we compute C to the power d: $C^d (\text{modulo } n)$.

The reason this works is as follows. $C^d = P^{ed} = P^{k(p-1)(q-1)+1}$, since $e \times d = 1 (\text{modulo } z)$. That reduction uses the theorem that $P^{(p-1)(q-1)} = P^0 (\text{modulo } n)$, a result by Euler, and now $P^{k(p-1)(q-1)+1} = P \times P^{k(p-1)(q-1)} = P \times P^0 = P \times 1 = P$. The theorem that the exponent $k(p-1)(q-1) = 0 (\text{modulo } n)$ is a result by Euler and Fermat (see I. Niven and H.S. Zuckerman, *An Introduction to the Theory of Numbers*, Wiley, New York, 1980).

An example with concrete numbers may illuminate the abstract mathematics. If one chooses $p = 47$, $q = 59$, and $e = 17$, then $d = 157$ because $e \times d = 1 (\text{modulo } 2668)$. This gives us two keys: $(17, 2773)$ and $(157, 2773)$. Now we can transform any P with a value between 0 and 2773. For example, if P is 31, $C = 587 = 31^{17} (\text{modulo } 2773)$. To reverse the transform, we compute $587^{157} = 31 (\text{modulo } 2773)$.

One way to break this scheme is to factor the modulus (n). In 1977 Ron Rivest (the R in RSA) estimated that factoring a 125-digit decimal number would take 40 quadrillion years, using the best known algorithms and state-of-the-art hardware running at 1 million instructions per second*. To test this claim and to encourage research into computational number theory and factoring, RSA Security, the company commercializing RSA, has posted several products of two primes, also called RSA numbers, as factoring challenges. Understanding the speed at which factoring can be done helps in choosing a suitable key length for a desired level of security.

In 1994, a group of researchers under the guidance of A.J. Lenstra factored a 129 digit decimal RSA number in 8 months using the Internet as a parallel computer, without paying for the cycles**. It required 5,000 MIPS years (i.e., 5,000 one-million-instructions-per-second computers each running for one year). Rivest's calculation is an example of the hazards involved in estimating an historic work factor. Better algorithms have been developed, allowing the computation to be performed in only 5,000 MIPS years instead of 40 quadrillion MIPS years, and communication technology has improved substantially, allowing 5,000 or more computers to be harnessed to perform that much computation in only one year.

In November 2005, the RSA challenge number of 193 decimal digits was factored in 3 months using even better algorithms and faster computers (80 2.2 Gigahertz Opteron processors). A 193-decimal digit number is 640 binary bits. Currently it is considered secure to use 1024-bit RSA numbers as keys. The RSA challenge numbers of 704, 768, 896, 1024, 1536, and 2048 bits are still open.

The security of RSA is based on its historical work factor. At this point, there are no known algorithms for factoring large numbers quickly. Although several other public-key ciphers exist, some of which are not covered by patents, to date no public-key system has been found for which one can *prove* a sufficiently large lower bound on the work factor. The best statement one can make now is the work factor based on the best known algorithms. It might be possible that some day a technique is discovered that may lead to fast factoring (e.g., using quantum computation), and thereby undermine the security of RSA.

RSA needs prime numbers; fortunately, there are many of them and generating them is much easier than factoring a product of two primes: "is n prime?" is a much easier question than "what are the factors of n?" There are approximately $n/\ln(n)$ prime numbers less than or equal to n. Thus, for numbers that can be expressed with 1024 bits or fewer, there are approximately 2^{1021} prime numbers. Therefore, we won't run out of prime numbers, if everyone needs two prime numbers different from everyone else's primes. In addition, an adversary won't have a lot of success creating a database that contains all prime numbers because there are so many.

---

* Martin Gardner, *Mathematical games: A new kind of cipher that would take million of years to break*, Scientific American 237, pages 120–124, August 1977.

** K. Leutwyler, *Superhack: forty quadrillion years early, 129-digit code is broken*, Scientific American, 271, 17–20, 1994.

5.9.5.2: Computing a Digital Signature

An important use of public-key ciphers is to implement the `SIGN` and `VERIFY` interface. If this interface is implemented using public-key cryptography, the authentication tag is called a digital signature. The basic idea—which needs refinement to be secure—for computing an RSA digital signature is as follows. `SIGN` produces an authentication tag by raising `M` to the private exponent. `VERIFY` raises the authentication tag to the public exponent, compares the result to the received message, and returns `ACCEPT` if they match and `REJECT` if they don't.

The implementation doesn't always guarantee authenticity, however. For example, if Lucifer succeeds in having Alice sign messages M_1 and M_2, then he can claim that Alice also signed M_3, where M_3 is the product of M_1 and M_2: $(M_3)^d = (M_1 \times M_2)^d = M_1^d \times M_2^d (\text{modulo } n)$. Thus, if Lucifer sends M_3 to Bob, when Bob uses Alice's public key to verify message M_3, that message will appear to have been signed by Alice.

To avoid this problem (and some others) `SIGN` usually computes a cryptographic hash of the message, and creates an authentication tag by raising this hash to the private exponent. This also has the pleasant side effect that it simplifies signing large messages because n only has to be larger than the value of the hash output, and we don't have to worry about splitting the message into blocks and signing each block. Upon receipt, `VERIFY` recomputes the hash from the received version of the message, raises the hash to the public exponent, and compares the result with the received authentication tag.

Using a cryptographic hash helps in constructing a secure `SIGN` and `VERIFY` , but isn't sufficient either. There is a substantial literature that presents even better schemes that also address other subtle issues that come up in the design of a good digital signature scheme.

5.9.5.3: A Public-Key Encrypting System

`ENCRYPT` and `DECRYPT` can also be implemented using public-key cryptography, but because operations in public-key systems are expensive (e.g., exponentiation in RSA instead of `XOR` in RC4), public-key implementations of `ENCRYPT` and `DECRYPT` are used sparingly. As described in Section 5.6, public-key encryption is used only to encrypt a newly-minted shared-secret key during the set up of a connection between a sender and a receiver, and then that secret-secret key is used for shared-secret encryption of further communication between the sender and the receiver. For example, SSL/TLS, which is described in Section 5.11, uses this approach.

The basic idea, which needs refinement to be secure, for implementing `ENCRYPT` and `DECRYPT` using RSA is as follows. Split the message M into fixed size blocks P so that the value of P is smaller than n, then `ENCRYPT` raises P to the public exponent (d). `DECRYPT` raises the encrypted block to the private exponent (e). This order is exactly the opposite of the one for `SIGN` ; `SIGN` raises to the private exponent and `VERIFY` raises to the public exponent.

That the order is the opposite doesn't matter because RSA is reversible. Since $(M^d)^e = (M^e)^d = M^{ed} (\text{modulo } n)$, one can raise to the public exponent (e) first, and raise to the private exponent (d) second, or vice versa, and either way get M back. It is claimed that the security of RSA is equally good both ways.

This basic implementation is relatively weak; there are a number of well-known attacks if the RSA cipher is used by itself for encrypting. To counter these attacks, `ENCRYPT` should pad short blocks with independent randomized variables so that the value of P is close to n, and then raise the padded P to the public exponent. In addition, `ENCRYPT` should run the message through what is called an **all or nothing transform (AONT)**. An AONT is a non-secret, reversible transformation of a message that ensures that the receiver must have all of the bits of the transformed message in order to recover any of the bits of the original message. Thus, an adversary cannot launch an attack by just concentrating on individual blocks of the message. Readers should consult the literature to learn what other measures are necessary to obtain a good implementation of `ENCRYPT` and `DECRYPT` using RSA.

This page titled 5.9: Cryptography as a Building Block (Advanced Topic) is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

5.10: Summary

Section 5.2 of this chapter provided a general perspective on how to think about building secure systems, including a set of design principles, and was then followed by 7 sections of details. One might expect, after reading all this text, that one should now know how to build secure computer systems.

Unfortunately, this expectation is incorrect. Section 5.12 relates several war stories of security system failures that have occurred over a 40-year time span. Failures from decades past might be explained as mistakes while learning that have helped lead to the better understanding now provided in this chapter. But most of the design principles presented in this chapter were formulated and published back in 1975. The section includes several examples of recent failures, which are reinforced by regular reports in the media about yet another virus, worm, distributed denial-of-service attack, identity theft, stolen credit card, or defaced Web site. If we know how to build secure systems, why does the real world of the Internet, corporate services, desktop computers, and personal computers seem to be so vulnerable?

The question does not have a single, simple answer. A lot of different things are tangled together. There are honest and dishonest opinions that the security problem isn't that important, and thus it is unnecessary to get it right. Since organizations prefer not to disclose security problems, it is even difficult to establish what the cost of a security compromise is. Some problems are due to designers just building systems that are too complex. Some problems come from lack of awareness. Some problems are due to designers attempting to build secure systems on Internet time, and not taking the time to do it properly. Some problems arise from ignorance. To get a handle on this general question it is helpful to split the question into several more specific questions:

- The Internet protocols do not provide a default of authentication of message source and privacy of message contents. Why? As discussed in Section 5.2, when the Internet was designed processors weren't fast enough to apply cryptographic transformations in software, the deployment of cryptographic-transformation hardware was hindered by government export regulations, and good key distribution protocols hadn't been designed yet. Since the Internet was originally primarily used by a cooperative set of academics, this lack of security was also not a serious omission. By the time it became economically feasible to do ciphers in software, key distribution was understood, and government export regulations were relaxed, the insecure protocols were so widespread that it was too hard to do a retrofit. Section 5.11 describes one of the now most widely-used secure protocols for Web transactions on the Internet.

- Personal computer systems do not come with enforced modularity that creates strong internal firewalls between applications. Why? The main reasons are keeping the cost low, and naivité. Initially, PCs were designed to be inexpensive computers for personal use. Few people, or perhaps nobody, anticipated that the rapid improvements in technology would lead to the current situation where PCs are the dominant platform for all computing. Furthermore, it took the PC designers and operating system vendors for PCs several iterations to get the designs for enforced modularity correct. Currently vendors are struggling to make PCs easier to configure and manage so that they aren't as vulnerable to attacks.

- Inadequately secured computers are attached to the Internet. Why? Most computers on the Internet are personal computers. When originally conceived personal computers were for *personal* computing, which at the time was editing documents and playing games. Network attacks were impossible, and thus network security was just not a requirement. But the value of being attached to the Internet grew rapidly as the number of available services increased. The result was that most users pursued that evident value, without much concern about the risks, which at first, despite warnings, seemed mostly hypothetical.

- UNIX systems, commonly used as services, have enforced modularity, but many UNIX services were originally (and some still seem to be) vulnerable to buffer-overrun attacks (Sidebar 5.2.4), which subvert modular boundaries. Why are these buffer overruns so difficult to eradicate? As explained in the sidebar, the main reason is the success of the C programming language, which was not designed to check array bounds. Much system software is written in C and has been deployed successfully for decades. A drastic change to the C programming language (or its library) is now difficult because change would break most existing C programs. As a result, each service program must be fixed individually.

- Why isn't software verified for security? Recent progress has been made in analyzing cryptographic algorithms, checking software for common security problems, and verifying security protocols within an adversary model. All these techniques are useful for verifying properties of a system, but they don't prove that a system is secure. In general, we don't know what

properties to verify to proof security.

- Why don't basic economic principles reward the company that produces secure systems? For example, why don't customers buy the more secure products, why don't firms that insure companies against security attacks cause software to be better, etc.? Economics is indeed a factor in information security, but the economic factors interact in surprising ways, and these questions don't have simple answers. Sidebar 5.10.1 summarizes some of the interactions, and their consequences.

Sidebar 5.10.1

Economics of computer security

Why is the company that produces software with fewest security vulnerabilities not the most successful one? Ross Anderson has studied some of the many economic factors in play and analyzed their impact on information security*. First, there are misaligned incentives. For example, under U.S. law it is the bank's burden to prove that a fraudulent withdrawal at an automated teller machine (ATM) is the customer's fault, but under U.K. law, it is the customer's burden to prove that a fraudulent ATM withdrawal is the bank's fault. One might think that U.K. banks spend less money on security, but Anderson reports that the opposite is true: U.K. banks spend more money on security and experience more fraud. It appears that U.K. banks became lazy and careless, knowing that customers complaints of fraud did not require a careful response on their part.

Second, there are network externalities: the larger the network of developers and users, the more valuable that network is to each of its members. Selecting a new operating system partly depends on the number of other people who made the same choice (i.e., because it simplifies exchanging files in closed formats). While an operating system vendor is building market dominance, it must appeal to vendors that complement the operating system as well as the customers. Since security could get in the way of vendors complementing the operating system, operating system vendors have a strong incentive to ignore security in the beginning in favor of features that might help obtain market leadership, and address security later. Unfortunately, adding on security later is never as good as security that is part of the original design.

Third, there are security externalities. For example, if a PC owner considers spending $40 to buy a good firewall, that owner is not the primary beneficiary; what the firewall really protects is targets like Google and Microsoft, because by avoiding becoming a bot, the firewall installer is helping prevent distributed denial-of-service attacks on other sites. Thus the incentive to purchase and install the firewall is low. Bot herders understand this phenomenon well, so they are careful not to attack the files stored on the bots themselves or otherwise give the owner of the bot any incentive to install the firewall.

Finally, security risks are interdependent. A firm's computer infrastructure is often connected to infrastructure under control of others (e.g., the Internet) or uses software written by others, and so the firm's efforts may be undermined by security failures elsewhere. In addition, attacks often exploit a weakness in a system used by many firms. This interdependence makes security risks unattractive to insurers, and as a result there are no market pressures from them.

The impact of economics on computer security is an emerging field of study, and as it develops the explanations might change, the actions of companies may change, but for now it is clear simple economic analysis may miss important interactions.

---

* Ross Anderson and Tyler Moore, *The Economics of Information Security*, Science, 314 (5799), Oct. 2006, pp. 610–613.

- Why doesn't security certification help more? There are no adequate standards for what kind of attacks a minimal secure system should protect against. Standards that do exist for security requirements are out of date because they don’t cover network security. Standardization organizations have a difficult time keeping up with the rate of change in technology.

- Many secure systems require a public key infrastructure, but no universal PKI exists. Why? PKIs exist only in isolated islands, limited to a single institution or application. For example, there is a specialized PKI that supports only the use of SSL/TLS in the World-Wide Web. Why doesn't a universal one exist? A reason is that realistically it is difficult to develop a single one that is satisfactory to everyone. Anyone trying to propose one has run into political and economic problems.

- Many organizations have installed network firewalls between their internal network and the Internet. Do they really help? Yes, but in a limited way, and they have the danger of creating a false sense of security. Because desktop and service operating systems have so many security problems (for the reasons mentioned above), end-to-end security is difficult to achieve. If firewalls are properly deployed they can keep the external, low-budget adversaries away from the vulnerable internal computers. But firewalls don't help against inside adversaries, nor against adversaries that find ways around the firewall to reach the inside network from the outside (e.g., by using the internal wireless network from outside, dialing into a desktop computer that is connected both to the internal network and the telephone system, by hitching rides on data or program files that inside users download through the firewall or load from detachable media, etc.)

- One hears reports that wireless network (WiFi or 802.11b/g) security is weak. This is a relatively new design. Why is it so vulnerable? As mentioned in Section 5.2, one reason appears to be that the security design was done by a committee that was expensive to join, and that only committee members were allowed to review the design. As a result, although the design was nominally open, it was effectively closed, and few security experts actually reviewed the design until after it was deployed, at which point several security weaknesses (for an example see Section 5.5.2) were identified.

- Cable TV scrambling systems, DSS (Satellite TV) security, the CSS system for protecting DVD movie content, and a proposed music watermarking system, were all compromised almost immediately following their deployment. Why were these systems so easy to break? Many of these systems used a closed design and the right people didn't review it. When the system was deployed, experts investigated the design and immediately found problems.

In addition to these more specific reasons, there are two general problems that contribute to the large number of security vulnerability. First, the rate of innovation is high in computer systems. New technologies emerge and are deployed must faster than their designers anticipated and the lack of a security plan in the initial versions becomes a problem suddenly. Furthermore, successful technologies become deployed for applications that the designer didn't anticipate and often turn out to have additional security requirements. Second, no one has a recipe for building secure systems because these systems try to achieve a negative goal. Designing and implementing secure systems requires experts that are extremely careful, have an eye for detail, and exhibit a paranoid attitude. As long as the rate of innovation is high and there is no recipe for engineering secure systems, it is likely that security exploits will be with us. The TLS example in Section 5.11 describes a successful secure protocol (with some growing pains to get it right) and the examples in Section 5.12 illustrate many ways to get things wrong.

This page titled 5.10: Summary is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

5.11: Case Study - Transport Layer Security (TLS) for the Web

The Transport Layer Security (TLS) protocol* is a widely used security protocol to establish a secure channel (confidential and authenticated) over the Internet. The TLS protocol is at the time of this writing a proposed international standard. TLS is a version of the Socket Security Layer (SSL) protocol, defined by Netscape in 1999, so current literature frequently uses the name "SSL/TLS" protocol. The TLS protocol has some improvements over the last version (3) of the SSL protocol, and this case study describes the TLS protocol, version 1.2.

The TLS protocol allows client/service applications to communicate in the face of eavesdroppers and adversaries who would tamper with and forge messages. In the handshake phase, the TLS protocol negotiates, using public-key cryptography, shared-secret keys for message authentication and confidentiality. After the handshake, messages are encrypted and authenticated using the shared-secret keys. This case study describes how TLS sets up a secure channel, its evolution from SSL, and how it authenticates principals.

* Tim Dierks and Eric Rescorla. The Transport Layer Security (TLS) protocol Version 1.2. *RFC 4346*. November 2007.

5.11.1: The TLS Handshake

The TSL protocol consists of several protocols, including the record protocol which specifies the format of messages between clients and services, the alert protocol to communicate errors, the change cipher protocol to apply a cipher suite to messages sent using the record layer protocol, and several handshaking protocols. We describe the handshake protocol for the case where an anonymous user is browsing a Web site and requires service authentication and a secure channel to that service.

Figure 5.11.1 shows the handshake protocol for establishing a connection from a client to a server. The `CLIENTHELLO` message announces to the service the version of the protocol that the client is running (SSL 2.0, SSL 3.0, TLS 1.0, etc.), a random sequence number, and a prioritized set of ciphers and compression methods that the client is willing to use. The *`session_id`* in the `CLIENTHELLO` message is null if the client hasn't connected to the service before.

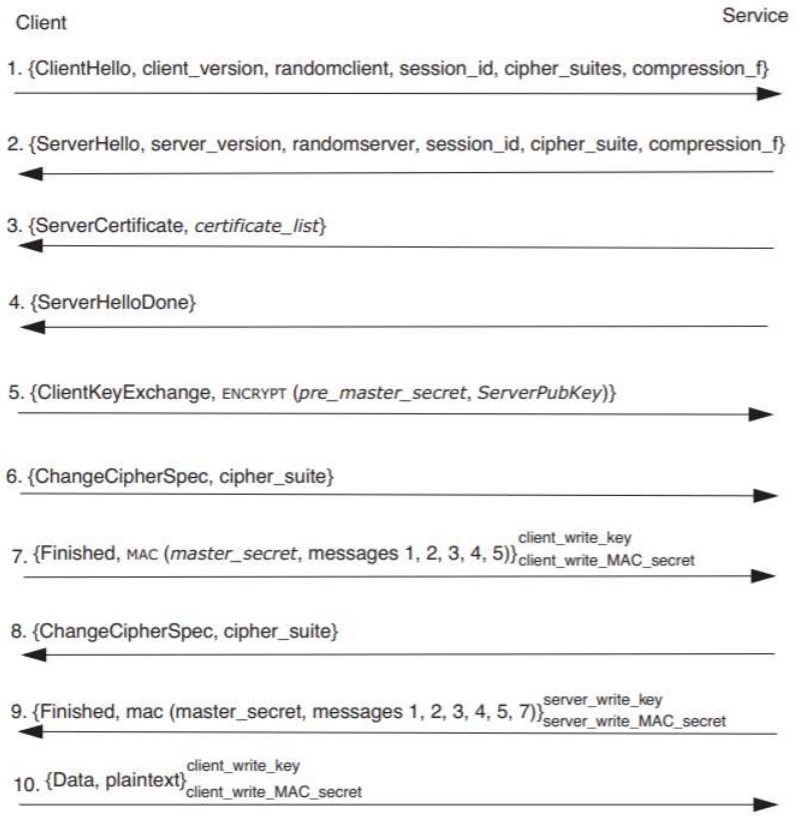


Figure 5.11.1: Typical TLS exchange of handshake protocol messages.

The service responds to the CLIENTHELLO message with 3 messages. It first replies with a SERVERHELLO message, announcing the version of the protocol that will be used (the lower of the one suggested by the client and the highest one supported by the service), a random number, a session identifier, and the cipher suite and compression method selected from the ones offered by the client.

To authenticate the service to the client, the service sends a SERVERCERTIFICATE message. This message contains a chain of certificates, ordered with the service's certificate first followed by any certificate authority certificates proceeding sequentially upward. Usually the list contains just two certificates: a certificate for the public key of the service and a certificate for the public key of the certification authority. (We will discuss certificates in more detail in Section 5.11.3 below.)

After the service sends its certificates, it sends a SERVERHELLODONE message to indicate that it is done with the first part of the handshake. After receiving this message and after satisfactorily verifying the authenticity of the service, the client generates a 48-byte *pre_master_secret*. TLS supports multiple public-key systems, and depending on the choice of the client and service, the *pre_master_secret* is communicated to the service in slightly different ways.

In practice, TLS typically uses a public-key system, in which the client encrypts the *pre_master_secret* with the public key of the service found in the certificate, and sends the result to the service in the CLIENTKEYEXCHANGE message. The *pre_master_secret* thus can be decrypted by any entity that knows the private key that corresponds to the public key in the certificate that the service presented. The security of this scheme therefore depends on the client carefully verifying that the certificate is valid and that it corresponds to the desired service. This point is explored in more detail in Section 5.11.3, below.

The *pre_master_secret* is used to compute the *master_secret* using the service and client nonce (" + " denotes concatenation):

$$master_secret \leftarrow \text{PRF}\ (pre_master_secret,\ \text{“master secret”},\ random_{client} + random_{server})$$

PRF is a pseudorandom function, which takes as input a secret, a label, and a seed. As output it generates pseudorandom bytes. TLS assigns the first 48 bytes of the PRF output to the master_secret. The TLS version 1.2 uses a PRF function that is based on the HMAC construction and the SHA-256 hash function (see Section 5.9 for the HMAC construction and the SHA family of hash functions).

It is important that the *master_secret* be dependent both on the *pre_master_secret* and the random values supplied by the service and client. For example, if the random number of the service were omitted from the protocol, an adversary could replay a recorded conversation without the service being able to tell that the conversation was old.

After the *master_secret* is computed, the *pre_master_secret* should be deleted from memory, since it is no longer needed and continuing to store it would just create an unnecessary security risk.

After sending the encrypted *pre_master_secret*, the client sends a CHANGECIPHERSPEC message. This message* specifies that all future message from the client will use the ciphers specified as the encrypting and authentication ciphers.

The keys for message encrypting and authentication ciphers are computed using the *master_secret*, $random_{client}$, and $random_{server}$ (which both the client and the service now have). Using this information a key block is computed:

$$key_block \leftarrow \text{PRF}\ (master_secret,\ \text{“key expansion”},\ random_{server} + random_{client})$$

until enough output has been produced to provide the following keys:

client_write_MAC_secret[CipherSpec.hash_size]
server_write_MAC_secret[CipherSpec.hash_size]
client_write_key[CipherSpec.key_material]
server_write_key[CipherSpec.key_material]
client_write_IV[CipherSpec.IV_size]
server_write_IV[CipherSpec.IV_size]

The first 4 variables are the keys for authentication and confidentiality, one for each direction. The last 2 variables are the initialization vectors, one for each direction, for ciphers using CBC mode (see Section 5.9). These variables together are the state necessary for the client and the service to communicate securely.

Now the client sends a FINISHED message to announce that it is done with the handshake. The FINISHED message contains at least 12** bytes of the following output:

PRF (*master_secret*, *finish_label*, HASH (*handshake_messages*))

The FINISHED message is a verifier of the protocol sequence so far (the value of all messages starting at the CLIENTHELLO message, but not including the FINISHED message). The client use the value "client finished" for *finish_label* . HASH is the same hash function used for the PRF , SHA-256. If the service verifies the hash, the service and client agree on the protocol sequence and the *master_secret* . TLS encrypts and authenticated the FINISHED message using the cipher suite that the client and service agreed on in the HELLO messages.

After the service receives the client's FINISHED message, it sends a CHANGECIPHERSPEC message, informing the client that all subsequent messages from service to client will be encrypted and authenticated with the specified ciphers. (The client and service can use different ciphers for their traffic.) Like the client, the service concludes the handshake with a FINISHED message, but uses the value "server finished" for *finish_label* . After both finish messages have been received and checked out correctly, the client and service have a secure (that is, encrypted and authenticated) channel over which they can carry on the remainder of their conversation.

---

* The TLS standard considers CHANGECIPHERSPEC not part of the handshake protocol, but part of the Change Cipher Spec protocol, even though the handshake protocol uses it.

** Clients may specify in the HELLO message that they prefer more bytes.

5.11.2: Evolution of TLS

The TLS handshake protocol is more complicated than some of the other protocols that we described in this chapter. In a large part, this complexity is due to all the options TLS supports. It allows a wide range of ciphers and key sizes. Service and client authentication are optional. Also, it supports different versions of the protocol. To support all these options, the TLS protocol needs a number of additional protocol messages. This makes reasoning about TLS difficult, since depending on the client and service constraints, the protocol has a different set of message exchanges, different ciphers, and different key sizes. Partly because of these features the predecessors of TLS 1.2, the earlier SSL protocols, were vulnerable to new attacks, such as cipher suite substitution and version rollback attacks.

In version 2 of SSL, the adversary could edit the CLIENTHELLO message undetected, convincing the service to use a weak cipher, for example one that is vulnerable to brute-force attacks. SSL Version 3 and TLS protect against this attack because the FINISHED message computes a MAC over all message values.

Version 3 of SSL accepts connection requests from version 2 of SSL. This opens a version-rollback attack, in which an adversary convinces the service to use version 2 of the protocol, which has a number of well-documented vulnerabilities, such as the cipher substitution attack. Version 3 appears to be carefully designed to withstand such attacks, but the specification doesn't forbid implementations of version 2 to resume connections that were started with version 3 of the protocol. The security implications of this design are unclear.

One curious aspect of version 3 of the SSL protocol is that the computation for the MAC of the FINISHED messages does not include the CHANGECIPHER messages. As pointed out by Wagner and Schneier, an adversary can intercept the CHANGECIPHER message and delete it, so that the service and client don't update their current cipher suite. Since messages during the handshake are not encrypted and authenticated, this can open a security hole. Wagner and Schneier describe an attack that exploits this observation [Suggestions for Further Reading 5.5.6]. Currently, widely used implementations of SSL 3.0 protect against this attack by accepting a FINISHED message only after receiving a CHANGECIPHER message.

TLS is the international standard version of SSL 3.0, but also improves over SSL 3.0. For example, it mandates that a FINISHED message must follow immediately after a CHANGECIPHER message. It also replaces ad-hoc ways of computing hash functions in various parts of the SSL protocol (e.g., in the FINISHED message and *master_secret*) with a single way, using the PRF function. TLS 1.1 has a number of small security improvements over 1.0. TLS 1.2 improves over TLS 1.1 by replacing an MD5/SHA-1 implementation of PRF with one specified in the cipher suite in the HELLO messages, preferable based on SHA-256. This allows TLS to evolve more easily when ciphers are becoming suspect (e.g., SHA-1).

5.11.3: Authenticating Services with TLS

TLS can be used for many client/service applications, but its main use is for secure Web transactions. In this case, a Web browser uses TLS to set up a message-authenticated, confidential communication connection with a Web service. HTTP requests and responses are sent over this secure connection. Since users typically visit Web sites and perform monetary transactions at these sites, it is important for users to authenticate the service. If users don't authenticate the service, the service might be one run by an adversary who can now record private information (e.g., credit card numbers) and supply fake information. Therefore, a key problem TLS addresses is service authentication.

The main challenge for a client is to convince itself that the service's public key is authentic. If a user visits a Web site, say amazon.com, then a user wants to make sure that the Web site the user connects to is indeed owned by Amazon.com, Inc. The basic idea is for Amazon to sign its name with its private key. Then, the client can verify the signed name using Amazon's public key. This approach reduces the problem to securely distributing the public key for Amazon. If it is done insecurely, an adversary can convince the client that the adversary has the public key of Amazon, but substitute the adversary's own public key and sign Amazon's name with the adversary's private key. This problem is an instance of the key-distribution problem, discussed in Section 5.6.

TLS relies on well-known certification authorities for key distribution. An organization owning a Web site buys a certificate from one or more certification authorities. Each authority runs a certification check to validate that the organization is the one it claims to be. For example, a certification authority might ask Amazon, Inc. for articles of incorporation to prove that it is the entity it claims to be. After the certification authority has verified the identity of the organization, it issues a certificate. The certificate contains the public key of the organization and the name of the organization, signed with the private key of the certificate authority. (The service sends the certificates in step 3 of the handshake protocol, described in Section 5.11.1 above.)

The client verifies the certificate as follows. First, it obtains in a secure way the public key of certification authorities that it is willing to trust. Typically a number of public keys come along with the distribution of a Web browser. Second, after receiving the service certificates, it uses the public keys of the authorities to verify one of the certificates. If one of the certificates verifies correctly, the client can be confident about the name of the organization owning the service. Whether a user can trust the organization that goes by that name is a different question and one that the user must resolve using psychological means.

TLS uses certificates that are standardized by the ISO X.509 standard. Some of the fields in Version 3 of X.509 certificates are shown below (the standard specifies them in a different order):

```
structure certificate
    version
    serial_number
    signature_cipher_identifier
    issuer_signature
    issuer_name
    subject_name
    subject_public_key_cipher_identifier
    subject_public_key
    validity_period
```

The version field specifies the version of the certificate (it would be 3 in this example). The `serial_number` field contains a nonce assigned by the issuing certification authority and is different for every certificate. The `signature_cipher_identifier` field identifies the algorithm used by the authority to sign this certificate. This information allows a client of the certification authority to know which of several standard algorithms to use to verify the `issuer_signature` field, which contains the value of the certificate's signature. If the signature checks out, the recipient can believe that the information in the certificate is authentic. The `issuer_name` field specifies the real-world name of the certificate authority. The `subject_name` field specifies the real-world name for the principal. The two other subject fields specify the public-key cipher the principal wants to use (say RSA), and the principal's public key.

The `validity_period` field specifies the time for which this signature is valid (the start and expiry dates and times). The `validity_period` field provides a weak method for key revocation. If Amazon obtains a certificate and the certificate is valid for 12 months (a typical number), and if the next day an adversary compromises the private key of amazon.com, then the adversary can impersonate Amazon for the next 12 months. To counter this problem a certification authority maintains a

certification revocation list, which contains compromised certificates (identified by the certificate's serial number). Anyone can download the certificate revocation list to check if a certificate is on this blacklist. Unfortunately, revocation lists are not in widespread use today. Good certificate revocation procedures are an open research problem.

The crucial security step for establishing a principal's identity is the certification process executed by the certification authority. If the authority issues certificates without checking out the identity of the organization owning the service, the certificate doesn't improve security. In that case, Lucifer could ask the certification authority to create a certificate for Amazon.com, Inc. If the authority doesn't check Lucifer's identity, Lucifer will obtain a certificate for Amazon, Inc. that binds the name Amazon, Inc. to Lucifer's public key, allowing Lucifer to impersonate Amazon, Inc. Thus, it is important that the certification authority do a careful job of certifying the principal's identity. A typical certification procedure includes paying money to the authority, and sending by surface mail the articles of incorporation (or equivalent) of the organization. The authority will run a partly manual check to validate the provided information before issuing the certificate.

Certification authorities face an inherent conflict between good security and convenience. The procedure must be thorough enough that the certificate means something. On the other hand, the certification procedure must be convenient enough that organizations are able or willing to obtain a certificate. If it is expensive in time and money to obtain a certificate, organizations might opt to go for an insecure solution (i.e., not authenticating their identity with TLS). In practice, certification authorities have a hard time striking the appropriate balance and therefore specialize for a particular market. For example, Verisign, a well-known certification authority, is mostly used by commercial organizations. Private parties who want to obtain a certificate from Verisign for their personal Web sites are likely to find Verisign's certification procedure impractical.

Ford and Baum provide a nice discussion of the current practice for secure electronic commerce using certificate authorities, certificates, etc., and the legal status of certificates [*].

---

[*] Warwick Ford and Michael S. Baum. *Secure Electronic Commerce: Building the Infrastructure for Digital Signatures and Encryption*. Prentice Hall, second edition, 2000. ISBN: 978–0–13–027276–8. 640 pages.

> Although the title implies more generality, this book is about public key infrastructure: certificate authorities, certificates, and their legal status in practice. The authors are a technologist (Ford) and a lawyer (Baum). The book provides thorough coverage and is a good way to learn a lot about the subject. Because the status of this topic changes rapidly, however, it should be considered a snapshot rather than the latest word.

5.11.4: User Authentication

User authentication can, in principle, be handled in the same way as server authentication. The user could obtain a certificate from an authority testifying to the user's identity. When the server asks for it, the user could provide the certificate and the server could verify the certificate (and thus the user's identity according to a certification authority) by using the public key of the authority that issued the certificate. Extensions of the TLS handshake protocol support this form of user authentication.

In practice, and in particular in the Web, user authentication doesn't rely on user certificates. Some organizations run a certificate authority and use it to authenticate members of their organization. However, often it is too much trouble for a user to obtain a certificate, so few Web users are willing to obtain one. Instead, many servers authenticate users based on the IP address of the client machine or based on a shared passphrase. Both methods are currently implemented insecurely.

Using the IP address for authentication is insecure because it is easy for an adversary to spoof an IP address. Thus, when the server checks whether a user on a machine with a particular IP address has access, the server has no guarantees. Typically, this method is used inside an organization that puts all its machines behind a firewall. The firewall attempts to keep adversaries out of the organization's network by monitoring all network traffic that is coming from the Internet and blocking bad traffic (e.g., a packet that is coming from outside the firewall but an internal IP address).

Passphrase authentication is better. In this case, the user sets up an account on the service and protects it with a passphrase that only the user and the service know. Later when the user visits the service again, the server puts up a login page and asks the user to provide the passphrase. If the passphrase is valid, the server assumes that the user is the principal who created the account.

To avoid having the user to type the password on each request, services can exploit a Web mechanism called **cookies**. A service sends a cookie, a service-specific piece of information, to the user's Web browser, which stores it for use in later requests to the service. The service sends the cookie by including in a response a `SET_COOKIE` directive containing data to be stored in the cookie. The browser stores the cookie in memory. (In practice, there may be many cookies, so they are named, but for this

description, assume that there is only one and no name is needed.) On subsequent calls (i.e., `GET` or `POST`) to the service that installed the cookie, the browser sends the installed cookie along with the other arguments to `GET` or `POST` .

Web services can use cookies for user authentication as follows. When the user logs in, the service creates a cookie that contains information to authenticate the user later and sends it to the user's browser, which stores it for use in future requests to this service. Every subsequent request from that browser will include a copy of the cookie, and the service can use the information stored in the cookie to learn which user issued this request. If the cookie is missing (for example, the user is using a different browser), the service will return an error to the browser and ask the user to login again. The security of this scheme depends on how careful the service is in constructing the authenticating cookie. One possibility is to create a nonce for a session and sign the nonce with a MAC. Kevin Fu et al. describe some ways to get it wrong and recommend a secure approach*. Problem Set 28 explores some of the issues in protecting and authenticating cookies.

Websites use cookies in many ways. For example, many sites use cookies to track the browsing patterns of returning visitors. Users who want to protect their privacy must disable cookie tracking in their browser.

---

* K. Fu, E. Sit, K. Smith, and N. Feamster, Dos and don'ts of client authentication on the Web, *Proceedings of the tenth USENIX Security Symposium*, Washington, August 2001.

This page titled 5.11: Case Study - Transport Layer Security (TLS) for the Web is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

5.12: War Stories - Security System Breaches

A designer responsible for system security can bring to the job three different, related assets. The first is an understanding of the fundamental security concepts discussed in the main body of this chapter. The second is knowledge of several different real security system designs; some examples have been discussed elsewhere in this chapter and more can be found in the Suggestions for Further Reading. This section concentrates on a third asset: familiarity with examples of real-world breaches of security systems. In addition to encouraging a certain amount of humility, one can develop from these case studies some intuition about approaches that are inherently fragile or difficult to implement correctly. They also provide evidence of the impressive range of considerations that a designer of a security system must consider.

The case studies selected for description all really happened, although inhibitions have probably colored some of the stories. Failures can be embarrassing, have legal consequences, or, if publicized, jeopardize production systems that have not yet been repaired or redesigned. For this reason, many of the cases described here were, when they first appeared in public, sanitized by omitting certain identifying details or adding misleading "facts". Years later, reconstructing the missing information is difficult, as is distinguishing the reality from any fantasy that was added as part of the disguise. To help separate fact from fiction, this section cites original sources wherever they are available.

The case studies start in the early 1960s, when the combination of shared computers and durable storage first brought the need for computer security into focus. In several examples, an anecdote describing a vulnerability discovered and a countermeasure devised decades ago is juxtaposed with a much more recent example of essentially the same vulnerability being again found in the field. The purpose is not to show that there is nothing new under the sun, but rather to emphasize Santayana's warning that "Those who cannot remember the past are condemned to repeat it."*

At the same time it is important to recognize that the rapid improvement of computer hardware technology over the last 40 years has created new vulnerabilities. Technology improvement has provided us with new case studies of security breaches in several ways:

- Adversaries can bring to bear new tools. For example, performance improvements have enabled previously infeasible attacks on security such as brute force key space searches.
- Cheap computers have increased the number of programmers much faster than the number of security-aware programmers.
- The attachment of computer systems to data communication networks has, from the point of view of a potential adversary, vastly increased the number of potential points of attack.
- Rapid technology change has encouraged giving high priority to rolling out new features and applications, so the priority of careful attention to security suffers.
- Technology improvement has enabled the creation of far more complex systems. Complexity is a progenitor of error, and error is a frequent cause of security vulnerabilities.

Although it is common to identify a single mistake that was the proximate cause of a security breach, if one *keeps digging* it is usually possible to establish that several violations of security principles contributed to making the breach possible, and thus to failure of defense in depth.

---

* George Santayana, *The Life of Reason, Volume 1, Introduction and Reason in Common Sense* (Scribner's: 1905).

5.12.1: Residues: Profitable Garbage

Security systems sometimes fail because they do not protect **residues**, the analyzable remains of a program or data after the program has finished. This general attack has been reported in many forms; adversaries have discovered secrets by reading the contents of newly allocated primary memory, secondhand hard disks, and recycled magnetic tapes as well as by pawing through piles of physical trash (popularly known as "dumpster diving").

5.12.1.1: 1963: Residues in CTSS

In the M.I.T. Compatible Time-Sharing System (CTSS), a user program ran in a memory region of an allocated size, and the program could request a change in allocation by calling the operating system. If the user requested a larger allocation, the system assigned an appropriate block of memory. Early versions of the system failed to clear the contents of the newly allocated block, so the residue of some previous program would be accessible to any other program that extended its memory size.

At first glance, this oversight seems to provide an attacker with the ability to read only an uncontrollable collection of garbage, which appears hard to exploit systematically. An industrious penetrator noticed that the system administrator ran a self-rescheduling job every midnight that updated the primary accounting and password files. On the assumption that the program processed the password file by first reading it into primary memory, the penetrator wrote a program that extended its own memory size from the minimum to the maximum, and then searched the residue in the newly assigned area for the penetrator's own password. If the program found that password, it copied the entire memory residue to a file for later analysis, expecting that it might also contain passwords of other users. The penetrator scheduled the program to go into operation just before midnight, and then reschedule itself every few seconds. It worked well. The penetrator soon found in the residue a section of the file relating user names and passwords.*

Lesson 5.12.1.1:

A design principle applies: use *fail-safe defaults*. In this case, the fail-safe default is for the operating system memory allocator to clear the contents of newly-allocated memory.

* Reported on CTSS by Maxim G. Smith in 1963. The identical problem was found in the General Electric GCOS system when its security was being reviewed by the U.S. Defense Department in the 1970's, as reported by Roger R. Schell. Computer Security: the Achilles' heel of the electronic Air Force? *Air University Review XXX*, 2 (January-February 1979) page 21.

5.12.1.2: 1997: Residues in Network Packets

If one sends a badly formed request to a Kerberos Version 4 server (Sidebar 5.6.1 describes the Kerberos authentication system), the service responds with a packet containing an error message. Since the error packet was shorter than the minimum frame size, it had to be padded out to reach the minimum frame size. The problem was that the padding region wasn't being cleared, so it contained the residue of the previous packet sent out by that Kerberos service. That previous packet was probably a response to a correctly formed request, which typically includes both the Kerberos realm name and the plaintext principal identifier of some authorized user. Although exposing the principal identifier of an authorized user to an adversary is not directly a security breach, the first step in mounting a dictionary attack (to which Kerberos is susceptible) is to obtain a principal identifier of an active user and the exact syntax of the realm name used by this Kerberos service.*

Lesson 5.12.1.2:

As in example 5.12.1.1, above, use *fail-safe defaults*. The packet buffer should have been cleared between uses.

* Reported by L0pht Heavy Industries in 1997, after the system had been in production use for ten years.

5.12.1.3: 2000: Residues in HTTP

To avoid retransmitting an entire file following a transmission failure, the HyperText Transfer Protocol (HTTP), the primary transport mechanism of the World Wide Web, allows a client to ask a service for just a portion of a file, describing that part by a starting address and a data length. If the requested region lies beyond the end of the file, the protocol specifies that the service return just the data up to the end of the file and alert the client about the error.

The Apple Macintosh AppleShare Internet Web service was discovered to return exactly as much data as the client requested. When the client asked for more data than was actually in the file, the service returned as much of the file as actually existed, followed by whatever data happened to be in the service's primary memory following the file. This implementation error allowed any client to mine data from the service.*

Lesson 5.12.1.3:

Apparently unimportant specifications, such as "return only as much data as is actually in the file," can sometimes be quite important.

* Reported Monday, 17 April 2000 to an (unidentified) Apple Computer technical support mailing list by Clint Ragsdale, followed up by analysis by Andy Griffin in *Macintouch* (Tuesday 18 April 2000), http://www.macintouch.com/.

5.12.1.4: Residues on Removed Disks

The potential for analysis of residues turns up in a slightly different form when a technician is asked to repair or replace a storage device such as a magnetic disk. Unless the device is cleared of data first, the technician may be able to read it. Clearing a disk is generally done by overwriting it with random data, but sometimes the reason for repair is that the write operation isn't working. Worse, if the hardware failure is data-dependent, it may be essential that the technician be allowed to read the residue to reproduce and diagnose the failure.

In November 1998, the dean of the Harvard Divinity School was sacked after he asked a University technician to upgrade his personal computer to use a new, larger hard disk and transfer the contents of the old disk to the new one. When the technician's supervisor asked why the job was taking so long, the technician, after some prodding, reluctantly replied that there seemed to be a large number of image files to transfer. That reply led to further questions, upon which it was discovered that the image files were pornographic.*

Lesson 5.12.1.4:

Physical possession of storage media usually allows bypass of security measures that are intended to control access within a system. The technician who removes a disk doesn't need a password to read it. Encryption of stored files can help minimize this problem.

* James Bandler. Harvard ouster linked to porn; Divinity School dean questioned. *Boston Globe* (Wednesday, 19 May 1999) City Edition, page B1, Metro/Region section.

5.12.1.5: Residues in Backup Copies

It is common practice for a data-storing system to make periodic backup copies of all files onto magnetic tape, often in several different formats. One format might allow quick reloading of all files, while another might allow efficient searching for a single file. Several backup copies, perhaps representing files at one-week intervals for a month, and at one-month intervals for a year, might be kept.

The administrator of a Cambridge University time-sharing system was served with an official government request to destroy all copies of a specific file belonging to a certain user. The user had compiled a list of secret telephone access codes, which could be used to place free long-distance calls. Removing the on-line file was straightforward, but the potential cost of locating and expunging the backup copies of that file—while maintaining backup copies of all other files—was enormous. (A compromise was reached, in which the backup tapes received special protection until they were due to be recycled.)[1]

A similar, more highly publicized backup residue incident occurred in November 1986, when Navy Vice-Admiral John M. Poindexter and Lieutenant Colonel Oliver North deleted 5,748 email messages in connection with the Iran-Contra affair. They apparently did not realize that the PROFS email system used by the National Security Council maintained backup copies. The messages found on the backup tapes became important evidence in subsequent trials of both individuals. An interesting aspect of this case was that the later investigation focused not just on the content of specific messages, but on their context in relation to other messages, which the backup system also preserved.[2,3]

Lesson 5.12.1.5:

There is a tension between reliability, which calls for maintaining multiple copies of data, and security, which is enhanced by minimizing extra copies.

[1] Incident ca. 1970, reported by Roger G. Needham.

[2] Lawrence E. Walsh. *Final report of the independent counsel for Iran/Contra matters* Volume 1, Chapter 3 (4 August 1993). U.S. Court of Appeals for the District of Columbia Circuit, Washington, D.C.

[3] The context issue is highlighted in *Armstrong v. Bush*, 721 F. Supp. 343, 345 n.1 (D.D.C. 1989).

5.12.1.6: Magnetic Residues: High-Tech Residue Analysis

A more sophisticated version of the residue problem is encountered when recording on continuous media such as magnetic tape or disk. If the residue is erased by overwriting, an ordinary read to the disk will no longer return the previous data. However, analysis

of the recording medium in the laboratory may disclose residual magnetic traces of previously recorded data. In addition, many disk controllers automatically redirect a write to a spare sector when the originally addressed sector fails, leaving on the original sector a residue that a laboratory can retrieve. For these reasons, certain U.S. Department of Defense agencies routinely burn magnetic tapes and destroy magnetic disk surfaces in an acid bath before discarding them.*

* *Remanence Security Guidebook*. Naval Staff Office Publication NAVSO P-5239-26 (September 1993:United States Naval Information Systems Management Center: Washington D.C.)

5.12.1.7: 2001 and 2002: More Low-Tech Garbage Analysis

The lessons about residues apparently have not yet been completely absorbed by system designers. In July 2001, a user of the latest version of the Microsoft Visual C++ compiler who regularly clears the unused part of his hard disk by overwriting it with a characteristic data pattern discovered copies of that pattern in binary executables created by the compiler. Apparently the compiler allocated space on the disk as temporary storage but did not clear that space before using it.* In January 2002, people who used the Macintosh operating system to create CDs for distribution were annoyed to find that most disk-burning software, in order to provide icons for the files on the CD, simply copied the current desktop database, which contains those icons, onto the CD. But this database file contains icons for every application program of the user, as well as incidental other information about many of the files on the user's personal hard disks—such as the World-Wide Web address from which they were downloaded. Thus users who received such CDs found that in addition to the intended files, there was a remarkable, and occasionally embarrassing, collection of personal information there, too.

Lesson 5.12.1.7:

"Visit with your predecessors… They know the ropes and can help you see around some corners. Try to make original mistakes, rather than needlessly repeating theirs."**

** Donald Rumsfeld, "Rumsfeld's Rules: Advice on Government, Business, and Life," 1974. A later version appeared as an op-ed submission in *The Wall Street Journal*, 29 January 2001.

* David Winfrey. "Uncleared disk space and MSVC." *Risks Forum Digest 21*, 50 (12 July 2001).

5.12.2: Plaintext Passwords Lead to Two Breaches

Some design choices, while not directly affecting the internal security strength of a system, can affect operational aspects enough to weaken system security.

In CTSS, as already mentioned, passwords were stored in the file system together with usernames. Since this file was effectively a master user list, the system administrator, whenever he changed the file, printed a copy for quick reference. His purpose was not to keep track of passwords. Rather, he needed the list of usernames to avoid duplication when adding new users. This printed copy, including the passwords, was processed by printer controller software, handled by the printer operator, placed in output bins, moved to the system administrator's office, and eventually discarded by his secretary when the next version arrived. At least one penetration of CTSS was accomplished by a student who discovered an old copy of this printed report in a wastebasket (another example of a residue problem).*

Lesson 5.12.2.1:

Pay attention to the *least privilege principle*: don't store your lunch (in this case, the user names) in the safe with the jewels (the passwords).

At a later time, another system administrator was reviewing and updating the master user list, using the standard text editor. The editor program, to ensure atomic update of the file, operated by creating a copy of the original file under a temporary name, making all changes to that copy, and at the end renaming the copy to make it the new original. Another system operator was working at the same time as the system administrator, using the same editor to update a different file in the same directory. The different file was the "message of the day," which the system automatically displayed whenever a user logged in. The two instances of the editor used the same name for their intermediate copies, with the result that the master user list, complete with passwords, was posted as the

message of the day. Analysis revealed that the designer of the editor had, as a simplification, chosen to use a fixed name for the editor's intermediate copy. That simplification seemed reasonable because the system had a restriction that prevented two different users from working in the same directory at the same time. But in an unrelated action, someone else on the system programming staff had decided that the restriction was inconvenient and unnecessary, and had removed the interlock.**

Lesson 5.12.2.2:

Removing interlocks can be risky because it is hard to track down every part of the system that depended on the interlock being there.

This lesson is not restricted to security.

---

* Reported by Richard G. Mills, 1963.

** Fernando J. Corbató. "On building systems that will fail." *Communications of the ACM 34*, 9 (September, 1991) page 77. This 1966 incident led to the use of one-way transformations for stored pass word records in Multics, the successor system to CTSS. But see item 5.12.3, which follows.

5.12.3: The Multiply Buggy Password Transformation

Having been burned by residues and weak designs on CTSS, the architects of the Multics system specified and implemented a (supposedly) one-way cryptographic transformation on passwords before storing them, using the same one-way transformation on typed passwords before comparing them with the stored version. A penetration team mathematically examined the one-way transformation algorithm and discovered that it wasn't one-way after all: an inverse transformation existed.

Lesson 5.12.3.1

Amateurs should not dabble in crypto-mathematics.

To their surprise, when they tried the inverse transformation it did not work. After much analysis, the penetration team figured out that the system procedure implementing the supposedly one-way transformation used a mathematical library subroutine that contained an error, and the passwords were being transformed incorrectly. Since the error was consistent, it did not interfere with later password comparisons, so the system performed password authentication correctly. Further, the erroneous algorithm turned out to be reversible too, so the system penetration was successful.

An interesting sidelight arose when penetration team reported the error in the mathematical subroutine and its implementers released a corrected update. Had the updated routine simply been installed in the library, the password-transforming algorithm would have begun working correctly. But then, correct user-supplied passwords would transform to values that did not match the stored values previously created using the incorrect algorithm. Thus, no one would be able to log in. A creative solution (which the reader may attempt to reinvent) was found for the dilemma. *

---

* Peter J. Downey. *Multics Security Evaluation: Password and File Encryption Techniques*. United States Air Force Electronics Systems Division Technical Report ESD–TR–74–193, Vol. III (June 1977).

5.12.4: Controlling the Configuration

Even if one has applied a consistent set of security techniques to the hardware and software of an installation, it can be hard to be sure that they are actually effective. Many aspects of security depend on the exact configuration of the hardware and software—that is, the versions being used and the controlling parameter settings. Mistakes in setting up or controlling the configuration can create an opportunity for an attacker to exploit. Before Internet-related security attacks dominated the news, security consultants usually advised their clients that their biggest security problem was likely to be unthinking or unauthorized action by an authorized person. In many systems the number of people authorized to tinker with the configuration is alarmingly large.

5.12.4.1: Authorized People Sometimes Do Unauthorized Things

A programmer was temporarily given the privilege of modifying the kernel of a university operating system as the most expeditious way of solving a problem. Although he properly made the changes appropriate to solve the problem, he also added a feature to a rarely-used metering entry of the kernel. If called with a certain argument value, the metering entry would reset the status of the current user's account to show no usage. This new "feature" was used by the programmer and his friends for months afterwards to obtain unlimited quantities of service time.*

* Reported by Richard G. Mills, 1965.

5.12.4.2: The System Release Trick

A Department of Defense operating system was claimed to be secured well enough that it could safely handle military classified information. A (fortunately) friendly penetration team looked over the system and its environment and came up with a straightforward attack. They constructed, on another similar computer, a modified version of the operating system that omitted certain key security checks. They then mailed to the DoD installation a copy of a tape containing this modified system, together with a copy of the most recent system update letter from the operating system vendor. The staff at the site received the letter and tape, and duly installed its contents as the standard operating system. A few days later one of the team members invited the management of the installation to watch as he took over the operating system without the benefit of either a user id or a password.*

Lesson 5.12.4.2:

Complete mediation includes checking the authenticity, integrity, and permission to install of software releases, whether they arrive in the mail or are downloaded over the Internet.

* This story has been in the folklore of security for at least 25 years, but it may be apocryphal. A similar tale is told of mailing a a bogus field change order, which would typically apply to the hardware, rather than the software, of a system. The folklore is probably based on a 1974 analysis of operating practices of United States Defense contractors and Defense Department sites that outlined this attack possibility in detail and suggested strongly that mailing a bogus software update would almost certainly result in its being installed at the target site. The authors never actually tried the attack. Paul A. Karger and Roger R. Schell. *MULTICS Security Evaluation: Vulnerability Analysis*. United States Air Force Electronics Systems Division Technical Report ESD–TR–74–193 Vol. II (June 1974), Section 3.4.5.1.

5.12.4.3: The Slammer Worm *

A malware program that copies itself from one computer to another over a network is known as a “worm”. In January 2003 an unusually virulent worm named Slammer struck, demonstrating the remarkable ease with which an attacker might paralyze the otherwise robust Internet. Slammer did not quite succeed because it happened to pick on an occasionally used interface that is not essential to the core operation of the Internet. If Slammer had found a target in a really popular interface, the Internet would have locked up before anyone could do anything about it, and getting things back to even a semblance of normal operation would probably have taken a long time.

The basic principle of operation of Slammer was stunningly simple:

1. Discover an Internet port that is enabled in many network-attached computers, and for which a popular listener implementation has a buffer overrun bug that a single, short packet can trigger. Internet Protocol UDP ports are thus a target of choice. Slammer exploited a bug in Microsoft SQL Server 2000 and Microsoft Server Desktop Engine 2000, both of which enable the SQL UDP port. This port is used for database queries, and it is vulnerable only on computers that run one of these database packages, so it is by no means universal.
2. Send to that port a packet that overruns a buffer, captures the execution point of the processor, and runs a program contained in the packet.
3. Write that program to go into a tight loop, generating an Internet address at random and sending a copy of the same packet to that address, as fast as possible. The smaller the packet, the more packets per second the program can launch. Slammer used packets that were, with headers, 404 bytes long, so a broadband-connected (1 megabit/second) machine could launch packets at a rate of 300 per second, a machine with a 10 megabits/second path to the Internet could launch packets at a rate of 3,000 per second and a high-powered server with a 155 megabits/second connection might be able to launch as many as 45,000 packets per second.

Forensics: Receipt of this single Slammer worm packet is enough to instantly recruit the target to help propagate the attack to other vulnerable systems. An interesting forensic problem is that recruitment modifies no files and leaves few traces because the worm exists only in volatile memory. If a suspicious analyst stops a recruited machine, disconnects it from the Internet, and reboots it, the analyst will find nothing. There may be some counters indicating that there was a lot of outbound network traffic, but no clue why. So one remarkable feature of this kind of worm is the potential difficulty of tracing its source. The only forensic information available is likely to be the payload of the intentionally tiny worm packet.

Exponential attack rate: A second interesting observation about the Slammer worm is how rapidly it increased its aggregate rate of attack. It recruited every vulnerable computer on the Internet as both a prolific propagator and also as an intense source of Internet traffic. The original launcher needed merely to find one vulnerable machine anywhere in the Internet and send it a single worm packet. This newly-recruited target immediately began sending copies of the worm packet to other addresses chosen at random. Internet version 4, with its 32-bit address fields, provided about 4 billion addresses, and even though many of them were unassigned, sooner or later one of these worm packets was likely to hit another machine with the same vulnerability. The worm packet immediately recruited this second machine to help with the attack. The expected time until a worm packet hit yet another vulnerable machine dropped in half and the volume of attack traffic doubled. Soon third and fourth machines were recruited to join the attack; thus the expected time to find new recruits halved again and the malevolent traffic rate doubled again. This epidemic process proceeded with exponential growth until either a shortage of new, vulnerable targets or bottlenecked network links slowed it down; the worm quickly recruited every vulnerable machine attached to the Internet.

The exponent of growth depends on the average time it takes to recruit the next target machine, which in turn depends on two things: the number of vulnerable targets and the rate of packet generation. From the observed rate of packet arrivals at the peak, a rough estimate is that there were 50 thousand or more recruits, launching at least 50 million packets per second into the Internet. The aggregate extra load on the Internet of these 3200-bit packets probably amounted to something over 150 Gigabits/second, but that is well below the aggregate capacity of the Internet, so reported disruptions were localized rather than universal.

With 50,000 vulnerable ports scattered through a space of 4 billion addresses, the chance that any single packet hits a vulnerable port is one in 120,000. If the first recruit sends one thousand packets per second, the expected time to hit a vulnerable port would be about two minutes. In four minutes there would be four recruits. In six minutes, eight recruits. In half an hour, nearly all of the 50,000 vulnerable machines would probably be participating.

Extrapolation: The real problem appears if we redo that analysis for a port to which five million vulnerable computers listen: the time scale drops by two orders of magnitude. With that many listeners, a second recruit would receive the worm and join the attack within one second, two more one second later, etc. In less than 30 seconds, most of the 5 million machines would be participating, each launching traffic onto the Internet at the fastest rate they (or their Internet connection) can sustain. This level of attack, about two orders of magnitude greater than the intensity of Slammer, would almost certainly paralyze every corner of the Internet. It could take quite a while to untangle because the overload of every router and link would hamper communication among people who are trying to resolve the problem. In particular, it could be difficult for owners of vulnerable machines to learn about and download any necessary patches.

Prior art: Slammer used a port that is not widely enabled, yet its recruitment rate, which determines its exponential growth rate, was at least one and perhaps two orders of magnitude faster than that reported for previous generations of fast-propagating worms. Those worms attacked much more widely-enabled ports, but they took longer to propagate because they used complex multipacket protocols that took much longer to set up. The Slammer attack demonstrates the power of brute force. By choosing a UDP port, infection can be accomplished by a single packet, so there is no need for a time-consuming protocol interchange. The smaller the packet size, the faster a recruit can then launch packets to discover other vulnerable ports.

Another risk: The worm also revealed a risk of networks that advertise a large number of addresses. At the time that individual computers that advertise a single address were receiving one Slammer worm packet every 80 seconds, a network that advertises 16 million addresses would have been receiving 200,000 packets/second, with a data rate of about 640 megabits/second. In confirmation, incoming traffic to the M.I.T. network border routers, which actually do advertise 16 million addresses, peaked at a measured rate of around 500 megabits/second with some of its links to the public Internet saturated. Being the home of 16 million Internet addresses has its hazards.

Lesson 5.12.4.3:

From this incident we can draw different lessons for different network participants: For users, the perennial but often-ignored advice to disable unused network ports does more than help a single computer resist attack, it helps protect the entire network. For vendors, shipping an operating system that by default activates a listener for a feature that the user does not explicitly request is hazardous to the health of the network (*use failsafe defaults*). For implementers, it emphasizes the importance of diligent care (and paranoid design) in network listener implementations, especially on widely activated UDP ports.**

** A detailed analysis of the Slammer worm and its effects on the Internet can be found in David Moore, et al., "Inside the Slammer Worm", *IEEE Security and Privacy 1*, 4 (July 2003) pages 33-39.

* This account is based on one originally published under the title "Slammer: an urgent wake-up call", pages 243–248 in *Computer Systems: theory, technology and applications/A tribute to Roger Needham*, Andrew Herbert & Karen Spärck Jones, editors. (Springer: New York: 2004).

5.12.5: The Kernel Trusts the User

5.12.5.1: Obvious Trust

In the first version of CTSS, a shortcut was taken in the design of the kernel entry that permitted a user to read a large directory as a series of small reads. Rather than remembering the current read cursor in a system-protected region, as part of each read call the kernel returned the cursor value to the caller. The caller was to provide that cursor as an argument when calling for the next record. A curious user printed out the cursor, concluded that it looked like a disk sector address, and wrote a program that specified sector zero, a starting block that contained the sector address of key system files. From there he was able to find his way to the master user table containing (as already mentioned, plaintext) passwords.*

Although this vulnerability seems obvious, many operating systems have been discovered to leave some critical piece of data in an unprotected user area, and later rely on its integrity. In OS/360, the operating system for the IBM System/360, each system module was allocated a limited quota of system-protected storage, as a strategy to keep the system small. Since the quota was unrealistically small in many cases, system programmers were effectively forced to place system data in unprotected user areas. Despite many later efforts to repair the situation, an acceptable level of security was never achieved in that system.

Lesson 5.12.5.1

A bit more attention to paranoid design would have avoided these problems.

* Noticed by the author, exploit developed by Maxim G. Smith, 1963.

5.12.5.2: Nonobvious Trust (Tocttou)

As a subtle variation of the previous problem, consider the following user-callable kernel entry point:

```
1 procedure DELETE_FILE (file_name)
2     auth ← CHECK_DELETE_PERMISSION (file_name, this_user_id)
3     if auth = PERMITTED
4         then DESTROY (file_name)
5         else signal ("You do not have permission to delete file_name")
```

This program seems to be correctly checking to verify that the current user (whose identity is found in the global variable `this_user_id`) has permission to delete the file `file_name` . But, because the code depends on the meaning of `file_name` not changing between the call to `CHECK_DELETE_PERMISSION` on line 2 and the call to `DESTROY` on line 4, in some systems there is a way to defeat the check.

Suppose that the system design uses indirection to decouple the name of a file from its permissions (as for example, in the UNIX file system, which stores its permissions in the inode). With such a design, the user can, in a concurrent thread, unlink and then relink the name `file_name` to a different file, thereby causing deletion of some other file that

CHECK_DELETE_PERMISSION would not have permitted. There is, of course a race—the user's concurrent thread must perform the unlinking and relinking in the brief interval between when CHECK_DELETE_PERMISSION looks up *file_name* in the file system and DESTROY looks up that same name again. Nevertheless, a window of opportunity does exist, and a clever adversary may also be able to find a way to stretch out the window.

This class of error is so common in kernel implementations that it has a name: "Time Of Check To Time Of Use" error, written "tocttou" and pronounced "tock-two."*

Lesson 5.12.5.2

For *complete mediation* to be effective, one must also consider the dynamics of the system. If the user can change something after the guard checks for authenticity, integrity, and permission, all bets are off.

* Richard Bisbey II, Gerald Popek, and Jim Carlstedt. *Protection errors in operating systems: inconsistency of a single data value over time*. USC/Information Sciences Institute Technical Report SR–75–4 (January 1976).

5.12.5.3: Tocttou 2: Virtualizing the DMA Channel

A common architecture for Direct Memory Access (DMA) input/output channel processors is the following: DMA channel programs refer to absolute memory addresses without any hardware protection. In addition, these channel programs may be able to modify themselves by reading data in over themselves. If the operating system permits the user to create and run DMA channel programs, it becomes difficult to enforce security constraints, and even more difficult for an operating system to create virtual DMA channels as part of a virtual machine implementation. Even if the channel programs are reviewed by the operating system to make sure that all memory addresses refer to areas assigned to the user who supplied the channel program, if the channel program is selfmodifying, the checks of its original content are meaningless. Some system designers try to deal with this problem by enforcing a prohibition on timing-dependent and self-modifying DMA channel programs. The problem with this approach was that it is difficult to methodically establish by inspection that a program conforms with the prohibition. The result is a battle of wits: for every ingenious technique developed to discover that a DMA channel program contains an obscure self-modification feature, some clever adversary may discover a still more obscure way to conceal self-modification. Precisely such a problem was noted with virtualization of I/O channels in the IBM System/360 architecture and its successors.*

Lesson 5.12.5.3:

It can be a major challenge to apply *complete mediation* to a legacy hardware architecture.

* This battle of wits is well known to people who have found themselves trying to "virtualize" existing computer architectures, but apparently the only specific example that has been documented is in C[lement]. R[ichard]. Attanasio, P[eter] W. Markstein and R[ay]. J. Philips, "Penetrating an operating system: a study of VM/370 integrity," *IBM System Journal 15*, 1 (1976), pages 102–117.

5.12.6: Technology Defeats Economic Barriers

5.12.6.1: An Attack on our System Would be Too Expensive

A Western Union vice president, when asked if the company was using encryption to protect the privacy of messages sent via geostationary satellites, dismissed the question by saying, "Our satellite ground stations cost millions of dollars apiece. Eavesdroppers don't have that kind of money."* This response seems oblivious of two things: (1) an eavesdropper may be able to accomplish the job with relatively inexpensive equipment that does not have to meet commercial standards of availability, reliability, durability, maintainability, compatibility, and noise immunity, and (2) improvements in technology can rapidly reduce an eavesdropper's cost. The next anecdote provides an example of the second concern.

Lesson 5.12.6.1:

Never underestimate the effect of technology improvement, and the effectiveness of the resources that a clever adversary may bring to bear.

* Reported by F. J. Corbató, ca. 1975.

5.12.6.2: Well, It Used to be Too Expensive

In 2003, the University of Texas and Georgia Tech were victims of an attack made possible by advancing computer and network technology. The setup went as follows: The database of student, staff, and alumni records included in each record a field containing that person's Social Security number. Furthermore, the Social Security number field was a key field, which means that it could be used to retrieve records. The assumption was that this feature was useful only to a client who knew a Social Security number.

The attackers realized that the universities had a high-performance database service attached to a high-bandwidth network, and it was therefore possible to systematically try all of the 999 million possible Social Security numbers in a reasonably short time—in other words, a dictionary attack. Most trials resulted in a "no such record" response, but each time an offered Social Security number happened to match a record in the database, the service returned the entire record for that person, thereby allowing the Social Security number to be matched with a name, address, and other personal information. The attacks were detected only when it was noticed that the service seemed to be experiencing an unusually heavy load.*

Lesson 5.12.6.2

As technology improves, so do the tools available for adversaries.

* Robert Lemos. "Data thieves nab 55,000 student records." *CNET News.com*, March 6, 2003. Robert Lemos. "Data thieves strike Georgia Tech." *CNET News.com*, March 31, 2003.

5.12.7: Mere Mortals Must be Able to Figure Out How to Use It

In an experiment at Carnegie-Mellon University, Alma Whitten and Doug Tygar engaged twelve subjects who were experienced users of email, but who had not previously tried to send secure email. The task for these subjects was to figure out how to send a signed and encrypted message, and decrypt and authenticate the response, within 90 minutes. They were to use the cryptographic package Pretty Good Privacy (PGP) together with the Eudora email system, both of which were already installed and configured to work together.

Of the twelve participants, four succeeded in sending the message correctly secured; three others sent the message in plaintext thinking that it was secure, and the remaining five never figured out how to complete the task. The report on this project provides a step-by-step analysis of the mistakes and misconceptions encountered by each of the twelve test subjects. It also includes a cognitive walkthrough analysis (that is, an *a priori review*) of the user interface of PGP.*

Lessons 5.12.7:

1. The mental model that a person needs to make correct use of public-key cryptography is hard for a non-expert to grasp; a simpler description is needed.
2. Any undetected mistake can compromise even the best security. Yet it is well known that it requires much subtlety to design a user interface that minimizes mistakes. The *principle of least astonishment* applies.

* Alma Whitten and J. D. Tygar. *Usability of Security: A Case Study*. Carnegie-Mellon University School of Computer Science Technical Report CMU–CS–98–155, December 1998. A less detailed version appeared in Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. *Proceedings of the eighth USENIX security symposium*, August 1999.

5.12.8: The Web can be a Dangerous Place

In the race to create the World Wide Web browser with the most useful features, security sometimes gets overlooked. One potentially useful feature is to launch the appropriate application program (called a helper) after downloading a file that is in a format not handled directly by the browser. However, launching an application program to act on a file whose contents are specified by someone else can be dangerous.

Cognizant of this problem, the Microsoft browser, named Internet Explorer, maintained a list of file types, the corresponding applications, and a flag for each that indicates whether or not launching should be automatic or the user should be asked first. When initially installed, Internet Explorer came with a pre-configured list, containing popular file types and popular application programs. Some flags were preset to allow automatic launch, indicating that the designer believed certain applications could not possibly cause any harm.

Apparently, it is harder than it looks to make such decisions. So far, three different file types whose default flags allow automatic launch have been identified as exploitable security holes on at least some client systems:

- Files of type ".LNK", which in Windows terminology are called "shortcuts" and are known elsewhere as symbolic links. Downloading one of these files causes the browser to install a symbolic link in the client's file system. If the internals of the link indicate a program at the other end of the link, the browser then attempts to launch that program, giving it arguments found in the link.
- Files of type ".URL", known as "Internet shortcuts", which contain a URL. The browser simply loads this URL, which would seem to be a relatively harmless thing to do. But a URL can be a pointer to a local file, in which case the browser does not apply security restrictions (for example, in running scripts in that file) that it would normally apply to files that came from elsewhere.
- Files of type ".ISP", which are intended to contain scripts used to set up an account with an Information Service Provider. Since the script interpreter was an undocumented Microsoft-provided application, deciding that a script cannot cause any harm was not particularly easy. Searching the binary representation of the program for character strings revealed a list of script keywords, one of which was "RUN". A little experimenting revealed that the application that interprets this keyword invokes the operating system to run whatever command line follows the RUN key word.

The first two of these file types are relatively hard to exploit because they operate by running a program already stored somewhere on the client's computer. A prospective attacker would have to either guess the location of an existing, exploitable application program or surreptitiously install a file in a known location. Both of these courses are, however, easier than they sound. Most system installations follow a standard pattern, which means that vendor-supplied command programs are stored in standard places with standard names, and many of those command programs can be exploited by passing them appropriate arguments. By judicious use of comments and other syntactic tricks one can create a file that can be interpreted either as a harmless HTML Web page or as a command script. If the client reads such an HTML Web page, the browser places a copy in its Web cache, where it can then be exploited as a command script, using either the .LNK or .URL type.

Lesson 5.12.8:

The fact that these security problems were not discovered before product release suggests that competitive pressures can easily dominate concern for security. One would expect that even a somewhat superficial security inspection would have quickly revealed each of these problems. Failure to adhere to the principle of open design is also probably implicated in this incident. Finally, the principle of least privilege suggests that automatically launched programs that could be under control of an adversary should be run in a distinct virtual machine, the computer equivalent of a padded cell, where they can't do much damage.*

* Chris Rioux provided details on this collection of browser problems, and discovered the .ISP exploitation, in 1998.

5.12.9: The Reused Password

A large corporation arranged to obtain network-accessible computing services from two competing outside suppliers. Employees of the corporation had individual accounts with each supplier.

Supplier A was quite careful about security. Among other things, it did not permit users to choose their own passwords. Instead, it assigned a randomly-chosen password to each new user. Supplier B was much more relaxed—users could choose their own

passwords for that system. The corporation that had contracted for the two services recognized the difference in security standards and instructed its employees not to store any company confidential or proprietary information on supplier B's more loosely managed system.

In keeping with their more relaxed approach to security, a system programmer for supplier B had the privilege of reading the file of passwords of users of that system. Knowing that this customer's staff also used services of supplier A, he guessed that some of them were probably lazy and had chosen as their password on system B the same password that they had been assigned by supplier A. He proceeded to log in to system A successfully, where he found a proprietary program of some interest and copied it back to his own system. He was discovered when he tried to sell a modified version of the program, and employees of the large corporation became suspicious.*

Lesson 5.12.9:

People aren't good at keeping secrets.

* This anecdote was reported in the 1970s, but its source has been lost.

5.12.10: Signaling with Clandestine Channels

5.12.10.1: Intentionally I: Banging on the Walls

Once information has been released to a program, it is difficult to be sure that the program does not pass the information along to someone else. Even though nondiscretionary controls may be in place, a program written by an adversary may still be able to signal to a conspirator outside the controlled region by using a clandestine channel. In an experiment with a virtual memory system that provides shared library procedures, an otherwise confined program used the following signaling technique: For the first bit of the message to be transmitted, it touched (if the bit value was `ONE`) or failed to touch (if the bit value was `ZERO`) a previously agreed-upon page of a large, infrequently used, shared library program. It then waited a while, and repeated the procedure for the second bit of the message. A receiving thread observed the presence of the agreed-upon page in memory by measuring the time required to read from a location in that page. A short (microsecond) time meant that the page was already in memory and a `ONE` value was recorded for that bit. Using an array of pages to send multiple bits, interspersed with pauses long enough to allow the kernel to page out the entire array, a data rate of about one bit per second was attained.* This technique of transmitting data by an otherwise confined program is known as "banging on the walls."

In 2005, Colin Percival noticed that when two processors share a cache, as do certain chips that contain multiple processors, this same technique can be used to transmit information at much higher rate. Percival estimates that the L1 cache of a 2.8 gigahertz Pentium 4 could be used to transmit data upwards of 400 kilobytes per second.**

Lesson 5.12.10.1:

Minimize common mechanisms. A common mechanism such as a shared virtual memory or a shared cache can provide an unintended communication path.

* Demonstrated by Robert E. Mullen ca. 1976, described by Tom Van Vleck in a poster session at the *IEEE Symposium on Research in Security and Privacy*, Oakland, California, May 1990. The description is posted on the Multics Web site, at http://www.multicians.org/thvv/timing-chn.html.

** C. Percival, Cache missing for fun and profit. *Proceedings of BSDCAN 2005*, Ottawa. http://www.deamonology.net/papers/htt.pdf (May 2005).

5.12.10.2: Intentionally II

In an interesting 1998 paper,* Marcus Kuhn and Ross Anderson describe how easy it is to write programs that surreptitiously transmit data to a cheap nearby radio receiver by careful choice of the patterns of pixels appearing on the computer's display screen. A display screen radiates energy in the form of radio waves whose shape depends on the particular pattern on the screen. They also discuss how to design fonts to minimize the ability for an adversary to interpret this unwanted radiation.

Lesson 5.12.10.2

Paranoid design requires considering *all* access paths.

* Markus G. Kuhn and Ross J. Anderson. Soft Tempest: Hidden Data Transmission Using Electro magnetic Emanations. In David Aucsmith (Ed.): *Information Hiding 1998, Lecture Notes in Computer Science 1525*, pages 124–142 (1998: Springer-Verlag: Berlin and Heidelberg).

5.12.10.3: Unintentionally

If an operating system is trying to avoid releasing a piece of information, it may still be possible to infer its value from externally observed behavior, such as the time it takes for the kernel to execute a system call or the pattern of pages in virtual memory after the kernel returns. An example of this attack was discovered in the Tenex time-sharing system, which provided virtual memory. Tenex allowed a program to acquire the privileges of another user if the program could supply that user's secret password. The kernel routine that examined the user-supplied password did so by comparing it, one character at a time, with the corresponding entry in the password table. As soon as a mismatch was detected, the password-checking routine terminated and returned, reporting a mismatch error.

This immediate termination turned out to be easily detectable by using two features of Tenex. The first feature was that the system reacted to an attempt to touch a nonexistent page by helpfully creating an empty page. The second feature was that the user can ask the kernel if a given page exists. In addition, the user-supplied password can be placed anywhere in user memory.

An attacker can place the first character of a password guess in the last byte of the last existing page, and then call the kernel asking for another user's privileges. When the kernel reports a password mismatch error, the attacker then can check to see whether or not the next page now exists. If so, the attacker concludes that the kernel touched the next page to look for the next byte of the password, which in turn implies that the first character of the password was guessed correctly. By cycling through the letters of the alphabet, watching for one that causes the system to create the next page, the attacker could systematically search for the first character of the password. Then, the attacker could move the password down in memory one character position and start a similar search for the second character. Continuing in this fashion, the entire password could be quickly exposed with an effort proportional to the length of the password rather than to the number of possible passwords.*

Lesson 5.12.10.3:

We have here another example of a common mechanism: the virtual memory shared between the user and the password checker inside the supervisor. Common mechanisms can provide unintended communication paths.

* This attack (apparently never actually exploited in the field before it was blocked) has been confirmed by Ray Tomlinson and Dan Murphy, the designers of Tenex. A slightly different description of the attack appears in Butler Lampson, "Hints for computer system design," *Operating Systems Review 17*, 5 (October 1983), pages 35–36.

5.12.11: It Seems to be Working Just Fine

A hazard with systems that are supposed to provide security is that there often is no obvious indication that they aren't actually doing their job. This hazard is especially acute in cryptographic systems.

5.12.11.1: I Thought It Was Secure

The Data Encryption Standard (DES) is a block cryptographic system that transforms each 64-bit plaintext input block into a 64-bit output ciphertext block under what appears to be a 64-bit key. Actually, the eighth bit of each key byte is a parity check on the other seven bits, so there are only 56 distinct key bits.

One of the many software implementations of DES works as follows. One first loads a key, say `my_key` , by invoking the entry

```
status ← LOAD_KEY (my_key)
```

The `LOAD_KEY` procedure first resets all the temporary variables of the cryptographic software, to prevent any interaction between successive uses. Then, it checks its argument value to verify that the parity bits of the key to be loaded are correct. If the parity does not check, `LOAD_KEY` returns a non-zero status. If the status argument indicates that the key loaded properly, the

application program can go on to perform other operations. For example, a cryptographic transformation can be performed by invoking

```
ciphertext ← ENCRYPT (plaintext)
```

for each 64-bit block to be transformed. To apply the inverse transformation, the application invokes `LOAD_KEY` with the same key value that was used for encryption and then executes

```
plaintext ← DECRYPT (ciphertext)
```

A network application used this DES implementation to encrypt messages. The client and the service agreed in advance on a key (the "permanent key"). To avoid exposing the permanent key by overuse, the first step in each session of the client/service protocol was for the client to randomly choose a temporary key to be used in this session, encipher it with the permanent key, and send the result to the service. The service decrypted the first block using the permanent key to obtain the temporary session key, and then both ends used the session key to encrypt and decrypt the streams of data exchanged for rest of that session.

The same programmer implemented the key exchange and loading program for both the client and the service. Not realizing that the DES key was structured as 56 bits of key with 8 parity bits, he wrote the program to simply use a random number generator to produce a 64-bit session key. In addition, not understanding the full implications of the status code returned by `LOAD_KEY` , he wrote the call to that program as follows (in the C language):

```
LOAD_KEY (tempkey)
```

thereby ignoring the returned status value. Everything seemed to work properly. The client generated a random session key, enciphered it, and sent it to the service. The service deciphered it, and then both the client and the service loaded the session key. But in 255 times out of 256, the parity bits of the session key did not check, and the cryptographic software did not load the key. With this particular implementation, failing to load a key after state initialization caused the program to perform the identity transformation. Consequently, in most sessions all the data of the session was actually transmitted across the network in the clear.*

Lesson 5.12.11.1:

The programmer who ignored the returned status value was not sufficiently paranoid in the implementation. The designer of `LOAD_KEY` , in implementing an encryption engine that performs the identity transformation when it is in the reset state did not apply the principle of *fail-safe defaults*, also did not apply the principle *be explicit*; the documentation of the package could have included a warning printed in large type of the importance of checking the returned status values.

* Reported by Theodore T'so in 1997.

5.12.11.2: How Large is the Key Space…Really?

When a client presents a Kerberos ticket to a service (see Sidebar 5.6.1 for a brief description of the Kerberos authentication system), the service obtains a relatively reliable certification that the client is who it claims to be. Kerberos includes in the ticket a newly-minted session key known only to it, the service, and the client. This new key is for use in continued interactions between this service and client, for example to encrypt the communication channel or to authenticate later messages.

Generating an unpredictable session key involves choosing a number at random from the 56-bit Data Encryption Standard key space. Since computers aren't good at doing things at random, generating a genuinely unpredictable key is quite difficult. This problem has been the downfall of many cryptographic systems. Recognizing the difficulty, the designers of Kerberos in 1986 chose to defer the design of a high-quality key generator until after they had worked out the design of the rest of the authentication system. As a placeholder, they implemented a temporary key generator which simply used the time of day as the initial seed for a pseudorandom-number generator. Since the time of day was measured in units of microseconds, using it as a starting point introduced enough unpredictability in the resulting key for testing.

When the public release of Kerberos was scheduled three years later, the project to design a good key generator bubbled to the top of the project list. A fairly good, hard-to-predict key generator was designed, implemented, and installed in the library. But, because Kerberos was already in trial use and the new key generator was not yet field-tested, modification of Kerberos to use the new key generator was deferred until experience with it and confidence in it could be accumulated.

In February of 1996, some 7 years later, two graduate students at Purdue University learned of a security problem attributed to a predictable key generator in a different network authentication system. They decided to see if they could attack the key generator in

Kerberos. When they examined the code they discovered that the temporary, time-of-day key generator had never been replaced, and that it was possible to exhaustively search its rather limited key space with a contemporary computer in just a few seconds. Upon hearing this report, the maintainers of Kerberos were able to resecure Kerberos quickly because the more sophisticated key-generator program was already in its library and only the key distribution center had to be modified to use the library program.*

Lesson 5.12.11.2:

This incident illustrates how difficult it is to verify proper operation of a function with negative specifications. From all appearances, the system with the predictable key generator was operating properly.

* Jared Sandberg, with contribution by Don Clark. Major flaw in Internet security system is discovered by two Purdue students. *Wall Street Journal CCXXVII*, 35 (Tuesday, 20 February 1996), Eastern Edition page B–7A.

5.12.11.3: How Long are the Keys?

A World Wide Web service can be configured, using the Secure Socket Layer, to apply either weak (40-bit key) or strong (128-bit key) cryptographic transformations in authenticating and encrypting communication with its clients. The Wells Fargo Bank sent the following letter to online customers in October, 1999:

"We have, from our initial introduction of Internet access to retirement account information nearly two years ago, recognized the value of requiring users to utilize browsers that support the strong, 128-bit encryption available in the United States and Canada. Following recent testing of an upgrade to our Internet service, we discovered that the site had been put into general use allowing access with standard 40-bit encryption. We fixed the problem as soon as it was discovered, and now, access is again only available using 128-bit encryption…We have carefully checked our Internet service and computer files and determined that at no time was the site accessed without proper authorization…"*

Some Web browsers display an indication, such as a padlock icon, that encryption is in use, but they give no clue about the size of the keys actually being used. As a result, a mistake such as this one will likely go unnoticed.

Lesson:

The same as for the preceding anecdote 5.12.11.2

* Jeremy Epstein. *Risks-Forum Digest 20*, 64 (Thursday, 4 November 1999).

5.12.12: Injection For Fun and Profit

A common way of attacking a system that is not well defended is to place control information in a typed input field, a method known as "injection". The programmer of the system provides an empty space, for example on a Web form, in which the user is supposed to type something such as a user name or an email address. The adversary types in that space a string of characters that, in addition to providing the requested information, invokes some control feature. The typical mistake is that the program that reads the input field simply passes the typed string along to some potentially powerful interpreter without first checking the string to make sure that it doesn't contain escape characters, control characters, or even entire program fragments. The interpreter may be anything from a human operator to a database management system, and the result can be that the adversary gains unauthorized control of some aspect of the system.

The countermeasure for injection is known as "sanitizing the input". In principle, sanitizing is simple: scan all input strings and delete inappropriate syntactical structures before passing them along. In practice, it it is sometimes quite challenging to distinguish acceptable strings from dangerous ones.

5.12.12.1: Injecting a Bogus Alert Message to the Operator

Some early time-sharing systems had a feature that allowed a logged-in user to send a message to the system operator: for example, to ask for a tape to be mounted. This message is displayed at the operator's terminal, intermixed with other messages from the operating system. The operating system normally displays a warning banner ahead of each user message so that the operator knows its source. In the Compatible Time Sharing System at M.I.T., the operating system placed no constraint on either the length or content of messages from users. A user could therefore send a single message that, first, cleared the display screen to eliminate the

warning banner, and then displayed what looked like a standard system alert message, such as a warning that the system was overheating, which would lead the operator to immediately shut down the system.*

* This vulnerability was noticed, and corrected, by staff programmers in the late 1960s. As far as is known, it was never actually exploited.

5.12.12.2: CardSystems Exposes 40,000,000 Credit Card Records to SQL Injection

A currently popular injection attack is known as "SQL injection". Structured Query Language (SQL) is a widely-implemented language for making queries of a database system. A typical use is that a Web form asks for a user name, and the program that receives the form inserts the typed string in place of `typedname` in an SQL statement such as this one:

```
select * from USERS where NAME = = 'typedname';
```

This SQL statement finds the record in the `USERS` table that has a `NAME` field equal to the value of the string that replaced `typedname` . Thus, if the user types "John Doe" in the space on the Web form, the SQL statement will look for and return the record for user John Doe.

Now, suppose that an adversary types the following string in the blank provided for the name field:

```
John Doe' ; drop USERS;
```

When that string replaces `typedname` , the result is to pass this input to the SQL interpreter:

```
select * from USERS where NAME = 'John Doe' ; drop USERS;';
```

The SQL interpreter considers that input to be three statements, separated by semicolons. The first statement returns the record corresponding to the name "John Doe". The second statement deletes the `USERS` table. The third statement consists of a single quote, which the interpreter probably treats as a syntax error, but the damage intended by the adversary has been done. The same scheme can be used to inject much more elaborate SQL code, as in the following incident, described by excerpts from published accounts.

Excerpt from `wired.com` , June 22, 2005: "MasterCard International announced last Friday that intruders had accessed the data from CardSystems Solutions, a payment processing company based in Arizona, after placing a malicious script on the company's network."[1] *The New York Times* reported that "…more than 40 million credit card accounts were exposed; data from about 200,000 accounts from MasterCard, Visa and other card issuers are known to have been stolen…"[2]

Excerpt from the testimony of the Chief Executive Officer of CardSystems Solutions before a Congressional committee: "An unauthorized script extracted data from 239,000 unique account numbers and exported it by FTP…"[3]

Excerpt from the FTC complaint, filed a year later: "6. Respondent has engaged in a number of practices that, taken together, failed to provide reasonable and appropriate security for personal information stored on its computer network. Among other things, respondent: (1) created unnecessary risks to the information by storing it in a vulnerable format for up to 30 days; (2) did not adequately assess the vulnerability of its Web application and computer network to commonly known or reasonably foreseeable attacks, including but not limited to "Structured Query Language" (or "SQL") injection attacks; (3) did not implement simple, low-cost, and readily available defenses to such attacks; (4) failed to use strong passwords to prevent a hacker from gaining control over computers on its computer network and access to personal information stored on the network; (5) did not use readily available security measures to limit access between computers on its network and between such computers and the Internet; and (6) failed to employ sufficient measures to detect unauthorized access to personal information or to conduct security investigations.

"7. In September 2004, a hacker exploited the failures set forth in Paragraph 6 by using an SQL injection attack on respondent's Web application and Web site to install common hacking programs on computers on respondent's computer network. The programs were set up to collect and transmit magnetic stripe data stored on the network to computers located outside the network every four days, beginning in November 2004. As a result, the hacker obtained unauthorized access to magnetic stripe data for tens of millions of credit and debit cards.

"8. In early 2005, issuing banks began discovering several million dollars in fraudulent credit and debit card purchases that had been made with counterfeit cards. The counterfeit cards contained complete and accurate magnetic stripe data, including the security code used to verify that a card is genuine, and thus appeared genuine in the authorization process. The magnetic stripe data matched the information respondent had stored on its computer network. In response, issuing banks cancelled and re-issued

thousands of credit and debit cards. Consumers holding these cards were unable to use them to access their credit and bank accounts until they received replacement cards."[4]

Visa and American Express cancelled their contracts with CardSystems, and the company is no longer in business.

Lesson 5.12.12.2

Injection attacks, and the countermeasure of sanitizing the input, have been recognized and understood for at least 40 years, yet another example is reported nearly every day. From earlier on this page, Lesson 5.12.1.7seems to apply here also.

[1] Kim Zetter. "CardSystems' Data Left Unsecured," *Wired.com* (22 June 2005), https://www.wired.com/2005/06/cardsystems-data-left-unsecured/.

[2] *The New York Times*, Tuesday, June 21, 2005.

[3] Statement of John M. Perry, President and CEO CardSystems Solutions, Inc., before the United States House of Representatives Subcommittee on Oversight and Investigations of the Committee on Financial Services, July 21, 2005.

[4] United States Federal Trade Commission Complaint, Case 0523148, Docket C-4168, September 5, 2006.

5.12.13: Hazards of Rarely-Used Components

In the General Electric 645 processor, the circuitry to check read and write permission was invoked as early in the instruction cycle as possible. When the instruction turned out to be a request to execute an instruction in another location, the execution of the second instruction was carried out with timing later in the cycle. Consequently, instead of the standard circuitry to check read and write permission, a special-case version of the circuit was used. Although originally designed correctly, a later field change to the processor accidentally disabled one part of the special-case protection-checking circuitry. Since instructions to execute other instructions are rarely encountered, the accidental disablement was not discovered until a penetration team began a systematic study and found the problem. The disablement was dependent on the address of both the executed instruction and its operand, and was therefore unlikely to have ever been noticed by anyone not intentionally looking for security holes.*

Lesson 5.12.13:

Most reliability design principles also apply to security: *avoid rarely-used components*.

* Paul A. Karger and Roger R. Schell. *Multics Security Evaluation: Vulnerability Analysis*. June 1974.

5.12.14: A Thorough System Penetration Job

One particularly thorough system penetration operation went as follows. First, the team of attackers legitimately obtained computer time at a different site that ran the same hardware and same operating system. On that system they performed several experiments, eventually finding an obscure error in protecting a kernel routine. The error, which permitted general changing of any kernel-accessible variable, could be used to modify the current thread's principal identifier. After perfecting the technique, the team of attackers shifted their activities to the site where the operating system was being used for development of the operating system itself. They used the privilege of the new principal identifier to modify one source program of the operating system. The change was a one-byte revision—replacing a "less than" test with a "greater than" test, thereby compromising a critical kernel security check. Having installed this change in the program, they covered their trail by changing the directory record of date-last-modified on that file, thereby leaving behind no traces except for one changed line of code in the source files of the operating system. The next version of the system to be distributed to customers contained the attacker's revision, which could then be exploited at the real target site.

This exploit was carried out by a tiger team that was engaged to discover security slipups. To avoid compromising the security of innocent customer sites, after verifying that the change did allow compromise, the tiger team further modified the change to one that was not exploitable, but was detectable by someone who knew where to look. They then waited until the next system release. As expected, the change did appear in that release.*

Lesson 5.12.14:

Complete mediation includes verifying the authenticity, integrity, and authorization of the software development process, too.

* Paul A. Karger and Roger R. Schell. *Multics Security Evaluation: Vulnerability Analysis*. June 1974.

5.12.15: Framing Enigma

Enigma is a family of encipherment machines designed in Poland and Germany in the 1920s and 1930s. An Enigma machine consists of a series of rotors, each with contacts on both sides, as in Figure 5.12.1. One can imagine a light bulb attached to each contact on one side of the rotor. If one touches a battery to a contact on the other side, one of the light bulbs will turn on, but which one depends on the internal wiring of that rotor. An Enigma rotor had 26 contacts on each side, thus providing a permutation of 26 letters, and the operator had a basket of up to eight such rotors, each wired to produce a different permutation.

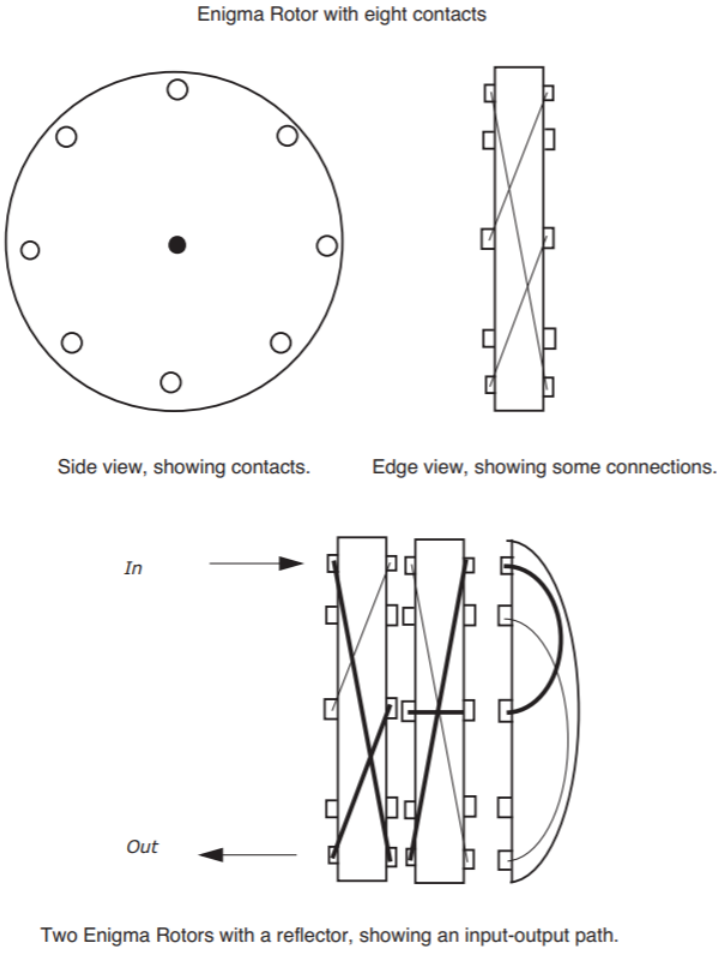


Figure 5.12.1: Enigma design concept (simplified for illustration).

The first step in enciphering was to choose four rotors from the basket [j, k, l and m] and place them on an axle in that order. This choice was the first component of the encoding key. The next step was to set each rotor into one of 26 initial rotational positions [a, b, c, d], which constituted the second component of the encoding key. The third step was to choose one of 26 offsets [e, f, g, h] for a tab on the edge of each rotor. The offsets were the final component of the encoding key. The Enigma key space was, in terms of the computational abilities available during World War II, fairly formidable against brute-force attack. After transforming one stream element of the message, the first rotor would turn clockwise one position, producing a different transformation for the next stream element. Each time the offset tab of the first rotor completed one revolution, it would strike a pawl on the second rotor,

causing the second rotor to rotate clockwise by one position, and so on. The four rotors taken together act as a continually changing substitution cipher in which any letter may transform into any letter, including itself.

The chink in the armor came about with an apparently helpful change, in which a reflecting rotor was added at one end—in the hope of increasing the difficulty of cryptanalysis. With this change, input to and output from the substitution were both done at the same end of the rotors. This change created a restriction: since the reflector had to connect some incoming character position into some *other* outgoing character position, no character could ever transform into itself. Thus the letter "E" never encodes into the letter "E".

This chink could be exploited as follows. Suppose that the cryptanalyst knew that every enciphered message began with the plaintext string of characters "The German High Command sends greetings to its field operations". Further, suppose that one has intercepted a long string of enciphered material, not knowing where messages begin and end. If one placed the known string (of length 60 characters) adjacent to a randomly selected adjacent set of 60 characters of intercepted ciphertext, there will probably be some positions where the ciphertext character is the same as the known string character. If so, the reflecting Enigma restriction guaranteed that this place could not be where that particular known plaintext was encoded. Thus, the cryptanalyst could simply slide the known plaintext along the ciphertext until he or she came to a place where no character matches and be reasonably certain that this ciphertext does correspond to the plaintext. (For a known or chosen plaintext string of 60 characters, there is a 9/10 probability that this framing is not a chance occurrence. For 120 characters, the probability rises to 99/100.)*

Lesson 5.12.15:

Being able systematically to frame most messages is a significant step toward breaking a code because it greatly reduces the number of trials required to discover the key.

* A thorough explanation of the mechanism of Enigma appeared in Alan M. Turing, "A description of the machine," (Chapter 1 of an undated typescript, sometimes identified as the *Treatise on Enigma* or "the prof's book", c. 1942) [United States National Archives and Records Administration, record group 457, National Security Agency Historical Collection, box 204, Nr. 964, as reported by Frode Weierude]. A nontechnical account of the flaws in Enigma and the ways they could be exploited can be found in Stephen Budianski, *Battle of Wits* [New York: Simon & Schuster: 2000].

This page titled 5.12: War Stories - Security System Breaches is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

5.13: Exercises

Exercise 5.13.1

1983-3-5b

Louis Reasoner has been using a simple RPC protocol that works as follows*:

```
client ⇒ service: {nonce, procedure, arguments}
service ⇒ client: {nonce, response}
```

The client sets a timer, and if it does not receive a response before the timer expires, it restarts the protocol from the beginning, repeating this sequence as many times as necessary until a response returns. The service maintains a table of nonces and responses, and when it receives a request containing a duplicate nonce it repeats the response, rather than repeating execution of the procedure. The client similarly maintains a list of nonces for which no response has yet been received, and it discards any responses for nonces not in that list, assuming that they are duplicates. One possible response is "unknown procedure", meaning that the service received a request it didn't know how to handle. The link layer checksums all frames and discards any that are damaged in transmission. All messages fit in one frame.

Louis wants to make this protocol secure against eavesdroppers. He has discovered that the client and the service already share a key, K_{cs} , for a shared-secret-key cryptographic system. So the first thing he tries is to encrypt the requests and responses of the simple RPC protocol:

```
client ⇒ service: ENCRYPT ({nonce, procedure, arguments}, Kcs)
service ⇒ client: ENCRYPT ({nonce, response}, Kcs)
```

This seems to work, but Louis has heard that if you use the same key to repeatedly transform predictable fields such as procedure names, someone may eventually discover the key by cryptanalysis. So he wants to use a different key for each RPC call. To minimize the coding effort, he changes the protocol to work as follows:

```
client ⇒ service: ENCRYPT ({Ktn}, Kcs)
client ⇒ service: ENCRYPT ({nonce, procedure, arguments}, Ktn)
service ⇒ client: ENCRYPT ({nonce, response}, Ktn)
```

in which K_{tn} is a one-time key chosen by the client to be used only for the *n*'th RPC call. When the service receives a key, it decrypts it and uses it until the service gets another key message. Louis figures that since K_{cs} is now being used only to encrypt temporary keys, which look like random numbers, it should be safer from cryptanalysis.

At first, this protocol, too, seems to work. Then Louis notices that the client is receiving the response “unknown procedure” much more often than it used to. Explain why, using a timing diagram to demonstrate an example of the failure, and offer a suggestion to fix the problem.

* Throughout the exercises, the notation `{a, b, c}` denotes a message constructed of the named items, marshaled in some unspecified way that is unimportant for the purposes of the problem so long as the recipient knows how to unmarshal the individual arguments.

Exercise 5.13.2

1994–2–1a...c

Lucifer is determined to figure out Alice's password by a brute-force attack. From watching her log in he knows that her password is eight characters long and all lower-case letters, of which there are 26. He sets out to try all possible combinations of eight lower-case letters.

a) Assuming that he has to try about half the possibilities before he runs across the right one, one trial can be done in one machine cycle, and he has a 600 mHz computer available, about how long will the project take?

b) How long will it take if Alice chooses an eight-character password that includes upper- and lower-case letters, numbers, and 16 special characters, creating a total of 78 possible characters?

c) Suppose processors continue to get faster, improving by a factor of three every two years. How long will it be until Alice's new password can be cracked as easily as her old one?

Exercise 5.13.3

1983–2–4b

Tracy Swallow has a bright idea for avoiding the need to store passwords securely. She suggests transforming the user's name with a key-driven cryptographic transformation using a systemwide "password key" and giving the result back to the user to present as a password. A user who wishes to log in simply presents his or her name and this password; the system can authenticate the user by again transforming the user's name with the password key to see if the result is the same as the presented password. Thus no central file of passwords is needed. What is wrong with Tracy's idea?

Exercise 5.13.4

1994–2–2a,b

Louis Reasoner is fascinated with the discovery that some cryptographic transformations are commutative. A commutative transformation has the interesting property that for every message and every pair of keys k_1 and k_2 ,

$$\texttt{TRANSFORM (TRANSFORM (}M\texttt{, }k_1\texttt{), }k_2\texttt{) = TRANSFORM (TRANSFORM (}M\texttt{, }k_2\texttt{), }k_1\texttt{)}$$

That is, you get the same result no matter in which order you do two transformations with different keys.

Louis did some further research, identified a high-quality commutative transformation, and used it to devise a commutative implementation of two confidentiality primitives he calls ENCRYPT_C and DECRYPT_C . He has proposed that Alice, in San Francisco, and Bob, in Boston, use the following scheme for secure private delivery of messages between their computers, which are connected via the Internet:

- Alice chooses a random key, k_a , encrypts her message M with that key, and sends the result, $\texttt{ENCRYPT_C (}M\texttt{, }k_a\texttt{)}$, to Bob.
- Bob chooses another random key, k_b , encrypts the already-encrypted message to produce $\texttt{ENCRYPT_C (ENCRYPT_C (}M\texttt{, }k_a\texttt{), }k_b\texttt{)}$, and sends the doubly-encrypted result back to Alice.
- By commutativity, this message is identical to $\texttt{ENCRYPT_C (ENCRYPT_C (}M\texttt{, }k_b\texttt{), }k_a\texttt{)}$, which is a message that Alice can decrypt with her key k_a . She does so, revealing $\texttt{ENCRYPT_C (}M\texttt{, }k_b\texttt{)}$.
- She sends this result back to Bob, who can now decrypt it with his key k_b to reveal M .

The appealing thing about this scheme is that Alice and Bob did not have to agree on a secret key in advance. Louis calls this the "No-Prior-Agreement" protocol.

a) Is it possible for a passive intruder (that is, one who just listens to the encrypted messages) to discover M ? If so, describe how. If not, explain why not.

b) Is it possible for an active intruder (that is, one who can also insert, delete, or replay messages) to discover M ? If so, describe how. If not, explain why not.

Exercise 5.13.5

1995–2–3a...d

Secure Inc. is developing a remote file system, Secure RFS (SRFS), which automatically encrypts files to guarantee better privacy of information. When a request to store a file arrives, SRFS encrypts the file using the client's key. On arrival of a request to read a file, SRFS looks up the client key, decrypts the file, and sends the file back to the client. SRFS keeps for each client a separate key.

a) The designers of Secure Inc. are wondering how long it would take to crack a file that is encrypted using RSA with a 512-bit key. To crack an RSA-encrypted file, one has to factor the key. The designers found a 1993 paper that reports that factoring a 100 decimal digit number takes about 1 month using idle cycles from 300 3 MIPS workstations. It is estimated that factoring an additional 3 decimal digits roughly doubles the computation time needed. How many 3-MIPS computers would be needed to factor a 155 decimal digit number (which corresponds to about 512 bits) in one month?

b) If processors are doubling computation performance per year, how many workstations would it take to factor a 512-bit key in one month in the year 2001?

c) Assume that the cryptographic transformations can be done at 250 kilobytes per second. How much would the throughput be reduced for reading files stored by SRFS, if the current maximum throughput without cryptographic transformations is 800 kilobytes per second? (Assume that the cryptographic transformations cannot be pipelined with sending and receiving.)

d) Secure Inc. is also considering adding automatic compression of files to SRFS. Compression reduces redundancy of information in a file so that the file takes less disk space. Should they first compress files, then encrypt them, or should they first encrypt files and then compress them? Explain.

Exercise 5.13.6

2008-3-12-13

Alice wants to communicate with Bob over an insecure network. She learned about one-time pads in Section 5.9, and decides to use a one-time pad to secure her communications. Since Alice wants to send a k-bit message to Bob in the future, she generates a random k-bit key r and hands it to Bob in person. When Alice comes to send Bob her message, she XOR 's the message m with the key r to produce a ciphertext c , and sends this on the network. Bob XOR 's c with r to retrieve m .

a) Assume that Alice's message m is a concatenation of a header followed by some data. Consider an eavesdropper Eve who snoops on Alice's conversation. If Eve can correctly guess the value of the header in Alice's message, which of the following are correct?

A. Eve's ability to decrypt the data bits in m is not improved by her knowledge of the header bits.

B. The data bits in Alice's message are confidential.

C. The data bits in Alice's message are securely authenticated.

Alice rapidly grows tired of the effort in exchanging one-time pads with Bob, and has an idea to simplify the key distribution process. Alice's idea works as follows:

To send a k-bit message m_1 to Bob, Alice picks a k-bit random number r_1 , computes ciphertext $c_1 = m_1 \oplus r_1$, and sends c_1 to Bob. Bob then picks his own k-bit random number r_2 , computes $c_2 = c_1 \oplus r_2$, and sends c_2 to Alice. Alice finally computes $c_3 = c_2 \oplus r_1$ and sends c_3 to Bob.

b) Which of the following statements are correct of Alice's new scheme?

A. Bob can correctly decrypt Alice's message m_1 , without receiving r_1 ahead of time, assuming all messages between Alice and Bob are correctly delivered.

B. An active attacker Lucifer (who can intercept, drop, and replay messages) can decrypt the message.

C. A passive eavesdropper Eve can decrypt the message.

Exercise 5.13.7

1995–2–4a

Bank of America is struggling to convince itself of the authenticity of a message it just received, and has asked your help in what to do next. So far, they know the following two facts to be true:

- Louis **says** (Ben **says** (Transfer $1,000,000 to Alyssa))
- Jim **speaks for** Ben

Ben's account has enough money for such a transaction, so if they can convince themselves that Ben really authorized the transaction, they will do the transfer. Which of the following things should they attempt to establish the truth of, and why?

A. Louis **speaks for** Jim

B. Ben **speaks for** Louis

C. Ben **says** (Jim **speaks for** Louis)

Exercise 5.13.8

1984–2–4

Ben Bitdiddle has hit on a bright idea for fixing the problem that capabilities are hard to revoke. His plan is to invent something called **timed capabilities**. One of the fields of a timed capability is its expiration time, which is the time of creation plus E. A timed capability can be used like any other capability until the system clock reaches the expiration time; after that time, it becomes worthless. Analyze this proposal with respect to:

a) Performance.

b) Propogation.

c) Revocation.

d) Auditing.

e) Ease of use.

Exercise 5.13.9

1984–2–3a,b

Two banks are developing an inter-bank funds transfer system. They are connected by a telephone line which runs in a duct along Main Street, and Alyssa P. Hacker is concerned that there might be foul play. The banks' expert, Ben Bitdiddle, says that the banks will use a shared-secret key $K1$ to encrypt their communications and a second shared-secret key $K2$ to authenticate their communications, using the following protocol:

$$\text{Bank 1} \Rightarrow \text{Bank 2}\{\{\text{“transfer from our Account Y”}\}_{K2}\}^{K1}$$
$$\text{Bank 1} \Rightarrow \text{Bank 2}\{\{\text{“to your Account X”}\}_{K2}\}^{K1}$$
$$\text{Bank 1} \Rightarrow \text{Bank 2}\{\{\text{“Amount Z”}\}_{K2}\}^{K1}$$
$$\text{Bank 2} \Rightarrow \text{Bank 1}\{\{\text{“OK”}\}_{K2}\}^{K1}$$

Alyssa immediately realizes that without knowing either $K1$ or $K2$ an intruder could subvert the banks.

a) With an Apple II in the manhole in middle of Main Street, describe how Alyssa could

- Increase or decrease the amount of a transfer.
- Cause a transfer to occur more than once.
- Cause a transfer not to occur at all without arousing suspicion at the requesting bank.

b) Design a new protocol that eliminates these problems and uses only two messages.

Exercise 5.13.10

2002–0–1,2

To attract attention to their Web site, OutofMoney.com has added a feature that broadcasts a stream of messages containing free stock market quotations. They intend the information to be public, so there is no need for confidentiality, but they are concerned about their reputation, so they want the stream of data to be authenticated.

Their current implementation signs every message with the company's private key, and clients authenticate the data by verifying it with the company's widely publicized public key. This technique works, but is proving problematic because the public-key algorithm uses too much computation time and the typical client, running a four-year-old pentium processor, can't keep up with the stream of messages on days when the stock market is crashing.

From reading this chapter, they learned that authentication using a shared-secretkey MAC is much faster. They have hired Ben Bitdiddle and Louis Reasoner as a consulting team to put this idea into practice. (Unfortunately, they didn't do any of the problem sets, so they don't know about the reputations of these two characters.) Louis's first proposal is as follows: any client who wishes to use the authenticated service starts by contacting the service and requesting a start message. The service signs

this start message with the company's public key. The start message contains the shared-secret key that is currently being used to authenticate the stream of messages containing the stock market quotations.

a) Ben's intuition is that this can't possibly work, but he isn't sure why. Give Ben some help by explaining why.

Undaunted, Louis has been reading about **delayed authentication** and decides it is the ideal way to tackle this problem. The idea is the following: since the service is sending a stream of messages, for each message use a *different* shared-secret key to create its authentication tag, and then publicly disclose that shared-secret key *after* all clients have received that message.

In Louis's design, each message P_i is constructed as follows:

$$raw_message_i \leftarrow \{i, D_i , K_{i-2}\}$$
$$authtag_i \leftarrow SIGN\ (raw_message_i , K_i)$$
$$P_i \leftarrow \{raw_message_i , authtag_i \}$$

Thus P_i contains

- its own sequence number, i
- some data, D_i
- the key K_{i-2} , which can be used to verify the data in message P_{i-2}
- an authentication tag created by signing the rest of the message with K_i

The key that authenticates this message will appear in message P_{i+2} . Louis argues that even though the key K_i is sent in plaintext, if the client receives D_i before the service sends K_i , by the time the attacker knows K_i , it is too late for the attacker to modify D_i . As with Louis's previous system, a client begins by requesting a start message. This time, the start message contains the same data as the next message in the broadcast stream, but it is signed with the company's private key.

b) Again, Ben is (rightly) suspicious of this system, but he can't figure out what is wrong with it. Help him out by explaining the flaw and how to fix it.

Exercise 5.13.11

2002–2–04

This chapter discusses both capabilities and access control lists as mechanisms for authorization. Which of the following statements are true?

A. A capability system associates a list of object references with each principal, indicating which objects the principal is allowed to use.

B. An access control list system associates a list of principals with each object, indicating which principals are allowed to use the object.

C. Revocation of a particular access permission of a principal is more difficult in an access control list system than in a capability system.

D. Protection in the UNIX file system is based on capabilities only.

Exercise 5.13.12

2003–3–5

Alice decided to try out a new RFID Student Tracking System, so she created an access control list that allows a few close friends to track her. One of those friends, Bob, wants to ask Alice to join his design project team, so this morning he requested that the tracking system give him a callback if Alice walks by the Administration building. Alice, working in a nearby laboratory, belatedly realizes that Bob is probably going to pop that question, so she logs in to the tracking system and removes Bob from her access control list. She then logs out and leaves for lunch. As she walks by the Administration building, Bob comes running out of the library to greet her, saying that he just received a callback from the tracking system.

The designer of the tracking system made a security blunder. Which of the following is the most likely explanation?

A. The tracking system didn’t properly erase residues.

B. In her rush to leave for lunch, Alice removed Lucy, rather than Bob, from her ACL.

C. The tracking system has a time-of-check to time-of-use bug.

D. The system used a version of SSL that is subject to cipher substitution attacks.

E. The system did not require a face-to-face rendezvous between users and system administrator.

Exercise 5.13.13

2007-3-3-4

Ben decides to start an Internet Service Provider. He buys an address space that contains 2^{24} addresses (out of the total of 2^{32} in the Internet) that have never been used before. A few days after he buys this address space, someone launches a new worm similar in design to the Slammer worm described in Section 5.12.4.3. The new worm targets a buffer overflow in the FOO server, which listens on UDP port 5044. Ben monitors all traffic sent to his part of the Internet address space on port 5044 and plots the number of worm probes versus time below:

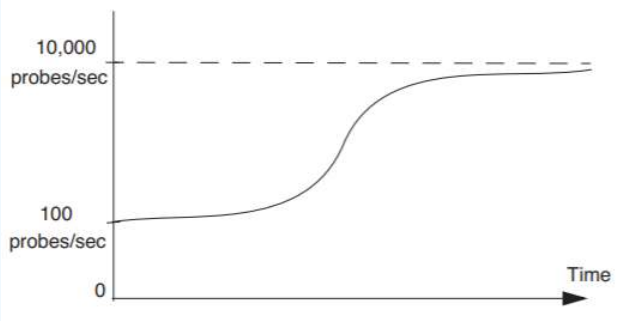


Figure 5.13.1: Graph of the frequency of worm probes in Ben's address space vs. time.

Assume the worm spreads by probing IP addresses chosen at random, and that its pseudorandom number generator is bug-free and generates a complete permutation of the integers before revisiting any integer. Ben learns from a security analyst that each infected machine sends 100 probes/second.

a) Give an estimate of the total number of machines that run the FOO server

A. 100 machines
B. 7.2×10^{18} machines
C. 25,600 machines
D. 8,000 machines

b) Ben thinks that the worm used a hit list of vulnerable addresses (i.e., addresses of FOO servers). Do you agree? If you do, what is the best estimate for the number of machines contained in the hit list?

A. no hit list
B. 100 machines
C. 256 machines
D. 25600 machines
E. 80 machines

Exercise 5.13.14

2008-3-6-7

Ben Bitdiddle, the new head of Cyber Security for the Department of Homeland Security, studied the war story about the Slammer worm in Section 5.12.4.3. He wants to build a system that will detect and stop future worm attacks before they can reach 50% of the vulnerable hosts. Ben makes the following assumptions about the worms to be defended against:

- Each worm instance sends 512 (2^9) probes per second.
- The worm's software probes all IP addresses at random.

- Of the 2^{32} possible addresses on the Internet, there are 32,768 (2^{15}) that are attached to active hosts that are vulnerable to the worm.
- The worm begins by infecting a single vulnerable host.

a) Given the assumptions above, roughly how many seconds will it take for the size of the infected population to double, during the early stages of a worm outbreak?

A. 16 seconds
B. 256 seconds
C. 1024 seconds

Ben convinces a consortium of router vendors to develop a new, remotelyconfigurable packet-filtering feature, and develops a system that can propagate filter updates to all routers in the Internet within 15 minutes (900 seconds) of a detected outbreak. Once all routers have the filter, the filters will prevent all further worm infections. Ben's detection mechanism is a network monitor that can observe 1/256-th of the Internet address space. His system automatically sends a filter update whenever worm traffic directed to the set of addresses he monitors reaches a predefined threshold.

b) At what traffic threshold should Ben choose to stop the worm before it reaches 50% of the vulnerable hosts?

A. 10 worm probes/second
B. 100 worm probes/second
C. 1000 worm probes/second
D. 10,000 worm probes/second
E. 100,000 worm probes/second

Exercise 5.13.15

Ben Bitdiddle visits the Web site `amazing.com` and obtains a fresh page signed with a private key. Which of these methods of obtaining the certificate for the server's public key can assure Ben that the private key used for the page's signature indeed belongs to the organization that owns the domain `amazing.com` ? (Assume that the certificate is signed by a trusted certificate authority and is valid.)

A. Using HTTP, Ben downloads the certificate from `http://amazing6033.com` .

B. Using HTTP, Ben downloads the certificate from the certificate authority.

C. Ben finds the certificate by doing a Web search on Google.

D. Ben gets the certificate via email from a spammer.

Exercise 5.13.16

1995–2–5a…c

Ben Bitdiddle and Louis Reasoner have founded a startup company, named Public Key Publication, Inc. (PKPI), whose business is distributing public keys. Their idea is that people who have a key pair for use with a public-key system need a way of letting other people know the public key of their key pair. Ben and Louis are not interested in creating keys, just in acting as a public key distributor.

Ben and Louis have designed the following protocol, in which Alice sends a private message to Bob. They need your help in debugging the protocol. K_{Pxyz} is the public key of principal `xyz` .

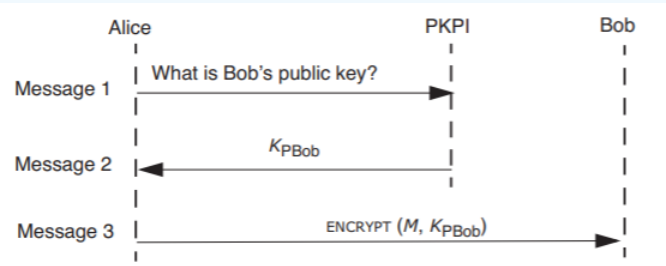


Figure 5.13.2 Messages 1 and 2 constitute the PKPI protocol. Message 3 is the beginning of Alice's protocol with Bob and is not under the control of PKPI; it is shown here only to place the PKPI protocol in context.

a) Louis believes that Eve, the passive eavesdropper, will find that she cannot learn anything by overhearing the PKPI protocol in use. Give an argument that supports Louis's position, or an example demonstrating that Louis is mistaken.

b) Louis originally hoped that Lucifer, the active attacker, wouldn't be able to cause any problems, either, but since reading this chapter he is not sure. Give an example of an active attack that demonstrates that Louis needs to revise the PKPI protocol to protect against Lucifer.

c) Ben suggests that the protocol could be improved by changing Message 2. What changes should be made so that Alice can be confident that no one but Bob can decrypt Message 3?

Exercise 5.13.17

1983–3–3

Louis Reasoner's cousin Norris has discovered the following interesting fact, and would like to put it to use:

- Interesting fact: 2^{150} proton-sized objects will compactly fill the known universe.

Since nonces are used in so many different applications, Norris proposes to create the Norris Nonce Service for use by everyone. If you send a request to Norris's service it will return the next 200-bit integer, in increasing order, for use as a nonce. (Norris chose 200 in case the size of the universe turns out to have been underestimated.) What are some of the things that make this proposal harder to do than Norris probably suspects?

This page titled 5.13: Exercises is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

CHAPTER OVERVIEW

6: Ancillary Materials

6.1: Powerpoint Presentations

6.2: Problem Sets

6.2.1: Problem Sets 1-10

6.2.2: Problem Sets 11-20

6.2.3: Problem Sets 21-30

6.2.4: Problem Sets 31-32

6.3: Suggestions for Further Reading

This page titled 6: Ancillary Materials is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

6.1: Powerpoint Presentations

Template:HideTOC

Attachments

File	Description	Last Modified
atomicity_figures.ppt		August 02, 2021
consistency_figures.ppt		August 02, 2021
faults_figures.ppt		August 02, 2021
networks_figures.ppt		August 02, 2021
protection_figures.ppt		August 02, 2021

This page titled 6.1: Powerpoint Presentations is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Jerome H. Saltzer & M. Frans Kaashoek (MIT OpenCourseWare) .

6.2: Problem Sets

These problem sets seek to make the student think carefully about how to apply the concepts of the text to new problems. These problems are derived from examinations given over the years while teaching the material in this textbook. Many of the problems are multiple choice with several right answers. The reader should try to identify all right options.

Some significant and interesting system concepts that are not mentioned in the main text, and therefore at first read seem to be missing from the book, are actually to be found within the exercises and problem sets. Definitions and discussion of these concepts can be found in the text of the exercise or problem set in which they appear .

Some of these problem sets span the topics of several different chapters. A parenthetical note at the beginning of each set indicates the primary chapters that it involves. Following each exercise or problem set question is an identifier of the form "1978–3–14" This identifier reports the year, examination number, and problem number of the examination in which some version of that problem first appeared. For those problem sets not developed by one of the authors, a credit line appears in a footnote on the first page of the problem set.

6.2: Problem Sets is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

6.2.1: Problem Sets 1-10

Problem Set 1: Quarria

1997–1–2a…m

Quarria is a new country formed on a 1-kilometer-wide rock island in the middle of the Pacific Ocean. The founders have organized the Quarria Stock Market in order to get the economy rolling. The stock market is very simple, since there is only one stock to trade (that of the Quarria Rock Company). Moreover, due to local religious convictions, the price of the stock is always precisely the wind velocity at the highest point on the island. Rocky, Quarria's president, proposes that the stock market be entirely network-based. He suggests running the stock market from a server machine, and requiring each investor to have a separate client machine which makes occasional requests to the server using a simple RPC protocol. The two remote procedures Rocky proposes supporting are

- `BALANCE()` : requests that the server return the cash balance of a client's account. This service is very fast, requiring a simple read of a memory location.
- `TRADE(nshares)` : requests that *nshares* be bought (assuming *nshares* is positive) or *nshares* be sold (if *nshares* is negative) at the current market price. This service is potentially slow, since it potentially involves network traffic in order to locate a willing trade partner.

Quarria implements a simple RPC protocol in which a client sends the server a request message with the following format:

```
structure Request
    integer Client     // Unique code for the client
    integer Opcode     // Code for operation requested
    integer Argument   // integer argument, if any
    integer Result     // integer return value, if any
```

The server replies by sending back the same message, with the *Result* field changed. We assume that all messages fit in one packet, that link- and network-layer error checking detect and discard garbled packets, and that Quarria investors are scrupulously honest; thus any received message was actually sent by some client (although sent messages might get lost).

Q1.1 Is this RPC design appropriate for a connectionless network model, or is a connection-based model assumed?

The client RPC stub blocks the client thread until a reply is received, but includes a timer expiration allowing any client RPC operation to return with the error code `TIME_EXPIRED` if no response is heard from the server after Q seconds.

Q1.2 Give a reason for preferring returning a `TIME_EXPIRED` error code over simply having the RPC operation block forever.

Q1.3 Give a reason for preferring returning a `TIME_EXPIRED` error code over having the RPC stub transparently retransmit the message.

Q1.4 Suppose you can bound the time taken for a request, including network and server time, at 3 seconds. What advantage is there to setting the expiration time, Q, to 4 seconds instead of 2 seconds?

Unfortunately, no such bound exists for Quarria's network.

Q1.5 What complication does client message retransmission introduce into the RPC semantics, in the absence of a time bound?

Rocky's initial implementation of the server is as follows:

```
integer Cash[1000]          // Cash balance of each client
integer Shares[1000]        // Stock owned by each client

procedure SERVER ()
    Request instance req, rep     // Pointer to request message
    do forever // loop forever...
        req ← GETNEXTREQUEST ()    // take next incoming request,
        if req.Opcode = 1 then     // …and dispatch on opcode.
            rep ← BALANCE (req)    // Request 1: return balance
            SEND (rep)
        if req.Opcode = 2 then     // Request 2: buy/sell stock
            rep ← TRADE (req);
            SEND (rep)
```

```
// Process a BALANCE request...
procedure BALANCE (Request instance req)
    client ← req.Client           // Get client number from request
    req.Result ← Cash[client]     // Return his cash balance
    return req                    // and return reply.

// Perform a trade: buy/sell Argument/-Argument shares of stock and
// return the total number of shares owned after trade.
procedure TRADE (Request instance req)
    client ← req.Client                         // The client who is requesting
    p ← STOCKPRICE ()                           // Price, using network requests
    nshares ← req.Argument                      // Number of shares to buy/sell
    actual ← CONFIRMTRADE (req, p, nshares)     // See how many shares we can trade
    Cash[client]← Cash[client] + p x actual     // Update our records
    Shares[client] ← Shares[client] + actual
    req.Result ← actual
    return req
```

Note that `CONFIRMTRADE` uses network communication to check on available shares, executes the trade, and returns the number of shares that have actually been bought or sold.

Rocky tests this implementation on a single server machine by having clients scattered around the island sending `BALANCE` requests as fast as they can. He discovers that after some point adding more clients doesn't increase the throughput—the server throughput tops out at 1000 requests per second.

Q1.6 Rocky is concerned about performance, and hires you to recommend steps for improvement. Which, if any, of the following steps might significantly improve Rocky's measured 1000 `BALANCE` requests per second?

A. Use faster client machines.
B. Use multiple client threads (each making `BALANCE` requests) on each client.
C. Use a faster server machine.
D. Use faster network technology.

Stone Galore, a local systems guru, has another suggestion to improve the performance generally. He proposes multithreading the server, replacing calls to service procedures like

```
BALANCE (req)                   // Run BALANCE, to service request
```

with

```
CREATE_THREAD (BALANCE, req)     // create thread to run BALANCE (req)
```

The `CREATE_THREAD` primitive creates a new thread, runs the supplied procedure (in this case `BALANCE`) in that thread, and deactivates the thread on completion. Stone's thread implementation is preemptive.

Stone changes the appropriate three lines of the original code according to the above model, and retries the experiment with `BALANCE` requests. He now measures a maximum server throughput of 500 requests per second.

Q1.7 Explain the performance degradation.

Q1.8 Is there an advantage to the use of threads in other requests? Explain.

Q1.9 Select the best advice for Rocky regarding server threads:

A. Don't use threads; stick with your original design.
B. Don't use threads for Balance requests, but use them for other requests.
C. Continue using them for all requests; the benefits outweigh the costs.

Independently of your advice, Stone is determined to stick with the multithreaded implementation.

Q1.10 Should the code for `TRADE` be changed to reflect the fact that it now operates in a multithreaded server environment? Explain, suggesting explicit changes as necessary.

Q1.11 What if the client is multithreaded and can have multiple request outstanding? Should the code for `TRADE` be changed? Explain, suggesting explicit changes as necessary.

Rocky decides that multithreaded clients are complicated and abandons that idea. He hasn't read about RPC, and isn't sure whether his server requires at-most-once RPC semantics.

Q1.12 Which of the requests require at-most-once RPC semantics? Explain

Q1.13 Suggest how one might modify Rocky's implementation to guarantee at-most-once semantics. Ignore the possibility of crashes, but consider lost messages and retransmissions.

Problem Set 2: PigeonExpress!.com I

1999–2–7…15

(Chapter 1)

Ben Bitdiddle cannot believe the high valuations of some Internet companies, so he is creating a startup, PigeonExpress!.com, which provides high-performance networking using pigeons. Ben's reasoning is that it is cheaper to build a network using pigeons than it is to dig up streets to lay new cables. Although there is a standard for transmitting Internet datagrams with avian carriers (see network RFC 1149) it is out of date, and Ben has modernized it.

When sending a pigeon, Ben's software prints out a little header on a sticky label and also writes a compact disk (CD) containing the data. Someone sticks the label on the disk and gives it to the pigeon. The header on the label contains the Global Positioning System (GPS) coordinates of the destination and the source (the point where the pigeon is taking off), a type field indicating the kind of message (REQUEST or ACKNOWLEDGMENT), and a sequence number:

```
structure header
    GPS source
    GPS destination
    integer type
    integer sequence_no
```

The CD holds a maximum of 640 megabytes of data, so some messages will require multiple CDs. The pigeon reads the header and delivers the labeled CD to the destination. The header and data are never corrupted and never separated. Even better, for purposes of this problem, computers don't fail. However, pigeons occasionally get lost, in which case they never reach their destination.

To make life tolerable on the pigeon network, Ben designs a simple end-to-end protocol (Ben's End-to-End Protocol, BEEP) to ensure reliable delivery. Suppose that there is a single sender and a single receiver. The sender's computer executes the following code:

```
shared next_sequence initially 0      // a global sequence number, starting at 0.

procedure BEEP (destination, n, CD[])     // send n CDs to destination
    header h                              // h is an instance of header.
    nextCD ← 0
    h.source ← MY_GPS                     // set source to my GPS coordinates
    h.destination ← destination           // set destination
    h.type ← REQUEST                      // this is a request message
    while nextCD < n do                   // send the CDs
        h.sequence_no ← next_sequence     // set seq number for this CD
        send pigeon with h, CD[nextCD]    // transmit
        wait 2,000 seconds
```

Pending and incoming acknowledgments are processed only when the sender is waiting:

```
procedure PROCESS_ACK (h)                   // process acknowledgment
    if h.sequence_no = sequence then        // ack for current outstanding CD?
        next_sequence ← next_sequence + 1
        nextCD ← nextCD + 1                 // allow next CD to be sent
```

The receiver's computer executes the following code. The arrival of a request triggers invocation of `PROCESS_REQUEST`:

```
procedure PROCESS_REQUEST (h, CD)     // process request
    PROCESS (CD)                      // process the data on the CD
    h.destination ← h.source          // send to where the pigeon came from
    h.source ← MY_GPS
    h.sequence_no ← h.sequence_no     // unchanged
    h.type ← ACKNOWLEDGMENT
    send pigeon with h                // send an acknowledgment back
```

Q2.1 If a pigeon travels at 100 meters per second (these are express pigeons!) and pigeons do not get lost, then what is the maximum data rate observed by the caller of `BEEP` on a pigeon link that is 50,000 meters (50 kilometers) long? Assume that the processing

delay at the sender and receiver are negligible.

Q2.2 Does at least one copy of each CD make it to the destination, even though some pigeons are lost?

A. Yes, because `nextCD` and `next_sequence` are incremented only on the arrival of a matching acknowledgment.
B. No, since there is no explicit loss-recovery procedure (such as a timer expiration procedure).
C. No, since both request and acknowledgments can get lost.
D. Yes, since the next acknowledgment will trigger a retransmission

Q2.3 To guarantee that each CD arrives at most once, what is required?

A. We must assume that a pigeon for each CD has to arrive eventually.
B. We must assume that acknowledgment pigeons do not get lost and must arrive within 2,000 seconds after the corresponding request pigeon is dispatched.
C. We must assume request pigeons must never get lost.
D. Nothing. The protocol guarantees at-most-once delivery.

Q2.4 Ignoring possible duplicates, what is needed to guarantee that CDs arrive in order?

A. We must assume that pigeons arrive in the order in which they were sent.
B. Nothing. The protocol guarantees that CDs arrive in order.
C. We must assume that request pigeons never get lost.
D. We must assume that acknowledgment pigeons never get lost.

To attract more users to PigeonExpress!, Ben improves the throughput of the 50-kilometer-long link by using a window-based flow-control scheme. He picks window (number of CDs) as the window size and rewrites the code. The code to be executed on the sender's computer now is:

```
procedure BEEP (destination, n, CD[])    // send n CDs to destination
    nextCD ← 0
    window ← 10                          // initial window size is 10 CDs
    h.source ← MY_GPS                    // set source to my GPS coordinates
    h.destination ← destination          // set destination to the destination
    h.type ← REQUEST                     // this is a request message
    while nextCD < n do                  // send the CDs
        CDsleft ← n - nextCD
        temp ← FOO (CDsleft, window)           // FOO computes how many pigeons to send
        for k from 0 to (temp - 1) do
            h.sequence_no ← next_sequence;     // set seq number for this CD
            send pigeon with h, CD[nextCD + k] // transmit
        wait 2,000 seconds
```

The procedures `PROCESS_ACK` and `PROCESS_REQUEST` are unchanged.

Q2.5 What should the procedure `FOO` compute in this code fragment?

A. minimum.
B. maximum.
C. sum.
D. absolute difference.

Q2.6 Alyssa looks at the code and tells Ben it may lose a CD. Ben is shocked and disappointed. What should Ben change to fix the problem?

A. Nothing. The protocol is fine; Alyssa is wrong.
B. Ben should modify `PROCESS_REQUEST` to accept and process CDs in the order of their sequence numbers.
C. Ben should set the value of `window` to the delay × bandwidth product.
D. Ben should ensure that the sender sends at least one CD after waiting for 2,000 seconds.

Q2.7 Assume pigeons do not get lost. Under what assumptions is the observed data rate for the window-based `BEEP` larger than the observed data rate for the previous `BEEP` implementation?

A. The time to process and launch a request pigeon is less than 2,000 seconds
B. The sender and receiver can process more than one request every 2,000 seconds
C. The receiver can process less than one pigeon every 2,000 seconds

After the initial success of PigeonExpress!, the pigeons have to travel farther and farther, and Ben notices that more and more pigeons don't make it to their destinations because they are running out of food. To solve this problem, Ben calls up a number of his friends in strategic locations and asks each of them to be a hub, where pigeons can reload on food.

To keep the hub design simple, each hub can feed one pigeon per second and each hub has space for 100 pigeons. Pigeons feed in first-in, first-out order at a hub. If a pigeon arrives at a full hub, the pigeon gets lucky and retires from PigeonExpress!. The hubs run a patented protocol to determine the best path that pigeons should travel from the source to the destination.

Q2.8 Which layer in the reference model of Chapter 1 provides functions most similar to the system of hubs?

A. the end-to-end layer
B. the network layer
C. the link layer
D. the network layer and the end-to-end layer
E. the feeding layer

Q2.9 Assume Ben is using the window-based `BEEP` implementation. What change can Ben make to this `BEEP` implementation in order to make it respond gracefully to congested hubs?

A. Start with a window size of 1 and increase it by 1 upon the arrival of each acknowledgment.
B. Have `PROCESS_REQUEST` delay acknowledgments and have a single pigeon deliver multiple acknowledgments.
C. Use a window size smaller than 100 CDs, since the hub can hold 100 pigeons.
D. Use multiplicative decrease and additive increase for the window size.

Problem Set 3: Monitoring Ants

2003–2–5…12

(Chapter 1 with a bit of Chapter 5)

Alice has learned that ants are a serious problem in the dorms. To monitor the ant population she acquires a large shipment of motes. Motes are tiny self-powered computers the size of a grain of sand, and they have wireless communication capability. She spreads hundreds of motes in her dorm, planning to create an **ad hoc wireless network**. Each mote can transmit a packet to another mote that is within its radio range. Motes forward packets containing messages on behalf of other motes to form a network that covers the whole dorm. The exact details of how this network of motes works are our topic.

Each mote runs a small program that every 1 millisecond senses if there are ants nearby. Each time the program senses ants, it increments a counter, called `SensorCount` . Every 16 milliseconds the program sends a message containing the value of `SensorCount` and the mote's identifier to the mote connected to Alice's desktop computer, which has identifier `A` . After sending the message, the mote resets `SensorCount` . All messages are small enough to fit into a single packet.

Only the radio consumes energy. The radio operates at a speed of 19.2 kilobits per second (using a 916.5 megahertz transceiver). When transmitting the radio draws 12 milliamperes at 3 volts DC. Although receiving draws 4.5 millamperes, for the moment assume that receiving is uses no power. The motes have a battery rated at 575 milliamperehours (mAh).

Q3.1 If a mote transmits continuously, about how long will it be until its battery runs down?

Q3.2 How much energy (voltage × current × time) does it take to transmit a single bit (1 watt-second = 1 joule)?

Because the radio range of the motes is only ten meters, the motes must cooperate to form a network that covers Alice's dorm. Motes forward packets on behalf of other motes to provide connectivity to Alice's computer, `A` . To allow the motes to find paths and to adapt to changes in the network (e.g., motes failing because their batteries run down), the motes run a routing protocol. Alice has adopted the path-vector routing protocol from Chapter 1. Each mote runs the following routing algorithm, which finds paths to mote `A` :

```
n ← MYID
if n = A then path ← []
else path ← NULL

procedure ADVERTISE ()
    if path ≠ NULL then TRANSMIT ({n, path}) // send marshaled message

procedure RECEIVE (p)
    if n in p then return
    else if (path = NULL) or (FIRST (p) = FIRST(path)) or (LENGTH (p) < LENGTH(path))
    then path ← p
```

```
procedure TIMER ()
    if HAVENOTHEARDFROMRECENTLY (FIRST (path)) then path ← NULL
```

When a mote starts it initializes its variables n and $path$. Each mode has a unique ID, which the mote stores in the local variable n . Each mote stores its path to A into the $path$ variable. A path contains a list of mote IDs; the last element of this list is A . The first element of the list $(FIRST\ (path))$ is the first mote on the path to A . When a mote starts it sets path to NULL , except for Alice's mote, which sets $path$ to the empty path ([]).

Every t seconds a mote creates a path that contains its own ID concatenated with the value of path, and transmits that path (see ADVERTISE) using its radio. Motes in radio range may receive this packet. Motes outside radio range will not receive this packet. When a mote receives a routing packet, it invokes the procedure RECEIVE with the argument set to the path p stored in the routing packet. If p contains n , then this routing packet circled back to n and the procedure RECEIVE just returns, rejecting the path. Otherwise, RECEIVE updates $path$ in three cases:

- if $path$ is NULL , because n doesn't have any path to A yet.
- if the first mote on $path$ is the mote from which we are receiving p , because that mote might have a new path to A .
- if p is a shorter path to A than $path$. (Assume that shorter paths are better.)

A mote also has a timer. If the timer goes off, it invokes the procedure TIMER . If since the last invocation of TIMER a mote hasn't heard from the mote at the head of the path to A , it resets $path$ to NULL because apparently the first node on $path$ is no longer reachable.

The forwarding protocol uses the paths found by the routing protocol to forward reports from a mote to A . Since A may not be in radio range, a report packet may have to be forwarded through several motes to reach A . This forwarding process works as follows:

```
structure report
    id
    data

procedure SEND (counter)
    report.id ← n
    report.data ← counter
    if path ≠ NULL then TRANSMIT (FIRST (path), report);

procedure FORWARD (nexthop, report)
    if n ≠ nexthop then return
if n = A then DELIVER (report)
else TRANSMIT (FIRST (path), report)
```

The procedure SEND creates a report (the mote's ID and its current counter value) and transmits a report packet. The report packet contains the first hop on path, that is FIRST (path), and the report:



Figure $PS.3.1$: Structure of the report packet transmited by SEND .

The $nexthop$ field contains the ID of the mote to which this packet is directed. The source field contains the ID of the mote that originated the packet. The $sensorcount$ field contains the sensor count. (If $path$ is NULL , SEND has no mote to forward the report to so the mote drops the packet.)

When a mote receives a report packet, it calls FORWARD . If a mote receives a report packet for which it is not the next hop, it ignores the packet. If a report packet has reached its final destination A , FORWARD delivers it to Alice's computer. Otherwise, the mote forwards the report by setting $nexthop$ to the first mote on its path variable. This process repeats until the report reaches A , and the packet is delivered.

Suppose we have the following arrangement of motes:

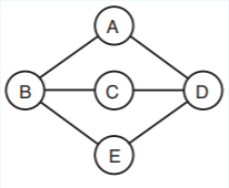


Figure $PS.3.2$: The arrangement of five motes, identified as A through E.

The circles represent motes with their node IDs. The edges connect motes that are in radio range of one another. For example, when mote A transmits a packet, it may be received by B and D, but not by C and E.

Packets may be lost and motes may also fail (e.g., if their batteries run down). If a mote fails, it just stops sensing, transmitting, and receiving packets.

Q3.3 If motes may fail and if packets may be lost, which of the following values could the variable `path` at node E have?

A. NULL
B. [B, C]
C. [D, A]
D. [B, A]
E. [B, C, D, A]
F. [C, D, A]
G. [D, A, B, C, D, A]

Q3.4 If no motes fail and packets are not lost, what properties hold for Alice's routing and forwarding protocols with the given arrangement of motes?

A. Every mote's path variable will contain a path to `A` .
B. After the routing algorithm reaches a stable state, every mote's path variable will contain a shortest path (in hops) to `A` .
C. After the routing algorithm reaches a stable state, the routing algorithm may have constructed a forwarding cycle.
D. After the routing algorithm reaches a stable state, the longest path is two hops.

Q3.5 If packets may be lost but motes don't fail, what properties hold for Alice's routing and forwarding protocols with the given arrangement of motes?

A. Every mote's path variable will contain a path to `A` .
B. The routing algorithm may construct forwarding cycles.
C. The routing algorithm may never reach a stable state.
D. When a mote sends a report packet using send, the report may or may not reach `A` .

A report is 13 bits: an 8-bit node ID and a 5-bit counter. A report packet is 21 bits: an 8-bit next hop and a report. Assume that your answer to **Q3.2** (the energy for transmitting one bit) is j joules. Further assume that to start the radio for transmission takes s joules. Thus, transmitting a packet with r bits takes $s + r \times j$ joules.

Q3.6 Assuming the routing algorithm reaches a stable state, with no node or packet failures, how much total energy does it take for every node to send one report to `A` (ignoring routing packets)?

Q3.7 Which of the following changes to Alice's system will reduce the amount of energy needed for each node to send one report to `A` ?

A. Add a nonce to a report so that Alice's computer can detect duplicates.
B. Delay forwarding packets and piggyback them on a report packet from this mote.
C. Use 4-bit node IDs.
D. Use a stop-and-wait protocol to transmit each report reliably from one mote to the next.

To be able to verify the integrity of the reports, Alice creates for each mote a public key pair. She manually loads the private key for mote `i` into mote `i` , and keeps the corresponding public keys on her computer. A mote signs the contents of the report (the counter and the source) with its private key and then transmits it.

Thus, a signed report consists of a 5-bit counter, a node ID, and a signature. When Alice's computer receives a signed report, it verifies the signed report and rejects the ones that don't check out.

Q3.8 Assuming that the private keys on the motes are not compromised, and the `SIGN` and `VERIFY` procedures are correctly implemented, which of the following properties hold for Alice's plan?

A. A mote that forwards a report is not able to read the report's content.
B. A mote that forwards a report may be able to duplicate a report without Alice's computer rejecting the duplicated report.
C. A mote that forwards a report can use its private key to verify the report.
D. When Alice receives a report for which `VERIFY` returns `ACCEPT` , she knows that the report was signed by the mote that is listed in the report and that the report is fresh.

Problem Set 4: Gnutella

2002–2–5…12

Chapters 1 and 5

Ben Bitdiddle is disappointed that the music industry is not publishing his CD, a rap production based on this textbook. Ben is convinced there is a large audience for his material. Having no alternative, he turns his CD into a set of MP3 files, the digital music standard understood by music playing programs, and publishes the songs through Gnutella.

Gnutella is a distributed file sharing application for the Internet. A user of Gnutella starts a Gnutella node, which presents a user interface to query for songs, talks to other nodes, and makes files from its local disk available to other remote users. The Gnutella nodes form what is called an **overlay network** on top of the existing Internet. The nodes in the overlay network are Gnutella nodes and the links between them are TCP connections. When a node starts it makes a TCP connection to various other nodes, which are connected through TCP connections to other nodes. When a node sends a message to another node, the message travels over the connections established between the nodes. Thus, a message from one node to another node travels through a number of intermediate Gnutella nodes.

To find a file, the user interface on the node sends a query (e.g., "System Design Rap") through the overlay network to other nodes. While the search propagates through the Gnutella network, nodes that have the desired file send a reply, and the user sees a list filling with file names that match the query. Both the queries and their replies travel through the overlay network. The user then selects one of the files to download and play. The user's node downloads the file directly from the node that has the file, instead of through the Gnutella network.

The format of the header of a Gnutella message is:

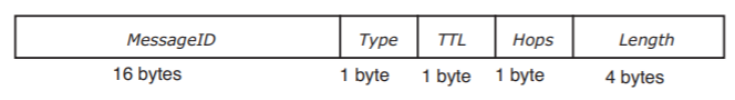


Figure $PS.4.1$: Format of the header of a Gnutella message.

This header is followed by the payload, which is `Length` bytes long. The main message Types in the Gnutella protocol are:

- `PING` : A node finds additional Gnutella nodes in the network using `PING` messages. A node wants to be connected to more than one other Gnutella node to provide a high degree of connectivity in the case of node failures. Gnutella nodes are not very reliable because a user might turn off his machine running a Gnutella node at any time. `PING` messages have no payload.
- `PONG` : A node responds by sending a `PONG` message via the Gnutella network whenever it receives a `PING` message. The `PONG` message has the same *`MessageID`* as the corresponding `PING` message. The payload of the `PONG` message is the Internet address of the node that is responding to the `PING` Message.
- `QUERY` : Used to search the Gnutella network for files; its payload contains the query string that the user typed.
- `QUERYHIT` : A node responds by sending a `QUERYHIT` message via the Gnutella network if it has a file that matches the query in a `QUERY` message it receives. The payload contains the Internet address of the node that has the file, so that the user's node can connect directly to the node that has the song and download it. The `QUERYHIT` message has the same *`MessageID`* as the corresponding `QUERY` message.

(The Gnutella protocol also has a `PUSH` message to deal with firewalls and network address translators, but we will ignore it.)

In order to join the Gnutella network, the user must discover and configure the local node with the addresses of one or more existing nodes. The local node connects to those nodes using TCP. Once connected, the node uses `PING` messages to find more nodes (more detail below), and then directly connects to some subset of the nodes that the `PING` message found.

For `QUERY` and `PING` messages, Gnutella uses a kind of broadcast protocol known as **flooding**. Any node that receives a `PING` or a `QUERY` message forwards that message to all the nodes it is connected to, except the one from which it received the message. A node decrements the *`TTL`* field and increments the Hops field before forwarding the message. If after decrementing the *`TTL`* field, the *`TTL`* field is zero, the node does not forward the message at all. The *`Hops`* field is set to zero by the originating user's node.

To limit flooding and to route `PONG` and `QUERYHIT` messages, a node maintains a message table, indexed by *`MessageID`* and *`Type`* , with an entry for each message seen recently. The entry also contains the Internet address of the Gnutella node that forwarded the message to it. The message table is used as follows:

- If a `PING` or `QUERY` message arrives and there is an entry in the message table with the same messageID and Type, then the node discards that message.
- For a `QUERYHIT` or `PONG` message for which there is a corresponding `QUERY` or `PONG` entry with the same *`MessageID`* in the message table, then the node forwards the `QUERYHIT` or `PONG` to the node from which the `QUERY` or `PING` was received.
- If the corresponding `QUERY` or `PING` message doesn't appear in the table, then the node discards the `QUERYHIT` or `PONG` message.

- Otherwise, the node makes a new entry in the table, and forwards the message to all the nodes it is connected to, except the one from which it received the message.

Q4.1 Assume one doesn't know the topology of the Gnutella network or the propagation delays of messages. According to the protocol, a node should forward all `QUERYHIT` messages for which it saw the corresponding `QUERY` message back to the node from which it received the QUERY message. If a node wants to guarantee that rule, when can the node remove the `QUERY` entry from the message table?

A. Never, in principle, because a node doesn't know if another `QUERYHIT` for the same Query will arrive.
B. Whenever it feels like, since the table is not necessary for correctness. It is only a performance optimization. **C.** As soon as it has forwarded the corresponding `QUERYHIT` message.
D. As soon as the entry becomes the least recently used entry.

Both the Internet and the Gnutella network form graphs. For the Internet, the nodes are routers and the edges are links between the routers. For the Gnutella network, the nodes are Gnutella nodes and the edges are TCP connections between the nodes. The shortest path in a graph between two nodes A and B is the path that connects A with B through the smallest number of nodes.

Q4.2 Assuming a stable Internet and Gnutella network, is the shortest path between two nodes in the Gnutella overlay network always the shortest path between those two nodes in the Internet?

A. Yes, because the Gnutella network uses the Internet to set up TCP connections between its nodes.
B. No, because TCP is slower than UDP.
C. Yes, because the topology of the Gnutella network is identical to the topology of the Internet.
D. No, because for node A to reach node B in the Gnutella network, it might have to go through node C, even though there is a direct, Internet link between A and B.

Q4.3 Which of the following relationships always hold? (`TTL(i)` and `HOP(i)` are the values of the *TTL* and *Hop* fields respectively after the message has traversed i hops)

A. `TTL(0) = HOPS(i) - TTL(i)`
B. `TTL(i) = TTL(i-1) - 1,` for $i > 0$
C. `TTL(0) = TTL(i) + HOPS(i)`
D. `TTL(0) = TTL(i) × HOPS(i)`

Q4.4 Ben observes that both `PING` and `QUERY` messages have the same forwarding rules, so he proposes to delete `PING` and `PONG` messages from the protocol and to use a `QUERY` message with a null query (which requires a node to respond with a `QUERYHIT` message) to replace `PING` messages. Is Ben's modified protocol a good replacement for the Gnutella protocol?

A. Yes, good question. Beats me why the Gnutella designers included both `PING` and `QUERY` messages.
B. No, a `PING` message will typically have a lower value in the *TTL* field than a `QUERY` message when it enters the network.
C. No, because `PONG` and `QUERYHIT` messages have different forwarding rules.
D. No, because there is no way to find nodes using `QUERY` messages.

Q4.5 Assume that only one node S stores the song "System Design Rap," and that the query enters the network at a node C. Further assume *TTL* is set to a value large enough to explore the whole network. Gnutella can still find the song "System Design Rap" despite the failures of some sets of nodes (either Gnutella nodes or Internet routers). On the other hand, there are sets of nodes whose failure would prevent Gnutella from finding the song. Which of the following are among the latter sets?

A. any set containing S
B. any set containing a single node on the shortest path from C to S
C. any set of nodes that collectively disconnects C from S in the Gnutella network
D. any set of nodes that collectively disconnects C from S in the Internet

Q4.6 To which of the following attacks is Gnutella vulnerable (i.e., an attacker can implement the described attack)?

A. A single malicious node can always prevent a client from finding a file by dropping `QUERYHIT` s.
B. A malicious node can respond with a file that doesn't match the query.
C. A malicious node can always change the contact information in a `QUERYHIT` message that goes through the node, for example, misleading the client to connect to it.
D. A single malicious node can always split the network into two disconnected networks by never forwarding `PING` and `QUERY` messages.
E. A single malicious node can always cause a `QUERY` message to circle forever in the network by incrementing the *TTL* field (instead of decrementing it).

Q4.7 Ben wants to protect the content of a song against eavesdroppers during downloads. Ben thinks a node should send `ENCRYPT (k, song)`, using a shared-secret algorithm, as the download, but Alyssa thinks the node should send `CSHA (song)`, where CSHA is a cryptographically secure hash algorithm. Who is right?

A. Ben is right because no one can compute song from the output of `CSHA (song)`, unless they already have `song`.
B. Alyssa is right because even if one doesn't know the shared-secret key `k`, anyone can compute the inverse of the output of `ENCRYPT (k, song)`.
C. Alyssa is right because `CSHA` doesn't require a key and therefore Ben doesn't have to design a protocol for key distribution.
D. Both are wrong because a public-key algorithm is the right choice, since encrypting with a public key algorithm is computationally more expensive than either `CSHA` or a shared-secret algorithm.

Ben is worried that an attacker might modify the "System Design Rap" song. He proposes that every node that originates a message signs the payload of a message with its private key. To discover the public keys of nodes, he modifies the `PONG` message to contain the public key of the responding node along with its Internet address. When a node is asked to serve a file it signs the response (including the file) with its private key.

Q4.8 Which attacks does this scheme prevent?

A. It prevents malicious nodes from claiming they have a copy of the "System Design Rap" song and then serving music written by Bach.
B. It prevents malicious nodes from modifying `QUERY` messages that they forward.
C. It prevents malicious nodes from discarding `QUERY` messages.
D. It prevents nodes from impersonating other nodes and thus prevents them from forging songs.
E. None. It doesn't help.

Problem Set 5: The Ottonet*

* Credit for developing this problem set goes to Hari Balakrishnan.

2004–2–5…13

(Chapter 1 with a bit of Chapter 5)

Inspired by the recent political success of his Austrian compatriot, "Arnie," in Caleeforneea, Otto Pilot decides to emigrate to Boston. After several months, he finds the local accent impenetrable, and the local politics extremely murky, but what really irks him are the traffic nightmares and long driving delays in the area.

After some research, he concludes that the traffic problems can be alleviated if cars were able to discover up-to-date information about traffic conditions at any specified location, and use this information as input to software that can dynamically suggest good paths to use to go from one place to another. He jettisons his fledgling political career to start a company whose modest goal is to solve Boston's traffic problems.

After talking to car manufacturers, Otto determines the following:

1. All cars have an on-board computer on which he can install his software. All cars have a variety of sensors that can be processed in the car to provide traffic status, including current traffic speed, traffic density, evidence of accidents, construction delays, etc.
2. It is easy to equip a car with a Global Positioning System (GPS) receiver (in fact, an increasing number of cars already have one built-in). With GPS, software in the car can determine the car's location in a well-known coordinate system. (Assume that the location information is sufficiently precise for our purposes.)
3. Each car's computer can be networked using an inexpensive 10 megabits per second radio. Each radio has a spherical range, R, of 250 meters; i.e., a radio transmission from a car has a non-zero probability of directly reaching any other car within 250 meters, and no chance of directly reaching any car outside that range.

Otto sets out to design the OttoNet, a network system to provide traffic status information to applications. OttoNet is an *ad hoc* wireless network formed by cars communicating with each other using cheap radios, cooperatively forwarding packets for one another.

Each car in OttoNet has a client application and a server application running on its computer. OttoNet provides two procedures that run on every car, which the client and server applications can use:

1. `QUERY (location)`: When the client application running on a car calls `QUERY (location)`, OttoNet delivers a packet containing a query message to at least one car within distance R (the radio range) of the specified location, according to a best-effort contract. A packet containing a query is 1,000 bits in size.

2. `RESPOND (status_info, query_packet)` : When the server application running on a car receives a query message, it processes the query and calls `RESPOND (status_info, query_packet)` . `RESPOND` causes a packet containing a response message to be delivered to the client that performed the query, again according to a best-effort contract. A response message summarizes local traffic information (`status_info`) collected from the car's sensors and is 10,000 bits in size.

For packets containing either query or response messages, the cars will forward the packet cooperatively in best-effort fashion toward the desired destination location or car. Cars may move arbitrarily, alternating between motion and rest. The maximum speed of a car is 30 meters per second (108 kilometers per hour or 67.5 miles per hour).

Q5.1 Which of the following properties is true of the OttoNet, as described thus far?

A. Because the OttoNet is "best-effort," it will attempt to deliver query and response messages between client and server cars, but messages may be lost and may arrive out of order.
B. Because the OttoNet is "best-effort," it will ensure that as long as there is some uncongested path between the client and server cars, query and response messages will be successfully delivered between them.
C. Because the OttoNet is "best-effort," it makes no guarantees on the delay encountered by a query or response message before it reaches the intended destination.
D. An OttoNet client may receive multiple responses to a query, even if no packet retransmissions occur in the system.

Otto develops the following packet format for OttoNet (all fields except `payload` are part of the packet header):

```
structure packet
    GPS dst_loc              // intended destination location
    integer_128 dst_id       // car's 128-bit unique ID picked at random
    GPS src_loc              // location of car where packet originated
    integer_128 src_id      // unique ID of car where packet originated
    integer hop_limit       // number of hops remaining (initialized to 100)
    integer type            // query or response
    integer size           // size of packet
    string payload         // query request string or response status info
packet instance pkt;       // pkt is an instance of the structure packet
```

Each car has a 128-bit unique ID, picked entirely at random. Each car's current location is given by its GPS coordinates. If the sender application does not know the intended receiver's unique ID, it sets the `dst_id` field to 0 (no valid car has an ID of 0).

The procedure `FORWARD (pkt)` runs in each car, and is called whenever a packet arrives from the network or when a packet needs to be sent by the application. `FORWARD` maintains a table of the cars within radio range and their locations, using broadcasts every second to determine the locations of neighboring cars, and implements the following steps:

F1. If the car's ID is `pkt.dst_id` then deliver to application (using `pkt.type` to identify whether the packet should be delivered to the client or server application), and stop forwarding the packet.

F2. If the car is within R of `pkt.dst_id` and `pkt.type` is `QUERY` , then deliver to server application, and forward to any one neighbor that is even closer to `dst_loc` .

F3. Geographic forwarding step: If neither F1 nor F2 is applicable, then among the cars that are closer to `pkt.dst_loc` , forward the packet to some car that is closer in distance to `pkt.dst_loc` . If no such car exists, drop the packet.

The OttoNet's `QUERY (location)` and `RESPOND (status_info, query_packet)` procedures have the following pseudocode:

```
1 procedure QUERY (location)
2     pkt.dst_lock ← location
3     pkt.dst_id ← X              // see question 5.2.
4     pkt.src_loc ← my_gps
5     pkt.src_id ← my_id
6     pkt.payload ← "What's the traffic status near you?"
7     SEND (pkt)
```

```
8  procedure RESPOND (status_info, query_packet)
9      pkt.dst_loc ← query_packet.src_loc
10     pkt.dst_id ← Y            // see question 5.2.
11     pkt.src_loc ← my_gps
12     pkt.src_id ← my_id
13     pkt.payload ← "My traffic status is:" + status_info   // "+" concatenates strings
14     SEND (pkt)
```

Q5.2 What are suitable values for `X` and `Y` in lines 3 and 10, such that the pseudocode conforms to the specification of `QUERY` and `RESPOND` ?

Q5.3 What kinds of names are the ID and the GPS location used in the OttoNet packets? Are they addresses? Are they pure names? Are they unique identifiers?

Q5.4 Otto outsources the implementation of the OttoNet according to these ideas and finds that there are times when a `QUERY` gets no response, and times when a receiver receives packets that are corrupted. Which of the following mechanisms is an example of an application of an end-to-end technique to cope with these problems?

A. Upon not receiving a response for a `QUERY` , when a timer expires retry the `QUERY` from the client.
B. If `FORWARD` fails to deliver a packet because no neighboring car is closer to the destination, store the packet at that car and deliver it to a closer neighboring car a little while later.
C. Implement a checksum in the client and server applications to verify if a message has been corrupted.
D. Run distinct TCP connections between each pair of cars along the path between a client and server to ensure reliable end-to-end packet delivery.

Otto decides to retry queries that don't receive a response. The speed of the radio in each car is 10 megabits per second, and the response and request sizes are 10,000 bits and 1,000 bits respectively. The car's computer is involved in both processing the packet, which takes 0.1 microsecond per bit, and in transmitting it out on the radio (i.e., there's no pipelining of packet processing and transmission). Each car's radio can transmit and receive packets at the same time.

The maximum queue size is 4 packets in each car, the maximum radio range for a single hop is 250 meters, and that the maximum possible number of hops in OttoNet is 100. Ignore media access protocol delays. The server application takes negligible time to process a request and generate a response to be sent.

Q5.5 What is the smallest "safe" timer expiration setting that ensures that the retry of a query will happen only when the original query or response packet is guaranteed not to still be in transit in the network?

Otto now proceeds to investigate why `FORWARD` sometimes has to drop a packet between a client and server, even though it appears that there is a sequence of nodes forming a path between them. The problem is that geographic forwarding does not always work, in that a car may have to drop a packet (rule F3) even though there is some path to the destination present in the network.

Q5.6 In the figure below, suppose the car at F is successfully able to forward a packet destined to location D using rule F3 via some neighbor, N. Assuming that neither F or N has moved, clearly mark the region in the figure where N must be located.



Figure $PS.5.1$: Problem diagram for Q5.6.

Q5.7 Otto decides to modify the client software to make pipelined `QUERY` calls in quick succession, sending a query before it gets a response to an earlier one. The client now needs to match each response it receives with the corresponding query. Which of these statements is correct?

A. As long as no two pipelined queries are addressed to the same destination location (the *dst_loc* field in the OttoNet header), the client can correctly identify the specific query that caused any given response it receives.
B. Suppose the OttoNet packet header includes a nonce set by the client, and the server includes a copy of the nonce in its response, and the client maintains state to match nonces to queries. This approach can always correctly match a response to a query, including when two pipelined queries are sent to the same destination location.

C. Both the client and the server need to set nonces that the other side acknowledges (i.e., both sides need to implement the mechanism in choice B above), to ensure that a response can always be correctly matched to the corresponding query.
D. None of the above.

Q5.8 After running the OttoNet for a few days, Otto notices that network congestion occasionally causes a congestion collapse because too many packets are sent into the network, only to be dropped before reaching the eventual destination. These packets consume valuable resources. Which of the following techniques is likely to reduce the likelihood of a congestion collapse?

A. Increase the size of the queue in each car from 4 packets to 8 packets.
B. Use exponential backoff for the timer expiration when retrying queries.
C. If a query is not answered within the timer expiration interval, multiplicatively reduce the maximum rate at which the client application sends OttoNet queries.
D. Use a flow control window at each receiver to prevent buffer overruns.

Q5.9 The OttoNet is not a secure system. Otto has an idea—he observes that the unique 128-bit ID of a car can be set to be the public key of the car! He proposes the following protocol. On a packet containing a query message, sign the packet with the client car's private key. On a packet containing a response, encrypt the packet with the client car's public key (that public key is in the packet that contained the query). To allow packets containing responses to be forwarded through the network, the server does not encrypt the destination location and ID fields of those packets. Assume that each car's private key is not compromised. Which of the following statements are true?

A. A car that just forwards a packet containing queries can read that packet's payload and verify it.
B. The only car in the network that can decrypt a response from a server is the car specified in the destination field.
C. The client cannot always verify the message integrity of a response, even though it is encrypted.
D. If every server at some queried location is honest and not compromised, the client can be sure that an encrypted response it receives for a query actually contains the correct traffic status information.

Problem Set 6: The Wireless EnergyNet

* Credit for developing this problem set goes to Hari Balakrishnan.

2005–2–7

(Chapter 1 and a little bit of 2)

Sara Brum, an undergraduate research assistant, is concerned about energy consumption in the Computer Science building and decides to design the EnergyNet, a wireless network of nodes with sensors to monitor the building. Each node has three sensors: a power consumption sensor to monitor the power drawn at the power outlet to which it is attached, a light sensor, and a temperature sensor. Sara plans to have these nodes communicate with each other via radio, forwarding data via each other, to report information to a central monitoring station. That station has a radio-equipped node attached to it, called the **sink**.

There are two kinds of communication in EnergyNet:

1. **Node-to-sink reports**: A node sends a report to the sink via zero or more other nodes.
2. **EnergyNet routing protocol**: The nodes run a distributed routing protocol to determine the next hop for each node to use to forward data to the sink. Each node's next hop en route to the sink is called its **parent**.

EnergyNet is a best-effort network. Sara remembers that layering is a good design principle for network protocols, and decides to adopt a three-layer design similar to the Chapter 1 reference model. Our job is to help Sara design the EnergyNet and its network protocols. We will first design the protocols needed for the node-to-sink reports without worrying about how the routing protocol determines the parent for each node.

To start, let's assume that each node has an unchanging parent, every node has a path to the sink, and nodes do not crash. Nodes may have hardware or software faults, and packets could get corrupted or lost, though. Sara develops the following simple design for the three-layer EnergyNet stack:

Layer	Header fields	Trailer fields
E2E report protocol	*location* *time*	*e2* (32-bit checksum)
Network	*dstaddr* (16-bit network address of destination)	
Layer	*recvid* (32-bit unique ID of link-layer destination) *sendid* (32-bit unique ID of link-layer source)	*l1* (32-bit checksum)

In addition to these fields, each report packet has a payload that contains a report of data observed by a node's sensors. When sending a report packet, the end-to-end layer at the reporting node sets the destination network-layer address to be a well-known 16- bit value, `SINK_ADDR` . The end-to-end layer at the sink node processes each report. Any node in the network can send a report to the sink.

If a layer has a checksum, it covers that layer's header and the data presented to that layer by the higher layer. Each EnergyNet node has a first-in, first-out (FIFO) queue at the network layer for packets waiting to be transmitted.

Q6.1 What does an EnergyNet report frame look like when sent over the radio from one node to another? Fill in the rectangle below to show the different header and trailer fields in the correct order, starting with the first field on the left. Be sure to show the payload as well. You do not need to show field sizes.

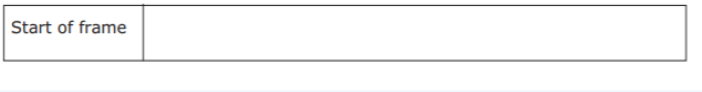


Figure $PS.6.1$: Answer framework for Q6.1.

Q6.2 Sara's goal is to ensure that the end-to-end layer at the sink passes on (to the application) only messages whose end-to-end header and payload are correct. Assume that the implementation of the functions to set and verify the checksum are correct, and that there are no faults when the end-to-end layer runs.

A. Will using just `ll_cksum` and not `e2e_cksum` achieve Sara's goal?
B. Will using just `e2e_cksum` and not `ll_cksum` achieve Sara's goal?
C. Must each node on the path from the reporting node to the sink recalculate `e2e_cksum` in order to achieve Sara's goal?

To recover lost frames, Sara decides to implement a link-layer retransmission scheme. When a node receives a frame whose `ll_cksum` is correct, it sends an acknowledgment (ACK) frame to the `sendid` of the frame. If a sender does not receive an ACK before a timer expires, it retransmits the frame. A sender attempts at most three retransmissions for a frame.

Q6.3 Which of these statements is true of Sara's link-layer retransmission scheme if no node changes its parent?

A. Duplicate error-free frames may be received by a receiver.
B. Duplicate error-free frames may be received by a receiver even if the sending node's timeout is longer than the maximum possible round trip time between sender and receiver.
C. If each new frame is sent on a link only after all link-layer retransmissions of previous frames, then the sink may receive packets from a given node in a different order from the way in which they were sent.
D. If Sara were to implement an end-to-end retransmission scheme in addition to this link-layer scheme, the resulting design would violate an end-to-end argument.

Q6.4 EnergyNet's radios use phase encoding with the Manchester code. Sara finds that if the frequency of level transitions of voltage is set to 500 kilohertz, the link has an acceptably low bit error rate when there is no radio channel interference, noise, or any other concurrent radio transmissions. What is the data rate corresponding to this level transition frequency? (Specify the correct units.)

Q6.5 Consider the transmission of an error-free frame (that is, one that never needed to be retransmitted) over one radio hop from node A to node B. Which of the delays in the right column of the table below contribute to the time duration specified in the left column? (There may be multiple contributors.)

1. Time lag between first bit leaving A and that bit reaching B	A. Processing delay
2. Time lag between first bit reaching B and last bit reaching B	B. Propagation delay
3. Time lag between when the last bit of the packet was received at A and the first bit of the same packet begins to be sent by A's link layer to B	C. Queuing delay
	D. Transmission delay

Q6.6 Sara finds that EnergyNet often suffers from congestion. Which of the following methods is likely to help reduce EnergyNet's congestion?

A. If no link-layer ACK is received, the sender should use exponential backoff before sending the next frame over the radio.
B. Provision the network-layer queue at each node to ensure that no packets ever get dropped for lack of queue space.
C. On each link-layer ACK, piggyback information about how much queue space is available at a parent, and slow down a node's rate of transmission when its parent's queue occupancy is above some threshold.

Now, let's assume that nodes may crash and each node's parent may change with time.

Let us now turn to designing the routing protocol that EnergyNet nodes use to form a routing tree rooted at the sink. Once each second, each node picks a parent by optimizing a "quality" metric and broadcasts a routing advertisement over its radio, as shown in the `BROADCAST_ADVERTISEMENT` procedure. Each node that receives an advertisement processes it and incorporates some information in its routing table, as shown in the `HANDLE_ADVERTISEMENT` procedure. These routing advertisements are not acknowledged by their recipients.

An advertisement contains one field in its payload: `quality` , calculated as shown in the pseudocode below. The **quality** of a path is a function of the success probability of frame delivery across each link on the path. The success probability of a link is the probability that a frame is received at the receiver and its ACK received by the sender.

In the pseudocode below, `quality_table` is a table indexed by `sendid` and stores an object with two fields: `quality` , the current estimate of the path quality to the parent via the corresponding `sendid` , and `lasttime` , the last time at which an advertisement was heard from the corresponding `sendid` .

```
procedure BROADCAST_ADVERTISEMENT () // runs once per second at each node
    if quality_table = EMPTY and node != sink then return
    REMOVE_OLD_ENTRIES (quality_table) // remove entries older than 5 seconds
    if node = sink then
        adv.quality ← 1.0
    else
        parent ← PICK_BEST(quality_table) // returns node with highest quality value
        adv.quality ← quality_table[parent].quality
    NETWORK_SEND (RTG_BCAST_ADDR, adv) // broadcasts adv over radio

procedure HANDLE_ADVERTISEMENT (sendid, adv)
    quality_table[sendid].lasttime ← CURRENT_TIME ()
    quality_table[sendid].quality ← adv.quality × SUCCESS_PROB (sendid)
```

When `BROADCAST_ADVERTISEMENT` runs (once per second), it first removes all entries older than 5 seconds in `quality_table` . Then, it finds the best parent by picking the sendid with maximum quality, and broadcasts an advertisement message out to the network-layer address (`RTG_BCAST_ADDR`) that corresponds to all nodes within one network hop.

Whenever a node receives an advertisement from another node, `sendid` , it runs `HANDLE_ADVERTISEMENT ()` . This procedure updates `quality_table[sendid]` . It calculates the path quality to reach the sink via `sendid` by multiplying the advertised quality with the success probability to this `sendid` , `SUCCESS_PROB (sendid)` . The implementation details of `SUCCESS_PROB ()` are not important here; just assume that all the link success probabilities are estimated correctly.

Assume that no "link" is perfect; i.e., for all `i` and `j` , p_{ij} is strictly less than 1, and that every received advertisement is processed within 100 ms after it was broadcast.

Q6.7 Ben Bitdiddle steps on and destroys the parent of node `N` at time `t` = 10 seconds. Assuming that node `N` has a current entry for its parent in its `quality_table` , to the nearest second, what are the earliest and latest times at which node `N` would remove the entry for its parent from its `quality_table` ?

See Figure $PS.6.2$. The picture shows the success probability for each pair of transmissions (only non-zero probabilities are shown). The number next to each radio link is the link's success probability, the probability of a frame being received by a receiver and its ACK being received successfully by the sender.

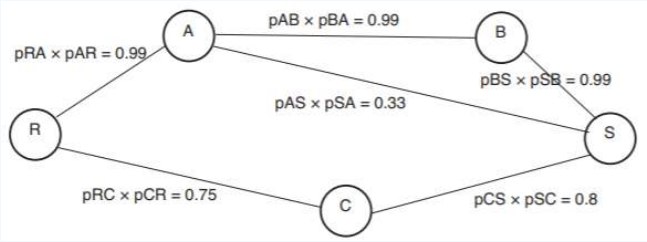


Figure $PS.6.2$: Network topology for some EnergyNet questions.

Q6.8 In Figure $PS.6.2$, suppose B is A's parent and B fails. Louis Reasoner asserts that as long as no routing advertisements are lost and there are no software or hardware bugs or failures, a routing loop can never form in the network. As usual, Louis is wrong. Explain why, giving a scenario or sequence of events that can create a routing loop.

Q6.9 Describe a modification to EnergyNet's routing advertisement that can prevent routing loops from forming in any EnergyNet deployment.

Q6.10 Suppose node B has been restored to service and the success probabilities are as shown. Which path between R and S would be chosen by Sara's routing protocol and why? Name the path as a sequence of nodes starting with R and ending with S.

Q6.11 Returning once again to Figure $PS.6.2$, recall that the nodes use link-layer retransmissions for report packets. If you want to minimize the total expected number of non-ACK radio transmissions needed to successfully deliver the packet from R to S, which path should you choose? You may assume that frames are lost independently over each link and that the link success probabilities are

independent of each other. (Hint: If a coin has a probability p of landing "heads", then the expected number of tosses before you see "heads" is $1/p$.)

Sara finds that each sensor's reported data is noisy, and that to obtain the correct data from a room, she needs to deploy $k > 1$ sensors in the room and take the average of the k reported values. However, she also finds that sensor nodes may fail in fail-fast fashion. Whenever there are fewer than k working sensors in a room, the room is considered to have "failed", and its data is "unavailable". When that occurs, an administrator has to go and replace the faulty sensors for the room to be "available" again, which takes time Tr. Tr is smaller than the MTTF of each sensor, but non-zero.

Assume that the sensor nodes fail independently and that Sara is able to detect the failure of a sensor node within a time much smaller than the node's MTTF. Sara deploys $m > k$ sensors in each room. Sara comes up with three strategies to deploy and replace sensors in a room:

A. Fix each faulty sensor as soon as it fails.
B. Fix the faulty sensors as soon as all but one fail.
C. Fix each faulty sensor as soon as data from the room becomes unavailable.

Q6.12 Rank these strategies in the order of highest to lowest availability for the room's sensor data.

Q6.13 Suppose that each sensor node's failure process is memoryless and that sensors fail independently. Sara picks strategy C from the choices in the previous question. What is the resulting MTTF of the room?

Problem Set 7: SureThing *

* Credit for developing this problem set goes to Barbara Liskov.

2006–2–7

(Chapter 1)

Alyssa P. Hacker decides to offer her own content delivery system, named SURETHING. A SURETHING system contains 1000 computers that communicate via the Internet. Each computer has a unique numerical identifier ID#, and the computers are thought of as (logically) being organized in a ring as in Figure $PS.7.1$. Each computer has successors as shown in the figure. The ring wraps around: the immediate successor of the computer with the highest ID# (computer N251 in the figure) is the computer with the lowest ID# (computer N8).

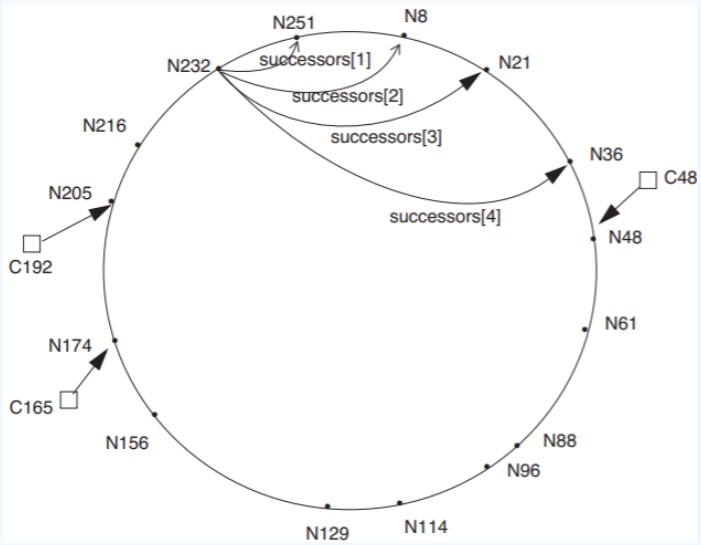


Figure $PS.7.1$: Arrangement of computers in a ring. Computer N232's pointers to its 4 successors lead to computers N251, N8, N21, and N36. The content item C192 is stored at computer N205 because #205 is the next largest computer ID# after C192's ID#. Similarly, content item C48 is stored at its immediate successor, computer N48, and item number C165 is stored at its immediate successor, computer N174.

Each content item also has a unique ID, `c` , and the content is stored at `c` 's **immediate successor**: the first computer in the ring whose ID# exceeds the ID# of `c` . This scheme is called **consistent hashing**.

Alyssa designs the system using two layers: a forwarding and routing layer (to find the IP address of the computer that stores the content) and a content layer (to store or retrieve the content).

Building a Forwarding and Routing Layer. Inspired by reading a paper on a system named Chord** that uses consistent hashing, Alyssa decides that the routing step will work as follows: Each computer has a local table, `successors[i]` , that contains the ID and IP

address of its 4 successors (the 4 computers whose IDs follow this computer's ID in the ring); the entries are ordered as they appear in the ring. These tables are set up when the system is initialized.

The forwarding and routing layer of each node provides a procedure `GET_LOCATION` that can be called by the content layer to find the IP address of the immediate successor of some content item `c`. This procedure checks its local `successors` table to see if it contains the immediate successor of the requested content; if not, it makes a remote procedure call to the `GET_LOCATION` procedure on the most distant successor in its own `successors` table. That computer returns the immediate successor of `c` if it is known locally in its `successors` table; otherwise that node returns its most distant successor, and the originating computer continues the search there, iterating in this way until it locates `c` 's immediate successor.

For example, if computer N232 is looking for the immediate successor of `c` = C165 in the system shown in Figure $PS.7.1$, it will first look in its local table; since this table doesn't contain the immediate successor of `c`, it will request information from computer N36. Computer N36 also doesn't have the immediate successor of C165 in its local `successors` table, and therefore it returns the IP address of computer N96. Computer N96 does have the immediate successor (computer N174) in its local `successors` table and it returns this information. This sequence of RPC requests and responses is shown in Figure $PS.7.2$.

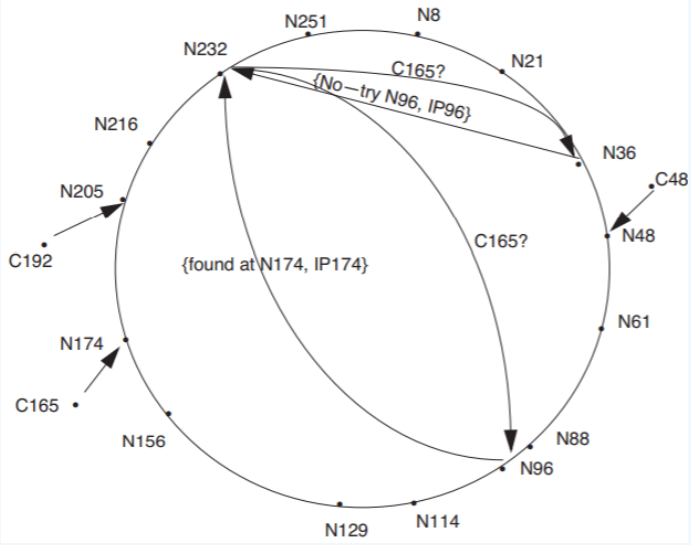


Figure $PS.7.2$: Sequence of RPCs and replies required for computer N232 to find the immediate successor of the content item with ID C165.

** Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications, *Proceedings of the ACM SIGCOMM '01 Conference*, August 2001.

Q7.1 While testing SURETHING, Alyssa notices that when the Internet attempts to deliver the RPC packets, they don't always arrive at their destination. Which of the following reasons might prevent a packet from arriving at its destination?

A. A router discards the packet.
B. The packet is corrupted in transit.
C. The payload of an RPC reply contains the wrong IP address.
D. The packet gets into a forwarding loop in the network.

For the next two questions, remember that computers don't fail and that all tables are initialized correctly.

Q7.2 Assume that `c` is an id whose immediate successor is not present in `successors`, and n is the number of computers in the system. In the *best case*, how many remote lookups are needed before `GET_LOCATION (c)` returns?

A. 0
B. 1
C. 2
D. $O(\log n)$
E. $O(n)$
F. $O(n^2)$

Q7.3 Assume that `c` is an id whose immediate successor is not present in `successors`, and n is the number of computers in the system. In the *worst case*, how many remote lookups are needed before `GET_LOCATION (c)` returns?

A. 0
B. 1
C. 2
D. $O(\log n)$
E. $O(n)$
F. $O(n^2)$

Building the Content Layer. Having built the forwarding and routing layer, Alyssa turns to building a content layer. At a high level, the system supports storing data that has an ID associated with it. Specifically, it supports two operations:

1. `PUT (c, content)` stores content in the system with ID `c` .
2. `GET (c)` returns the content that was stored with ID `c` .

Content IDs are integers that can be used as arguments to `GET_LOCATION` . (In practice, one can ensure that IDs are integers by using a hash function that maps human-readable names to integers.)

Alyssa implements the content layer by using the forwarding and routing layer to choose which computers to use to store the content. For reliability, she decides to store every piece of content on two computers: the two immediate successors of the content's ID. She modifies `GET_LOCATION` to return both successors, calling the new version `GET_2LOCATIONS` . For example, in Figure $PS.7.1$, if `GET_2LOCATIONS` is asked to find the content item with ID C165, it returns the IP addresses of computers N174 and N205.

Once the correct computers are located using the forwarding and routing layer, Alyssa's implementation sends a `PUT` RPC to each of these computers to store the content in a file in both places. (If one of the computers is the local computer, it performs that store with a local call to `PUT` rather than an RPC.)

To retrieve the content associated with a given ID, if either ID returned by `GET_2LOCATIONS` is local it reads the file with a local `GET` . If not, it sends a `GET` RPC to the computer with the first ID, requesting that the computer load the appropriate file from disk, if it exists, and return its contents. If that RPC fails for some reason, it tries the second ID.

Q7.4 Which of the following are the end-to-end properties of the content layer? Assume that there are no failures of computers or disks while the system is running, that all tables are initialized correctly, and that the network delivers every message correctly.

A. `GET (c)` always returns the same content that was stored with ID `c` .
B. `PUT (c, content)` stores the content at the two immediate successors of `c` .
C. `GET` returns the content from the immediate successor of `c` .
D. If the content has been stored on some computer, `GET` will find it.

Q7.5 Now, suppose that individual computers may crash but the network continues to deliver every message correctly. Which of the following properties of the content layer are true?

A. One of the computers returned by `GET_2LOCATIONS` might not answer the `GET` or `PUT` call.
B. `PUT` will sometimes be unable to store the content at the content's two immediate successors.
C. `GET` will successfully return the requested content, assuming it was stored previously.
D. If one of the two computers on which `PUT` stored the content has not crashed when `GET` runs, `GET` will succeed in retrieving the content.

Improving Forwarding Performance. We now return to the forwarding and routing layer and ignore the content layer.

Alyssa isn't happy with the performance of the system, in particular `GET_LOCATION` . Her friend Lem E. Tweakit suggests the following change: each computer maintains a `node_cache` , which contains information about the IDs and IP addresses of computers in the system. The `node_cache` table initially contains information about the computers in `successors` .

For example, initially the `node_cache` at computer N232 contains entries for computers N251, N8, N21, N36. But after computer N232 communicates with computer N36 and learns the ID and IP address of computer N96, N232's `node_cache` would contain entries for computers N251, N8, N21, N36, and N96.

Q7.6 Assume that `c` is a content ID whose immediate successor is not one of the computers listed in `successors` , and n is the number of computers in the system. In the *best case*, how many remote lookups are needed before `GET_LOCATION (c)` returns?

A. 0
B. 1
C. 2
D. $O(\log n)$
E. $O(n)$
F. $O(n^2)$

Problem Set 8: Sliding Window*

* Credit for developing this problem set goes to Dina Katabi.

2008–2–7

(Chapter 1)

Consider the sliding window algorithm described in Chapter 1. Assume the topology in the figure below, where all links are duplex and have the same capacity and delay in both directions. The capacities of the two links on the left are very large and can be assumed infinite, while their propagation delays are negligible and can be assumed zero. Both sources send to the same destination node.

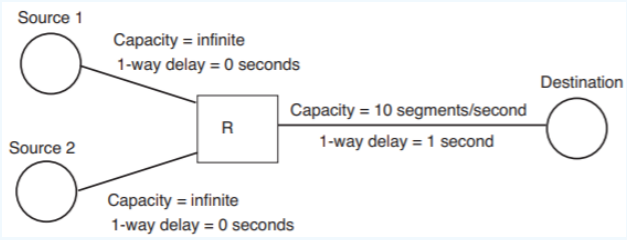


Figure $PS.8.1$: Topology of the sliding window algorithm used for Problem Set 8.

Q8.1 Assume the window size is fixed and only Source 1 is active. (Source 2 does not send any traffic.) What is the smallest sliding window that allows Source 1 to achieve the maximum throughput?

Source 1 does not know the bottleneck capacity and hence cannot compute the smallest window size that allows it to achieve the maximum throughput. Ben has an idea to allow Source 1 to compute the bottleneck capacity. Source 1 transmits two data segments back-to-back, i.e., as fast as possible. The destination sends an acknowledgment for each data segment immediately.

Q8.2 Assume that acknowledgements (acks) are significantly smaller than data segments, all data segments are the same size, all acks are the same size, and only Source 1 has any traffic to transmit. In this case, which option is the best way for Source 1 to compute the bottleneck capacity?

A. Divide the size of a data segment by the interarrival time of two consecutive acks.
B. Divide the size of an ack by the interarrival time of two acks.
C. Sum the size of a data segment with an ack segment and divide the sum by the ack interarrival time.

Now assume both Source 1 and Source 2 are active. Router R uses a large queue with space for about 10 times the size of the sliding window of question 8.1. If a data segment arrives at the router when the buffer is full, R discards that segment.

Source 2 uses standard TCP congestion control to control its window size. Source 1 also uses standard TCP, but hacks its congestion control algorithm to always use a fixed-size window, set to the size calculated in question 8.1.

Q8.3 Which of the following is true?

A. Source 1 will have a higher average throughput than Source 2.
B. Source 2 will have a higher average throughput than Source 1.
C. Both sources get the same average throughput.

Problem Set 9.1: Geographic Routing *

* Credit for developing this problem set goes to Dina Katabi.

2008–2–3

(Chapter 1)

Ben Bitdiddle is excited about a novel routing protocol that he came up with. Ben argues that since Global Positioning System (GPS) receivers are getting very cheap, one can equip every router with a GPS receiver so that the router can know its location and route packets based on location information.

Assume that all nodes in a network are in the same plane and nodes never move. Each node is identified by a tuple (x, y), where x and y are its GPS-derived coordinates, and no two nodes have the same coordinates. Each node is joined by links to its neighbors, forming a connected network graph. A node informs its neighbors of its coordinates when it joins the network and whenever it recovers after a failure.

When a source sends a packet, in place of a destination IP address, it puts the destination coordinates in the packet header. (A sender can learn the coordinates of its destination by asking Ben's modified DNS, which he calls the Domain Name Location Service.) When a router

wants to forward a packet, it checks whether any of its neighbors are closer to the destination in Euclidean distance than itself. If none of its neighbors is closer, the router drops the packet. Otherwise the router forwards the packet to the neighbor closest to the destination. Forwarding of a packet stops when that packet either reaches a node that has the destination coordinates or is dropped.

Q9.1 Which of these statements are true about Ben's geographic routing algorithm?

A. If there are no failures, and no nodes join the network while packets are en route, no packet will experience a routing loop.
B. If nodes fail while packets are en route, a packet may experience a routing loop.
C. If nodes join the network while packets are en route, a packet may experience a routing loop.

Suppose that that there are no failures of either links or nodes, and also that no node joins the network.

Q9.2 Can Ben's algorithm deliver packets between any source-destination pair in a network? If yes, explain. If no, draw a counterexample in the grid below, placing nodes on grid intersections and making sure that links connect all nodes.

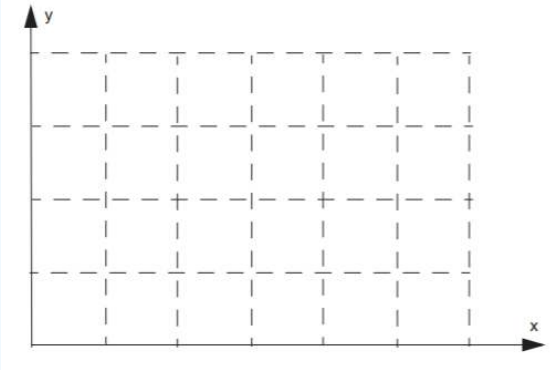


Figure $PS.9.1$: Answer grid for Q9.2.

Q9.3 For all packets that Ben's algorithm delivers to their corresponding destinations, does Ben's algorithm use the same route as the path vector algorithm described in Section 1.5.2? If your answer is yes, then explain it. If your answer is no, then draw a counter example.



Figure $PS.9.2$: Answer grid for Q9.3.

Problem Set 10: Carl's Satellite*

* Credit for developing this problem set goes to Robert T. Morris.

2001–3–6…13

(Chapter 2)

Carl Coder decides to quit his job at an e-commerce start-up and go to graduate school. He's curious about the possibility of broadcasting data files through satellites, and decides to build a prototype that does so.

Carl decides to start simple. He launches a satellite into a geosynchronous orbit, so that the satellite is visible from all points in the United States. The satellite listens for incoming bits on a radio up-channel, and instantly retransmits each bit on a separate down-channel. Carl builds two ground stations, a sender and a receiver. The sending station sends on a radio to the satellite's up-channel; the receiving station listens to the satellite's down-channel.

Carl's test application is to send Associated Press (AP) news stories from a sending station, through the satellite, to a receiving station; the receiving station prints each story on a printer. AP stories always happen to consist of 1024 characters. Carl writes the code below to run on computers at the sending and receiving stations (Scheme 1).

```
procedure SEND_BUFFER (byte buffer[1024])
    for i from 0 to 1024 do
        SEND_8_BITS (buffer[i])

procedure RECEIVER ()
    byte buffer[1024]
    do forever
        ok ← RECV_BUFFER (buffer)
        if ok = TRUE then
            print buffer on a printer

procedure RECV_BUFFER (byte buffer[1024])
    for i from 0 to 1024 do
        buffer[i] ← RECV_8_BITS ()
    return (TRUE)
```

The receiving radio hardware receives a bit if and only if the sending radio sends a bit. This means the receiver receives the same number of bits that the sender sent. However, the receiving radio may receive a bit incorrectly, due to interference from sources near the receiver. The radio doesn't detect such errors; it just hands the incorrect bit to the computer at the receiving ground station with no warning. These incorrect bits are the only potential faults in the system, other than (perhaps) flaws in Carl's design. If the computer tells the printer to print an unprintable character, the printer prints a question mark instead.

After running the system for a while, Carl observes that it doesn't always work correctly. He compares the stories that are sent by the sender with the stories printed at the receiver.

Q10.1 What kind of errors might Carl see at the receiver's printer?

A. Sometimes one or more characters in a printed story are incorrect.
B. Sometimes a story is repeated.
C. Sometimes stories are printed out of order.
D. Sometimes a story is entirely missing.

Q10.2 The receiver radio manufacturer claims that the probability of receiving a bit incorrectly is one in 10^5, and that such errors are independent. If these claims are true, what fraction of stories is likely to be printed correctly?

Carl wants to make his system more reliable. He modifies his sender code to calculate the sum of the bytes in each story and append the low 8 bits of that sum to the story. He modifies the receiver to check whether the low 8 bits of the sum of the received bytes match the received sum. His new code (Scheme 2) is below.

```
procedure SEND_BUFFER (byte buffer[1024])
    byte sum ← 0                 // byte is an eight-bit unsigned integer
    for i from 0 to 1024 do
        SEND_8_BITS (buffer[i])
        sum ← sum + buffer[i]
    SEND_8_BITS (sum)

procedure RECV_BUFFER (byte buffer[1024])
    byte sum1, sum2
    sum1 ← 0
    for i from 0 to 1024 do
        buffer[i] ← RECV_8_BITS()
        sum1 ← sum1 + buffer[i]
    sum2 ← RECV_8_BITS()
    if sum1 = sum2 then return TRUE
    else return FALSE
```

Q10.3 What kind of errors might Carl see at the receiver's printer with this new system?

A. Sometimes one or more characters in a printed story are incorrect.
B. Sometimes a story is repeated.

C. Sometimes stories are printed out of order.
D. Sometimes a story is entirely missing.

Q10.4 Suppose the sender sends 10,000 stories. Which scheme is likely to print a larger number of these 10,000 stories correctly?

Carl decides his new system is good enough to test on a larger scale, and sets up 3 new receiving stations scattered around the country, for a total of 4. All of the stations can hear the AP stories from his satellite. Users at each of the receivers call him up periodically with a list of articles that don't appear in their printer output so Carl can have the system re-send them. Users can recognize which stories don't appear because the Associated Press includes a number in each story, and assigns numbers sequentially to successive stories.

Q10.5 Carl visits the sites after the system has been in operation for a week, and looks at the accumulated printouts (in the order they were printed) at each site. Carl notes that the first and last stories were received by all sites and all sites have received all retransmissions they have requested. What kind of errors might he see in these printouts?

A. Sometimes one or more characters in a printed story are incorrect.
B. Sometimes a story is repeated.
C. Sometimes stories are printed out of order.
D. Sometimes a story is entirely missing.

Q10.6 Suppose Carl sends out four AP stories. Site 1 detects an error in just the first story; site 2 detects an error in just the second story; site 3 detects an error in just the third story; and site 4 receives all 4 stories correctly. How many stories will Carl have to re-send? Assume any re-sent stories are received and printed correctly.

After hearing about RAID, Carl realizes he could improve his system even more. He modifies his sender to send an extra "parity story" after every four AP stories; the parity story consists of the exclusive or of the previous four real stories. If one of the four stories is damaged, Carl's new receiver reconstructs it as the exclusive or of the parity and the other three stories.

His new pseudocode uses the checksumming versions of `SEND_BUFFER ()` and `RECV_BUFFER ()` to detect damaged stories.

```
procedure SENDER ()
    byte buffer[1024]
    byte parity[1024]
    do forever
        clear parity[] to all zeroes
        for i from 0 to 4 do
            read next AP story into buffer
            SEND_BUFFER (buffer)
            parity ← parity ⊕ buffer // XOR's the whole buffer
        SEND_BUFFER (parity)

procedure RECEIVER ()
    byte buffers[5][1024]       // holds the 4 stories and the parity
    boolean ok[5]               // records which ones have been received correctly
    integer n                   // count buffers received correctly
    do forever
        n ← 0
        for i from 0 to 5 do
            ok[i] ← RECV_BUFFER (buffers[i])
            if ok[i] then n ← n + 1
        for i from 0 to 4 do
            if ok[i] then print buffers[i]                          // buffers[i] is correct
            else if n = 4 then
                clear buffers[i] to all zeroes
                for j from 0 to 5 do
                    if i ≠ j then
                        buffers[i] ← buffers[i] ⊕ buffers[j]     // XOR two buffers
                print buffers[i]                                // don't print if you cannot
reconstruct
```

Q10.7 Suppose Carl sends out four AP stories with his new system, followed by a parity story. Site 1 is just missing the first story; site 2 is just missing the second story; site 3 is just missing the third story; and site 4 receives all stories correctly. How many stories will Carl have to re-send? Assume any re-sent stories are received and printed correctly.

Q10.8 Carrie, Carl's younger sister, points out that Carl is using two forms of redundancy: the parity story and the checksum for each story. Carrie claims that Carl could do just as well with the parity alone, and that the checksum serves no useful function. Is Carrie right? Why or why not?

6.2.1: Problem Sets 1-10 is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

6.2.2: Problem Sets 11-20

Problem Set 11: RaidCo*

* Credit for developing this problem set goes to Stephen A. Ward.

2007–2–11

(Chapter 2)

RaidCo is a company that makes pin-compatible hard disk replacements using tiny, chip-sized hard disks ("microdrives") that have become available cheaply. Each RaidCo product behaves like a hard disk, supporting the operations:

- `error ← GET (nblocks, starting_block_number, buffer_address)`
- `error ← PUT (nblocks, starting_block_number, buffer_address)`

to get or put an integral number of consecutive blocks from or to the disk array. Each operation returns a status value indicating whether an error has occurred.

The models are as follows:

- R0: sector-level striping across all twelve microdrives, no redundancy/error correction
- R1: six pairs of two mirrored microdrives, no striping
- R2: 12-microdrive RAID 2, using bit-level striping, error detection, and error correction); microdrive's internal sector-level error detection is disabled.
- R3: 12-microdrive RAID 3, using sector-level striping and error correction.
- R4: 12-microdrive RAID 4, no striping, dedicated parity disk
- R5: 12-microdrive RAID 5, no striping, distributed parity

The microdrives each conform to the same read/write API sketched above, each microdrive providing 100,000 sectors of 1,000 bytes each, and offering a uniform 10 millisecond seek time and a read/write bandwidth of 100 megabytes per second; thus the entire 100 megabytes of data on a microdrive can be fetched using a single GET operation in one second. The RaidCo products do no caching or buffering: each GET or PUT involves actual data transfer to or from the involved microdrives. Since the microdrives have uniform seek time, the RaidCo products do not need, and do not use, any seek optimizations.

> **Q11.1** As good as the students were at programming, they unfortunately left the documentation unfinished. Your job is to complete the following table, showing certain specifications for each model drive (i.e., the size and performance parameters of the API supported by each RAID system). Entries assume error-free operation, and ignore transfer times that are small compared to seek times encountered.

	R0	R1	R2	R3	R4	R5
Block size (kilobytes) exposed to `GET/PUT`	1 kB	1 kB		11 kB		1 kB
Capacity, in blocks						1,100,000
Max time for a single 100-megabyte `GET` (seconds)	1/12 s				1 s	1 s
Time for a 1-block `PUT` (milliseconds)	10 ms	10 ms	10 ms			20 ms

	R0	R1	R2	R3	R4	R5
Typical number of microdrives involved in a 1-block `GET`	1					1
Typical number of microdrives involved in a 2-block `GET`		2			1	1
Typical number of microdrives involved in a 1-block `PUT`		2				

Add exercises text here.

Problem Set 12: ColdFusion *

* Credit for developing this problem set goes to Hari Balakrishnan.

2000–3–8…15

(Chapter 2, with a bit of Chapter 3)

Alyssa P. Hacker and Ben Bitdiddle are designing a hot new system, called ColdFusion, whose goal is to allow users to back up their storage systems with copies stored in a distributed network of ColdFusion servers. Users interact with ColdFusion using `PUT` and `GET` operations.

- `PUT (data, fid)` takes a data buffer and reliably stores it under a unique identifier `fid` , a positive integer, on some subset of the servers. It returns `SUCCESS` if it was successful in storing it on all the machines in the chosen subset, and `FAILURE` otherwise.
- `GET (fid)` returns the contents of the most recent successful `PUT` to the system for the file identified by `fid` .

Because high availability is a key competitive advantage, Alyssa decides to replicate user data on more than one server machine. But rather than replicate each file on every server, she decides to be clever and use only a subset of the servers for each file. Thus, the `PUT` of a file stores it on some number (A) of the servers, invoking a `SERVER_PUT` operation on each server. `SERVER_PUT` is atomic and is implemented by each server.

If `PUT` is unable to successfully store the file on A servers, it returns `FAILURE` .

When a client does a `GET` of the file, the `GET` software attempts to retrieve the file from some subset of the servers and picks the version using an election algorithm. It chooses B servers to read data from, using an atomic `SERVER_GET` operation implemented by each server, following which it calls `PICK_MAJORITY ()` . `PICK_MAJORITY` returns valid data corresponding to a version that is shared by *more than 50%* of the B copies retrieved, and `NULL` otherwise. Even though the client may not know which specific servers hold the current copy, the number of servers (A) in `PUT` and the number (B) in `GET` are chosen so that if a client `GET` succeeds, it is certain to have received the most recent copy.

Alyssa and Ben write the following code for `PUT` and `GET` . There are S servers in all, and $S > 2$. The particular ordering of the servers in the code below may be different at different clients, but all clients have the same list. They hire you as a consultant to help them figure out the missing parameters (A and B) and analyze the system for correctness.

```
// Internet addresses of the S servers, S > 2
ip_address server[S] // each client may have a different ordering
```

```
procedure PUT (byte data[], integer fid)
    integer ntried, nputs ← 0     // # of servers tried and # successfully put
     for ntried from 0 to S do      // Put “data” into a file identified by fid at
server[ntried]
        status ← SERVER_PUT (data, fid, server[ntried])
        if status = success then nputs ← nputs + 1
        if nputs ≥ A then // yes! have SERVER_PUT () to A servers!
            return SUCCESS;
    return FAILURE // found < A servers to SERVER_PUT() to

procedure GET (integer fid, byte data[])
     integer ntried, ngets ← 0           // # times tried and # times server_get
returned success
    byte files[S][MAX_FILE_LENGTH]    // array of files; an entry is a copy from a
server
    integer index
    byte data[]
    for ntried from 0 to S do        // Get file fid into buffer files[ntried] from
server[ntried]
        status ← SERVER_GET (fid, files[ntried], server[ntried])
        if status = success then ngets ← ngets + 1
        if ngets ≥ B then            // yes! have gotten data from B servers
             // PICK_MAJORITY () takes the array of files and magically knows which
ones are
              // valid. It scans the ngets valid ones and returns an index in the
files[] array
                  // for one of the good copies, which corresponds to a version
returned by more
            // than 50% of the servers. Otherwise, it returns –1. If ngets = 1,
            // PICK_MAJORITY () simply returns an index to that version.
        index ← PICK_MAJORITY (files, ngets);
        if index ≠ –1 then
            COPY (data, files[index])     // copy into data buffer
            return SUCCESS
        else return FAILURE
    return FAILURE                    // didn’t find B servers to SERVER_GET from
```

For questions 12.1 through 12.4, assume that operations execute serially (i.e., there is no concurrency). Assume also that the end-to-end protocol correctly handles all packet losses and delivers messages in to a recipient in the same order that the sender dispatched them. In other words, no operations are prevented from completing because of lost or reordered packets. However, servers may crash and subsequently recover.

Q12.1 Which reliability technique is the best description of the one being attempted by Alyssa and Ben?

A. Fail-safe design.
B. N-modular redundancy.
C. Pair-and-compare.
D. Temporal redundancy.

Q12.2 Which of the following combinations of A and B in the code above ensures that `GET` returns the results of the last successful `PUT` , as long as no servers fail? (Here, $\lfloor x \rfloor$ is the largest integer $\leq x$, and $\lceil y \rceil$ is the smallest integer $\geq y$. Thus $\lfloor 2.3 \rfloor = 2$, and $\lceil 2.3 \rceil = 3$. Remember also that $S > 2$.)

A. $A = 1$ $B = S$
B. $A = \lceil S/3 \rceil$ $B = S$
C. $A = \lceil S/2 \rceil$ $B = S$

D. $A = \lceil (3S)/4 \rceil \quad B = \lfloor S+2 \rfloor + 1$
E. $A = S \quad B = 1$

Q12.3 Suppose that the number of servers S is an odd number larger than 2, and that the number of servers used for PUT is $A = S/2$. If only PUT and no GET operations are done, how does the mean time to failure (MTTF) of the PUT operation change as S increases? The PUT operation fails if the return value from PUT is FAILURE. Assume that the process that causes servers to fail is memoryless, and that no repairs are done.

A. As S increases, there is more redundancy. So, the MTTF increases.
B. As S increases, one still needs about one-half of the servers to be accessible for a successful PUT. So, the MTTF does not change with S.
C. As S increases the MTTF decreases even though we have more servers in the system.
D. The MTTF is not a monotonic function of S; it first decreases and then increases.

Q12.4 Which of the following is true of ColdFusion's PUT and GET operations, for choices of A and B that guarantee that GET successfully returns the data from the last successful PUT when no servers fail?

A. A PUT that fails because some server was unavailable to it, done after a successful PUT, may cause subsequent GET attempts to fail, even if B servers are available.
B. A failed PUT attempt done after a successful PUT cannot cause subsequent GET attempts to fail if B servers are available.
C. A failed PUT attempt done after a successful PUT always causes subsequent GET attempts to fail, even if B servers are available.
D. None of the above.

ColdFusion unveils their system for use on the Internet with S = 15 servers, using $A = 2S/3$ and $B = 1 + 2S/3$. However, they find that the specifications are not always met—several times, GET does not return the data from the last PUT that returned SUCCESS.

In questions 12.5 through 12.8, assume that there may be concurrent operations.

Q12.5 Under which of these scenarios does ColdFusion always meet its specification (i.e., GET returns SUCCESS and the data corresponding to the last successful PUT)?

A. There is no scenario under which ColdFusion meets its specification for this choice of A and B.
B. When a user PUT's data to a file with some *fid*, and at about the same time someone else PUT's different data to the same fid.
C. When a user PUT's a file successfully from her computer at home, drives to work and attempts to GET the file an hour later. In the meantime, no one performs any PUT operations to the same file, but three of the servers crash and are unavailable when she does her GET.
D. When the PUT of a file succeeds at some point in time, but some subsequent PUT's fail because some servers are unavailable, and then a GET is done to that file, which returns SUCCESS.

You tell Ben to pay attention to multisite coordination, and he implements his version of the two-phase commit protocol. Here, each server maintains a log containing READY (a new record he has invented), ABORT, and COMMIT records. The server always returns to a client the version of a file that last underwent COMMIT.

In Ben's protocol, when the client PUT's a file, the server returns SUCCESS or FAILURE as before. If it returns SUCCESS, the server appends a READY entry for this fid in its log. If the client sees that all the servers it asked returns SUCCESS, it sends a message asking them all to COMMIT. When a server receives this message, it writes a COMMIT entry in its log together with the file's *fid*. On the other hand, if even one of the servers returns FAILURE, the client sends a message to all the servers asking them to abort the operation, and each server writes an ABORT entry in its log. Finally, if a server gets a server put request for some fid that is in the READY state, it returns FAILURE to the requesting client.

Q12.6 Under which of these scenarios does ColdFusion always meet its specification (i.e., GET returns SUCCESS and the data corresponding to the last successful PUT)?

A. There is no scenario under which ColdFusion meets its specification for this choice of A and B.
B. When a user PUT's data to a file with some *fid*, and at about the same time someone else PUT's different data to the same fid.

C. When a user `PUT` 's a file successfully from her computer at home, drives to work and attempts to `GET` the file an hour later. In the meantime, no one performs any `PUT` operations to the same file, but three of the servers crash and are unavailable when she does her `GET` .

D. When the `PUT` of a file succeeds at some point in time and the corresponding `COMMIT` messages have reached the servers, but some subsequent `PUT` s fail because some servers are unavailable, and then a `GET` is done to that file, which returns `SUCCESS` .

Q12.7 When a server crashes and recovers, the original clients that initiated `PUT` s may be unreachable. This makes it hard for a server to know the status of its `READY` actions, since it cannot ask the clients that originated them. Assuming that no more than one server is unavailable at any time in the system, which of the following strategies allows a server to correctly learn the status of a past `READY` action when it recovers from a crash?

A. Contact any server that is up and running and call `SERVER_GET` with the file's *`fid`* ; if the server responds, change `READY` to `COMMIT` in the log.

B. Ask all the other servers that are up and running using server `GET()` with the file's *`fid`* ; if more than 50% of the other servers respond with identical data, just change `READY` to `COMMIT` .

C. Pretend to be a client and invoke `GET` with the file's fid; if `GET` is successful and the data returned is the same as what is at this server, just change `READY` to `COMMIT` .

D. None of the above.

To accommodate the possibility of users operating on entire directories at once, ColdFusion adds a two-phase locking protocol on individual files within a directory. Alyssa and Ben find that although this sometimes works, deadlocks do occur when a `GET` owns some locks that a `PUT` needs, and vice versa.

Q12.8 Ben analyzes the problem and comes up with several "solutions" (as usual). Which of his proposals will actually work, *always* preventing deadlocks from happening?

A. Ensure that the actions grab locks for individual files in increasing order of the *`fid`* of the file.

B. Ensure that no two actions grab locks for individual files in the same order.

C. Assign an incrementing timestamp to each action when it starts. If action A_i needs a lock owned by action A_j with a larger timestamp, abort action A_i and continue.

D. Assign an incrementing timestamp to each action when it starts. If action A_i needs a lock owned by action A_j with a smaller timestamp, abort action A_i and continue.

Problem Set 13: PigeonExpress.com

1999–3–5…14

(Chapter 3, based on Chapter 1)

After selling PigeonExpress!.com and taking a trip around the world, Ben Bitdiddle is planning his next start-up, AtomicPigeon!.com. AtomicPigeon improves over PigeonExpress by offering an atomic data delivery system.

Recall from problem set 2 that when sending a pigeon, Ben's software prints out a little header and writes a CD, both of which are given to the pigeon. The header contains the GPS coordinates of the sender and receiver, a type (`REQUEST` or `ACKNOWLEDGMENT`), and a sequence number:

```
structure header
    GPS source
    GPS destination
    integer type
    integer sequence_no
```

Ben starts with the code for the simple end-to-end protocol (`BEEP`) for PigeonExpress!.com. He makes a number of modifications to the sending and receiving code.

At the sender, Ben simplifies the code. The `BEEP` protocol transfers only a single CD:

```
shared next_sequence initially 0     // a globally shared sequence number.
```

```
procedure BEEP (target, CD[])          // send 1 CD to target
    header h                           // h is an instance of header.
    h.source ← MY_GPS                  // set source to my GPS coordinates
    h.destination ← target             // set destination
    h.type ← REQUEST                   // this is a request message
    h.sequence_no ← next_sequence      // set seq number
    // loop until we receive the corresponding ack, retransmitting if needed
    while h.sequence_no = next_sequence do
        send pigeon with h, CD         // transmit
        wait 2,000 seconds
```

As before, pending and incoming acknowledgments are processed only when the sender is waiting:

```
procedure PROCESS_ACK (h)                 // process acknowledgment
    if h.sequence_no = sequence then      // ack for current outstanding CD?
        next_sequence ← next_sequence + 1
```

Ben makes a small change to the code running on the receiving computer. He adds a variable `expected_sequence` at the receiver, which is used by `PROCESS_REQUEST` to filter duplicates:

```
integer expected_sequence initially 0             // duplicate filter.

procedure PROCESS_REQUEST (h, CD)                 // process request
    if h.sequence_no = expected_sequence then     // the expected seq #?
        PROCESS (CD) // yes, process data
        expected_sequence ← expected_sequence + 1 // increase expectation
    h.destination ← h.source                       // send to where the pigeon came
from
    h.source ← MY_GPS
    h.sequence_no ← h.sequence_no                 // unchanged
    h.type ← ACKNOWLEDGMENT;
    send pigeon with h                            // send an acknowledgment back
```

The assumptions for the pigeon network are the same as in problem set 2:

- Some pigeons might get lost, but, if they arrive, they deliver data correctly (uncorrupted)
- The network has one sender and one receiver
- The sender and the receiver are single-threaded

Q13.1 Assume the sender and receiver do not fail (i.e., the only failures are that some pigeons may get lost). Does `PROCESS` in `PROCESS_REQUEST` process the value of CD exactly once?

A. Yes, since `next_sequence` is a nonce and the receiver processes data only when it sees a new nonce.
B. No, since `next_sequence` and `expected_sequence` may get out of sync because the receiver acknowledges requests even when it skips processing.
C. No, since if the acknowledgment isn't received within 2,000 seconds, the sender will send the same data again.
D. Yes, since pigeons with the same data are never retransmitted.

Ben's new goal is to provide atomicity, even in the presence of sender or receiver failures. The reason Ben is interested in providing atomicity is that he wants to use the pigeon network to provide P-commerce (something similar to E-commerce). He would like to write applications of the form:

```
procedure TRANSFER (amount, destination)
    WRITE (amount, CD) // write amount on a CD
    BEEP (destination, CD) // send amount
```

The amount always fits on a single CD.

If the sender or receiver fails, the failure is fail-fast. For now, let's assume that if the sender or receiver fails, it just stops and does not reboot; later, we will relax this constraint.

Q13.2 Given the current implementation of the `BEEP` protocol and assuming that only the sender may fail, what could happen during, say, the 100th call to `TRANSFER` ?

A. That `TRANSFER` might never succeed.
B. That `TRANSFER` might succeed.
C. `PROCESS` in `PROCESS_REQUEST` might process *`amount`* more than once.
D. `PROCESS` in `PROCESS_REQUEST` might process *`amount`* exactly once.

Ben's goal is to make transfer always succeed by allowing the sender to reboot and finish failed transfers. That is, after the sender fails, it clears volatile memory (including the nonce counter) and restarts the application. The application starts by running a recovery procedure, named `RECOVER_SENDER` , which retries a failed transfer if any exist.

To allow for restartable transfers, Ben supplies the sender and the receiver with durable storage that never fails. On the durable storage, Ben stores a log, in which each entry has the following form:

```
structure log_entry
    integer type         // STARTED or COMMITTED
    integer sequence_no  // a sequence number
```

The main objective of the sender's log is to allow `RECOVER_SENDER` to restore the value of `next_sequence` and to allow the application to restart an unfinished transfer, if any. Ben edits `TRANSFER` to use the log:

```
1 procedure TRANSFER (amount, destination)
2     WRITE (amount, CD)                    // write amount on a CD
3     ADD_LOG (STARTED, next_sequence)      // append STARTED record
4       BEEP (destination, CD)                    // send amount (BEEP increases
next_sequence)
5     ADD_LOG (COMMITTED, next_sequence – 1) // append COMMITTED record
```

`ADD_LOG` atomically appends a record to the log on durable storage. If `ADD_LOG` returns, the entry has been appended. Logs contain sufficient space for new records and they don't have to be garbage-collected.

Q13.3 Identify the line in this new version of `TRANSFER` that is the commit point.

Q13.4 How can the sender discover that a failure caused the transfer not to complete?

A. The log contains a `STARTED` record with no corresponding `COMMITTED` record.
B. The log contains a `STARTED` record with a corresponding `ABORTED` record.
C. The log contains a `STARTED` record with a corresponding `COMMITTED` record.
D. The log contains a `COMMITTED` record with no corresponding `STARTED` record.

Ben tries to write `RECOVER_SENDER` to recover *`next_sequence`* , but his editor crashes before committing the final editing and the expression in the `if` statement is missing, as indicated by a `?` in the code below. Your job is to edit the code such that the correct expression is evaluated.

```
procedure RECOVER_SENDER ()
    next_sequence ← 0
    starting at end of log…
    for each entry in log do
        if ( ? ) then          // What goes here? (See question 13.6)
            next_sequence ← (sequence_no of entry) + 1
            break              // terminate scan of log
```

Q13.5 After you edit `RECOVER_SENDER` for Ben, which of the following sequences could appear in the log? (The log records are represented as `<type sequence_no>` .

A. `..., <STARTED 1>, <COMMITTED 1>, <STARTED 2>, <COMMITTED 2>`
B. `..., <STARTED 1>, <STARTED 1>, <COMMITTED 1>`
C. `..., <STARTED 1>, <STARTED 1>, <STARTED 2>, <COMMITTED 1>, <STARTED 2>`
D. `..., <STARTED 1>, <COMMITTED 1>, <COMMITTED 1>`

Q13.6 What expression should replace the `?` in the `RECOVER_SENDER` code above?

A. *entry.type* = COMMITTED
B. *entry.type* = STARTED
C. *entry.type* = ABORTED
D. FALSE

Q13.7 Given the current implementation of the BEEP protocol what could happen, say, during the 100th call to TRANSFER ? (Remember only the sending computer may fail.)

A. If the sending computer keeps failing during recovery, that TRANSFER might never succeed.
B. That TRANSFER might succeed.
C. PROCESS in PROCESS_REQUEST might process *amount* more than once.
D. PROCESS in PROCESS_REQUEST might process *amount* exactly once.

Ben's next goal is to make PROCESS_REQUEST all-or-nothing. In the following questions, assume that whenever the receiving computer fails, it reboots, calls RECOVER_RECEIVER , and after RECOVER_RECEIVER is finished, it waits for messages and calls PROCESS_REQUEST on each message.

To make *expected_sequence* all-or-nothing, Ben tries to change the receiver in a way similar to the change he made to the sender. Again, his editor didn't commit all the changes in time. The missing code is marked by ? and # . The missing expression marked by ? evaluates the same expression as did the ? in RECOVER_SENDER . The new missing expressions are marked by # in PROCESS_REQUEST :

```
procedure RECOVER_RECEIVER ()
    expected_sequence ← 0
    starting at end of log…
    for each entry in log do
        if ( ? ) then                          // The expression of question 13.6
            expected_sequence ← sequence_no of entry + 1
            break                              // terminate scan of log
```

```
1 procedure PROCESS_REQUEST (h, CD)
2     if h.sequence_no = expected_sequence then         // the expected seq #?
3         ADD_LOG (#, #)                     // See question 13.8.
4         PROCESS (CD)                       // yes, process data
5         expected_sequence ← expected_sequence + 1     // increase expectation
6         ADD_LOG (#, #) // See question 13.8.
7     h.destination ← source of h            // send to where the pigeon came from
8     source of h.source ← MY_GPS
9     h.sequence_no ← h.sequence_no          // unchanged
10    h.type ← ACKNOWLEDGMENT
11    send pigeon with h                     // send an acknowledgment back
```

As you can see from the code, Ben chose not to implement a write-ahead protocol because PROCESS is implemented by a third party, for example, a bank: PROCESS might be a call into the bank's transaction database system.

Q13.8 Complete the ADD_LOG calls on the lines 3 and 6 in PROCESS_REQUEST such that *expected_sequence* will be all-or-nothing.

```
3    ADD_LOG (        ,            )
6    ADD_LOG (        ,            )
```

Q13.9 Can PROCESS in PROCESS_REQUEST be called multiple times for a particular call to TRANSFER ?

A. No, because *expected_sequence* is recovered and *h.sequence_no* is checked against it.
B. Yes, because failed transfers will be restarted and result in the acknowledgment being retransmitted.
C. Yes, because after 2,000 seconds a request will be retransmitted.
D. Yes, because the receiver may fail after PROCESS , but before it commits.

Q13.10 How should PROCESS , called by PROCESS_REQUEST , be implemented to guarantee exactly-once semantics for transfers? (Remember that the sender is persistent.)

A. As a normal procedure call;
B. As a remote procedure call;
C. As a nested transaction;
D. As a top-level transaction.

Problem Set 14: Sick Transit *

* Credit for developing this problem set goes to Stephen A. Ward.

1997–3–6…15

(Chapter 3)

Gloria Mundi, who stopped reading the text before getting to Chapter 3, is undertaking to resurrect the failed London Ambulance Service as a new streamlined company called Sick Transit. She has built a new computer she intends to use for processing ST's activities.

A key component in Gloria's machine is a highly reliable sequential-access infinite tape, which she plans to use as an **append-only** log. Records can be appended to the tape, but once written are immutable and durable. Records on the tape can be read any number of times, from front-to-back or from back-to-front. There is no disk in the ST system; the tape is the only non-volatile storage.

Because of the high cost of the infinite tape, Gloria compromised on the quality of more conventional components like RAM and CPU, which fail frequently but fortunately are fail-fast: every error causes an immediate system crash. Gloria plans to ensure that, after a crash, a consistent state can be reconstructed from the log on the infinite tape.

Gloria's code uses transactions, each identified by a unique transaction ID. The visible effect of a completed transaction is confined to changes in global variables whose `WRITE` operations are logged. The log will contain entries recording the following operations:

```
BEGIN (tid)          // start a new transaction, whose unique ID is tid
    COMMIT (tid)     // commit a transaction
    ABORT (tid)      // abort a transaction
    WRITE (tid, variable, old_value, new_value)
                     // write a global variable, specifying previous & new values.
```

To keep the system simple, Gloria plans to use the above forms as the application-code interface, in addition to a `READ (tid, variable)` call which returns the current value of `variable` . Each of the calls will perform the indicated operation and write a log entry as appropriate. Reading an unwritten variable is to return `ZERO` .

Gloria begins by considering the single-threaded case (only one transaction is active at any time). She stores values of global variables in a table in RAM. Gloria is now trying to figure out how to reset variables to committed values following a crash, using the log tape.

Q14.1 In the single-threaded case, what value should variable `v` be restored to following a crash?

A. 37
B. `new_value` from the last logged `WRITE (tid, v, old_value, new_value)` , or `ZERO` if unwritten
C. `new_value` from the last logged `WRITE (tid, v, old_value, new_value)` that is not followed by an `ABORT (tid)` , or `ZERO` if unwritten.
D. new_value from the last logged `WRITE (tid, v, old_value, new_value)` that is followed by a `COMMIT (tid)` , or `ZERO` otherwise.
E. Either `old_value` or `new_value` from the last logged `WRITE (tid, v, old_value, new_value)` , depending on whether that `WRITE` is followed by a `COMMIT` on the same `tid` , or `ZERO` if unwritten.

Gloria now tries running concurrent transactions on her system. Accesses to the log are serialized by the sequential-access tape drive.

Her first trial involves concurrent execution of these two transactions:

```
BEGIN (t1)
     t1x ← READ (x)
     t1y ← READ (y)
     WRITE (t1, x, t1x, t1y + 1)
COMMIT (t1)
```

```
BEGIN (t2)
     t2x ← READ (x)
     t2y ← READ (y)
     WRITE (t2, y, t2y, t2x + 2)
COMMIT (t2)
```

The initial values of x and y are ZERO , as are all uninitialized variables in her system. Here the READ primitive simply returns the most recently written value of a variable from the RAM table, ignoring COMMIT s.

Q14.2 In the absence of locks or other synchronization mechanism, will the result necessarily correspond to some serial execution of the two transactions?

A. Yes.
B. No, since the execution might result in $x = 3$, $y = 3$.
C. No, since the execution might result in $x = 1$, $y = 3$.
D. No, since the execution might result in $x = 1$, $y = 2$.

Gloria is considering using locks, and automatically adding code to each transaction to guarantee before-or-after atomicity. She would like to maximize concurrency; she is, however, anxious to avoid deadlocks. For each of the following proposals, decide whether the approach (1) yields semantics consistent with before-or-after atomicity and (2) introduces potential deadlocks.

Q14.3 A single, global lock where ACQUIRE is at the start of each transaction and RELEASE occurs at COMMIT .

Q14.4 A lock for each variable. Every READ or WRITE operation is immediately surrounded by an ACQUIRE and RELEASE of that variable's lock.

Q14.5 A lock for each variable on which a transaction performs READ or WRITE , acquired immediately prior to the first reference to that variable in the transaction; all locks are released at COMMIT .

Q14.6 A lock for each variable on which a transaction performs READ or WRITE , acquired in alphabetical order, immediately following the BEGIN . All locks are released at COMMIT .

Q14.7 A lock for each variable on which a transaction performs WRITE , acquired, in alphabetical order, immediately following the BEGIN . All locks are released at COMMIT .

In the general case (concurrent transactions) Gloria would like to avoid having to read the entire log during crash recovery. She proposes periodically adding a CHECKPOINT entry to the log, and reading the log backwards from the end restoring committed values to RAM. The backwards scan should end as soon as committed values have been restored to all variables. Each CHECKPOINT entry in the log contains current values of all variables and a list of uncommitted transactions at the time of the CHECKPOINT .

Q14.8 What portion of the tape must be read to properly restore values committed at the time of the crash?

A. All of the tape; checkpoints don't help.
B. Enough to include the STARTED record from each transaction that was uncommitted at the time of the crash.
C. Enough to include the last CHECKPOINT , as well as the STARTED record from each transaction that was uncommitted at the time of the crash.
D. Enough to include the last CHECKPOINT , as well as the STARTED record from each transaction that was uncommitted at the time of the last checkpoint.

Simplicity Winns, Gloria's one-time classmate, observes that since global variable values can be reconstructed from the log their storage in RAM is redundant. She proposes eliminating the RAM as well as all of Gloria's proposed locks, and implementing a READ (tid, var) primitive which returns an appropriate value of var by examining the log.

Simplicity's plan is to implement READ so that each transaction a "sees" the values of global values at the time of BEGIN (a) , as well as changes made within a . She quickly sketches an implementation of READ which she claims gives appropriate atomicity semantics:

```
procedure READ(tid, var)
    winners ← EMPTY     // winners is a list.
    prior ← FALSE
    for each entry in log do
        if entry is STARTED (tid) then prior ← TRUE
        if entry is COMMITTED (Etid) and prior = TRUE then add Etid to winners
        if entry is WRITE (Etid, var, old_value, new_value) then
            if Etid = tid then return new_value
            if Etid is in winners then return new_value
    return 0
```

Gloria is a little dazed by Simplicity's quick synopsis, but thinks that Simplicity is likely correct. Gloria asks for your help in figuring out what Simplicity's algorithm actually does.

Q14.9 Suppose transaction `t` performs a `READ` of variable `x` but does not write it. Will each `READ` of `x` in `t` see the same value? If so, concisely describe the value returned by each `READ` ; if not, explain.

Q14.10 Does Simplicity's scheme REALLY offer transaction semantics yet avoid deadlocks?

A. Yup. Read it and weep, Gloria.
B. It doesn't introduce deadlocks, but doesn't guarantee before-or-after transaction semantics either.
C. It gives before-or-after transactions, but introduces possible deadlocks.
D. Simplicity's approach doesn't work even when there's no concurrency—it gives wrong answers.

Q14.11 The real motivation of the Sick Transit problem is a stupid pun. What does *sic transit gloria mundi* actually mean?

A. Thus passes a glorious Monday.
B. Thus passes the glory of the world.
C. Gloria threw up on the T Monday.
D. This bus for First Class and Coach.
E. This is the last straw! If I wanted to take @#%!*% *Latin*, I'd have gone to Oxford.

Problem Set 15: The Bank of Central Peoria, Limited

1998–3–7…14

(Chapter 3)

Ben Bitdiddle decides to go into business. He bids $1 at a Resolution Trust Corporation auction and becomes the owner of the Bank of Central Peoria, Limited (BCPL).

When he arrives at BCPL, Ben is shocked to learn that the only programmer who understood BCPL's database has left the company to work on new animation techniques for South Park. Hiring programmers is difficult in Peoria, so Ben decides to take over the database code himself.

Ben learns that an account is represented as a structure with the following information:

```
structure account
    integer account_id          // account identification number
    integer balance             // account balance
```

The BCPL system implements a standard transaction interface for accessing accounts:

```
tid ← BEGIN ()            // Starts a new transaction that will be identified as
number tid
balance ← READ (tid, account.account_id)            // Returns the balance of an
account
WRITE (tid, account.account_id, newbalance)         // Updates the balance of an
account
COMMIT (tid)            // Makes the updates of transaction tid visible to other
transactions
```

The BCPL system uses two disks, both accessed synchronously (i.e., `GET` and `PUT` operations on the disks won't return until the data is read from the disk or has safely been written to the disk, respectively). One disk contains nothing but the account balances, indexed by number. This disk is called the **database disk**. The other disk is called the **log disk** and exclusively stores, in chronological order, a sequence of records of the form:

```
structure logrecord
    integer op              // WRITE, COMMIT, or END
    integer tid             // transaction number
    integer account_id      // account number
    integer new_balance     // new balance for account “account”
```

where the meaning of each record is given by the `op` field:

```
op = WRITE          // Update of an account to a new balance by transaction tid
op = COMMITTED          // Transaction tid’s updates are now visible to other
transactions                          // and durable across crashes
op = END              // Transaction tid’s writes have all been installed on the
database disk
```

For each active transaction, the BCPL system keeps a list in volatile memory called `intentions` containing pairs (`account_id` , `new_balance`). The implementation of `READ` is as follows:

```
procedure READ (tid, account_id)
    if account_id is in intentions of tid then
        pair ← last pair containing account_id from intentions of tid
        return pair.new_balance
    else
        GET account containing account_id from database
        return account.balance
```

A. Recovery

For this section, assume that there are no concurrent transactions.

Ben asks whether the database computer has ever crashed and learns that it crashed frequently due to intense sound vibrations from the jail next door. Ben decides he had better understand how recovery works in the BCPL system. He examines the implementation of the recovery procedure. He finds the following code:

```
1 procedure RECOVERY ()
2     winners ← NULL
3     reading the log from oldest to newest,
4     for each record in log do
5         if record.op = COMMITTED then add record.tid to winners
6         if record.op = END then remove record.tid from winners
7     again reading the log from oldest to newest,
8     for each record in log do
9         if record.op = WRITE and record.tid is in winners then
10            INSTALL (record.new_balance in database for record.account_id
11    for each tid in winners do
12        LOG {END, tid}
```

Q15.1 What would happen if lines 11 and 12 were omitted?

A. The system might fail to recover correctly from the first crash that occurs.
B. The system would recover correctly from the first crash but the log would be corrupt so the system might fail to recover correctly from the second crash.
C. The system would recover correctly from multiple crashes but would have to do more work when recovering from the second and subsequent crashes.

Q15.2 For the `RECOVERY` and `READ` procedures to be correct, which of the following could be correct implementations of the `COMMIT` procedure?

A.
```
procedure COMMIT (tid) {}
```

B.
```
procedure COMMIT (tid)
    for each pair in tid.intentions do
        INSTALL (pair.new_balance in database for pair.account_id)
    tid.intentions ← NULL
    LOG {COMMITTED, tid}
```

C.
```
procedure COMMIT (tid)
    LOG {END, tid}
    for each pair in tid.intentions do
        INSTALL (pair.new_balance in database for pair.account_id)
    tid.intentions ← NULL
    LOG {COMMITTED, tid}
```

D.
```
procedure COMMIT (tid)
    LOG {COMMITTED, tid}
    for each pair in tid.intentions do
        INSTALL (pair.new_balance in database for pair.account_id)
    tid.intentions ← NULL
    LOG {END, tid}
```

Q15.3 For the `RECOVERY` and `READ` procedures to be correct, which of the following could be correct implementations of the `WRITE` procedure?

A.
```
procedure WRITE (tid, account_id, new_balance)
    LOG {WRITE, tid, account_id, new_balance}
```

B.
```
procedure WRITE (tid, account_id, new_balance)
    add the pair {account_id, new_balance} to tid.intentions
    LOG {WRITE, tid, account_id, new_balance}
```

C.
```
procedure WRITE (tid, account_id, new_balance)
    LOG {WRITE, tid, account_id, new_balance}
    add the pair {account_id, new_balance} to tid.intentions
```

D.
```
procedure WRITE (tid, account_id, new_balance)
    LOG {WRITE, tid, account_id, new_balance}
    add the pair {account_id, new_balance} to tid.intentions
    INSTALL new_balance in database for account_id
```

Ben is rather surprised to see there is no `ABORT (tid)` procedure that terminates a transaction and erases its database updates. He calls up the database developer, who says it should be easy to add. Ben figures he might as well add the feature now, and adds a new log record type `ABORTED` .

Q15.4 Which of the following could be correct implementations of the `ABORT` procedure? Assume that the `RECOVERY` procedure is changed correspondingly.

A.
```
procedure ABORT (tid)
    tid.intentions ← NULL
```

```
B.  procedure ABORT (tid)
        LOG {ABORTED, tid}

C.  procedure ABORT (tid)
        LOG {ABORTED, tid}
        tid.intentions ← NULL

D.  procedure ABORT (tid)
        tid.intentions ← NULL
        LOG {ABORTED, tid}
```

B. Buffer cache

In section B, again assume that there are no concurrent transactions.

BCPL is in intense competition with the nearby branch of Peoria Authorized Savings, Credit and Loan. BCPL's competitive edge is lower account fees. Ben decides to save the cost of upgrading the computer system hardware by adding a volatile memory buffer cache, which will make the database much more efficient on the current hardware. The buffer cache is used for `GET` s and `PUT` s to the database disk only; `GET` s and `PUT` s to the log disk remain write-through and synchronous.

The buffer cache uses an LRU replacement policy. Each account record on the database disk is cached or replaced separately. In other words, the cache block size, disk block size, and account record sizes are all identical.

Q15.5 Why will adding a buffer cache for the database disk make the system more efficient?

A. It is faster to copy from the buffer cache than to `GET` from the disk.
B. If common access patterns can be identified, performance can be improved by prefetching multiple account balances into the cache.
C. It reduces the total number of disk `GET` s when one transaction reads the same account balance multiple times without updating it.
D. It reduces the total number of disk `GET` s when multiple consecutive transactions read the same account balance.

Ben then makes a mistake. He reasons that the intentions list described in section A is now unnecessary, since the list just keeps in-memory copies of database data, which is the same thing done by the buffer cache. He deletes the intentions list code and modifies `PUT` so it updates the copy of the account balance in the buffer cache. He also modifies the system to delay writing the `END` record until all buffered accounts modified by that transaction have been written back to the database disk. Much to his horror, the next time the inmates next door try an escape and the resulting commotion causes the BCPL system to crash, the database does not recover to a consistent state.

Q15.6 What might have caused recovery to fail?

A. The system crashed when only some of the modifications made by a committed transaction had reached the database disk.
B. The LRU replacement policy updated the database disk with data modified by an uncommitted transaction, which later committed before the crash.
C. The LRU replacement policy updated the database disk with data modified by an uncommitted transaction, which failed to commit before the crash.
D. The LRU replacement policy updated the database disk with data modified by a committed transaction, which later completed before the crash.
E. The LRU replacement policy updated the database disk with data modified by a committed transaction, which did not complete before the crash.

C. Concurrency

Ben restores the intention-list code, deletes the buffer cache code and goes back to the simpler system described in section A. He is finally ready to investigate how the BCPL system manages concurrent transactions. He calls up the developer and she tells him that there is a lock stored in main memory for each account in the database, used by the `CONCURRENT_BEGIN` and `CONCURRENT_COMMIT` procedures. Since BCPL runs concurrent transactions, all its applications actually use these two procedures rather than the lower-level `BEGIN` and `COMMIT` procedures described earlier.

An application doing a concurrent transaction must declare the list of accounts it will use as an argument to the `CONCURRENT_BEGIN` procedure.

```
procedure CONCURRENT_BEGIN (account_list)
    do atomically
        for each account in account_list do
            ACQUIRE (account.lock)
    tid ← BEGIN ()
    tid.account_list ← account_list
    return tid

procedure CONCURRENT_COMMIT (tid)
    COMMIT (tid)
    for each account in tid.account_list do
        RELEASE (account.lock)
```

Ben runs two transactions concurrently. Both transactions update account number 2:

```
tida ← CONCURRENT_BEGIN (MAKELIST (2))
tmpa ← READ (tida, 2)
WRITE (tida, 2, tmpa + 1)
CONCURRENT_COMMIT (tida)
```

```
tidb ← CONCURRENT_BEGIN (MAKELIST (2))
tmpb ← READ (tidb, 2)
WRITE (tidb, 2, tmpb + 2)
CONCURRENT_COMMIT (tidb)
```

`MAKELIST` creates a list from its arguments; in this case the list has just one element. The initial balance of account 2 before these transactions start is 0.

Q15.7 What possible values can account 2 have after completing these two transactions (assuming no crashes)?

A. 0
B. 1
C. 2
D. 3
E. 4

Ben is surprised by the order of the operations in `CONCURRENT_COMMIT` , since `COMMIT` is expensive (requiring synchronous writes to the log disk). It would be faster to release the locks first.

Q15.8 If the initial balance of account 2 is zero, what possible values can account 2 have after completing these two transactions (assuming no crashes) if the locks are released before the call to `COMMIT` ?

A. 0
B. 1
C. 2
D. 3
E. 4

Problem Set 16: Whisks*

* Credit for developing this problem set goes to Eddie Kohler.

1996–3–4a…d

(Chapter 3)

The Odd Disk Company (ODC) has just invented a new kind of non-volatile storage, the Whisk. A Whisk is unlike a disk in the following ways:

- Compared with disks, Whisks have very low read and write latencies.
- On the other hand, the data rate when reading and writing a Whisk is much less than that of a disk.

- Whisks are **associative**. Where disks use sector addresses, a Whisk block is named with a pair of items: an address and a **tag**. We write these pairs as A/t, where A is the address and t is the tag. Thus, for example, there might be three blocks on the Whisk with address 49, each with different tags: 49/1, 49/2, and 49/97.

The Whisk provides four important operations:

- `data ← GET (A/t)` : This is the normal read operation.
- `PUT (A/t, data)` : Just like a normal disk. If the system crashes during a write operation, a partially written block may result.
- `boolean ← EXISTS (A/t)` : Returns `TRUE` if block `A/t` exists on the Whisk.
- `CHANGE_TAG (A/m, n)` : Atomically changes the tag `m` of block `A/m` to `n` (deleting any previous block `A/n` in the process). The atomicity includes both all-or-nothing atomicity and before-or-after atomicity.

Ben Bitdiddle is excited about the properties of Whisks. Help him develop different storage systems using Whisks as the medium.

Q16.1 At first, Ben emulates a normal disk by writing all blocks with tag 0. But now he wants to add an `ATOMIC_PUT` operation. Design an `ATOMIC_PUT` for Ben's Whisk, and identify the step that is the commit point.

Ben has started work on a Whisk transaction system; he'd like you to help him finish it. Looking through his notes, you see that Ben's system will use no caches or logs: all writes go straight to the Whisk. One sentence particularly catches your eye: a joyfully scrawled "Transaction IDs Are Tags!!" Ben's basic idea is this. The current state of the database will be stored in blocks with tag 0. When a transaction `t` writes a block, the data is stored in the separate block `A/t` until the transaction commits.

Ben has set aside a special disk address, ComRec, to hold commit records for all running transactions. For a transaction `t` , the contents of `ComRec/t` is either committed, aborted, or pending, depending on the state of transaction `t` .

So far, three procedures have been implemented. In these programs, `t` is a transaction ID, `A` is a Whisk block address, and `data` is a data block.

```
procedure AA_BEGIN (t)
     PUT (ComRec/t, PENDING)
```

```
procedure AA_READ (t, A)
     if EXISTS (A/t) then
          return GET (A/t)
     else // uninitialized!
          return GET (A/0)
```

```
procedure AA_WRITE (t, A, data)
     PUT (A/t, data)
```

The following questions are concerned only with all-or-nothing atomicity; there are no concurrent transactions.

Q16.2 Write pseudocode for `AA_COMMIT (t)` and `AA_ABORT (t)` , and identify the commit point in `AA_COMMIT` . Assume that the variable `dirty` , an array with `num_dirty` elements, holds all the addresses to which `t` has written. (Don't worry about any garbage an aborted transaction might leave on disk, and assume transaction IDs are never reused.)

Q16.3 Write the pseudocode for `AA_RECOVER` , the program that handles recovery after a crash. Ben has already done some of the work: his code examines the ComRec blocks and determines which transactions are `COMMITTED` , `ABORTED` , or `PENDING` . When your pseudocode is called, he has already set 6 variables for you (you might not need them all):

```
num_committed          // the number of committed transactions
committed[i]           // an array holding the committed transactions' IDs
num_aborted            // the number of aborted transactions
aborted[i]             // an array holding the aborted transactions' IDs
num_pending            // the number of transactions in progress
in_progress[i]         // an array holding the in-progress transactions' IDs
```

Whisk addresses run from 0 to N.

Problem Set 17: ANTS (Advanced "Nonce-ensical Transcation System)*

* Credit for developing this problem set goes to Hari Balakrishnan.

2002–3–6…13

(Chapter 3)

Sara Bellum, forever searching for elegance, sets out to design a new transaction system called ANTS, based on the idea of nonces. She observes that the locking schemes she learned in Chapter 3 cause transactions to wait for locks held by other transactions. She observes that it is possible for a transaction to simply abort and retry, instead of waiting for a lock. A little bit more work convinces her that this idea may allow her to design a system in which transactions don't need to use locks for before-or-after atomicity.

Sara sets out to write pseudocode for the following operations: `BEGIN ()` , `READ ()` , `WRITE ()` , `COMMIT ()` , `ABORT ()` , and `RECOVER ()` . She intends that, together, these operations will provide transaction semantics: before-or-after atomicity, all-or-nothing atomicity, and durability. You may assume that once any of these operations starts, it runs to completion without preemption or failure, and that no other thread is running any of the procedures at the same time. The system may interleave the execution of multiple transactions, however.

Sara's implementation assigns a transaction identifier (TID) to a transaction when it calls `BEGIN ()` . The TIDs are integers, and ANTS assigns them in numerically increasing order. Sara's plan for the transaction system's storage is to maintain cell storage for variables, and a write-ahead log for recovery. Sara implements both the cell storage and the log using non-volatile storage. The log contains the following types of records:

- `STARTED` *`TID`*
- `COMMITTED` *`TID`*
- `ABORTED` *`TID`*
- `UPDATED` *`TID, Variable Name, Old Value`*

Sara implements `BEGIN ()` , `COMMIT ()` , `ABORT ()` , and `RECOVER ()` as follows:

- `BEGIN ()` allocates the next TID , appends a `STARTED` record to the log, and returns the TID.
- `COMMIT ()` appends a `COMMITTED` record to the log and returns.
- `ABORT (`*`TID`*`)` undoes all of transaction *`TID`* 's `WRITE ()` operations by scanning the log backwards and writing the old values from the transaction's `UPDATED` records back to the cell storage. After completing the undo, `ABORT (`*`TID`*`)` appends an `ABORTED` entry to the log, and returns.
- `RECOVER ()` is called after a crash and restart, before starting any more transactions. It scans the log backwards, undoing each `WRITE` record of each transaction that had neither committed nor aborted at the time of the crash. `RECOVER ()` appends one `ABORTED` record to the log for each such transaction.

Sara's **before-or-after intention** is that the result of executing multiple transactions concurrently is the same as executing those same transactions one at a time, in increasing TID order. Sara wants her `READ ()` and `WRITE ()` implementations to provide before-or-after atomicity by adhering to the following rule:

Suppose a transaction with TID *`t`* executes `READ (`*`X`*`)` . Let *`u`* be the highest TID less than *`t`* that calls `WRITE (`*`X`*`)` and commits. The `READ (`*`X`*`)` executed by *`t`* should return the value that *`u`* writes.

Sara observes that this rule does not require her system to execute transactions in strict TID order. For example, the fact that two transactions call `READ ()` on the same variable does not (by itself) constrain the order in which the transactions must execute.

To see how Sara intends ANTS to work, consider the following two transactions:

	Transaction T_A	Transaction T_B
1	tid_a ← BEGIN () // returns 15	
2		tid_b ← BEGIN () // returns 16

3	$v_a \leftarrow$ READ (tid_a, X)	
4	$v_a \leftarrow v_a + 1$	
5		$v_b \leftarrow$ READ (tid_b, X)
6		$v_b \leftarrow v_b + 1$
α	WRITE (tid_a, X, v_a)	WRITE (tid_b, X, v_b)
β	COMMIT (tid_a)	COMMIT (tid_b)

Each transaction marks its start with a call to BEGIN , then reads the variable X from the cell store and stores it in a local variable, then adds one to that local variable, then writes the local variable to X in the cell store, and then commits. Each transaction passes its TID (tid_a and tid_b respectively) to the READ , WRITE , and COMMIT procedures.

These transactions both read and write the same piece of data, X . Suppose that TA starts just before TB , and Sara's BEGIN allocates TIDs 15 and 16 to T_A and T_B, respectively. Suppose that ANTS interleaves the execution of the transactions as shown through line 6, but that ANTS has not yet executed lines α and β . No other transactions are executing, and no failures occur.

Q17.1 In this situation, which of the following actions can ANTS take in order to ensure before-or-after atomicity?

A. Force just T_A to abort, and let T_B proceed.
B. Force just T_B to abort, and let T_A proceed.
C. Force neither T_A nor T_B to abort, and let both proceed.
D. Force both T_A and T_B to abort.

To help enforce the before-or-after intention, Sara's implementation of ANTS maintains the following two pieces of information for each variable:

- *ReadID* — the TID of the highest-numbered transaction that has successfully read this variable using READ .
- *WriteID* — the TID of the highest-numbered transaction that has successfully written this variable using WRITE .

Sara defines the following utility procedures in her implementation of ANTS:

- INPROGRESS (*TID*) returns FALSE if *TID* has committed or aborted, and otherwise TRUE . (All transactions interrupted by a crash are aborted by the RECOVER procedure.)
- EXIT () terminates the current thread immediately.
- LOG () appends a record to the log and waits for the write to the log to complete.
- READ_DATA (*x*) reads cell storage and returns the corresponding value.
- WRITE_DATA (*x*, *v*) writes value *v* into cell storage *x* .

Sara now sets out to write pseudocode for READ and WRITE :

```
1 procedure READ (tid, x)              // Return the value stored in cell x
2     if tid < x.WriteID then
3         ABORT (tid)
4         EXIT ()
5     if tid > x.WriteID and INPROGRESS (x.WriteID) then
6          ABORT (tid)                    // Last transaction to have written x is
still in progress
7         EXIT ()
8     v ← READ_DATA (x)                 // In all other cases execute the read
9     x.ReadID ← MAX (tid, x.ReadID)  // Update ReadID of x
10    return v
```

```
11 procedure WRITE (tid, x, v)          // Store value v in cell storage x
12    if tid < x.ReadID then
13        ABORT (tid)
14        EXIT ()
15    else if tid < x.WriteID then
16        [Mystery Statement I]          // See question 17.3
17    else if tid > x.WriteID and INPROGRESS(x.WriteID) then
18        ABORT (tid)
19        EXIT ()
20    LOG (WRITE, tid, x, READ_DATA (x))
21    WRITE_DATA (x, v)
22    [Mystery Statement II]           // Now update ReadID of x (see question 17.5)
```

Help Sara complete the design above by answering the following questions.

Q17.2 Consider lines 5–7 of READ. Sara is not sure if these lines are necessary. If lines 5–7 are removed, will the implementation preserve Sara's before-or-after intention?

A. Yes, the lines can be removed. Because the previous `WRITE` to x, by the transaction identified by `x.WriteID`, cannot be affected by transaction `tid`, `READ_DATA (x)` can safely execute.
B. Yes, the lines can be removed. Suppose transaction T_1 successfully executes `WRITE (x)`, and then transaction T_2 executes `READ (x)` before T_1 commits. After this, T_1 cannot execute `WRITE (x)` successfully, so T_2 would have correctly read the last written value of x from T_1.
C. No, the lines cannot be removed. One reason is: The only transaction that can correctly execute `READ_DATA (x)` is the transaction with TID equal to `x.WriteID`. Therefore, the condition on line 5 of `READ` should simply read: " `if tid > x.WriteID` ".
D. No, the lines cannot be removed. One reason is that before-or-after atomicity might not be preserved when transactions abort.

Q17.3 Consider Mystery Statement I on line 16 of `WRITE`. Which of the following operations for this statement preserves Sara's before-or-after intention?

A. `ABORT (tid)` ; `EXIT ()` ;
B. `return` (without aborting `tid`)
C. Find the higher-numbered transaction T_h corresponding to `x.WriteID` ; `ABORT` (T_h) and terminate the thread that was running T_h; perform `WRITE_DATA (x, v)` in transaction `tid` ; and return.
D. All of the above choices.

Q17.4 Consider lines 17–19 of `WRITE` . Sara is not sure if these lines are necessary. If lines 17–19 are removed, will Sara's implementation preserve her before-or-after intention? Why or why not?

A. Yes, the lines can be removed. We can always recover the correct values from the log.
B. Yes, the lines can be removed since this is the `WRITE` call; it's only on a `READ` call that we need to be worried about the partial results from a previous transaction being visible to another running transaction.
C. No, the lines cannot be removed. One reason is: If transaction T_1 writes to cell x and then transaction T_2 writes to cell x, then an abort of T_2 followed by an abort of T_1 may leave x in an incorrect state.
D. No, the lines cannot be removed. One reason is: If transaction T_1 writes to cell x and then transaction T_2 writes to cell x, then an abort of T_1 followed by an abort of T_2 may leave x in an incorrect state.

Q17.5 Which of these operations for Mystery Statement II on line 22 of `WRITE` preserves Sara's before-or-after intention?

A. `(x.WriteID)` ← `tid`
B. `(x.WriteID)` ← `MIN(x.WriteID, tid)`
C. `(x.WriteID)` ← `MAX(x.WriteID, tid)`
D. `(x.WriteID)` ← `MAX(x.WriteID, x.ReadID)`

Ben Bitdiddle looks at the `READ` and `WRITE` pseudocode shown above for Sara's system and concludes that her system is in fact nonsensical! To make his case, he constructs the following concurrent transactions:

	Transaction T_1	Transaction T_2
1	tid_1 ← BEGIN ()	
2		tid_2 ← BEGIN ()
3	WRITE (tid_1, A, v_1)	
4		v_2 ← READ (tid_2, A)
5		WRITE (tid_2, B, v_2)
6		COMMIT (tid_2)
7	v_1 ← READ (tid_1, B)	
8	COMMIT (tid_1)	

The two transactions are interleaved in the order shown above. Note that T_1 begins before T_2. Ben argues that this leads to a deadlock.

Q17.6 Why is Ben's argument incorrect?

A. Both transactions will abort, but they can both retry if they like.
B. Only T_2 will abort on line 4. So, T_1 can proceed.
C. Only T_1 will abort on line 7. So, T_2 can proceed.
D. Sara's system does not suffer from deadlocks, though concurrent transactions may repeatedly abort and never commit.

Recall that Sara uses a write-ahead log for crash recovery.

Q17.7 Which of these statements is true about log entries in Sara's ANTS implementation?

A. The order of `STARTED` entries in the log is in increasing TID order.
B. The order of `COMMITTED` entries in the log is in increasing TID order.
C. The order of `ABORTED` entries in the log is in increasing TID order.
D. The order of `UPDATED` entries in the log for any given variable is in increasing TID order.

Q17.8 The `WRITE` procedure appends the `UPDATED` record to the log before it installs in cell storage. Sara wants to improve performance by caching the non-volatile cell storage in the volatile main memory. She changes `READ_DATA` to read the value from the cache if it is there; if it isn't, `READ_DATA` reads from non-volatile cell storage. She changes `WRITE_DATA` to update just the cache; ANTS will install to non-volatile cell storage later. Can ANTS delay the install to non-volatile cell storage until after the `COMMITTED` record has been written to the log, and still ensure transaction semantics?

A. No, because if the system crashed between the `COMMIT` and the write to non-volatile storage, `RECOVER` would not recover cell storage correctly.
B. Yes, because the log contains enough information to undo uncommitted transactions after a crash.
C. Yes, because line 3 of `READ` won't let another transaction read the data until after the write to non-volatile storage completes.
D. None of the above.

Problem Set 18: KeyDB*

* Credit for developing this problem set goes to Hari Balakrishnan.

2005–3–13

(Chapter 3)

Keys–R–Us has contracted with you to implement an in-memory key-value transactional store named KeyDB. KeyDB provides a hash table interface to store key-value bindings and to retrieve the value previously associated with a key.

You decide to use locks to provide before-or-after atomicity. Lock L_k is a lock for key k, which corresponds to the entry $KeyDB[k]$. A single transaction may read or write multiple KeyDB entries. Your goal is to achieve correct before-or-after atomicity for all transactions that use KeyDB. Transactions may abort. ACQUIRE (L_k) is called before the first READ or WRITE to $KeyDB[k]$ and RELEASE (L_k) is called after the last access to $KeyDB[k]$.

Q18.1 For each of the following locking rules, is the rule is **necessary**, **sufficient**, or **neither necessary nor sufficient** to *always* guarantee correct before-or-after atomicity between any set of concurrent transactions?

A. An ACQUIRE (L_k) must be performed after the start of a transaction and before the first READ or WRITE of $KeyDB[k]$, and a RELEASE (L_k) must be performed some time after the last READ or WRITE of $KeyDB[k]$ and before the end of the transaction.

B. ACQUIRE s of every needed lock must occur after the start of a transaction and before any other operation, and there can be no RELEASE of a lock before COMMIT or ABORT if the corresponding data item was modified by the thread.

C. ACQUIRE s of every needed lock must occur after the start of a transaction and before the first RELEASE, and there can be no RELEASE of a lock before COMMIT or ABORT if the corresponding data item was modified by the thread.

D. All threads that ACQUIRE more than one lock must ACQUIRE the locks in the same order, and there may be no RELEASE s of locks before COMMIT or ABORT.

E. ACQUIRE s of every needed lock must occur after the start of a transaction and before the first RELEASE, and a lock may be RELEASE d at at any time after the last READ or WRITE of the corresponding data before COMMIT or ABORT.

Q18.2 Determine whether each of the following locking rules either avoids or is likely (with probability approaching 1 as time goes to infinity) to eliminate permanent deadlock between any set of concurrent transactions.

A. ACQUIRE s of every needed lock must occur after the start of a transaction and before any other operation, and there can be no RELEASE of a lock before COMMIT or ABORT.

B. ACQUIRE s of every needed lock must occur after the start of a transaction and before the first RELEASE, and there can be no RELEASE of a lock before COMMIT or ABORT.

C. All threads that ACQUIRE more than one lock must ACQUIRE the locks in the same order.

D. When a transaction begins, set a timer to a value longer than the transaction is expected to take. If the timer expires, ABORT the transaction and try it again with a timer set to a value chosen with random exponential backoff.

Problem Set 19: Alice's Reliable Book Store*

* Credit for developing this problem set goes to Barbara Liskov

2006–3–9

(Chapter 3)

Alice has implemented a version of ALL_OR_NOTHING_PUT and ALL_OR_NOTHING_GET using only two copies, based an idea she got by reading Section 3.8.1. Her implementation appears below. In her implementation each virtual all-or-nothing sector x is stored at two disk locations, $x.D0$ and $x.D1$, which are updated and read as follows:

```
// Write the bits in data at item x
1 procedure ALL_OR_NOTHING_PUT (data, x)
2     flag ← CAREFUL_GET (buffer, x.D0); // read into a temporary buffer
3     if flag = OK then
4         CAREFUL_PUT (data, x.D1);
5         CAREFUL_PUT (data, x.D0);
6     else
7         CAREFUL_PUT (data, x.D0);
8         CAREFUL_PUT (data, x.D1);
```

```
// Read the bits of item x and return them in data
1 procedure ALL_OR_NOTHING_GET (reference data, x)
2     flag ← CAREFUL_GET (data, x.D0);
3     if flag = OK then
4         return;
5     CAREFUL_GET (data, x.D1);
```

The CAREFUL_GET and CAREFUL_PUT procedures are as specified in Section 2.6.4.5 and Figure 2.6.1. The property of these two procedures that is relevant is that CAREFUL_GET can detect cases when the original data is damaged by a system crash during CAREFUL_PUT .

Assume that the only failure to be considered is a fail-stop failure of the system during the execution of ALL_OR_NOTHING_GET or ALL_OR_NOTHING_PUT . After a fail-stop failure the system restarts.

Q19.1 Which of the following statements are true and which are false for Alice's implementation of ALL_OR_NOTHING_PUT and ALL_OR_NOTHING_GET ?

A. Her code obeys the rule "never overwrite the only copy."
B. ALL_OR_NOTHING_PUT and ALL_OR_NOTHING_GET ensure that if just one of the two copies is good (i.e., CAREFUL_GET will succeed for one of the two copies), the caller of ALL_OR_NOTHING_GET will see it.
C. ALL_OR_NOTHING_PUT and ALL_OR_NOTHING_GET ensure that the caller will always see the result of the last ALL_OR_NOTHING_PUT that wrote at least one copy to disk.

Q19.2 Suppose that when ALL_OR_NOTHING_PUT starts running, the copy at *x.D0* is good. Which statement's completion is the commit point of ALL_OR_NOTHING_PUT ?

Q19.3 Suppose that when ALL_OR_NOTHING_PUT starts running, the copy at *x.D0* is bad. Which statement's completion is the commit point of ALL_OR_NOTHING_PUT ?

Consider the following chart showing possible states that the data could be in prior running ALL_OR_NOTHING_PUT :

	State 1	State 2	State 3
x.D0	old	old	bad
x.D1	old	bad	old

For example, when the system is in state 2, *x.D0* contains an old value and *x.D1* contains a bad value, meaning that CAREFUL_GET will return an error if someone tries to read *x.D1* .

Q19.4 Assume that ALL_OR_NOTHING_PUT is attempting to store a new value into item *x* and the system fails. Which of the following statements are true?

A. (*x.D0* = new, *x.D1* = new) is a potential outcome of ALL_OR_NOTHING_PUT , starting in any of the three states.
B. Starting in state S1, a possible outcome is (*x.D0* = bad, *x.D1* = old).
C. Starting in state S2, a possible outcome is (*x.D0* = bad, *x.D1* = new).
D. Starting in state S3, a possible outcome is (*x.D0* = old, *x.D1* = new).
E. Starting in state S1, a possible outcome is (*x.D0* = old, *x.D1* = new).

Ben Bitdiddle proposes a simpler version of ALL_OR_NOTHING_PUT . His simpler version, named SIMPLE_PUT , would be used with the existing ALL_OR_NOTHING_GET .

```
procedure SIMPLE_PUT (data, x)
    CAREFUL_PUT (data, x.D0)
    CAREFUL_PUT (data, x.D1)
```

Q19.5 Will the system work correctly if Ben replaces ALL_OR_NOTHING_PUT with SIMPLE_PUT ? Explain.

Q19.6 Now consider failures other than system failures while running the original `ALL_OR_NOTHING_PUT` . Which of the following statements is true and which false?

A. Suppose `x.D0` and `x.D1` are stored on different disks. Then `ALL_OR_NOTHING_PUT` and `ALL_OR_NOTHING_GET` also mask a single head crash (i.e., the disk head hits the surface of a spinning platter), assuming no other failures.

B. Suppose `x.D0` and `x.D1` are stored as different sectors on the same track. Then `ALL_OR_NOTHING_PUT` and `ALL_OR_NOTHING_GET` also mask a single head crash, assuming no other failures.

C. Suppose that the failure is that the operating system overwrites the in-memory copy of the data being written to disk by `ALL_OR_NOTHING_PUT` . Nevertheless, `ALL_OR_NOTHING_PUT` and `ALL_OR_NOTHING_GET` mask this failure, assuming no other failures.

Now consider how to handle decay failures. The approach is to periodically correct them by running a `SALVAGE` routine. This routine checks each replicated item periodically and if one of the two copies is bad, it overwrites that copy with the good copy. The code for `SALVAGE` is in Figure 3.8.1.

Assume that there is a decay interval D such that at least one copy of a duplicated sector will probably still be good D seconds after the last execution of `ALL_OR_NOTHING_PUT` or `SALVAGE` on that duplicated sector. Further assume that the system recovers from a failure in less than F seconds, where $F << D$, and that system failures happen so infrequently that it is unlikely that more than one will happen in a period of D seconds.

Q19.7 Which of the following methods ensures that the approach handles decay failures with very high probability?

A. `SALVAGE` runs only in a background thread that cycles through the disk with the guarantee that each replicated sector is salvaged every P seconds, where P is less than $D - F$.

B. `SALVAGE` runs as the first step of `ALL_OR_NOTHING_PUT` , and only then.

C. `SALVAGE` runs as the first step of both `ALL_OR_NOTHING_PUT` and `ALL_OR_NOTHING_GET` , and only then.

D. `SALVAGE` runs on all duplicated sectors as part of recovering from a fail-stop failure, and only then.

Problem Set 20: Establishing Serializability*

* Credit for developing this problem set goes to Barbara Liskov.

2006–3–17

(Chapter 3)

Chapter 3 explained that one technique of ensuring correctness is to serialize concurrent transactions that act on shared variables, and it offered methods such as version histories or two-phase locking to ensure serialization. Louis Reasoner has come up with his own locking scheme that does not have an easy proof of correctness, and he wants to know whether or not it actually leads to correct results. Louis implements his locking scheme, runs a particular set of three transactions two different times, and observes the order in which individual actions of the transactions occur. The observed order is known as a **schedule**.

Here are Louis's three transactions:

- T_1: `BEGIN (); WRITE (x); READ (y); WRITE (z); COMMIT ();`
- T_2: `BEGIN (); READ (x); WRITE (z); COMMIT ();`
- T_3: `BEGIN (); READ (z); WRITE (y); COMMIT ();`

The records `x` , `y` and `z` are stored on disk. Louis's first run produces schedule 1 and his second run produces schedule 2:

	Schedule 1	Schedule 2
1	T_1: WRITE (x)	T_3: READ (z)
2	T_2: READ (x)	T_2: READ (x)
3	T_1: READ (y)	T_1: WRITE (x)

	Schedule 1	Schedule 2
4	T_3: READ (z)	T_3: WRITE (y)
5	T_3: WRITE (y)	T_1: READ (y)
6	T_2: WRITE (z)	T_2: WRITE (z)
7	T_1: WRITE (z)	T_1: WRITE (z)

The question Louis needs to answer is whether or not these two schedules can be serialized. One way to establish serializability is to create what is called an **action graph**. An action graph contains one node for each transaction and an arrow (directed edge) from T_i to T_j if T_i and T_j both use the same record r in conflicting modes (that is, both transactions write r or one writes r before the other reads r) and T_i uses r first. If for a particular schedule there is a cycle in its action graph, that schedule is not serializable. If there is no cycle, then the arrows reveal a serialization of those transactions.

Q20.1 The table below lists all of the possible arrows that might lead from one transaction to another. For schedule 1, fill in the table, showing whether or not that arrow exists, and if so list the two steps that create that arrow. To get you started, one row is filled in. Also, draw the arrows in Figure $PS.20.1$.

Arrow	Exists?	Steps
T1 → T2		
T1 → T3		
T2 → T1		
T2 → T3		
T3 → T1		
T3 → T2	Yes	4 and 6

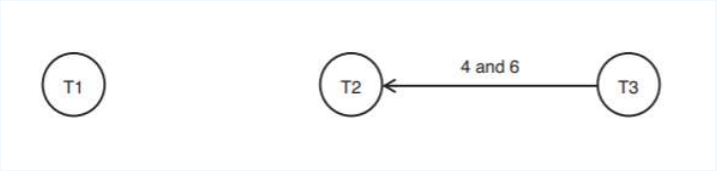


Figure \PS.20.1: Answer template for Q20.1.

Q20.2 Is schedule 1 serializable? If not, explain briefly why not. If so, give a serial schedule for it.

Q20.3 Now fill in the table for schedule 2. This time you get to fill in the whole table yourself. Also, draw in the arrows in Figure $PS.20.2$.

Arrow	Exists?	Steps
T1 → T2		
T1 → T3		
T2 → T1		
T2 → T3		
T3 → T1		
T3 → T2		



Figure $PS.20.2$: Answer template for Q20.2.

Q20.4 Is schedule 2 serializable? If not, explain why not. If so, give a serial schedule for it.

Q20.5 Could schedule 2 have been produced by two-phase locking, in which a transaction acquires a lock on an object as the first part of the step in which it first uses that object? For example, step 3 of schedule 2 is the first time that transaction T1 uses record `x`, so it would start that step by acquiring a lock for `x`. Explain.

Louis is also concerned about recovery. When he ran the three transactions and obtained schedule 2, he found that the system generated the following log:

```
1  BEGIN (transaction: T1)
2  BEGIN (transaction: T2)
3  BEGIN (transaction: T3)
4  CHANGE (transaction: T1, record: x, undo: 1, redo: 2)
5  CHANGE (transaction: T3, record: y, undo: 1, redo: 2)
6  CHANGE (transaction: T2, record: z, undo: 1, redo: 2)
7  COMMIT (transaction: T3)
8  COMMIT (transaction: T2)
9  CHANGE (transaction: T1, record: z, undo: 2, redo: 3)
10 COMMIT (transaction: T1)
```

In a `CHANGE` record the undo field gives the old value before this change, and the redo field gives the new value afterwards. For example, entry 4 indicates that the old value of `x` was 1 and the new value is 2. The system uses the redo/undo recovery procedure of Figure 3.4.7.

Q20.6 Suppose the system crashed after record 7 of the log has made it to disk but before record 8 is written. What states do `x`, `y`, and `z` have after recovery is complete?

Q20.7 Suppose instead that the system crashed after record 9 of the log has made it to disk but before record 10 is written. What states do `x`, `y`, and `z` have after recovery is complete?

Louis's database consists of a collection of integer objects stored on disk. Each `WRITE` operation increments by 1 the object being modified. The system is using a write-ahead logging protocol and there is an in-memory cache that the system periodically flushes to disk, without checking to see if the cached objects belong to committed transactions.

To save space in the log, Louis's friend Ben Bitdiddle suggests that `CHANGE` records could just indicate the operation that was performed. For example, log entry 4 would be:

```
4  CHANGE (transaction: T1, record: x, operation: increment)
```

When the recovery manager sees this entry, it performs the specified operation: increment `x` by 1. Ben makes no other changes to the recovery protocol.

Q20.8 All objects are initialized to 0. Louis tries Ben's plan, but after the first system crash and recovery he discovers that it doesn't work. Explain why.

6.2.2: Problem Sets 11-20 is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

6.2.3: Problem Sets 21-30

Problem Set 21: Speedy Taxi Company*

* Credit for developing this problem set goes to Robert T. Morris

2008–2–9

(Chapter 3)

The Speedy Taxi company uses a computer to help its dispatcher, Arnie. Customers call Arnie, each asking for a taxi to be sent to a particular address, which Arnie enters into the computer. Arnie can also ask the computer to assign the next waiting address to an idle taxi; the computer indicates the address and taxi number to Arnie, who informs that taxi over his two-way radio.

Arnie's computer stores the set of requested addresses and the current destination address of each taxi (if not idle) in an in-memory database. To ensure that this information is not lost in a power failure, the database logs all updates to an on-disk log. Since the database is kept in volatile memory only, the state must be completely reconstructed after a power failure and restart, as in Figure 3.4.6. The database uses write-ahead logging as in Chapter 3: it always appends each update to the log on disk, and waits for the disk write to the log to complete before modifying the cell storage in main memory. The database processes only one transaction at a time (since Arnie is the only user, there is no concurrency).

The database stores the list of addresses waiting to be assigned to taxis as a single variable; thus any change results in the system logging the entire new list. The database stores each taxi's current destination as a separate variable. A taxi is idle if it has no address assigned to it.

Consider one action that uses the database: `DISPATCH_ONE_TAXI` . Arnie's computer presents a UI to him consisting of a button marked `DISPATCH_ONE_TAXI` . When Arnie presses the button, and there are no failures, the computer takes one address from the list of addresses waiting to be assigned, assigns it to an idle taxi, and displays the address and taxi to Arnie.

Here is the code for `DISPATCH_ONE_TAXI` :

```
1  procedure DISPATCH_ONE_TAXI ()
2      BEGIN_TRANSACTION
3                  // read and delete the first address in list
4          list ← READ ()
5          if LENGTH (list) < 1 then
6              ABORT_TRANSACTION
7          address ← list[0]
8          DELETE (list[0])
9          WRITE (list)
10                 // find first free taxi
11         taxi_index ← -1
12         for i from 0 until NUMBER_OF_TAXIS - 1
13             taxis[i] ← READ ()
14             if taxis[i] = NULL and taxi_index = -1 then
15                 taxi_index ← i
16             if taxi_index = -1 then
17                 ABORT_TRANSACTION
18                 // record address as the taxi's destination
19         taxis[taxi_index] ← address
20         WRITE (taxis[taxi_index])
21     COMMIT_TRANSACTION
22     display "DISPATCH TAXI " + taxi_index + " TO " + address
```

When Arnie starts work, `list` contains exactly two addresses: a_1 and a_2. There are two taxis (`taxis[0]` and `taxis[1]`) and both are idle (`NULL`). Arnie pushes the `DISPATCH_ONE_TAXI` button, but he sees no DISPATCH TAXI display, and the computer crashes, restarts, and runs database recovery. Arnie pushes the button a second time and again

sees no DISPATCH TAXI display, and again the computer crashes, restarts, and runs recovery. There is no activity beyond that described or necessarily implied.

Q21.1 If you were to look at last few entries of the database log at this point, which of the following might you see, and which are not possible? B_x stands for a `BEGIN` record for transaction ID *x* , M_x is a `MODIFY` (i.e. change) record for the indicated variable and new value, and C_x is a `COMMIT` record.

A. No log records corresponding to Arnie's actions.
B. B_101; M_101 *list* = a_2; M_101 *taxis* [0] = a_1; C_101; B_102; M_102 *list* = (empty); M_102 *taxis* [1] = a_2; C_102
C. B_101; M_101 *list* = a_2; M_101 *taxis*[0] = a_1; B_102; M_102 *list* = (empty); M_102 *taxis*[1] = a_2
D. B_101; M_101 *list* = a_2; M_101 *taxis*[0] = a_1; C_101; B_102; M_102 *list* = a_2; M_102 *taxis*[0] = a_1
E. B_101; M_101 *list* = a_2; M_101 *taxis*[0] = a_1; B_102; M_102 *list* = a_2; M_102 *taxis*[0] = a_1

Suppose again the same starting state (the address list contains a_1 and a_2, both taxis are idle). Arnie pushes the button, the system crashes without displaying a DISPATCH TAXI message, the system reboots and runs recovery, and Arnie pushes the button again. This time the system does display a DISPATCH TAXI message. Again, there is no activity beyond that described or necessarily implied.

Q21.2 Which of the following are possible messages?

A. DISPATCH TAXI 0 TO a_1
B. DISPATCH TAXI 0 TO a_2
C. DISPATCH TAXI 1 TO a_1
D. DISPATCH TAXI 1 TO a_2

Arnie questions whether it's necessary to make the whole of `DISPATCH_ONE_TAXI` a single transaction. He suggests that it would work equally well to split the program into two transactions, the first comprising lines 2 through 9, and the other comprising lines 12 through 21. Arnie makes this change to the code.

Suppose again the same starting state and no other activity. Arnie pushes the button, the system crashes without displaying a DISPATCH TAXI message, the system reboots and runs recovery, and Arnie pushes button again. This time the system displays a DISPATCH TAXI message.

Q21.3 Which of the following are possible messages?

A. DISPATCH TAXI 0 TO a_1
B. DISPATCH TAXI 0 TO a_2
C. DISPATCH TAXI 1 TO a_1
D. DISPATCH TAXI 1 TO a_2

Problem Set 22: Locking for Transactions Exercise*

* Credit for developing this problem set goes to Robert T. Morris.

2008–3–14

(Chapter 3)

Alyssa has devised a database that uses logs as described in Section 3.4. The logging and recovery works as shown in Figure 3.4.6 (the in-memory database with write-ahead logging). Alyssa claims that if programmers insert `ACQUIRE` and `RELEASE` calls properly they can have transactions with both before-or-after and all-or-nothing atomicity.

Alyssa has programmed the following transaction as a demonstration. As Alyssa claims, it has both before-or-after and all-or-nothing atomicity.

```
T1:
    BEGIN_TRANSACTION ()
    ACQUIRE (X.lock)
    ACQUIRE (Y.lock)
    X ← X + 1
    if X = 1 then
        Y ← Y + 1
    COMMIT_TRANSACTION()
    RELEASE (X.lock)
    RELEASE (Y.lock)
```

X and Y are the names of particular database fields, not parameters of the transaction.

Q22.1 The database starts with contents X=0 and Y=0. Two instances of T_1 are started at about the same time. There are no crashes, and no other activity. After both transactions have finished, which of the following are possible database contents?

A. X=1, Y=1
B. X=2, Y=0
C. X=2, Y=1
D. X=2, Y=2

Ben changes the code for T_1 to `RELEASE` the locks earlier:

```
T1b:
    BEGIN_TRANSACTION ()
    ACQUIRE (X.lock)
    ACQUIRE (Y.lock)
    X ← X + 1
    if X = 1 then
        Y ← Y + 1
    RELEASE (X.lock)
    RELEASE (Y.lock)
    COMMIT_TRANSACTION ()
```

With this change, Louis suspects that there may be a flaw in the program.

Q22.2 The database starts with contents X=0 and Y=0. Two instances of T_{1b} are started at about the same time. There is a crash, a restart, and recovery. After recovery completes, which of the following are possible database contents?

A. X=1, Y=1
B. X=2, Y=0
C. X=2, Y=1
D. X=2, Y=2

Ben and Louis devise the following three transactions. Beware: the locking in T_2 is flawed.

```
T2:
    BEGIN_TRANSACTION ()
    ACQUIRE (M.lock)
    temp ← M
    RELEASE (M.lock)
    ACQUIRE (N.lock)
    N ← N + temp
    COMMIT_TRANSACTION ()
    RELEASE (N.lock)
```

```
T3:
    BEGIN_TRANSACTION ()
    ACQUIRE (M.lock)
    M ← 1
    COMMIT_TRANSACTION ()
    RELEASE (M.lock)

T4:
    BEGIN_TRANSACTION ()
    ACQUIRE (M.lock)
    ACQUIRE (N.lock)
    M ← 1
    N ← 1
    COMMIT_TRANSACTION ()
    RELEASE (M.lock)
    RELEASE (N.lock)
```

Q22.3 The initial values of M and N in the database are M=2 and N=3. Two of the above transactions are executed at about the same time. There are no crashes, and there is no other activity. For each of the following pairs of transactions, decide whether concurrent execution of that pair could result in an incorrect result. If the result is always correct, give an argument why. If an incorrect result could occur, give an example of such a result and describe a scenario that leads to that result.

A. T_2 and T_2:
B. T_2 and T_3:
C. T_2 and T_4:

Problem Set 23: "Log"-ical Calendaring *

* Credit for developing this problem set goes to Hari Balakrishnan.

2004–3–7…15

(Chapters 3 and 4)

Ally Fant is designing a calendar server to store her appointments. A calendar client contacts the server using the following remote procedure calls (RPCs):

- `ADD (timeslot, descr)` : Adds the appointment description (`descr`) to the calendar at time slot `timeslot` .
- `SHOW (timeslot)` : Reads the appointment at time slot `timeslot` from the calendar and displays it to the user. (If there is no appointment, `SHOW` displays an empty slot.)

The RPC between client and server runs over a transport protocol that provides "at-most-once" semantics. The server runs on a separate computer and it stores appointments in an append-only log on disk. The server implements `ADD` in response to the corresponding client request by appending an appointment entry to the log. Each appointment entry has the following format:

```
structure appt_entry
    integer id          // unique id of action that created this entry
    string timeslot     // the timeslot for this appointment
    string descr        // description of this appointment
```

Ally would like to make the `ADD` action atomic. She realizes that she can use `ALL_OR_NOTHING_PUT (data, sector)` and `ALL_OR_NOTHING_GET (data, sector)` as described in Section 3.3.1. These procedures guarantee that a single all-or-nothing sector is written either completely or not at all.

Each appointment entry is for one `timeslot` , which specifies the time interval of the appointment (e.g., 1:30 pm–3:00 pm on May 20, 2005). Each appointment entry is exactly as large as a single all-or-nothing sector (512 bytes). The first all-or-nothing sector on disk, numbered 0, is the `master_sector` , which stores the all-or-nothing sector number where the next

log record will be written. The number stored in *master_sector* is called the end of the log, *end_of_log* , and is initialized to 1.

Ally designs the following procedure:

```
1 procedure ADD (timeslot; descr)
2     id ← NEW_ACTION_ID () // returns a unique identifier
3     appt ← MAKE_NEW_APPT (id; timeslot; descr) // make and fill in an appt entry
4     if ALL_OR_NOTHING_GET (end_of_log; master_sector) ≠ OK then return
5     if ALL_OR_NOTHING_PUT (appt; end_of_log) ≠ OK then return
6     end_of_log ← end_of_log + 1
7     if ALL_OR_NOTHING_PUT (end_of_log; master_sector) ≠ OK then return
```

The procedure NEW_ACTION_ID returns a unique action identifier. The procedure MAKE_NEW_APPT allocates an *appt_entry* structure and fills it in, padding it to 512 bytes.

Ally implements SHOW as follows:

1. Use ALL_OR_NOTHING_GET to read the master sector to determine the end of the log.
2. Scan the log backwards starting from the last written all-or-nothing sector (end_of_log – 1), using ALL_OR_NOTHING_GET on each sector, and stopping as soon as an entry for the timeslot is found.

To help understand if her implementation of the calendar system is correct or not, Ally defines the following properties that her calendar server should ensure:

- P1: SHOW (*timeslot*) should display the appointment corresponding to the last committed ADD to that timeslot, even if system crashes occur during calls to ADD .
- P2: The calendar server must store the appointments corresponding to all committed ADD actions for at least three years.
- P3: If multiple ADD and SHOW actions run concurrently, their execution should be serializable and property P1 should hold.
- P4: No ADD should be committed if it has a time slot that overlaps with an existing appointment.

Ally has learned a number of apparently relevant concepts: before-or-after atomicity, all-or-nothing atomicity, constraint, durability, and transaction.

Q23.1 Which of the apparently relevant concepts does ADD correctly implement?

Q23.2 For each of the properties P2, P3, and P4, identify the apparently relevant concept that best describes it.

Q23.3 What is the earliest point in the execution of the ADD procedure that ensures that a subsequent SHOW is guaranteed to observe the changes made by the ADD . (Assume that the SHOW does not fail.)

A. The successful completion of ALL_OR_NOTHING_PUT in line 5 of ADD .
B. The successful completion of line 6.
C. The successful completion of ALL_OR_NOTHING_PUT in line 7.
D. The instant that ADD returns to its caller.

Q23.4 Ally sometimes uses the calendar server concurrently from different client machines. Which of these statements is true of properties P3 and P4? (Assume that no failures occur, but that the server may be processing multiple RPCs concurrently.)

A. If exactly one ADD and several SHOW actions run concurrently on the server, then property P3 is satisfied even if those actions are for the same timeslot.
B. If more than one ADD and exactly one SHOW run concurrently on the server, then property P3 is satisfied as long as the actions are for different timeslots.
C. Suppose ADD (*timeslot, descr*) calls SHOW (*timeslot*) before line 7 and immediately returns to its caller if the timeslot already has an appointment. If multiple ADD and SHOW actions run concurrently on the server, then property P4 is satisfied whether or not property P3 holds.
D. Suppose ADD (*timeslot, descr*) calls SHOW (timeslot) before line 7 and immediately returns to its caller if the timeslot already has an appointment. If multiple ADD and SHOW actions run concurrently on the server, then property P4 is satisfied as long as property P3 holds.

Q23.5 Ally finds two disks A and B whose conditional failure probabilities follow the "bathtub curve", shown below. She also learns that the disk manufacturers sell units that have been "burned in" but otherwise are unused.Which disk should she buy new to have a higher likelihood of meeting property P2 for at least one year?

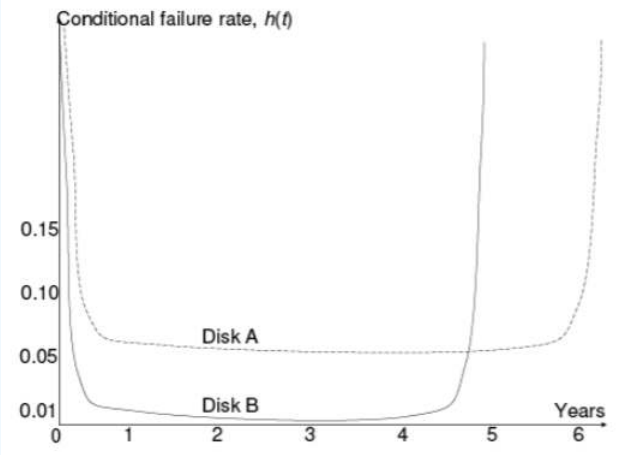


Figure $PS.23.1$: Graph of conditional failure probabilities over time in years for disks A and B.

Multi-user calendar.

Ally becomes president of Scholarly University and opens her server calendar to the entire University community to add and show entries. People start complaining that it takes a long time for them to `SHOW` Ally's appointments. Ally's new provost, Lem E. Fixit, tells her that a single log makes reading slow.

Lem convinces Ally to use the log as a recovery log, and use a volatile in-memory table, named table, to store the appointments to improve the performance of `SHOW` . The table is indexed by the timeslot. `SHOW` is now a simple table lookup, keyed by the timeslot. If the system crashes, the table is lost; when the system recovers, the recovery procedure reinstalls the table. Lem shows Ally how to modify the recovery log to include an "undo" entry in it, as well as a "redo" entry. All the log writes are done using `ALL_OR_NOTHING_PUT` .

Ally writes the following lines in her `NEW_ADD` pseudocode. (For now, the writes to the log are only shown in `COMMIT` .)

```
1  procedure NEW_ADD (timeslot, descr)
2      id ← NEW_ACTION_ID ()
3      appt ← MAKE_NEW_APPT (id, timeslot, descr)
4      table[timeslot] ← appt
5      if OVERLAPPING (table, appt) then ABORT (id)
6      COMMIT (id)
7  procedure COMMIT (id)
8      if ALL_OR_NOTHING_GET (end_of_log, master_sector) ≠ OK then ABORT (id)
9      if ALL_OR_NOTHING_PUT (“COMMITTED”, id, end_of_log) ≠ OK then ABORT (id)
10     end_of_log ← end_of_log + 1
11     if ALL_OR_NOTHING_PUT (end_of_log, master_sector) ≠ OK then ABORT (id)
```

The procedure named `OVERLAPPING` checks *table* to see if *appt* overlaps with a previously committed appointment (property P4). `ABORT` uses the log to undo any changes to table made by `NEW_ADD` , releases any locks that `NEW_ADD` set, and then terminates the action.

Ally modifies `SHOW` to look up an appointment in table, instead of scanning the log.

Q23.6 Which of the following statements is true for `NEW_ADD` with respect to property P1? (Assume that there are no concurrent actions.)

A. If `NEW_ADD` writes the log entry corresponding to the table write just before line 4, then P1 holds.
B. If `NEW_ADD` writes the log entry corresponding to the table write just before line 6, then P1 holds.
C. Because *table* is in volatile memory, there is no need for `ABORT` to undo any changes made by

NEW_ADD in order for P1 to hold.

D. If Ally had designed *table* to be in non-volatile storage, and NEW_ADD inserts the log entry just before line 4, then P1 holds.

Lem convinces Ally that using locks can be a good way to ensure property P3. Ally uses two locks, λ_t and λ_g. λ_t protects *table[timeslot]* and λ_g protects accesses to the log. She needs help to figure out where to place the lock ACQUIRE and RELEASE statements to ensure that property P3 holds when multiple concurrent NEW_ADD and SHOW actions run.

Q23.7 Which of the following placements of ACQUIRE and RELEASE statements in NEW_ADD correctly ensures property P3? Assume that SHOW implements correct locking.

A. ACQUIRE (λ_t) just before line 3,
RELEASE (λ_t) just after line 6,
ACQUIRE (λ_g) just before line 3,
RELEASE (λ_g) just after line 6.

B. ACQUIRE (λ_t) just before line 4,
RELEASE (λ_t) just after line 5,
ACQUIRE (λ_g) just before line 6 but after RELEASE(λ_t) ,
RELEASE (λ_g) just after line 6.

C. None of the above.

Disconnected calendar.

Ally Fant wants to use her calendar in disconnected operation, for example, from her PDA, cell phone, and laptop. Ally modifies the client software as follows. Just before a client disconnects, the client copies the log from the calendar server atomically, and then reinstalls *table* locally. When the user (i.e., Ally) adds an item, the client runs NEW_ADD on the client, updating the local copy of the log and *table* .

When the client can connect to the calendar server or any other client, it reconciles. When reconciling, one of the two machines is the primary. If a client connects to the calendar server, the server is the primary; if a client connects to another client, then one of them is the primary. The client that is not the primary calls RECONCILE , which runs locally on the client:

```
1 procedure RECONCILE (primary, client_log)
2     for each entry ∈ client_log do
3         if entry.state = COMMITTED then
4             invoke NEW_ADD (entry.timeslot, entry.descr) at primary
5     COPY (primary.log, client_log)          // overwrite client_log
6     DELETE (table)
7     rebuild table from client_log          // create new table
```

Assume that RECONCILE is atomic and that no crashes occur during reconciliation. Assume also that between any pair of nodes there is at most one active RECONCILE at any time.

Q23.8 Which of the following statements is true about the implementation that supports disconnected operation?

A. RECONCILE will resolve overlapping appointments in favor of appointments already present on the primary.
B. Some appointments added on a disconnected client may not appear in the output of SHOW after the reconciliation is completed.
C. The result of client C1 reconciling with client C2 (with C2 as the primary), and then reconciling C2 with the calendar server, is the same as reconciling C2 with client C1 (with C1 as the primary), and then reconciling C1 with the calendar server.
D. Suppose Ally stops making changes, and then reconciles all clients with the server once. After doing that, the logs on all machines will be the same.

Lem E. Fixit notices that the procedure RECONCILE is slow. To speed it up, Lem invents a new kind of record, called the RECONCILED record. Each time RECONCILE runs, it appends a RECONCILED record listing the client's unique identifier to the primary's log just before line 5.

Q23.9 Which of the following uses of the `RECONCILED` record speeds up `RECONCILE` correctly? (Assume that clients reconcile only with the calendar server.)

A. Modify line 2 to scan the client log backwards (from the end of the log), terminating the scan if a `RECONCILED` record with the client's identifier is found, and then scan forward until the end of the log calling `NEW_ADD` on the appointment entries in the log.

B. Modify line 2 to scan the client log forwards (from the beginning of the log) calling `NEW_ADD` on the appointment entries in the log, but terminating the scan if a `RECONCILED` record with the client's identifier is found.

C. Don't reinstall table from scratch at the end of reconciliation, but instead update it by adding the entries in the primary log (which the client just copied) that are between the previous `RECONCILED` record and the `RECONCILED` record from the current reconciliation. If an entry in the log overlaps with an entry in the table, then replace the table entry with the one in the log.

D. Assign Lem E. Fixit a different job. None of these optimizations maintains correctness.

Problem Set 24: Ben's Calendar Exercise*

* Credit for developing this problem set goes to Robert T. Morris.

2001–3–14…19

(Chapter 4)

Ben Bitdiddle has just been promoted to Engineering Manager. He quickly notices two facts about his new job. First, keeping an accurate appointment calendar is crucial. Second, he no longer has any programming responsibilities. He decides to address both problems at once by building his own highly available replicated calendar system.

Ben runs a client user interface on his workstation. The client talks over the network to one of three replicated servers. Ben places the three servers, called S1, S2, and S3, in three different cities to try to ensure independent failure. Ben only runs one client at a time.

Each server keeps a simple database of Ben's appointments. The database holds a string for every hour of every day, describing the appointment for that hour. The string for each hour starts out empty. A server can perform just two operations on its own database:

- `DBREAD (day, hour)` returns the appointments for a particular day and hour. The argument day indicates the desired day, where 0 means January 1st, 2000. The argument hour is an integer between 0 and 23, inclusive.
- `DBWRITE (day, hour, string)` changes the string for the hour `hour` . Writing an empty string to an hour effectively deletes any existing appointment for that hour.

Each server allows Ben's client to invoke these operations by RPC. The RPC system uses a powerful checksum that detects all errors and discards any corrupted message. If the RPC client implementation doesn't receive a response from the server within a few seconds, it times out, sets the variable `rpc_OK` to false, and returns `NIL` . If the client does receive a reply from the server, the RPC implementation sets `rpc_OK` to true and returns the result from the server, if any. The RPC system does not resend requests. Thus, for example, if the network discards or corrupts the request or response message, the RPC call returns with `rpc_OK` set to false.

Ben's client user interface can display the appointments for a day and also change an appointment. To support these operations, Ben writes client software based on this pseudocode (the notation `S[i].F` indicates an RPC call to procedure `F` on server `i`):

```
procedure CLIENTREAD (day, hour)
    string s
    for i from 1 to 4 do            // try each server one by one
        s ← S[i].DBREAD (day, hour)
        if rpc_OK then return s     // return with the first reply
    return "Error"
```

```
procedure CLIENTWRITE (day, hour, what)
    for i from 1 to 4 do            // write to all three servers
        boolean done ← FALSE
        while done = FALSE do
            S[i].DBWRITE (day, hour, what)
            if rpc_OK then done ← TRUE
```

Q24.1 Suppose the network connecting Ben's client to servers S1 and S2 is fast and reliable, but the network between the client and S3 often stops working for a few minutes at a time. How will Ben's system behave in this situation?

A. `CLIENTWRITE` will often take a few minutes or more to complete.
B. `CLIENTREAD` will often take a few minutes or more to complete.
C. `CLIENTWRITE` will often fail to update all of the servers.
D. `CLIENTREAD` will often fail, returning "Error".

Ben tests his system by reading and writing the entry for January 1st, 2000, 10 a.m.; he calls:

```
CLIENTWRITE (0, 10, "Staff Meeting")
CLIENTWRITE (0, 10, "Breakfast")
CLIENTREAD (0, 10)
```

Q24.2 Suppose there are no faults. What string will the `CLIENTREAD` call return?

Just to be sure, Ben tries a different test, involving moving a meeting from 10 a.m. to 11 a.m., and scheduling breakfast:

```
CLIENTWRITE (0, 10, "Free at 10")
CLIENTWRITE (0, 11, "Free at 11")
CLIENTWRITE (0, 10, "Talk to Frans at 10")
CLIENTWRITE (0, 11, "Talk to Frans at 11")
CLIENTWRITE (0, 10, "Breakfast at 10")
```

Ben starts the test, but trips over the power cord of his client computer while the test is running, causing the client to reboot. The client forgets that it was executing the test after the reboot; it doesn't restart or continue the test. After the reboot Ben calls `CLIENTREAD (0, 10)` and `CLIENTREAD (0, 11)` . Other than the mentioned client reboot, the only faults that might occur during the test are lost messages (and thus RPC failures).

Q24.3 Which of the following results might Ben see?

A. Breakfast at 10, Talk to Frans at 11
B. Talk to Frans at 10, Talk to Frans at 11
C. Breakfast at 10, Free at 11
D. Free at 10, Talk to Frans at 11

Q24.4 Ben is getting a little paranoid, so he calls `ClientREAD (0, 10)` twice, to see how consistent his database is. Which of the following results might Ben see?

A. Breakfast at 10, Breakfast at 10
B. Talk to Frans at 10, Talk to Frans at 10
C. Free at 10, Breakfast at 10
D. Talk to Frans at 10, Free at 10

Ben feels this behavior is acceptable. Before he starts to use the system, however, his younger brother Mark points out that Ben's system won't be able to complete updates if one of the servers is down. Mark says that if a server is down, a `DBWRITE` RPC to that server will time out, so `CLIENTWRITE` will have higher availability if it ignores RPC timer expirations. Mark suggests the following changed `CLIENTWRITE` :

```
procedure CLIENTWRITE (day, hour, what)
    for i from 1 to 4 do
        S[i].DBWRITE (day, hour, what)
        // Ignore RPC failure
```

Ben adopts this change, and starts using the system to keep his appointments. However, his co-workers start to complain that he is missing meetings. Suspicious of Mark's change, Ben tests the system by manually clearing all database entries on all servers to empty strings, then executing the following code on his client:

```
CLIENTWRITE (0, 10, "X")
v1 ← CLIENTREAD (0, 10)
CLIENTWRITE (0, 10, "Y")
v2 ← CLIENTREAD (0, 10)
CLIENTWRITE (0, 10, "Z")
v3 ← CLIENTREAD (0, 10)
```

Assume that the only possible faults are losses of RPC messages, and that RPC messages are delivered instantly.

Q24.5 With Mark's change, which of the following sequences of `v1` , `v2` , and `v3` are possible?

A. X, Y, Z
B. X, Z, Y
C. X, X, Z
D. X, Y, X

Q24.6 In Ben's original system, what sequences of `v1` , `v2` , and `v3` would have been possible?

A. X, Y, Z
B. X, Z, Y
C. X, X, Z
D. X, Y, X

Problem Set 25: Alice's Replicas

2003–3–6…12

(Chapter 4)

After reading Chapter 4 and the end-to-end argument, Alice explores designing an application for reconciling UNIX file systems. Her program, `RECONCILE` , takes as input the names of two directory trees and attempts to reconcile the differences between the two trees. The typical scenario for `RECONCILE` is that one of the directory trees is on a file service. The other one is a replica of that same directory tree, located on Alice's laptop.

When Alice is disconnected from the service, she modifies files on her laptop while her friends may modify files on the service When Alice reconnects to the service, she runs `RECONCILE` to reconcile the differences between the directory tree on her laptop and the service so that they are identical again. For example, if a file has changed on her laptop, but not on the service, `RECONCILE` copies the file from the laptop to the service. If the file has changed on both the laptop and server, then `RECONCILE` requires guidance to resolve conflicting changes.

The `RECONCILE` program maintains on each host a database named `fsinfo` , which is stored outside the directory tree being reconciled. This database is indexed by path name, and it stores:

```
character pathname[1024]     // path name of this file
integer160 hash              // cryptographic hash of the content of the file
```

On disk a UNIX file consists of metadata (the inode) and content (the data blocks). The cryptographic hash is computed using only the file's content. Path names are less than 1024 bytes. For this problem, ignore the details of reconciling directories, and assume that Alice has permission to read and write everything in both directory trees.

The `RECONCILE` program operates in 4 phases:

- Phase 1: Compute the set of changes on the laptop since the last reconciliation and the set of changes on the server since the last reconciliation.
- Phase 2: The laptop retrieves the set of changes from the service. Using the two change sets, the laptop computes a set of actions that must be taken to reconcile the two directory trees. In this phase, reconcile might decide that some files cannot be reconciled because of conflicting changes.

- Phase 3: The laptop carries out the actions determined in phase 2. The laptop updates the files and directories in its local directory tree, and retrieves files, sends files, and sends instructions to the server to update its directory tree.
- Phase 4: Both the laptop and the service update the *fsinfo* they used to reflect the new content of files that were successfully reconciled on this run.

Assume that RECONCILE runs to completion before starting again. Furthermore, assume that when reconcile runs no concurrent threads are allowed to modify the file system. Also assume that initially the *fsinfo* databases are identical in content and computed correctly, and that after reconciliation they also end up in an identical state.

Assume that RECONCILE runs to completion before starting again. Furthermore, assume that when reconcile runs no concurrent threads are allowed to modify the file system. Also assume that initially the *fsinfo* databases are identical in content and computed correctly, and that after reconciliation they also end up in an identical state.

The first phase of RECONCILE runs the procedure COMPUTEMODIFICATIONS on both the laptop and the service. On each machine COMPUTEMODIFICATIONS produces two sets: a set of files that changed on that machine and a set of files that were deleted on that machine.

```
set changeset, deleteset;

procedure COMPUTEMODIFICATIONS (path, fsinfo)
    changeset ← NULL
    deleteset ← NULL
    COMPUTECHANGESET (path, fsinfo)
    COMPUTEDELETESET (fsinfo)

procedure COMPUTECHANGESET (path, fsinfo)
    info ← LOOKUP (path, fsinfo)
    if info = NULL then ADD (path, changeset)
    else if ISDIRECTORY (path) then
        for each file in path do COMPUTECHANGESET (path/file, fsinfo)
    else if CSHA (CONTENT (path) ≠ info.hash then ADD (path, changeset)

procedure COMPUTEDELETESET (fsinfo)
    for each entry in fsinfo do
        if not (EXIST (entry.pathname)) then ADD (pathname, deleteset)
```

The COMPUTEMODIFICATIONS procedure takes as arguments the path name of the directory tree to be reconciled and the *fsinfo* database. The procedure COMPUTECHANGESET walks the directory tree and checks every file to see if it was created or changed since the last time RECONCILE ran. CSHA is a cryptographic hash function, which has the property that it is computationally infeasible to find two different inputs *i* and *j* such that

```
CSHA (i) = CSHA (j)
```

The COMPUTEDELETESET procedure checks for each entry in the database whether the corresponding file still exists; if not, it adds it to the set of files that have been deleted since the last run of RECONCILE .

Q25.1 What files will RECONCILE add to *changeset* or *deleteset* ?

A. Files whose content has decayed.
B. Files whose content has been modified.
C. Files that have been created.
D. Files whose inodes have been modified.
E. Files that have been deleted.
F. Files that have been deleted but recreated with identical content.
G. Files that have been renamed.

The second phase of reconcile compares the two *changesets* and the two *deletesets* to compute a set of actions for reconciling the two directory trees. To avoid confusion we call *changeset* on the laptop *changeLeft* , and *changeset* on the service *changeRight* . Similarly, *deleteset* on the laptop is *deleteLeft* and

deleteset on the service is *deleteRight* . The second phase consists of running the procedure COMPUTEACTIONS with the 4 sets as input. COMPUTEACTIONS produces 5 sets:

- *additionsLeft* : files that must be copied from server to the laptop
- *additionsRight* : files that must be copied from laptop to the service
- *removeLeft* : file that must be removed from laptop
- *removeRight* : file that must be removed from service
- *conflicts* : files that have conflicting changes

In the following code fragment, the notation *A - B* indicates all elements of the set *A* after removing the elements that also occur in the set *B* . With this notation, the 5 sets are computed as follows:

```
conflicts ← NULL;

procedure COMPUTEACTIONS (changeLeft, changeRight, deleteLeft, deleteRight)
    for each file ∈ changeLeft do
        if file ∈ (changeRight ∪ deleteRight) then ADD (file, conflicts)
    for each file ∈ (deleteLeft) do
        if file ∈ (changeRight) then ADD (file, conflicts)
    additionsRight ← changeLeft - conflicts
    additionsLeft ← changeRight - conflicts
    removeRight ← deleteLeft - conflicts
    removeLeft ← deleteRight - conflicts
```

Q25.2 What files end up in the set *additionsRight* ?

A. Files created on the laptop that don't exist on the service.
B. Files that have been removed on the server but not changed on the laptop.
C. Files that have been removed on the laptop but not on the service.
D. Files that have been modified on the laptop but not on the service.
E. Files that have been modified on the laptop and on the service.

Q25.3 What files end up in the set conflicts?

A. Files that have been modified on the laptop and on the service.
B. Files that have been removed on the laptop but that exist unmodified on the service.
C. Files that have been removed on the laptop and on the service.
D. Files that have been modified on the service but not on the laptop.
E. Files that have been created on the laptop and on the service but with different content.
F. Files that have been created on the laptop and on the service with the same content.

Phase 3 of the reconcile executes the actions: deleting files, transferring files, and resolving conflicts. All conflicts are resolved by asking the user.

We focus on transferring files from laptop to service. Alice wants to ensure that transfers of files are atomic. Assume that all file system calls execute atomically. The RECONCILE program transfers files from *additionsRight* by invoking the remote procedure RECEIVE on the service:

```
procedure RECEIVE (data, size, path)
    tname ← UNIQUENAME ()
    fd ← CREATE_FILE (tname)
    if fd ≥ 0 then
        n ← WRITE (fd, data, size)
        CLOSE (fd)
        if n = size then RENAME (tname, path)
        else DELETE (tname)
        return (n = size)          // boolean result tells success or failure
    else return (FALSE)
```

The RECEIVE procedure takes as arguments the new content of the file (*data* and *size*) and the name (*path*) of the file to be updated or created. As its first step, RECEIVE creates a temporary file with a unique name (*tname*) and writes the data into in it. After the write is successful, RECEIVE renames the temporary file to its real name (*path*), which incidentally removes any existing old version of *path* ; otherwise, it cleans up and deletes the temporary file. Assume that RENAME always executes successfully.

Q25.4 Where is the commit point in the procedure RECEIVE ?

A. right after RENAME completes
B. right after CLOSE completes
C. right after CREATE_FILE completes
D. right after DELETE completes
E. right after WRITE completes
F. none of the above

After the server or laptop fails, it calls a recovery procedure to back out or roll forward a RECEIVE operation that was in progress when the host failed.

Q25.5 What must this recovery procedure do?

A. Remove any temporary files left by receive.
B. Nothing.
C. Send a message to the sender to restart the file transfer.
D. Rename any temporary files left by receive to their corresponding path name.

Q25.6 Which advantages does this version of RECONCILE have over the reconciliation procedure described in Chapter 4?

A. This RECONCILE repairs files that decay.
B. This RECONCILE doesn't require changes to the underlying file system implementation.
C. This version of RECONCILE doesn't require a log on the laptop.
D. This RECONCILE propagates changes from the laptop to the service, and vice versa.
E. This RECONCILE will run much faster on big files.

Alice wonders if her code extends to reconciling more than two file systems. Consider 3 hosts (A, B, and C) that all have an identical copy of a file *f* , and the following sequence of events:

- at noon B modifies file *f*
- at 1 pm B reconciles with A
- at 2 pm C modifies *f*
- at 3 pm B reconciles with C
- at 4 pm A modifies *f*
- at 5 pm B reconciles with A

Assume that B has two distinct *fsinfo* databases, one used for reconciling with A and one for reconciling with C.

Q25.7 Which of the following statements are correct, given this sequence of events and Alice's implementation of RECONCILE ?

A. If the conflict at 3 pm is reconciled in favor of B's copy, then RECONCILE will not report a conflict at 5 pm.
B. If the conflict at 3 pm is reconciled in favor of C's copy, then RECONCILE will report a conflict at 5 pm.
C. If the conflict at 3 pm is resolved by a modification to *f* that merges B's and C's versions, then RECONCILE will report a conflict at 5 pm.
D. If the conflict at 3 pm is resolved by removing f from B and C, then RECONCILE will not report a conflict at 5 pm.

Problem Set 26: JailNet*

* Credit for developing this problem set goes to Stephen A. Ward.

1998–2–7…14

(Some Chapter 1, but mostly Chapter 5)

The Computer Crimes Correction Facility, a federal prison for perpetrators of information-related crimes, has observed curious behavior among their inmates. Prisoners have discovered that they can broadcast arbitrary binary strings to each other by banging cell bars with either the tops or bottoms of their tin cups, making distinct sounds for "0" and "1". Since such sounds made in any cell can typically be heard in every other cell, they have devised an Ethernet-like scheme for communicating varying-length packets among themselves.

The basic communication scheme was devised by Annette Laire, a CCCF lifer convicted of illegal exportation of restricted information when the GIF she emailed to her cousin in El Salvador was found to have some bits in common with a competent cryptographic algorithm.

Annette defined the following basic communication primitive:

```
procedure SEND (message, from, to)
    BANG (ALL_ONES)                          // Start with a byte of eight 1's
    BANG (to)                                // destination inmate number
    BANG (from)                              // source inmate number
    BANG (message)                           // the message data
    BANG (CHECKSUM ({to, from, message}))    // Checksum of whole message
```

where the operation `BANG (data)` is executed by banging ones tin cup to signal the sequence of bits corresponding to the specified null-terminated character string, including the zero byte at its end. The special string `ALL_ONES` sent initially has a single byte of (eight) 1 bits (followed by the terminating null byte). The high-order bit of each 8-bit character (in `to` , `from` , `message` , and the result of `CHECKSUM`) is zero.

Annette specified that the `to` and `from` strings be the unique numbers printed on every inmate's uniform, since all of the nerd-inmates quickly learn the numbers of each of their colleagues. Each inmate listens more or less continuously for packets addressed to them, ignoring all packets whose `to` field don't match their number or whose checksums are invalid.

Q26.1 What function(s) are served by sending the initial byte of all 1's?

A. Bit framing.
B. Byte (character) framing.
C. Packet framing.
D. Packet Reassembly.
E. None of the above.

Typical higher-level protocols involve sequences of packets exchanged between inmates, for example:

```
Annette ⇒ Ty: SEND ("I thought the lobster bisque was good tonight", ANNETTE, TY);
Ty ⇒ Annette: SEND ("Yes, but the filet was a bit underdone", TY, ANNETTE);
```

where the symbols `ANNETTE` and `TY` are bound to character strings containing the uniform numbers of Annette and Ty, respectively.

Of course, prison guards quickly catch on to the communication scheme, listen in on the conversations, and sometimes even inject messages of their own, typically with false source addresses:

```
Guard: SEND ("Yeah? Then it's dog food for you tomorrow!", JIMMIETHEGEEK, ANNETTE);
```

Such experiences motivate Ty Debole, the inmate in charge of cleaning, to add security measures to the JailNet protocols. Ty reads up on public-key cryptography and decides to use it as the basis for JailNet security. He chooses a public-key algorithm and asks each inmate to generate a public/private key pair and tell him the public key.

- `KEY` represents the inmate's public key. Since Ty runs the CCCF laundry, he prints the numbers on inmates' uniforms. He replaces each inmate's assigned number with a representation of KEY;

- `$KEY` is the inmate's private key. This key is known only to the inmate whose uniform bears `KEY` .

Ty assures each inmate that so long as they don't reveal their private `$KEY` , nobody else—inmates or guards—will be able to determine it. Inmates continue to address each other by the numbers on their uniforms, which now specify their public Keys.

Q26.2 What is an assumption on which Ty bases the security of the secret `$KEY` ?

A. `$KEY` is theoretically impossible to compute from `KEY` .
B. `$KEY` takes an intractably long time to compute from `KEY` .
C. `$KEY` takes at least as long to compute from `KEY` as the generation of the `KEY` , `$KEY` pair.
D. There is a reasonably efficient way to compute `$KEY` , but it's not generally known by guards and inmates.

Ty then teaches inmates the 4 security primitives for messages of up to 1,500 bytes:

```
ENCRYPT (plaintext, KEY)          // returns a message string
DECRYPT (ciphertext, $KEY)        // returns a message string
SIGN (message, $KEY)              // returns an authentication tag
VERIFY (message, KEY, signature) // returns ACCEPT or REJECT
```

These primitives have the properties described in Chapter 5.

Ty proposes improving the security of communications by replacing calls to `SEND` with calls like:

```
SEND (TYCODE (message, from, to), from, to);
```

where `TYCODE` is defined as

```
procedure TYCODE (message, from, to)
    return ENCRYPT (message, to)
```

Ty and Annette are smugly confident that although the guards might hear their conversation, they won't be able to understand it since the encrypted message appears as gibberish until properly decoded. The first use of `TYCODE` involves the following message, received by Annette:

```
SEND (TYCODE (“Meet by the wall at ten for the escape”, TY, ANNETTE), TY, ANNETTE)
```

Q26.3 What computation did Annette perform to decode Ty's message? Assume *`rmessage`* is the message as received, *`message`* is to be the decoded plaintext, and that *`$Annette`* and *`$Ty`* contain the secret keys of Annette and Ty, respectively.

A. `message ← VERIFY (rmessage, Ty, $Annette);`
B. `message ← ENCRYPT (rmessage, $Ty);`
C. `message ← ENCRYPT (rmessage, Ty);`
D. `message ← DECRYPT (rmessage, $Annette);`
E. `message ← SIGN (rmessage, $Ty);`
F. `message ← DECRYPT (rmessage, Annette);`

After receiving the message, Annette sneaks out at ten to meet Ty, whom she expects will help her climb over the prison wall. Unfortunately Ty never shows up, and Annette gets caught by a giggling guard and is punished severely (early bed, no dessert). When she talks to Ty the next day, she learns that he never sent the message. She concludes that it must have been sent by a guard, but is puzzled since the cryptography is secure.

Q26.4 What is the most likely explanation?

A. Annette's secret key was compromised during a search of her cell.
B. Some other message Ty sent was garbled in transmission, and accidentally came out "Meet me by the wall at ten for the escape".
C. Annette’s secret key was broken by a dictionary attack.
D. Ty's secret key was broken by a dictionary attack.
E. Annette was victimized by a replay attack.

Annette's friend Cert Defy, on hearing this story, comes up with a new cryptographic procedure:

```
procedure CERT (message, A)
    signature ← SIGN (message, A)
    return {message, signature}
```

Unfortunately, Cert is placed in solitary confinement before fully explaining how to use this procedure, though he did state that sending a message with

```
SEND (CERT (message, A), from, to)
```

can assure the receiver of the integrity of the message body and the authenticity of the sender's identity. So the inmates need some help from you.

Q26.5 When Ty sends a message to Annette, what value should he supply for `A` ?

A. `ENCRYPT (Annette, $Ty)`
B. `Ty`
C. `$Ty`
D. `Annette`
E. `$Annette`

After Ty determinines the answer to Q26.5, Annette receives a packet purportedly from Ty. She splits the received packet into `message` and `signature` , and `VERIFY (message, TY, signature)` returns `ACCEPT` .

Q26.6 Which of the following can Annette conclude about `message` ?

A. `message` was initially sent by Ty.
B. The packet was sent by Ty.
C. `message` was initially sent to Annette.
D. Only Annette and Ty know the contents of `message` .
E. If Ty sent `message` to Annette and Annette only, then only they know its contents.
F. `message` was not corrupted in transmission.

Annette, intrigued by Cert's contribution, decides to combine `SEND` , `TYCODE` , and `CERT` to achieve both authentication and confidentiality. She proposes to use `NEWSEND` , combining both features:

```
procedure NEWSEND (message, A, from, to)
    SEND (TYCODE (CERT (message, A), from, to), from, to)
```

Annette engages in the following conversation:

```
Ty ⇒ Annette: NEWSEND (“Let’s escape tonight at ten”, TY, ANNETTE);
Ty ⇒ Annette: NEWSEND (“Not tonight, Survivor is on”, ANNETTE, TY);
```

The following night, Annette gets the message

```
Ty ⇒ Annette: NEWSEND (“Let’s escape tonight at ten”, TY, ANNETTE);
```

Once again Annette goes to meet Ty at ten, but Ty never shows up. Eventually Annette gets bored and returns. Ty subsequently disclaims having sent the message. Again, Annette is puzzled by the failure of her allegedly secure system. She suspects that a guard has figured out how to break the system.

Q26.7 Explain why this happened, yet no guard showed up at the wall to punish Annette for plotting to escape. Suggest a change that Ty could have made that would have eliminated the problem.

Pete O’Fender, who has been in and out of CCCF at regular intervals, wants to extend the security protocols to deal with JailNet key distribution. Whenever he’s jailed, Pete is placed directly into solitary confinement where he has no contact with inmates (except via bar banging), and where the TV gets only 3 channels. The problem is complicated by the facts that (a) Everyone (including Pete) forgets Pete's uniform number as soon as he leaves, so when he returns he can't just re-use the old key; (b) Pete may not even remember the key for Ty or other trusted long-term inmates; (c) Pete is issued an unnumbered uniform while in solitary, and (d) guards often pose as newly-jailed solitary occupants to learn inmate secrets. Pete asks you to devise JailNet key distribution protocols to address these problems.

Q26.8 Which of the following are true of the best protocol you can devise, given the assumptions stated about `ENCRYPT` , `DECRYPT` , `SIGN` , and `VERIFY` ?

A. Assuming Pete is thrust into Solitary remembering no keys, he can devise a new `Key/$Key` pair and broadcast `Key` . Using this `Key` , Ty can be assured that messages he sends to Pete are confidential.

B. Assuming Pete is thrust into Solitary remembering no keys, he can't convince inmates that they aren't communicating with a guard.

C. If Pete remembers Ty's uniform number and trusts Ty, an authenticated broadcast message from Ty could be used to remind Pete of other inmates' uniform numbers without danger of deluding Pete.

D. Even if Pete remembers a trusted inmate's uniform number, any communication from Pete can be understood by guards.

E. Even if Pete remembers a trusted inmate's uniform number, any communication to Pete might have been forged by guards.

Problem Set 27: PigeonExpress!.com II

1999–2–16,17

(Chapter 5)

To drive up the stock value of PigeonExpress!.com at the planned Initial Public Offering (IPO), Ben needs to make the pigeon net secure. To focus on just security issues, assume for this problem that pigeons never get lost.

First, Ben goes for achieving confidentiality. Ben generates 20 CDs (`KCD[0]` through `KCD[19]`) filled with random numbers, makes two copies of each CD, and mails the copies through a secure channel to the sender and receiver. He plans to use the CDs as a onetime pad.

Ben slightly modifies the original `BEEP` code (which appeared just before question Q2.1) to use the key CDs. The sender's computer runs these two procedures:

```
shared next_sequence initially 0     // a global sequence number, starting at 0.
shared nextKCD initially 0           // index in the array of key CDs.

procedure SECURE_BEEP (destination, n, CD[])     // send n CDs to destination
    header h                         // h is an instance of header.
    nextCD ← 0
    h.source ← MY_GPS                // set source to my GPS coordinates
    h.destination ← destination      // set destination
    h.type ← REQUEST                 // this is a request message
    while nextCD < n do              // send the CDs
        h.sequence_no ← next_sequence            // set seq number for this CD
        send pigeon with {h, (CD[nextCD] ⊕ KCD[nextKCD])}   // send encrypted
        wait 2,000 seconds

procedure SECURE_PROCESS_ACK (h)         // process acknowledgment
    if h.sequence_no = sequence then     // ack for current outstanding CD?
        next_sequence ← next_sequence + 1
        nextCD ← nextCD + 1              // allow next CD to be sent
        nextKCD ← (nextKCD + 1) modulo 20     // increment with wrap-around
```

Ben also modifies the procedures running on the receiver's computer to match:

```
integer nextkcd initially 0           // index in array of KCDs.
```

```
procedure SECURE_PROCESS_REQUEST (h, CD)
    PROCESS (CD ⊕ KCD[nextKCD])       // decrypt and process the data on the CD
    nextKCD ← (nextKCD + 1) modulo 20      // increment with wrap-around
    h.destination ← h.source          // send to where the pigeon came from
    h.source ← MY_GPS
    h.sequence_no ← h.sequence_no     // unchanged
    h.type ← ACKNOWLEDGMENT
    send pigeon with h                // send an acknowledgment back
```

Q27.1 Do SECURE_BEEP , SECURE_PROCESS_ACK , and SECURE_PROCESS_REQUEST provide confidentiality of the data on the CDs?

A. No, since acknowledgments are not signed;
B. No, since the KCD s are reused, SECURE_BEEP is vulnerable to a known-plaintext attack;
C. Yes, since one-time pads are unbreakable;
D. No, since one can invert XOR .

To make the system more practical, Ben decides to switch to a short key and to exchange the key over the pigeon net itself instead of using an outside secure channel. Every principal has a key pair for a public-key system. He designs the following key-distribution protocol:

```
Alice ⇒ Bob: "I propose we use key k" (signed with Alice's private key)
Bob ⇒ Alice: "OK, key k is fine" (signed with Bob's private key)
```

The two key-distribution messages are written on a CD and sent with BEEP (not SECURE_BEEP). From key k the sender and receiver generate a bit string using a well-known pseudorandom number generator, and employ the bit string in SECURE_BEEP and SECURE_PROCESS_REQUEST to encrypt and decrypt CDs.

Q27.2 Which statements are true of the above protocol?

A. It is insecure because key k travels in the clear, and therefore an intruder can find out key k and listen in on future instances of SECURE_BEEP .
B. It is secure because only Bob can verify the message from Alice.
C. It is insecure because Alice's public key is widely known.
D. It is secure, since the messages are signed and key k is only used as a seed to a pseudorandom number generator.

Problem 28: WebTrust.com

2001–2–6…17

(Chapter 5)

A group of accomplished 16-year-olds wants to go into business as a technology provider. Reading many war stories about security has convinced the kid wizards that there should be a market for a secure client authentication product for Web services. The kids incorporate as WebTrust.com, and study up on how the Web works. They discover that HTTP 1.0 is a simple protocol whose essence consists of two remote procedure calls:

```
GET (document)            // returns a document
POST (document, form)     // sends a form and returns a document
```

The GET procedure gets the document identified by the Uniform Resource Locator (URL) document from a Web service. The POST procedure sends back to the service the entries that the user filled out on a form that was in a previously retrieved document. The POST procedure also gets a document. The browser invokes the POST procedure when the user hits the submit button on the form.

These remote procedure calls are sent over a reliable transport protocol, TCP. A Web browser opens a TCP connection, calls a procedure (GET or POST), and waits for a result from the service. The Web service waits for a TCP connection request, accepts the connection, and waits for a GET or POST call. Once a call arrives, the service executes it, sends the result over

the connection, and closes the connection. The browser displays the response and closes the connection on its side. Thus, a new connection must be set up for each request.

Simple URLs are of the form: `http://www.WebTrust.com/index.html`

Q28.1 " `www.WebTrust.com` " in the above URL is

A. a DNS name
B. a protocol name
C. a path name for a file
D. an Internet address

The objective of WebTrust.com's product is to authenticate users of online services. The intended use is for a user to login once per session and to allow only logged-in users access to the rest of the site. The product consists of a collection of Web pages and some server software. The company employs its own product to authenticate customers of the company's Web site.

To allow Internet users to create an account, WebTrust.com has a Web form in which a user types in a user name and two copies of a proposed password. When the user types the password, the browser doesn't echo it, but instead displays a "•" for each typed character. When the user hits the submit button, the user's browser calls the `POST` procedure to send the form to the server.

When the server receives a `CREATE_ACCOUNT` request, it makes sure that the two copies of the password match and makes sure that the proposed user name hasn't already been taken by someone else. If these conditions are true, then it creates an entry in its local password file. If either of the conditions is false, the server returns an error.

The form to create an account is stored in the document " `create.html` " on WebTrust's Web site. Another document on the server contains:

```
<a href="create.html">Create an account<\a>
```

Q28.2 What is the source of the context reference that identifies the context in which the name " `create.html` " will be resolved?

A. The browser derives a default context reference from the URL of the document that contains the relative URL.
B. It is configured in the Web browser when the browser is installed.
C. The server derives it from information it remembers about previous documents it sent to this client.
D. The user types it into the browser.

Q28.3 Why does the form for creating an account ask a user to type in the password twice?

A. To allow a password not to be echoed on the screen while enabling users to catch typos.
B. To detect transmission errors between the keyboard and the browser.
C. To reduce the probability that a packet with a password has to be retransmitted if the network deletes the packet.
D. To make it harder for users to create fake accounts.

Q28.4 In this system, to what attacks is creating an account vulnerable? (Assume an active attacker.)

A. An attacker can learn the password for a user by eavesdropping
B. An attacker can modify the password
C. An attacker can overwhelm the service by creating many accounts
D. An attacker can run a service that pretends to be " `www.WebTrust.com` "

To login, the user visits the Web page " `login.html` ", which asks the user for a user name and password. When the user hits the submit button, the browser invokes the `POST` procedure, which sends the user name and password to the service. The service checks the stored password against the password in the login request. If they match, the user is allowed to access the service; otherwise, the service returns an error.

Q28.5 To what attacks is the login procedure vulnerable? (Assume an active attacker.)

A. An attacker can login by replaying a recorded `POST` from a legitimate login
B. An attacker can login as any user by replaying a single recorded `POST` for login

C. An attacker can impersonate WebTrust.com to any registered user

D. An attacker can impersonate WebTrust.com to an unregistered user

To authenticate subsequent Web requests from a user after logging in, WebTrust.com exploits a Web mechanism called **cookies**. A service can install some state (called a cookie) in the Web browser. The service installs this state by including in a response a `SET_COOKIE` directive containing data to be stored in the cookie. WebTrust.com's use of cookies is summarized in the figure below.

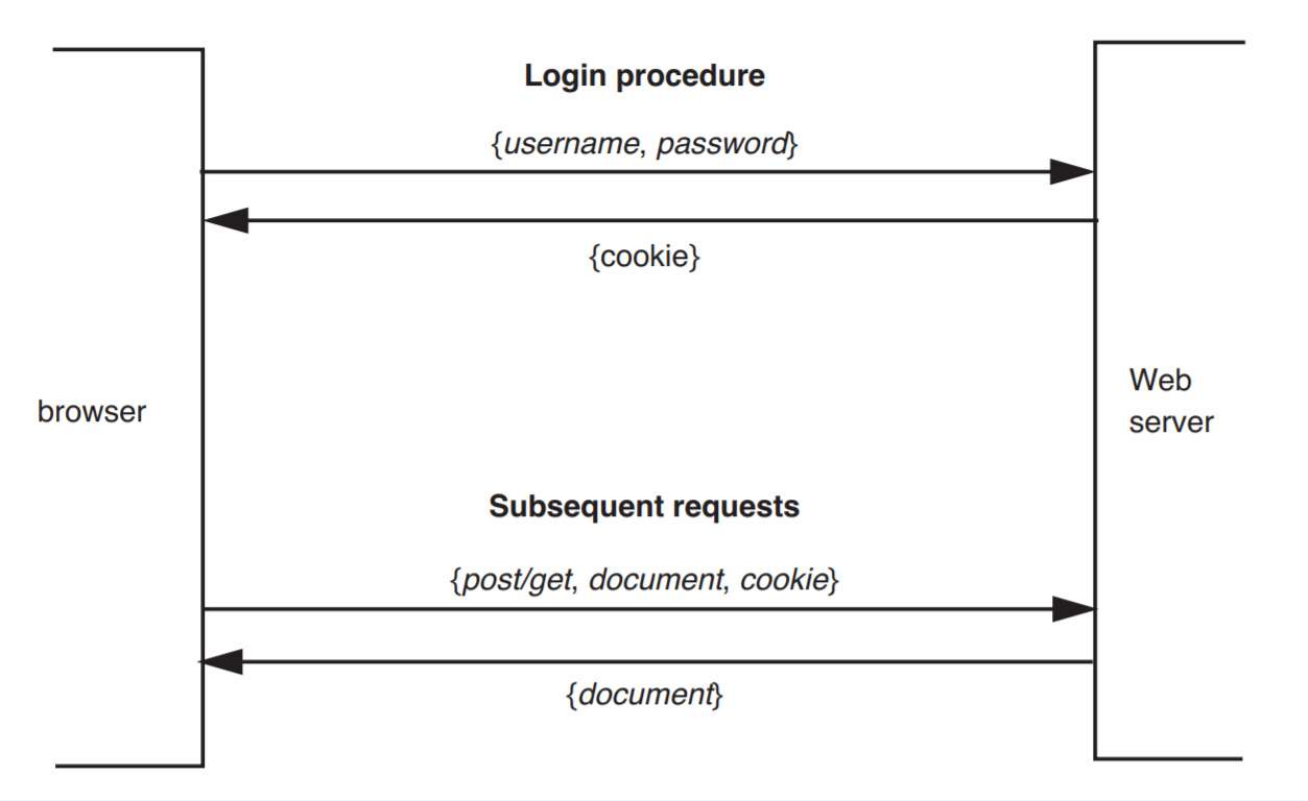


Figure $PS.28.1$: How WebTrust.com makes use of cookies.

The document containing the response to a login request comes with the directive:

```
POST (webtrustcookie)
```

The browser stores the cookie in memory. (In practice, there may be many cookies, so they are named, but for this problem, assume that there is only one and no name is needed.) On subsequent calls (i.e., `GET` or `POST`) to the service that installed the cookie, the browser sends the installed cookie along with the other arguments to `GET` or `POST` . Thus, once WebTrust.com has set a cookie in a browser, it will see that cookie on every subsequent request from that browser.

The service requires that the browser send the cookie along with all instances of `GET` , and also all instances of `POST` except those posting a `CREATE` or `LOGIN` form. If the cookie is missing (for example, the browser has lost the cookie because the client computer crashed, or an attacker is leaving the cookie out on purpose), the service will return an error to the browser and ask the user to log in again.

An important issue is to determine suitable contents for *webtrustcookie* . WebTrust.com offers a number of alternatives.

The first option is to compute the cookie as follows:

$$cookie \leftarrow \{expiration_time\}_{key}$$

using a MAC with a shared-secret *key* . The key is known only to the service, which remembers it for just 24 hours. All cookies in that period use the same key. All cookies expire at 5 a.m., at which time the service changes to a new *key* .

When the server receives the cookie, it checks it for authenticity and expiration using:

```
procedure CHECK (cookie)
    if VERIFY (cookie, key) = ACCEPT then
        if cookie.expiration_time ≤ CURRENT_TIME () then
            return ACCEPT
    return REJECT
```

The procedure `VERIFY` recomputes and checks the MAC. If the MAC is valid, then the service checks whether cookie is still fresh (i.e., if the expiration time is later than the current time). If it is, then `CHECK` returns `ACCEPT` ; the server can now execute the request. In all other cases, `CHECK` returns `REJECT` and the server returns an error to the browser.

Q28.6 What is the role of the MAC in this protocol?

A. To help detect transmission errors
B. To privately communicate key from the server to the browser
C. To privately communicate expiration-time from the server to the browser.
D. To help detect a forged cookie.

Q28.7 Which of these attacks does this protocol prevent?

A. Replayed cookies
B. Forged expiration times
C. Forged cookies
D. Dictionary attacks on passwords

Another option supported by webtrust.com is to compute cookie as follows:

$$cookie \leftarrow \{expiration_time, username\}_{key}$$

The server uses for *username* the name of the user in the login request. The usage of this cookie is similar to before and the checking procedure is unchanged.

Q28.8 If the service receives a cookie with "Alice" as username and `CHECK` returns `ACCEPT` , what does the service know? (Assume active attacks.)

A. No one modified the cookie
B. The server accepted a login from "Alice" recently
C. The cookie was generated recently
D. The browser of the user "Alice" sent this cookie recently

Q28.9 Assume temporarily that all of Alice's Web requests are sent over a single TCP connection that is encrypted and authenticated, and that the setup all has been done appropriately (i.e., only the browser and server know the encryption and authentication keys). After Alice has logged in over this connection, the server has received a cookie with "Alice" as the username over this connection, and has verified it successfully (i.e., `VERIFY` returns `ACCEPT`), what does the server know? (Assume active attacks.)

A. No one but the server and the browser of the user "Alice" knows the cookie
B. The server accepted a login from "Alice" recently
C. The cookie was generated recently
D. The browser of the user "Alice" sent this cookie recently

Q28.10 Is there any additional security risk with storing cookies durably (i.e., the browser stores them in a file on the local operating system) instead of in the run-time memory of the browser? (Assume the operating system is a multi-user operating system such as Linux or Windows, including a virtual memory system.)

A. Yes, because the file with cookies might be accessible to other users.
B. Yes, because the next user to login to the client machine might have access to the file with cookies.
C. Yes, because it expands the trusted computing base to include the local operating system.
D. Yes, because it expands the trusted computing base to include the hard disk.

Q28.11 For what applications is WebTrust's product (without the encrypting and authenticating TCP connection) appropriate (i.e., usable without grave risk)?

A. For protecting access to bank accounts of an electronic bank
B. For restricting access to electronic news articles to clients that have a subscription service
C. For protecting access to student data on a university's online student services
D. For electronic shopping, say, at amazon.com
E. None of the above

Mark Bitdiddle—Ben's 16-year kid brother—proposes to change the protocol slightly. Instead of computing cookie as:

$cookie \leftarrow \{expiration_time, username\}_{key}$

Mark suggests that the code be simplified to:

$cookie \leftarrow \{\{expiration_time\}_{key}, username\}$

He also suggests the corresponding change for the procedure `VERIFY` . The protocol, as originally, runs over an ordinary unencrypted and unauthenticated TCP connection.

Q28.12 Describe one attack that this change opens up and illustrate the attack by describing a scenario (e.g., "Lucifer can now … by …").

Problem Set 29: ByteStream Encryption

1997–2–3a…c

(Chapter 5)

Observing recent interest in security in the popular press, ByteStream Inc. decides to offer products that have the function of obtaining confidentiality by encryption. ByteStream decides to use the simple shared-secret system shown below:

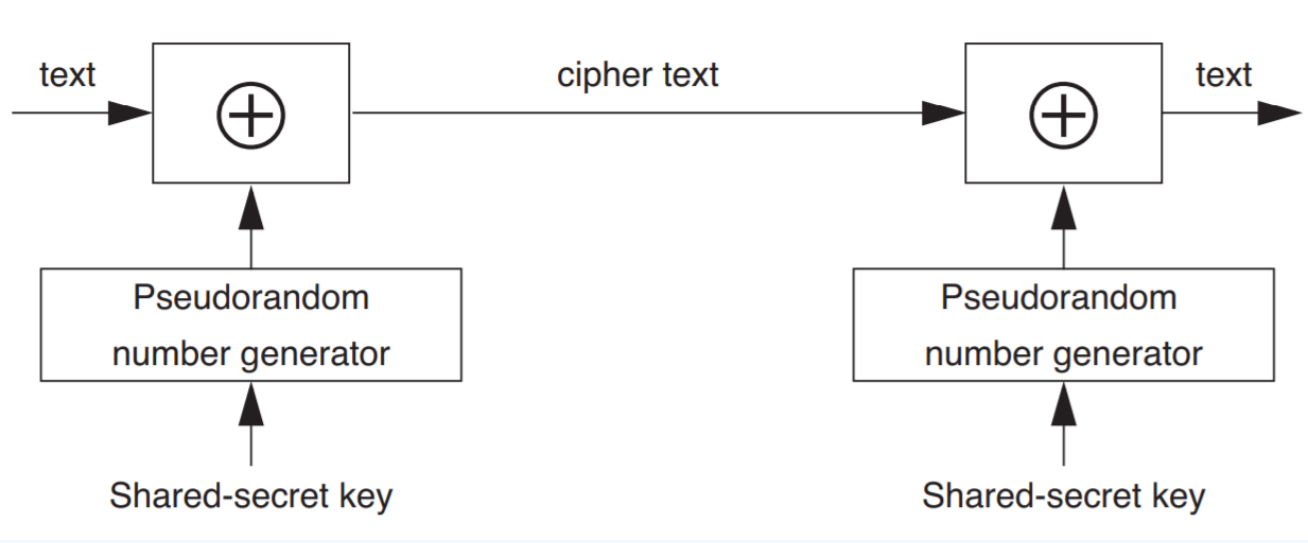


Figure $PS.29.1$: A simple shared-secret encryption/decryption system.

ByteStream uses the exclusive- `OR` (`XOR` , shown as ⊕) function. The pseudorandom number generator (PRNG) generates a stream of hard-to-predict bits, using the shared-secret key as a seed. Whenever it is seeded with the same key, it will generate the same bit stream. Messages are encrypted by computing the `XOR` of the message and the bit stream produced by the generator. The resulting ciphertext is decrypted by computing the `XOR` of the ciphertext and the bit stream produced by the PRNG, seeded with some key. The code for the PRNG is publicly available.

To check the implementation, ByteStream Inc. hires a tiger team that include Eve S. Dropper and Lucy Fer. The tiger team verifies that the code for computing the `XOR` is bug-free and the PRNG does not contain cryptographic weaknesses. The tiger team subsequently studies the following scenario. Alice shares a 200-bit key K with Bob. Alice encrypts a message with K and sends the resulting ciphertext to Bob. Bob decrypts this message with K . The result after decryption is Alice's message. Assume that every message is equally likely (i.e., Alice's message contains no redundancy whatever).

Q29.1 Given that Eve sees only the cipher text, can she cryptanalyze Alice's message?

A. No, since only Alice and Bob know the key, and the PRNG generates a 0 or 1 with equal probability, Eve has no way of telling what the content of Alice's message is.

B. Yes, since with a supercomputer Eve could try out all possible combinations of 0s and 1s for K and check whether they match the cipher text.

C. No, since it is hard to compute the `XOR` of two bit streams.

D. Yes, since `XOR` is a simple function, Eve can just compute the inverse of `XOR` .

Q29.2 Alice and Bob switch to a new shared key. Lucy mounts an active attack by tricking Alice into sending a message that begins with 500 ones, followed by Alice's original message. Given the ciphertext can Lucy cryptanalyze Alice's message?

A. Yes, since the key is smaller than 500 bits.

B. Maybe, but with probability so low that it is negligible.

C. No, since only Alice and Bob know the key and the PRNG generates a 0 or 1 with equal probability, Lucy cannot extract Alice's message.

D. No, since it is hard to compute the `XOR` of two bits.

ByteStream is interested in a product that supports two-way communication. ByteStream implements two-way communication by having one stream for requests and another stream for replies. ByteStream seeds both streams with the same key. Since ByteStream worries that using the same key in both directions might be a weakness, it asks the tiger team to check the implementation.

The tiger team studies the following scenario. Alice seeds the PRNG for the request stream with K and sends Bob a message. Upon receiving Alice's message, Bob seeds the PRNG for the reply stream with K , and sends a response to Alice. Again, assume that every request and response is equally likely.

Q29.3 What can Eve deduce about the content of the messages?

A. Nothing.

B. The content of the request, but not the reply.

C. The `XOR` of the request and the reply.

D. The content of both the request and the reply.

Problem Set 30: Stamp Out Spam*

* Credit for developing this problem set goes to Hari Balakrishnan.

2005–3–6

(Chapter 5)

Spam, defined as unsolicited messages sent in large quantities, now forms the majority of all email and short message service (SMS) traffic worldwide. Studies in 2005 estimated that about 100 billion (100×10^9) emails and SMS messages were sent per day, two-thirds of which were spam. Alyssa P. Hacker realizes that spam is a problem because it costs virtually nothing to send email, which makes it attractive for a spammer to send a large number of messages every day. Alyssa starts designing a spam control system called SOS, which uses the following approach:

A. Allocation. Every sender is given some number of stamps in exchange for payment. A newly issued stamp is fresh, while one that has been used can be cancelled to ensure that it is used only once.

B. Sending. The sender (an outgoing mail server) attaches a fresh stamp to each email message.

C. Receiving. The receiver (an incoming mail server) tests the incoming stamp for freshness by contacting a quota enforcer that runs on a trusted server using a `TEST_AND_CANCEL` remote procedure call (RPC), which is described below. If the stamp is fresh, then the receiver delivers the message to the human user. If the stamp is found to be cancelled, then the receiver discards the message as spam.

D. Quota enforcement. The quota enforcer implements the `TEST_AND_CANCEL` RPC interface for receivers to use. If the stamp was not already cancelled, the quota enforcer cancels it in this procedure by storing cancellations in a database.

Alyssa's hope is that allocating reasonable quotas to everyone and then enforcing those quotas would cripple spammers (because it would cost them a lot), while leaving legitimate users largely unaffected (because it would cost them little).

Like postage stamps, SOS's stamps need to be unforgeable, for which cryptography can help. SOS relies on a central trusted stamp authority, SA, with a well-known public key, SA_{pub}, and a corresponding private key, SA_{priv}. Each sender S generates a public/private key pair, (S_{pub}, S_{priv}), and presents S_{pub} to SA along with some payment. In return, the stamp authority SA gives sender S a certificate (C_S) and allocates it a stamp quota.

$$C_S = \{S_{pub}, expiration_time, daily_quota\}_{SApriv}$$

The notation $\{msg\}_k$ stands for the marshaling of msg and the signature (signed with key k) of msg into a buffer. We assume that signing the same message with the same key always generates the same bit string. In the certificate, $expiration_time$ is set to a time one year from the time that SA issued the certificate, and $daily_quota$ is a positive integer that specifies the maximum number of messages per day that S can send.

S is allowed to make up to $daily_quota$ stamps, each with a unique integer id between 1 and $daily_quota$, and the current date. To send a message, S constructs and attaches a stamp with the following format:

$$stamp = \{C_S, \{id, date\}_{Spriv}\}$$

When a receiver gets a stamp, it first checks that the stamp is valid by running $CHECK_STAMP_VALIDITY\ (stamp)$. This procedure verifies that C_S is a properly signed, unexpired certificate, and that the contents of the stamp have not been altered. It also checks that the id is in the range specified in C_S, and that the date is either yesterday's date or today's date (thus a stamp has a two-day validity period).

If any check fails the receiver assumes that the message is spam and discards it. If all the checks pass, then the stamp is considered valid. The receiver calls $TEST_AND_CANCEL$ on valid stamps.

Unless otherwise mentioned, assume that:

- No entity's private key is compromised.
- All of the cryptographic algorithms are computationally secure.
- SA is trusted by all participants and no aspect of its operation is compromised.
- Senders may be malicious. A malicious sender will attempt to exceed his quota; for example, he may attempt to send many messages with the same stamp, or steal another sender's unused stamps.
- Receivers may be malicious; for example, a malicious receiver may attempt to cancel stamps belonging to other senders that it has not seen.
- Most receivers cancel stamps that they have seen, especially those attached to spam messages.
- Each message has exactly one recipient (don't worry about messages sent to mailing lists).
- Spammers and other unsavory parties may mount denial-of-service and other resource exhaustion attacks on the quota enforcer, which SOS should protect against.

Alyssa implements $TEST_AND_CANCEL$ as shown below.

```
1 procedure TEST_AND_CANCEL (stamp, client)
2     // assume that client is not a spoofed network address
3     if CHECK_STAMP_VALIDITY (stamp) ≠ VALID then return
4     u ← GET (num_uses, stamp)
5     if u > 0 then status ← CANCELLED
6     else status ← FRESH
7     u ← u + 1
8     PUT(num_uses, stamp, u)
9     SEND(client, status);          // assume reliable data delivery
```

Because spammers have an incentive to reuse stamps, she wants to keep track of the total number of $TEST_AND_CANCEL$ requests done on each stamp. num_uses is a hash table keyed by stamp that keeps track of this number. The hash table supports two interfaces:

- $PUT\ (table, key, value)$ inserts the ($key, value$) pair into table.

- `GET (table, key)` returns the value associated with `key` in table, if one was previously `PUT` , and 0 otherwise. A value of 0 is never `PUT` .

Q30.1 Louis Reasoner looks at the `TEST_AND_CANCEL` procedure and declares, "Alyssa, the client would already have checked that the stamp is valid, so you don't need to call `CHECK_STAMP_VALIDITY` again." Alyssa thinks about it, and decides to keep the check. Why?

Q30.2 Suppose that a recipient R gets an email message that includes a valid stamp belonging to S. Then, which of the following assertions is true?

A. R can be certain that the email message came from S.
B. R can be certain of both the data integrity and the origin integrity of the certificate in the stamp.
C. R may be able to use the information in this stamp to cancel another stamp belonging to S with a different id.
D. If an attacker breaks into a computer that has fresh stamps on it, he may be able to use those stamps for his own messages, even though the stamps were signed by another entity.
E. S can tell whether or not R received an e-mail message by calling `TEST_AND_CANCEL` to see if the stamp attached to that message has been cancelled at the quota enforcer.
F. If S has encrypted the email message with R_{pub} , then no entity other than S or R could have read the contents of the message without S or R knowing.

The United Nations Privacy Organization looks at Alyssa's proposal and throws a fit, arguing that SOS compromises the privacy of sender-receiver email communication because the stamp authority, which also runs the quota enforcer, may be able to guess that a given sender communicated with a given receiver. Alyssa decides that the SOS protocol should be amended to meet two goals:

- G1. It should be computationally infeasible for the stamp authority (quota enforcer) to associate cancelled stamps with a sender-receiver pair.
- G2. It should still be possible for a receiver to call `TEST_AND_CANCEL` and correctly determine a stamp's freshness.

Alyssa considers several alternatives to achieve this task. Louis proposes using an encryption method he calls `DETERMINISTIC_ENCRYPT (msg, k)` , which always produces the same output string for the same `(msg, k)` input. A second scheme involves an off-the-shelf `ENCRYPT (msg, k)` that, because it adds a timestamp to the plaintext message, always produces different output for the same `(msg, k)` input. A third alternative uses `HASH (msg)` , a cryptographically secure one-way hash function of `msg` . Alyssa removes line 3 of `TEST_AND_CANCEL` so that it no longer calls `CHECK_STAMP_VALIDITY` and she checks to make sure that `TEST_AND_CANCEL` will accept any bit-string as its first argument. S_{pub} is S's public key (from the certificate in the stamp) and R_{pub} is R's public key.

Q30.3 Which of these methods achieves goals G1? Which achieves G2?

A. The receiving client R extracts $u = \{C_S, \{id, date\}_{Spriv}\}$ from the stamp, and computes $e_1 = DETERMINISTIC_ENCRYPT(u, S_{pub})$. It then calls $TEST_AND_CANCEL(e_1, R)$.

B. The receiving client R extracts $u = \{C_S, \{id, date\}_{Spriv}\}$ from the stamp, and computes $e_2 = ENCRYPT(u, R_{pub})$. It then calls $TEST_AND_CANCEL(e_2, R)$.

C. The receiving client R extracts $u = \{C_S, \{id, date\}_{Spriv}\}$ from the stamp, and computes $h = HASH(u)$. It then calls $TEST_AND_CANCEL(h, R)$.

Alyssa realizes that if SOS is to be widely used she will need several computers to run the quota enforcer to handle the daily `TEST_AND_CANCEL` load. Alyssa finds that storing the `num_uses` hash table used by `TEST_AND_CANCEL` on disk gives poor performance because the accesses to the hash table are random. When Alyssa stores this hash table in RAM, she finds that one computer can handle 50,000 `TEST_AND_CANCEL` RPCs per second on a realistic input workload, including the work required to find the machine storing the key (compared to ≈100 RPCs per second for a disk-based hash table implementation). The network connecting clients to the quota enforcer servers has extra capacity and is thus not the bottleneck.

The space required to store stamps in Alyssa's current design is rather large. She decides to save space by storing `HASH (stamp)` rather than the much larger stamp. With this optimization, storing each cancellation in the `num_uses` hash table consumes 20 bytes of space. Assume that `num_uses` stores only stamps that are from today or yesterday. Alyssa purchases computers that each have one gigabyte of RAM available for stamp storage.

Q30.4 Alyssa finds that the peak `TEST_AND_CANCEL` request rate is 10 times the average. Estimate the number of servers that Alyssa needs for SOS in order to handle 100 billion `TEST_AND_CANCEL` operations per day. (Use the approximation that there are 10^5 seconds in one day.) Be sure to consider all of the potential bottlenecks.

Alyssa builds a prototype SOS system with multiple servers. She runs multiple `TEST_AND_CANCEL` threads on each server. Alyssa wants each thread to be recoverable and for all cancelled stamps to be durable for at least two days. She also wants the different concurrent threads to be isolated from one another.

Alyssa decides that a good way to implement the quota enforcer is to use transactions. She inserts a call to `BEGIN_TRANSACTION` at the beginning of `TEST_AND_CANCEL` and a call to `COMMIT` just before the call to `SEND` . She implements a disk-based undo/redo log of updates to the *`num_uses`* hash table using the write-ahead log protocol (each disk sector write is recoverable). She uses locks for isolation.

Because all stamp cancellations are stored in RAM, Alyssa finds that a server crash loses the entire in-RAM hash table of previously cancelled stamps. A thread could also `ABORT` at any time before it `COMMIT` s (for example, the operating system could decide to `ABORT` a thread that is running too long).

Q30.5 Which of these statements about SOS's recoverability and durability is true?

A. When a thread `ABORT` s, under some circumstances, the `ABORT` procedure must undo some operations from the log.
B. When a thread `ABORT` s, under some circumstances, the `ABORT` procedure must redo some operations from the log.
C. The failure recovery process, under some circumstances, must undo some operations from the log.
D. The failure recovery process, under some circumstances, must redo some operations from the log.
E. When the failure recovery process is recovering from the log after a failure, there is no need for it to `ACQUIRE` any locks as long as no new threads run until recovery completes.

Q30.6 Recall that an important goal in SOS is to detect if any stamp is used more than once. Louis Reasoner asserts, "Alyssa, any reuse of stamps will be caught even if you don't worry about before-or-after atomicity between `TEST_AND_CANCEL` threads." Give an example to show why before-or-after atomicity is necessary.

Satisfied that her prototype works and that it can handle global message volumes, Alyssa turns to the problem of pricing stamps. Her goal is "modest": to reduce spam by a factor of 10. She realizes that her answer depends on a number of assumptions and is only a first-cut approximation.

Q30.7 Alyssa reads various surveys and concludes that spammers would be willing to spend at most US $11 million per day on sending spam. She also concludes that 66% (two-thirds) of the 100 billion daily messages sent today are spam. Under these assumptions, what should the price of each stamp be in order to reduce the number of spam messages by at least a factor of 10?

6.2.3: Problem Sets 21-30 is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

6.2.4: Problem Sets 31-32

Problem Set 31: Confidential Bitdiddler*

* Credit for developing this problem set goes to Sam Madden.

(Chapter 5)

2007–3–16

Ben Bitdiddle is designing a file system for a new handheld computer, the Bitdiddler, which is designed to be especially simple and has a file system without directories. The disk is physically partitioned into three regions: an inode list, a free list, and a collection of 4K data blocks, much like the UNIX file system. Unlike in the UNIX file system, each inode contains the name of the file it corresponds to, as well as a bit indicating whether or not the inode is in use. Like the UNIX file system, the inode also contains a list of blocks that compose the file, as well as metadata about the file, including permission bits, its length in bytes, and modification and creation timestamps. The free list is a bitmap, with one bit per data block indicating whether that block is free or in use. There are no indirect blocks in Ben's file system. The file system provides six primary calls: `CREATE`, `OPEN`, `READ`, `WRITE`, `CLOSE`, and `UNLINK`. Ben implements all six correctly and in a straightforward way. All updates to the disk are synchronous; that is, when a call to write a block of data to the disk returns, that block is definitely installed on the disk. Individual block writes are atomic.

Ben stores many files in the file system on this handheld computer, and runs out of disk space quickly. He looks at the blocks on the disk and discovers that many blocks have the same content. To reduce space consumption he augments the file system implementation as follows:

- The file system keeps a table in memory that records for each allocated block a 32-bit non-cryptographic hash of that block. (When the file system starts, it computes this table from the on-disk state.) Ben talks to a hashing expert, who tells Ben to use the b-bit (here $b = 32$) non-cryptographic hash function $H(block) = block \bmod P$ where P is a large b-bit prime number that yields a uniform distribution of hash values throughout the interval $[0... 2^b - 1]$.
- When the file system writes a block b for a file, it checks if the table contains a block number d whose block content on disk has the same hash value as the hash value for block b. If so, the file system frees b and inserts d into the file's inode. If there is no block d, the file system writes b to the disk, and puts b's block number and its hash in the table.

To keep things simple, let's ignore what happens when a user unlinks a file.

Q31.1 Occasionally, Ben finds that his system has files with incorrect contents. He suspects hash collisions are to blame. These might be caused by:

A. Accidental collisions: different data blocks hash to the same 32-bit value.
B. Engineered collisions: adversaries can fabricate blocks that hash to the same 32 bit value.
C. A block whose hash is the same as its block number.

Q31.2 For each of the following proposed fixes, list which of the problem causes listed in Question 31.1 (A, B, or C) it is likely to fix:

A. Use a $b = 160$-bit non-cryptographic hash in step A of the algorithm.
B. Use a 160-bit cryptographic hash such as SHA-1 in step A of the algorithm.
C. Modify step B of the algorithm so that when a matching hash is found, it compares the contents of the stored block to the data block and treats the blocks as different unless their contents match.

Ben decides he wants to encrypt the contents of the files on disk so that if someone steals his handheld computer, they cannot read the files stored on it. Ben considers two encryption plans:

- **User-key encryption**: One plan is to give each user a different key and use a secure block encryption scheme with no cryptographic vulnerabilities to encrypt the user's files. Ben implements this by storing a table of (user name, key) pairs, which the system stores securely on disk.

- **Convergent encryption**: One problem with user-key encryption is that it doesn't provide the space saving if blocks in different files of different users have the same content. To address this problem, Ben proposes to use **convergent encryption** (also called "content hash keying"), which encrypts a block using a cryptographic hash of the content of that block as a shared-secret key (that is, `ENCRYPT (block, HASH (block))` . Ben reasons that since the output of the cryptographic hash is pseudorandom, this is just as good as choosing a fresh random key. Ben implements this scheme by modifying the file system to use the table of hash values as before, but now the file system writes encrypted blocks to the disk instead of plaintext ones. This way blocks are encrypted but, because duplicate blocks have the same hash and thus encrypt to the same ciphertext, Ben still gets the space savings for blocks with the same content. The file system maintains a secure table of block hash values so that it can decrypt blocks when an authorized user requests a read operation.

  Q31.3 Which of the following statements are true of convergent encryption?

  A. If Alyssa can guess the contents of a block (by enumerating all possibilities, or by guessing based on the file metadata, etc), it is easy for her to verify whether her guess of a block's data is correct.
  B. If Alyssa can discover the 32-bit block numbers referenced by inodes in the file system, she can learn something about the contents of Ben's files.
  C. The file system can detect when an adversary changes the content of a block on disk.

  Q31.4 Which of the following statements are true of user-key encryption?

  A. If Alyssa can guess the contents of a block but doesn't know Ben's key, it is easy for her to verify whether her guess of a block's data is correct.
  B. If Alyssa can discover the 32-bit block numbers referenced by inodes in the file system, she can learn something about the contents of Ben's files.
  C. The file system can detect when an adversary changes the content of a block on disk.

Problem Set 32: Beyond Stack Smashing*

* Credit for developing this problem set goes to Lewis D. Girod. This problem set was inspired by a paper by Jonathan Pincus and Brandon Baker, "Beyond stack smashing: recent advances in exploiting buffer overruns," *IEEE Security & Privacy 2*, 4 (July/August 2004) pages 20–27. Further details and explanations can be found in that paper.

(Chapter 5)

2008–3-8

You are hired by a well-known OS vendor to help them defend their products against buffer overrun attacks of the kind described in Sidebar 5.2.4. Their team presents several proposed strategies to foil buffer exploits:

- **Random stack**: Place the stack in an area of memory randomly chosen for each new process, rather than at the same address for every process.
- **Non-executable stack**: Set the permissions on the virtual memory containing the stack to allow reading and writing but not execution as a program. Set the permissions on the memory containing the program instructions to read and execute but not write.
- **Bounds checking**: Use a language such as Java or Scheme that checks that all array/buffer indices are valid.

You are aware of several buffer overrun attacks, including the following:

- **Simple buffer overrun**: The victim program has an array on the stack as follows:

```
procedure VICTIM (data, length)
    integer buffer[100]
    COPY (buffer, data, length) // overruns array buffer if length > 100
    // ....
```

  The attacker supplies a `length` > 100 together with an array data that includes some new instructions and places the address of the first instruction in the position where the procedure return is stored. When `VICTIM` reaches its `return` , it returns to the attacker's code in the stack rather than the program that originally called `VICTIM` .

- **Trampoline**: The victim program has an array on the stack as in the code fragment above, but the attacker cannot predict its address, so replacing the procedure return address with the address of the attacker's code won't work. However, the attacker knows that subroutine `VICTIM ()` leaves an address in some register (say `R5`) that points to a known, fixed offset within the array. The other thing that is needed is an instruction anywhere in memory at a known address *x* that jumps to wherever `R5` is pointing. The attacker overruns the array with his new instructions and overwrites the procedure return address with the address *x* . When `VICTIM` reaches its return, it returns to address *x* , which jumps to the address in `R5` , which transfers control to the attacker's code.
- **Arc injection** (return-to-library): Taking advantage again of knowledge that `VICTIM` leaves the address of a fixed offset within the array in register `R5` , the attacker provides some carefully selected data at that offset and also overruns the buffer with a new procedure return address. The new procedure return address is chosen to be in some system or library program known to reside elsewhere in the current address space, preferably to a place within that library program after it has checked the validity of its parameters, and is about to do something using the contents of register `R5` as the address of one of its parameters. A particularly good library program to jump into is one that calls a procedure whose string name is supplied as an argument. The attacker's carefully selected data is chosen to be the string name of an existing program that the attacker would like to execute.

In the following questions, an attack is considered **prevented** if the attacker can no longer execute the intended malicious code, even if an overflow can still overwrite data or crash or disrupt the program.

Q32.1 Which of the following attack methods are prevented by the use of the **random** stack technique?

A. Simple buffer overrun
B. Trampoline
C. Arc injection (return-to-library)

Q32.2 Which of the following attack methods are prevented by the use of the **non-executable** stack technique?

A. Simple buffer overrun
B. Trampoline
C. Arc injection (return-to-library)

Q32.3 Which of the following attack methods are prevented by the use of the bounds checking technique?

A. Simple buffer overrun
B. Trampoline
C. Arc injection (return-to-library)

6.2.4: Problem Sets 31-32 is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

6.3: Suggestions for Further Reading

Chapter 1: The Network as a System and as a System Component

Proceedings of the IEEE 66, 11 (November 1978), a special issue of that journal devoted to packet switching, contains several papers mentioned under various topics here. Collectively, they provide an extensive early bibliography on computer communications.

Topic 1.1: Networks

1.1.1 Radia Perlman. *Interconnections, Second Edition: Bridges, Routers, Switches, and Internetworking Protocols*. Addison-Wesley, 1999. ISBN: 978–0–201–63448–8. 560 pages.

> This book presents everything you could possibly want to know about how the network layer actually works. The style is engagingly informal, but the content is absolutely first-class, and every possible variation is explored. The previous edition was simply titled *Interconnections: Bridges and Routers*.

1.1.2 David D. Clark, Kenneth T. Pogran, and David P. Reed. An introduction to local area networks. *Proceedings of the IEEE 66*, 11 (November 1978), pages 1497–1517.

> This basic tutorial on local area network communications characterizes the various modular components of a local area network, both interface and protocols, gives specific examples, and explains how local area networks relate to larger, interconnected networks. The specific examples are now out of date, but the rest of the material is timeless.

1.1.3 Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM 19*, 7 (July 1976), pages 395–404.

> This paper provides the design of what has proven to be the most popular local area network technology.

Topic 1.2: Protocols

1.2.1 Louis Pouzin and Hubert Zimmerman. A tutorial on protocols. *Proceedings of the IEEE 66*, 11 (November 1978), pages 1346–1370.

> This paper is well written and provides perspective along with the details. The fact that it was written a long time ago turns out to be its major appeal. Because networks were not widely understood at the time, it was necessary to fully explain all of the assumptions and offer extensive analogies. This paper does an excellent job of both, and as a consequence it provides a useful complement to modern texts. While reading this paper, anyone who is familiar with current network technology will frequently exclaim, "So that's why the Internet works that way."

1.2.2 Vinton G. Cerf and Peter T. Kirstein. Issues in packet-network interconnection. *Proceedings of the IEEE 66*, 11 (November 1978), pages 1386–1408.

> At the time this paper was written, an emerging problem was the interconnection of independently administered data communication networks. This paper explores the issues in both breadth and depth, a combination that more recent papers do not provide.

1.2.3 David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. *ACM SIGCOMM '91 Conference: Communications Architectures and Protocols*, in *Computer Communication Review 20*, 4 (September 1990), pages 200–208.

> This paper captures 20 years of experience in protocol design and implementation and lays out the requirements for the next few rounds of protocol design. The basic observation is that the performance requirements of future high-speed networks and applications will require that the layers used for protocol description not constrain implementations to be similarly layered. This paper is required reading for anyone who is developing a new protocol or protocol suite.

1.2.4 Danny Cohen. On holy wars and a plea for peace. *IEEE Computer 14*, 10 (October 1981), pages 48–54.

> This is an entertaining discussion of big-endian and little-endian arguments in protocol design.

1.2.5 Danny Cohen. Flow control for real-time communication. *Computer Communication Review 10*, 1–2 (January/April 1980), pages 41–47.

This brief item is the source of the "servant's dilemma", a parable that provides helpful insight into why flow control decisions must involve the application.

1.2.6 Geoff Huston. Anatomy: A look inside network address translators. *The Internet Protocol Journal 7*, 3 (September 2004), pages 2–32.

Network address translators (NATs) break down the universal connectivity property of the Internet: when NATs are in use one, can no longer assume that every computer in the Internet can communicate with every other computer in the Internet. This paper discusses the motivation for NATs, how they work, and in what ways they create havoc for some Internet applications.

1.2.7 Van Jacobson. Congestion avoidance and control. *Proceedings of the Symposium on Communications Architectures and Protocols* (SIGCOMM '88), pages 314–329. Also in *Computer Communication Review 18*, 4 (August 1988).

Sidebar 1.7.2 gives a simplified description of the congestion avoidance and control mechanism of TCP, the most commonly used transport protocol in the Internet. This paper explains those mechanisms in full detail. They are surprisingly simple but have proven to be effective.

1.2.8 Jordan Ritter. Why Gnutella can't scale. No, really. Unpublished grey literature. <http://www.darkridge.com/~jpr5/doc/gnutella.html.>

This paper offers a simple performance model to explain why the Gnutella protocol (see problem set 20) cannot support large networks of Gnutella peers. The problem is incommensurate scaling of its bandwidth requirements.

1.2.9 David B. Johnson. Scalable support for transparent mobile host internetworking. *Wireless Networks 1*, 3 (1995), pages 311–321.

Addressing a laptop computer that is connected to a network by a radio link and that can move from place to place without disrupting network connections can be a challenge. This paper proposes a systematic approach based on maintaining a tunnel between the laptop computer's current location and an agent located at its usual home location. Variations of this paper (based on the authors 1993 Ph.D. thesis at Carnegie-Mellon University and available as CMU Computer Science Technical Report CS–93–128) have appeared in several 1993 and 1994 workshops and conferences, as well as in the book *Mobile Computing*, Tomasz Imielinski and Henry F. Korth, editors, Kluwer Academic Publishers, c. 1996. ISBN: 079239697–9.

Subtopic 1.2.1: Remote Procedure Call

1.2.1.1 Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems 2*, 1 (February 1984), pages 39–59.

A well-written paper that shows first, the simplicity of the basic idea, second, the complexity required to deal with real implementations, and third, the refinements needed for high effectiveness.

1.2.1.2 Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, third edition, 2008. ISBN 978–0–13–600663-3 (hardcover). 952 pages.

This book provides a thorough tutorial introduction to the world of operating systems but with a tendancy to emphasize the mechanics. Insight into why things are designed the way they are is there, but in many cases requires teasing out. Nevertheless, as a starting point, it is filled with street knowledge that is needed to get into the rest of the literature. It includes useful case studies of GNU/Linux, Windows Vista, and Symbian OS, an operating system for mobile phones. Section 4.4 covers the protocol of remote procedure call in detail.

Topic 1.3: Organization for communication

1.3.1 Leonard Kleinrock. Principles and lessons in packet communications. *Proceedings of the IEEE 66*, 11 (November 1978), pages 1320–1329.

1.3.2 Lawrence G. Roberts. The evolution of packet switching. *Proceedings of the IEEE 66*, 11 (November 1978), pages 1307–1313.

These two papers discuss experience with the ARPANET. Anyone faced with the need to design a network should look over these two papers, which focus on lessons learned and the sources of surprise.

1.3.3 J[erome] H. Saltzer, D[avid]. P. Reed, and D[avid]. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems 2*, 4 (November 1984), pages 277–288. An earlier version appears in the *Proceedings of the Second International Conference on Distributed Computing Systems* (April 1981), pages 504–512.

This paper proposes a design rationale for deciding which functions belong in which layers of a layered network implementation. It is one of the few papers available that provides a system design principle.

1.3.4 Leonard Kleinrock. The latency/bandwidth trade-off in gigabit networks. *IEEE Communications Magazine 30*, 4 (April 1992), pages 36–40.

Technology has made gigabit/second data rates economically feasible over long distances. But long distances and high data rates conspire to change some fundamental properties of a packet network—latency becomes the dominant factor that limits applications. This paper provides a good explanation of the problem.

Topic 1.4: Practical aspects

For the complete word on the Internet protocols, check out the following series of books.

1.4.1 W. Richard Stevens. *TCP/IP Illustrated*. Addison-Wesley; v. 1, 1994, ISBN 0–201–63346–9, 576 pages; v. 2 (with co-author Gary R. Wright) 1995, ISBN 0–201–63354–x, 1174 pages.; v. 3, 1996, ISBN 0–201–63495–3, 328 pages. *Volume 1: The Protocols*. *Volume 2: The Implementation*. *Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX® Domain Protocols*.

These three volumes will tell you more than you wanted to know about how TCP/IP is implemented, using the network implementation of the Berkeley System Distribution for reference. The word "illustrated" refers more to computer printouts —listings of packet traces and programs—than to diagrams. If you want to know how some aspect of the Internet protocol suite is actually implemented, this is the place to look—though it does not often explain why particular implementation choices were made.

Chapter 2: Fault Tolerance

Topic 2.1: Fault tolerance

2.1.1 Jim [N.] Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, California, 1993 (Look for the low-bulk paper edition, which became available with the third printing in 1994). ISBN: 978–1–55860–190–1. 1,070 pages.

All aspects of fault tolerance, atomicity, coordination, recovery, rollback, logs, locks, transactions, and engineering trade-offs for performance are pulled together in this comprehensive book. This is the definitive work on transactions. Though not intended for beginners, given the high quality of its explanations, this complex material is surprisingly accessible. The glossary of terms is excellent, whereas the historical notes are good as far as they go, but are somewhat database-centric and should not be taken as the final word. In particular, Chapter 3 provides a bedrock text on the subject of fault tolerance.

2.1.2 Jim [N.] Gray and Daniel P. Siewiorek. High-availability computer systems. *Computer 24*, 9 (September 1991), pages 39–48.

This is a nice, easy-to-read overview of how high availability can be achieved.

2.1.3 Daniel P. Siewiorek. Architecture of fault-tolerant computers. *Computer 17*, 8 (August 1984), pages 9–18.

This paper provides an excellent taxonomy, as well as a good overview of several architectural approaches to designing computers that continue running even when a single hardware component fails.

Topic 2.2: Software errors

2.2.1 Dawson Engler et al. Bugs as deviant behavior: A general approach to inferring errors in systems code. *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001, in *Operating Systems Review 35*, 5 (December 2001), pages 57–72.

This paper describes a method for finding possible programming faults in large systems by looking for inconsistencies. For example, if in most cases an invocation of a certain function is preceded by disabling interrupts but in a few cases it is not, there is a good chance that a programming fault is present. The paper uses this insight to create a tool for finding potential faults in large systems.

2.2.2 Michael M. Swift et al. Recovering device drivers. *Proceedings of the Sixth Symposium on Operating System Design and Implementation* (December 2004), pages 1–16.

> This paper observes that software faults in device drivers often lead to fatal errors that cause operating systems to fail and thus require a reboot. It then describes how virtual memory techniques can be used to enforce modularity between device drivers and the rest of the operating system kernel, and how the operating system can recover device drivers when they fail, reducing the number of reboots.

Topic 2.3: Disk failures

2.3.1 Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? *Proceedings of the fifth USENIX Conference on File and Storage Technologies* (2007), pages 1–16.

> As explained in Section 2.2, it is not uncommon that data sheets for disk drives specify MTTFs of one hundred years or more, many times the actual observed lifetimes of those drives in the field. This paper looks at disk replacement data for 100,000 disk drives and discusses what MTTF means for those disk drives.

2.3.2 Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Andre Barroso. Failure trends in a large disk drive population. *Proceedings of the fifth USENIX Conference on File and Storage Technologies* (2007), pages 17–28.

> Recently, outfits such as Google have deployed large enough numbers of off-the-shelf disk drives for a long enough time that they can make their own evaluations of disk drive failure rates and lifetimes, for comparison with the *a priori* reliability models of the disk vendors. This paper reports data collected from such observations. It analyzes the correlation between failures and several parameters that are generally believed to impact the lifetime of disk and finds some surprises. For example, it reports that temperature is less correlated with disk drive failure than was previously reported, as long as the temperature is within a certain range and stable.

Chapter 3: Atomicity

Topic 3.1: Atomicity, Coordination, and Recovery

The best source on this topic is the book by Gray and Reuter listed above as reading number **2.1.1**, but at a thousand pages long it can be a bit overwhelming.

3.1.1 Warren A. Montgomery. *Robust Concurrency Control for a Distributed Information System*. Ph.D. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, December 1978. Also available as M.I.T. Laboratory for Computer Science Technical Report TR–207, January 1979. 197 pages.

> This work describes alternative strategies that maximize concurrent activity while achieving atomicity: maintaining multiple values for some variables, atomic broadcast of messages to achieve proper sequence.

3.1.2 D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *Proceedings of an ACM Conference on Language Design for Reliable Software* (March 1977), pages 128–137. Published as *ACM SIGPLAN Notices 12*, 3 (March 1977); *Operating Systems Review 11*, 2 (April 1977); and *Software Engineering Notes 2*, 2 (March 1977).

> This is one of the first attempts to link atomicity to both recovery and coordination. It is written from a language, rather than an implementation, perspective.

Topic 3.2: Databases

3.2.1 Jim [N.] Gray et al. The recovery manager of the System R database manager. *ACM Computing Surveys 13*, 2 (June 1981), pages 223–242.

> This paper is a case study of a sophisticated, real, high-performance logging and locking system. It is one of the most interesting case studies of its type because it shows the number of different, interacting mechanisms needed to construct a system that performs well.

3.2.2 C. Mohan et al. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems 17*, 1 (1992), pages 94-162.

This paper describes all the intricate design details of a fully featured, commercialquality database transaction system that uses write-ahead logging.

3.2.3 C. Mohan, Bruce Lindsey, and Ron Obermarck. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems (TODS) 11*, 4 (December 1986), pages 378–396.

This paper deals with transaction management for distributed databases, and introduces two new protocols (Presumed Abort and Presumed Commit) that optimize two-phase commit (see Section 3.7), resulting in fewer messages and log writes. Presumed Abort is optimized for transactions that perform only read operations, and Presumed Commit is optimized for transactions with updates that involve several distributed databases.

3.2.4 Tom Barclay, Jim Gray, and Don Slutz. Microsoft TerraServer: A spatial data warehouse. *Microsoft Technical Report MS-TR-99-29*. June 1999.

The authors report on building a popular Web site that hosts aerial, satellite, and topographic images of Earth using off-the-shelf components, including a standard database system for storing the terabytes of data.

3.2.5 Ben Vandiver et al. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. *Proceedings of the Twenty-first ACM Symposium on Operating Systems Principles*, in *Operating Systems Review 41*, 6 (December 2005), pages 59–79.

This paper describes a replication scheme for handling Byzantine faults in database systems. It issues queries and updates to multiple replicas of unmodified, off-the-shelf database systems, and it compares their responses, thus creating a single database that is Byzantine fault tolerant (see Section 2.7 for the definition of *Byzantine*).

Topic 3.3: Atomicity-related topics

3.3.1 Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (February 1992), pages 26–52. Originally published in *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, in *Operating Systems Review 25*, 5 (December 1991), pages 1–15.

Although it has long been suggested that one could in principle store the contents of a file system on disk in the form of a finite log, this design is one of the few that demonstrates the full implications of that design strategy. The paper also presents a fine example of how to approach a system problem by carefully defining the objective, measuring previous systems to obtain a benchmark, and then comparing performance as well as functional aspects that cannot be measured.

3.3.2 H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems 9*, 4 (June 1981), pages 213–226.

This early paper introduced the idea of using optimistic approaches to controlling updates to shared data. An optimistic scheme is one in which a transaction proceeds in the hope that its updates are not conflicting with concurrent updates of another transaction. At commit time, the transaction checks to see if the hope was justified. If so, the transaction commits. If not, the transaction aborts and tries again. Applications that use a database in which contention for particular records is infrequent may run more efficiently with this optimistic scheme than with a scheme that always acquires locks to coordinate updates.

3.3.3 Butler W. Lampson and Howard Sturgis. Crash recovery in a distributed data storage system. Working paper, Xerox Palo Alto Research Center, November 1976, and April 1979. (Never published)

Jim Gray called the 1976 version of this paper "an underground classic." The 1979 version presents the first good definition of models of failure. Both describe algorithms for coordinating distributed updates; they are sufficiently different that both are worth reading.

3.3.4 Gregory R. Ganger and Yale N. Patt. Metadata update performance in file systems. *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation* (November 1994), pages 49–60.

This paper is an application to file systems of some recovery and consistency concepts originally developed for database systems. It describes a few simple rules (e.g., an inode should be written to the disk after writing the disk blocks to which it points) that allow a system designer to implement a file system that is high performance and always keeps its on-disk data structures consistent in the presence of failures. As applications perform file operations, the rules create dependencies between data blocks in the write-behind cache. A disk driver that knows about these dependencies can write the cached blocks to disk in an order that maintains consistency of on-disk data structures despite system crashes.

Chapter 4: Consistency and Durable Storage

Topic 4.1: Consistency

4.1.1 J. R. Goodman. Using cache memory to reduce processor-memory traffic. *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124–132 (1983).

The paper that introduced a protocol for cache-coherent shared memory using snoopy caches. The paper also sparked much research in more scalable designs for cache-coherent shared memory

4.1.2 James J. Kistler and M[ahadarev] Satyanarayanan. Disconnected operation in the Coda file system. *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, in *Operating Systems Review 25*, 5 (December 1991), pages 213–225.

Coda is a variation of the Andrew File System (AFS) that provides extra fault tolerance features. It is notable for using the same underlying mechanism to deal both with accidental disconnection due to network partition and the intentional disconnection associated with portable computers. This paper is well written.

4.1.3 Jim Gray et al. The dangers of replication and a solution. *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, in *ACM SIGMOD Record 25*, 2 (June 1996), pages 173–182.

This paper describes the challenges for replication protocols in situations where the replicas are stored on mobile computers that are frequently disconnected. The paper argues that trying to provide transactional semantics for an optimistic replication protocol in this setting is unstable because there will be too many reconciliation conflicts. It proposes a new two-tier protocol for reconciling disconnected replicas that addresses this problem.

4.1.4 Leslie Lamport. Paxos made simple. Distributed computing (column), *ACM SIGACT News 32*, 4 (Whole Number 121, December 2001), pages 51–58.

This paper describes an intricate protocol, Paxos, in a simple way. The Paxos protocol allows several computers to agree on a value (e.g., the list of available computers in a replicated service) in the face of network and computer failures. It is an important building block in building fault-tolerant services.

4.1.5 Fred Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys 22*, 4 (1990), pages 299–319.

This paper provides a clear description of one of the most popular approaches for building fault tolerant services, the replicated-state machine approach.

4.1.6 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), pages 558–565.

This paper introduces an idea that is now known as Lamport clocks. A Lamport clock provides a global, logical clock for a distributed system that respects the physical clocks of the computers comprising the distributed system and the communication between them. The paper also introduces the idea of replicated state machines.

4.1.7 David K. Gifford. Weighted voting for replicated data. *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, in *Operating Systems Review 13*, 5 (December 1979), pages 150–162. Also available as Xerox Palo Alto Research Center Technical Report CSL–79–14 (September 1979).

The work discusses a replicated data algorithm that allows the trade-off between reliability and performance to be adjusted by assigning weights to each data copy and requiring transactions to collect a quorum of those weights before reading or writing.

4.1.8 Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer System 7*, 4 (November 1989), pages 321–359.

This paper describes a method to create a shared virtual memory across several separated computers that can communicate only with messages. It uses hardware support for virtual memory to cause the results of a write to a page to be observed by readers of that page on other computers. The goal is to allow programmers to write parallel applications on a distributed computer system in shared-memory style instead of a message-passing style.

4.1.9 Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (October 2003), pages 29–43. Also in *Operating Systems Review 37*, 5 (December 2003).

> This paper introduces a file system used in many of Google's applications. It aggregates the disks of thousands of computers in a cluster into a single storage system with a simple file system interface. Its design is optimized for large files and replicates files for fault tolerance. The Google File System is used in the storage back-end of many of Google's applications, including Google search.

4.1.10 F[ay] Chang et al. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems 26*, 2, article 4 (2008), pages 1–26.

> This paper describes a database-like system for storing petabytes of structured data on thousands of commodity servers.

Topic 4.2: Durable Storage

4.2.1 Raymond A. Lorie. The long-term preservation of digital information. *Proceedings of the first ACM/IEEE Joint Conference on Digital Libraries* (2001), pages 346–352.

> This is a thoughtful discussion of the problems of archiving digital information despite medium and technology obsolescence.

4.2.2 Randy H. Katz, Garth A. Gibson, and David A. Patterson. Disk system architectures for high performance computing. *Proceedings of the IEEE 77*, 12 (December 1989), pages 1842–1857.

> The first part of this reference paper on Redundant Arrays of Independent Disks (RAID) reviews disk technology; the important material is the catalog of six varieties of RAID organization.

4.2.3 Petros Maniatis et al. LOCKSS: A peer-to-peer digital preservation system. *ACM Transactions on Computer Systems 23*, 1 (February 2005), pages 2–50.

> This paper describes a peer-to-peer system for preserving access to journals and other archival information published on the Web. Its design is based on the mantra "lots of copies keep stuff safe" (LOCKSS). A large number of persistent Web caches keep copies and cooperate to detect and repair damage to their copies using a new voting scheme.

4.2.4 A[lan J.] Demers et al. Epidemic algorithms for replicated database maintenance. *Proceedings of the Sixth Symposium on Principles of Distributed Computing* (August 1987), pages 1-12. Also in *Operating Systems Review 22*, 1 (January 1988), pages 8-32.

> This paper describes an epidemic protocol to update data that is replicated on many machines. The essence of an epidemic protocol is that each computer periodically gossips with some other, randomly chosen computer and exchanges information; multiple computers thus learn about all updates in a viral fashion. Epidemic protocols can be simple and robust, yet can spread updates relatively quickly.

Topic 4.3: Reconciliation

4.3.1 Douglas B. Terry et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. *Proceedings of the Fifteenth Symposium on Operating Systems Principles* (December 1995), in *Operating Systems Review 29*, 5 (December 1995), pages 172–183.

> This paper introduces a replication scheme for computers that share data but are not always connected. For example, each computer may have a copy of a calendar, which it can update optimistically. Bayou will propagate these updates, detect conflicts, and attempt to resolve conflicts, if possible.

4.3.2 Trevor Jim, Benjamin C. Pierce, and Jérôme Vouillon. How to build a file synchronizer. (A widely circulated piece of grey literature—dated February 22, 2002 but never published.)

This paper describes the nuts and bolts of Unison, a tool that efficiently synchronizes the files stored on two computers. Unison is targeted to users who have their files stored in several places (e.g., on a server at work, a laptop to carry while traveling, and a desktop at home) and would like to have all the files on the different computers be the same.

Chapter 5: Information Security

Topic 5.1: Privacy

5.1.1 Alan F. Westin. *Privacy and Freedom*. Atheneum Press, 1967. 487 pages. (Out of print.)

If you have any interest in privacy, track down a copy of this book in a library or used-book store. It is the comprehensive treatment, by a constitutional lawyer, of what privacy is, why it matters, and how it is positioned in the U.S. legal framework.

5.1.2 Arthur R. Miller. *The Assault on Privacy*. University of Michigan Press, Ann Arbor, Michigan, 1971. ISBN: 0–47265500–0. 333 pages. (Out of print.)

This book articulately spells out the potential effect of computerized data-gathering systems on privacy, and of possible approaches to improving legal protection. Part of the latter is now out of date because of advances in legislation, but most of this book is still of much interest.

5.1.3 Daniel J. Weitzner et al. Information accountability. *Communications of the ACM 51*, 6 (June 2008), pages 82–87.

The paper suggests that in the modern world, Westin's definition covers only a subset of privacy. See Sidebar 5.2.1 for a discussion of the paper's proposed extended definition.

Topic 5.2: Protection Architectures

5.2.1 Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (September 1975), pages 1278–1308.

After more than 30 years, this paper (an early version of the current Chapter 5) still provides an effective treatment of protection mechanics in multiuser systems. Its emphasis on protection inside a single system, rather than between systems connected to a network, is one of its chief shortcomings, along with antique examples and omission of newer techniques of certification such as authentication logic.

5.2.2 R[oger] M. Needham. Protection systems and protection implementations. *AFIPS Fall Joint Conference 41*, Part I (December 1972), pages 571–578.

This paper is probably as clear an explanation of capability systems as one is likely to find. For another important paper on capabilities, see reading 5.2.3 below, by Fabry.

5.2.3 R[obert] S. Fabry. Capability-based addressing. *Communications of the ACM 17*, 7 (July 1974), pages 403–412.

This is the first comprehensive treatment of capabilities, a mechanism introduced to enforce modularity but actually more of a naming feature.

Topic 5.3: Certification, Trusted Computer Systems and Security Kernels

5.3.1 Butler [W.] Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems 10*, 4 (November 1992), pages 265–310.

This paper, one of a series on a logic that can be used to reason systematically about authentication, provides a relatively complete explication of the theory and shows how to apply it to the protocols of a distributed system.

5.3.2 Edward Wobber, Martín Abadi, Michael Burrows, and Butler W. Lampson. Authentication in the Taos operating system. *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, in *Operating Systems Review 27*, 5 (December 1993), pages 256–269.

This paper applies the authentication logic developed in reading 5.3.1 to an experimental operating system. In addition to providing a concrete example, the explanation of the authentication logic itself is a little more accessible than that in the other paper.

5.3.3 Ken L. Thompson. Reflections on trusting trust. *Communications of the ACM 27*, 8 (August 1984), pages 761–763.

Anyone seriously interested in developing trusted computer systems should think hard about the implications for verification that this paper raises. Thompson demonstrates the ease with which a compiler expert can insert undetectable Trojan Horses into a system. Reading 5.3.4 describes a way to detect a Trojan horse. [The original idea that Thompson describes came from a paper whose identity he could not recall at the time, and which is credited with a footnote asking for help locating it. The paper was a technical report of the United States Air Force Electronic Systems Division at Hanscom Air Force Base. Paul A.

Karger and Roger R. Schell. *Multics Security Evaluation: Vulnerability Analysis*. *ESD–TR–74–193, Volume II* (June 1974), page 52.]

5.3.4 David A. Wheeler. Countering trusting trust through diverse double-compiling. *Proceedings of the 21st Annual Computer Security Applications Conference* (2005), pages 28–40.

This paper proposes a solution that the author calls "diverse double compiling", to detect the attack discussed in Thompson's paper on trusting trust (see reading 5.3.3). The idea is to recompile a new, untrusted compiler's source code twice: first using a trusted compiler, and second using the result of this compilation. If the resulting binary for the compiler is bit-for-bit identical with the untrusted compiler's original binary, then the source code accurately represents the untrusted binary, which is the first step in developing trust in the new compiler.

5.3.5 Paul A. Karger et al. A VMM security kernel for the VAX architecture. *1990 IEEE Computer Society Symposium on Security and Privacy* (May 1990), pages 2–19.

In the 1970s, the U.S. Department of Defense undertook a research effort to create trusted computer systems for defense purposes and in the process created a large body of literature on the subject. This paper distills most of the relevant ideas from that literature into a single, readable case study, and it also provides pointers to other key papers for those seeking more details on these ideas.

5.3.6 David D. Clark and David. R. Wilson. A comparison of commercial and military computer security policies. *1987 IEEE Symposium on Security and Privacy* (April 1987), pages 184–194.

This thought-provoking paper outlines the requirements for security policy in commercial settings and argues that the lattice model is often not applicable. It suggests that these applications require a more object-oriented model in which data may be modified only by trusted programs.

5.3.7 Jaap-Henk Hoepman and Bart Jacobs. Increased security through open source. *Communications of the ACM 50*, 1 (January 2007), pages 79–83.

It has long been argued that the open design principle (see Section 5.2.4) is important to designing secure systems. This paper extends that argument by making the case that the availability of source code for a system is important in ensuring the security of its implementation.

5.3.8 Simson Garfinkel and Gene [Eugene H.] Spafford. *Practical UNIX and Internet Security*. O'Reilly & Associates, Sebastopol, California, third edition, 2003. ISBN 978–59600323–4 (paperback). 986 pages.

This is a really comprehensive guide to how to run a network-attached UNIX system with some confidence that it is relatively safe against casual intruders. In addition to providing practical information for a system manager, it incidentally gives the reader quite a bit of insight into the style of thinking and design needed to provide security.

5.3.9 Butler W. Lampson and Howard E. Sturgis. Reflections on an operating system design. *Communications of the ACM 19*, 5 (May 1976), pages 251–265.

An operating system named CAL, designed at the University of California at Berkeley, appears to be the first system to make explicit use of types in the interface to the operating system. In addition to introducing this idea, Lampson and Sturgis also give good insight into the pros and cons of various design decisions. Documented late, the system was actually implemented in 1969.

5.3.10 Michael D. Schroeder, David D. Clark, and Jerome H. Saltzer. The Multics kernel design project. *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, in *Operating Systems Review 11*, 5 (November 1977), pages 43–56.

This paper addresses a wide range of issues encountered in applying type extension (as well as microkernel thinking, though it wasn't called that at the time) to Multics in order to simplify its internal organization and reduce the size of its trusted base. Many of these ideas were explored in even more depth in Philippe Janson's Ph.D. thesis, Using Type Extension to Organize Virtual Memory Mechanisms, M.I.T. Department of Electrical Engineering and Computer Science, August 1976. That thesis is also available as M.I.T. Laboratory for Computer Science Technical Report TR–167, September 1976.

Topic 5.4: Authentication

5.4.1 Robert [H.] Morris and Ken [L.] Thompson. Password security: A case history. *Communications of the ACM 22*, 11 (November 1979), pages 594–597.

This paper is a model of how to explain something in an accessible way. With a minimum of jargon and a historical development designed to simplify things for the reader, it describes the UNIX password security mechanism.

5.4.2 Frank Stajano and Ross J. Anderson. The resurrecting duckling: Security issues for ad-hoc wireless networks. *Security Protocols Workshop 1999*, pages 172–194.

This paper discusses the problem of how a new device (e.g., a surveillance camera) can establish a secure relationship with the remote controller of the device's owner, instead of its neighbor's or adversary's. The paper's solution is that a device will recognize as its owner the first principal that sends it an authentication key. As soon as the device receives a key, its status changes from newborn to imprinted, and it stays faithful to that key until its death. The paper illustrates the problem and solution, using a vivid analogy of how ducklings authenticate their mother (see Sidebar 5.4.1).

5.4.3 David Mazières. *Self-certifying file system*. Ph.D. thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science (May 2000).

This thesis proposes a design for a cross-administrative domain file system that separates the file system from the security mechanism using an idea called self-certifying path names. Self-certifying names can be found in several other systems.

5.4.4 Bryan Ford et al. Persistent personal names for globally connected mobile devices. *Proceedings of the Seventh USENIX Symposium on Operating Systems Design and Implementation* (November 2006), pages 233–248.

This paper describes a naming system for personal devices. Each device is a root of its own naming network and can use short, convenient names for other devices belonging to the same user or belonging to people in the user's social network. The implementation of the naming system allows devices to be disconnected from the Internet and resolve names of devices that are reachable. The first five pages lay out the basic naming plan. Later sections explain security properties and a security-based implementation, which makes this portion of the paper the most relevant to the material of Chapter 5.

For more on authentication, see Sidebar 5.6.1 on Kerberos.

Topic 5.5: Cryptographic techniques

The fundamental books about cryptography applied to computer systems are the two readings that follow:

5.5.1 Bruce Schneier. *Applied Cryptography*. John Wiley and Sons, second edition, 1996. ISBN: 978–0–471–12845–8 (hardcover), 978–0–471–11709–4 (paperback). 784 pages.

Here is everything you might want to know about cryptography and cryptographic protocols, including a well-balanced perspective on what works and what doesn't. This book saves the need to read and sort through the thousand or so technical papers on the subject. Protocols, techniques, algorithms, real-world considerations, and source code can all be found here. In addition to being competent, it is also entertainingly written and articulate. Be aware that a number of minor errors have been reported in this book; if you are implementing code, it would be a good idea to verify the details by consulting reading 5.5.2.

5.5.2 A[lfred] J. Menezes, Paul C. Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997. ISBN: 978–08493–8523–0. 816 pages.

This book is exactly what its title claims: a complete handbook on putting cryptography to work. It lacks the background and perspective of reading 5.5.1, and it is extremely technical, which makes parts of it inaccessible to less mathematically inclined readers. But its precise definitions and careful explanations make this by far the best reference book available on the subject.

In light of these two books, the first few papers from the 1970s listed below are primarily of historical interest. There is also a good, more elementary, treatment of cryptography in the book by Simson Garfinkel, which is listed above as reading 5.3.8.

5.5.3 R[onald] L. Rivest, A[di] Shamir, and L[en] Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM 21*, 2 (February 1978), pages 120–126.

This paper was the first to suggest a possibly workable public key system.

5.5.4 Whitfield Diffie and Martin E. Hellman. Exhaustive cryptanalysis of the NBS Data Encryption Standard. *Computer 10*, 6 (June 1977), pages 74–84.

This is the unofficial analysis of how to break the DES by brute force—by building special-purpose chips and arraying them in parallel. Twenty-five years later, brute force still seems to be the only promising attack on DES, but the intervening

improvements in hardware technology make special chips unnecessary—an array of personal computers on the Internet can do the job. The Advanced Encryption Standard (AES) is DES's successor (see Section 5.9.3.1).

5.5.5 Ross J. Anderson. Why cryptosystems fail. *Communications of the ACM 37*, 11 (November 1994), pages 32–40.

Anderson presents a nice analysis of what goes wrong in real-world cryptosystems—secure modules don't necessarily lead to secure systems—and the applicability of systems thinking in their design. He points out that merely doing the best possible design isn't enough; a feedback loop that corrects errors in the design following experience in the field is an equally important component that is sometimes forgotten.

5.5.6 David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. *Proceedings of the Second USENIX Workshop on Electronic Commerce, Volume 2* (November 1996), pages 29–40.

This paper is useful not only because it provides a careful analysis of the security of the subject protocol, but it also explains how the protocol works in a form that is more accessible than the protocol specification documents. The originally published version was almost immediately revised with corrections. The revised version is available on the World Wide Web at <http://www.counterpane.com/ssl.html>.

5.5.7 M[ihir] Bellare, R[an] Canetti, and H[ugo] Krawczyk. Keying hash functions for message authentication. *Proceedings of the Sixteenth International Cryptograhy Conference* (August 1996), pages 1–15. (Also see H. Krawczyk, M. Bellare, and R. Canetti, HMAC: Keyed-hashing for message authentication, *Request for Comments RFC 2104*, Internet Engineering Task Force (February 1997).

This paper and the RFC introduce and define HMAC, a hash function used in widely deployed protocols.

5.5.8 David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM 24*, 2 (February 1981), pages 84–88.

This paper introduces a system design, named mixnet, that allows a sender of a message to hide its true identity from a receiver but still allow the receiver to respond.

Note that all of these books and papers focus on the application of cryptography, not on crypto-mathematics, which is a distinct area of specialization not covered in this reading list. An accessible crypto-mathematics reference is the following reading:

5.5.9 Johannes A. Buchman. *Introduction to Cryptography*. Springer, 2nd edition, 2004. ISBN 978–0–387–21156–5 (hardcover), 978–0–387–20756–8 (paperback). 335 pages.

Buchman provides a nice, concise introduction to number theory for cryptography.

Topic 5.6: Adversaries (the dark side)

Section 5.12 on war stories gives a wide range of examples of how adversaries can break a system's security. This section lists a few papers that provide a longer and more detailed descriptions of attacks. This is a fast-moving area; as soon as designers fend off new attacks, adversaries try to find new attacks. This arms race is reflected in some of the following readings, and although some of the attacks described have become ineffective (or will over time), these papers provide valuable insights. The proceedings of *Usenix Security* and *Computer and Communication Security* often contain papers explaining current attacks, and conferences run by the so-called "black hat" community document the "progress" on the dark side.

5.6.1 Eugene Spafford. Crisis and aftermath. *Communications of the ACM 32*, 6 (June 1989), pages 678–687.

This paper documents how the Morris worm works. It was one of the first worms, as well as one of the most sophisticated.

5.6.2 Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy 2*, 4 (August 2004), pages 20–27.

This paper describes how buffer overrun attacks have evolved since the Morris worm.

5.6.3 Abhishek Kumar, Vern Paxson, and Nicholas Weaver. Exploiting underlying structure for detailed reconstruction of an Internet scale event. *Proceedings of the ACM Internet Measurement Conference* (October 2005), pages 351-364.

This paper describes the Witty worm and how the authors were able to track down its source. The work contains many interesting nuggets of information.

5.6.4 Vern Paxson. An analysis of using reflectors for distributed denial-of-service attacks. *Computer Communications Review 31*, 3 (July 2001), pages 38-47.

This paper describes how an adversary can trick a large set of Internet servers to send their combined replies to a victim and in that way launch a denial-of-service attack on the victim. It speculates on several possible directions for defending against such attacks.

5.6.5 Chris Kanich et al. Spamalytics: an empirical analysis of spam marketing conversion. *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Arlington, Virginia (October 2008), pages 3–14.

This paper describes the infrastructure that spammers use to send unsolicited email and tries to establish what the financial reward system is for spammers. This paper has its shortcomings, but it is one of the few papers that tries to understand the economics behind spam.

5.6.6 Tom Jagatic, Nathaniel Johnson, Markus Jakobsson, and Filippo Menczer. Social phishing. *Communications of the ACM 50*, 10 (October 2007), pages 94–100.

This study investigates the success rate of individual phishing attacks.

6.3: Suggestions for Further Reading is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

Index

A

actions
- 3.1: Overview

application protocol
- 1.3: Getting Organized - Layers

atomicity
- 3.1: Overview

B

bootstrapping
- 3.3: All-Or-Nothing Atomicity I - Concepts

C

congestion collapse
- 1.7: A Network System Design Issue - Congestion Control

I

isochronous communication
- 1.2: Interesting Properties of Networks

L

link layer
- 1.4: The Link Layer

M

multiplexed communication
- 1.2: Interesting Properties of Networks

P

parallel transmission
- 1.4: The Link Layer

presentation protocol
- 1.3: Getting Organized - Layers

protocols
- 1.3: Getting Organized - Layers

R

ready/acknowledge protocol
- 1.4: The Link Layer

reconciliation
- 4.5: Reconciliation

S

serial transmission
- 1.4: The Link Layer

service time
- 1.7: A Network System Design Issue - Congestion Control

snoopy cache
- 4.3: Cache Coherence

V

version history
- 3.3: All-Or-Nothing Atomicity I - Concepts

Glossary

abort—Upon deciding that an all | or-nothing action cannot or should not commit, to undo all of the changes previously made by that all-or-nothing action. After aborting, the state of the system, as viewed by anyone above the layer that implements the all-or-nothing action, is as if the all-or-nothing action never existed. Compare with commit.

absolute path name | In a naming hierarchy, a path name that a name resolver resolves by using a universal context known as the root context.

abstraction | The separation of the interface specification of a module from its internal implementation so that one can understand and make use of that module with no need to know how it is implemented internally.

access control list (ACL) | A list of principals authorized to have access to some object.

acknowledgment (ACK) | A status report from the recipient of a communication to the originator. Depending on the protocol, an acknowledgment may imply or explicitly state any of several things, for example, that the communication was received, that its checksum verified correctly, that delivery to a higher level was successful, or that buffer space is available for another communication. Compare with negative acknowledgment.

action | An operation performed by an interpreter. Examples include a microcode step, a machine instruction, a higher-level language instruction, a procedure invocation, a shell command line, a response to a gesture at a graphical interface, or a database update.

active fault | A fault that is currently causing an error. Compare with latent fault.

adaptive routing | A method for setting up forwarding tables so that they change automatically when links are added to and deleted from the network or when congestion makes a path less desirable. Compare with static routing.

address | A name that is overloaded with information useful for locating the named object. In a computer system, an address is usually of fixed length and resolved by hardware into a physical location by mapping to geometric coordinates. Examples of addresses include the names for a byte of memory and for a disk track. Also see network address.

address resolution protocol (ARP) | A protocol used when a broadcast network is a component of a packet-forwarding network. The protocol dynamically constructs tables that map station identifiers of the broadcast network to network attachment point identifiers of the packet-forwarding network.

address space | The name space of a location-addressed memory, usually a set of contiguous integers (0, 1, 2,…).

adversary | An entity that intentionally tries to defeat the security measures of a computer system. The entity may be malicious, out for profit, or just a hacker. A friendly adversary is one that tests the security of a computer system.

advertise | In a network-layer routing protocol, for a participant to tell other participants which network addresses it knows how to reach.

alias | One of multiple names that map to the same value; another term for synonym. (Beware: some operating systems define alias to mean an indirect name.)

all-or-nothing atomicity | A property of a multistep action that if an anticipated failure occurs during the steps of the action, the effect of the action from the point of view of its invoker is either never to have started or else to have been accomplished completely. Compare with before-or-after atomicity and atomic.

any-to-any connection | A desirable property of a communication network, that any node be able to communicate with any other.

archive | A record, usually kept in the form of a log, of old data values, for auditing, recovery from application mistakes, or historical interest.

asynchronous | (From Greek roots meaning "not timed") 1. Describes concurrent activities that are not coordinated by a common clock and thus may make progress at different rates. For example, multiple processors are usually asynchronous, and I/O operations are typically performed by an I/O channel processor that is asynchronous with respect to the processor that initiated the I/O. 2. In a communication network, describes a communication link over which data is sent in frames whose timing relative to other frames is unpredictable and whose lengths may not be uniform. Compare with isochronous.

At the end | to-end layer of a network, the end that initiates actions. Compare with service.

At the end | to-end layer of a network, the end that responds to actions initiated by the other end. Compare with client.

at-least-once | A protocol assurance that the intended operation or message delivery was performed at least one time. It may have been performed several times.

at-most-once | A protocol assurance that the intended operation or message delivery was performed no more than one time. It may not have been performed at all.

atomic (adj.); atomicity (n.) | A property of a multistep action that there be no evidence that it is composite above the layer that implements it. An atomic action can be beforeor-after, which means that its effect is as if it occurred either completely before or completely after any other before-or-after action. An atomic action can also be all-ornothing, which means that if an anticipated failure occurs during the action, the effect of the action as seen by higher layers is either never to have started or else to have completed successfully. An atomic action that is both all-or-nothing and before-or-after is known as a transaction.

atomic storage | Cell storage for which a multicell PUT can have only two possible outcomes: (1) it stores all data successfully, or (2) it does not change the previous data at all. In consequence, either a concurrent thread or (following a failure) a later thread doing a GET will always read either all old data or all new data. Computer architectures in which multicell PUTs are not atomic are said to be subject to write tearing.

authentication | Verifying the identity of a principal or the authenticity of a message.

authentication tag | A cryptographically computed string, associated with a message, that allows a receiver to verify the authenticity of the message.

authorization | A decision made by an authority to grant a principal permission to perform some operation, such as reading certain information.

automatic rate adaptation | A technique by which a sender automatically adjusts the rate at which it introduces packets into a network to match the maximum rate that the narrowest bottleneck can handle.

availability | A measure of the time that a system was actually usable, as a fraction of the time that it was intended to be usable. Compare with its complement, down time.

backup copy | Of a set of replicas that is not written or updated synchronously, one that is written later. Compare with primary copy and mirror.

backward error correction | A technique for correcting errors in which the source of the data or control signal applies enough redundancy to allow errors to be detected and, if an error does occur, that source is asked to redo the calculation or repeat the transmission. Compare with forward error correction.

bad-news diode | An undesirable tendency of people in organizations that design and implement systems: good news, for example, that a module is ready for delivery ahead of schedule, tends to be passed immediately throughout the organization, but bad news, for example, that a module did not pass its acceptance tests, tends to be held locally until either the problem can be fixed or it cannot be concealed any longer.

bandwidth | A measure of analog spectrum spacefor a communication channel. The bandwidth, the acceptable signal power, and the noise level of a channel together determine the maximum possible data rate for that channel. In digital systems, this term is so often misused as a synonym for maximum data rate that it has now entered the vocabulary of digital designers with that additional meaning. Analog engineers, however, still cringe at that usage.

batching | A technique to improve performance by combining several operations into a single operation to reduce setup overhead.

before-or-after atomicity | A property of concurrent actions: Concurrent actions are before-or-after actions if their effect from the point of view of their invokers is the same as if the actions occurred either completely before or completely after one another. One consequence is that concurrent before-or-after software actions cannot discover the composite nature of one another (that is, one action cannot tell that another has multiple steps). A consequence in the case of hardware is that concurrent before-or-after WRITEs to the same memory cell will be performed in some order, so there is no danger that the cell will end up containing, for example, the OR of several WRITE values. The database literature uses the words "isolation" and "serializable", the operating system literature uses the words "mutual exclusion" and "critical section", and the computer architecture literature uses the unqualified word "atomicity" for this concept.

best-effort contract | The promise given by a forwarding network when it accepts a packet: it will use its best effort to deliver the packet, but the time to delivery is not fixed, the order of delivery relative to other packets sent to the same destination is unpredictable, and the packet may be duplicated or lost.

binding (n.); bind (v.) | As used in naming, a mapping from a specified name to a particular value in a specified context. When a binding exists, the name is said to be bound. Binding may occur at any time up to and including the instant that a name is resolved. The term is also used more generally, meaning to choose a specific lower-layer implementation for some higher-layer feature.

bit error rate | In a digital transmission system, the rate at which bits that have incorrect values arrive at the receiver, expressed as a fraction of the bits transmitted, for example, one in 1010.

bit stuffing | The technique of inserting a bit pattern as a marker in a stream of bits and then inserting bits elsewhere in the stream to ensure that payload data never matches the marker bit pattern.

blind write | An update to a data value X by a transaction that did not previously read X.

bootstrapping | A systematic approach to solving a general problem, consisting of a method for reducing the general problem to a specialized instance of the same problem and a method for solving the specialized instance.

bottleneck | The stage in a multistage pipeline that takes longer to perform its task than any of the other stages.

broadcast | To send a packet that is intended to be received by many (ideally, all) of the stations of a broadcast link (link-layer broadcast), or all the destination addresses of a network (network-layer broadcast).

burst | A batch of related bits that is irregular in size and timing relative to other such batches. Bursts of data are the usual content of messages and the usual payload of packets. One can also have bursts of noise and bursts of packets.

Byzantine fault | A fault that generates inconsistent errors (perhaps maliciously) that can confuse or disrupt fault tolerance or security mechanisms.

cache | A performance-enhancing module that remembers the result of an expensive computation on the chance that the result may soon be needed again.

cache coherence | Read/write coherence for a multilevel memory system that has a cache. It is a specification that the cache provide strict consistency at its interface.

capability | In a computer system, an unforgeable ticket, which when presented is taken as incontestable proof that the presenter is authorized to have access to the object named in the ticket.

capacity | Any consistent measure of the size or amount of a resource.

cell storage | Storage in which a WRITE or PUT operates by overwriting, thus destroying previously stored information. Many physical storage devices, including magnetic disk and CMOS random access memory, implement cell storage. Compare with journal storage.

certificate | A message that attests the binding of a principal identifier to a cryptographic key.

certificate authority (CA) | A principal that issues and signs certificates.

certify | To check the accuracy, correctness, and completeness of a security mechanism.

checkpoint | 1. (n.) Information written to non-volatile storage that is intended to speed up recovery from a crash. 2 (v.) To write a checkpoint.

checksum | A stylized error-detection code in which the data is unchanged from its uncoded form and additional, redundant data is placed in a distinct, separately architected field.

cipher | Synonym for a cryptographic transformation.

ciphertext | The result of encryption. Compare with plaintext.

circuit switch | A device with many electrical circuits coming in to it that can connect any circuit to any other circuit; it may be able to perform many such connections simultaneously. Historically, telephone systems were constructed of circuit switches.

cleartext | Synonym for plaintext.

client | A module that initiates actions, such as sending a request to a service.

client/service organization | An organization that enforces modularity among modules of a computer system by limiting the interaction among the modules to messages.

close-to-open consistency | A consistency model for file operations. When a thread opens a file and performs several write operations, all of the modifications weill be visible to concurrent threads only after the first thread closes the file.

closure | In a programming language, an object that consists of a reference to the text of a procedure and a reference to the context in which the program interpreter is to resolve the variables of the procedure.

coheerence | See read/write coherence or cache coherence. collision—1. In naming, a particular kind of name conflict in which an algorithmic name generator accidentally generates the same name more than once in what is intended to be a unique identifier name space. 2. In networks, an event when two stations attempt to send a message over the same physical medium at the same time. See also Ethernet.

commit | To renounce the ability to abandon an all-or-nothing action unilaterally. One usually commits an all-or-nothing action before making its results available to concurrent or later all-or-nothing actions. Before committing, the all-or-nothing action can be abandoned and one can pretend that it had never been undertaken. After committing, the all-or-nothing action must be able to complete. A committed all-ornothing action cannot be abandoned; if it can be determined precisely how far its results have propagated, it may be possible to reverse some or all of its effects by compensation. Commitment also usually includes an expectation that the results preserve any appropriate invariants and will be durable to the extent that the application requires those properties. Compare with compensate and abort.

communication link | a data communication path between physically separated components.

Compare with all | or-nothing atomicity and atomic.

compensate (adj.); compensation (n.) | To perform an action that reverses the effect of some previously committed action. Compensation is intrinsically application dependent; it is easier to reverse an incorrect accounting entry than it is to undrill an unwanted hole.

complexity | A loosely defined notion that a system has so many components, interconnections, and irregularities that it is difficult to understand, implement, and maintain.

confidentiality | Limiting information access to authorized principals. Secrecy is a synonym.

confinement | Allowing a potentially untrusted program to have access to data, while ensuring that the program cannot release information.

congestion | Overload of a resource that persists for significantly longer than the average service time of the resource. (Since significance is in the eye of the beholder, the concept is not a precise one.)

congestion collapse | When an increase in offered load causes a catastrophic decrease in useful work accomplished.

connection | A communication path that requires maintaining state between successive messages. See set up and tear down.

connectionless | Describes a communication path that does not require coordinated state and can be used without set up or tear down. See connection.

consensus | Agreement at separated sites on a data value despite communication failures.

consistency | A particular constraint on the memory model of a storage system that allows concurrency and uses replicas: that all readers see the same result. Also used in some professional literature as a synonym for coherence.

constraint—An application | defined invariant on a set of data values or externally visible actions. Example: a requirement that the balances of all the accounts of a bank sum to zero, or a requirement that a majority of the copies of a set of data be identical.

context | One of the inputs required by a name-mapping algorithm in order to resolve a name. A common form for a context is a set of name-to-value bindings.

context reference | The name of a context.

continuous operation | An availability goal, that a system be capable of running indefinitely. The primary requirement of continuous operation is that it must be possible to perform repair and maintenance without stopping the system.

control point | An entity that can adjust the capacity of a limited resource or change the load that a source offers.

cooperative scheduling | A style of thread scheduling in which each thread on its own initiative releases the processor periodically to allow other threads to run.

covert channel | In a flow-control security system, a way of leaking information into or out of a secure area. For example, a program with access to a secret might touch several shared but normally unused virtual memory pages in a pattern to bring them into real memory; a conspirator outside the secure area may be able to detect the pattern by measuring the time required to read those same shared pages.

cryptographic hash function | A cryptographic function that maps messages to short values in such a way that it is difficult to (1) reconstruct a message from its hash value; and (2) construct two different messages having the same value.

cryptographic key | The easily changeable component of a key-driven cryptographic transformation. A cryptographic key is a string of bits. The bits may be generated randomly, or they may be a transformed version of a password. The cryptographic key, or at least part of it, usually must be kept secret, while all other components of the transformation can be made public.

cryptographic transformation | Mathematical transformation used as a building block for implementing security primitives. Such building blocks include functions for implementing encryption and decryption, creating and verifying authentication tags, cryptographic hashes, and pseudorandom number generators.

cryptography | A discipline of theoretical computer science that specializes in the study of cryptographic transformations and protocols.

cut-through | A forwarding technique in which transmission of a packet or frame on an outgoing link begins while the packet or frame is still being received on the incoming link.

dallying | A technique to improve performance by delaying a request on the chance that the operation won't be needed, or to create more opportunities for batching.

dangling reference | Use of a name that has outlived the binding of that name.

data integrity | Authenticity of the apparent content of a message or file. In a network, a transport protocol assurance that the data delivered to the recipient is identical to the original data the sender provided. Compare with origin authenticity.

data rate | The rate, usually measured in bits per second, at which bits are sent over a communication link. When talking of the data rate of an asynchronous communication link, the term is often used to mean the maximum data rate that the link allows.

deadlock | Undesirable interaction among a group of threads in which each thread is waiting for some other thread in the group to make progress.

decay | Unintended loss of stored state with the passage of time.

decay set | A set of storage blocks, words, tracks, or other physical groupings, in which all members of the set may spontaneously fail together, but independently of any other decay set.

decrypt | To perform a reverse cryptographic transformation on a previously encrypted message to obtain the plaintext. Compare with encrypt.

default context reference | A context reference chosen by the name resolver rather than specified as part of the name or by the object that used the name. Compare with explicit context reference.

demand paging | A class of page-movement algorithm that moves pages into the primary device only at the instant that they are used. Compare with prepaging.

destination | The network attachment point to which the payload of a packet is to be delivered. Sometimes used as shorthand for destination address.

destination address | An identifier of the destination of a packet, usually carried as a field in the header of the packet.

detectable error | An error or class of errors for which a reliable detection plan can be devised. An error that is not detectable usually leads to a failure, unless some mechanism that is intended to mask some other error accidentally happens to mask the undetectable error. Compare with maskable error and tolerated error.

digital signature | An authentication tag computed with public-key cryptography.

directory | In a file system, an object consisting of a table of bindings between symbolic file names and some description (e.g., a file number or a file map) of the corresponding file. Other terms used for this concept include catalog and folder. A directory is an example of a context.

discretionary access control | A property of an access control system. In a discretionary access control system, the owner of an object has the authority to decide which principals have access to that object. Compare with non-discretionary access control.

do action | (n.) Term used in some systems for a redo action.

domain | A range of addresses to which a thread has access. It is the abstraction that enforces modularity within a memory, separating modules and allowing for controlled sharing.

down time | A measure of the time that a system was not usable, as a fraction of the time that it was intended to be usable. Compare with its complement, availability.

duplex | Describes a link or connection between two stations that can be used in both directions. Compare with simplex, half-duplex, and full-duplex.

duplicate suppression | A transport protocol mechanism for achieving at-most-once delivery assurance, by identifying and discarding extra copies of packets or messages.

durability | A property of a storage medium that, once written, it can be read for as long as the application requires. Compare with stability and persistence, terms that have different technical definitions as explained in Sidebar 2.7.

durable storage | Storage with the property that it (ideally) is decay-free, so it never fails to return on a GET the data that was stored by a previously successful PUT. Since that ideal is impossibly strict, in practice, storage is considered durable when the probability of failure is sufficiently low that the application can tolerate it. Durability is thus an application-defined specification of how long the results of an action, once completed, must be preserved. Durable is distinct from non-volatile, which describes storage that maintains its memory while the power is off, but may still have an intolerable probability of decay. The term persistent is sometimes used as a synonym for durable, as explained in Sidebar 2.7, but to minimize confusion this text avoids that usage.

dynamic scope | An example of a default context, used to resolve names of program variables in some programming languages. The name resolver searches backward in the call stack for a binding, starting with the stack frame of the procedure that used the name, then the stack of its caller, then the caller's caller, and so on. Compare with static scope.

earliest deadline first scheduling policy | A scheduling policy for real-time systems that gives priority to the thread with the earliest deadline.

early drop | A predictive strategy for managing an overloaded resource: the system refuses service to some customers before the queue is full.

emergent property | A property of an assemblage of components that would not be predicted by examining the components individually. Emergent properties are a surprise when first encountered.

emulation | Faithfully simulating some physical hardware so that the simulated hardware can run any software that the physical hardware can.

encrypt | To perform a cryptographic transformation on a message with the objective of achieving confidentiality. The cryptographic transformation is usually key-driven. Compare with the inverse operation, decrypt, which can recover the original message.

end-to-end | Describes communication between network attachment points, as contrasted with communication between points within the network or across a single link.

end-to-end layer | The communication system layer that manages end-to-end communications.

enforced modularity | Modularity that prevents accidental errors from propagating from one module to another. Compare with soft modularity.

enumerate | To generate a list of all the names that can currently be resolved (that is, that have bindings) in a particular context.

environment | 1. In a discussion of systems, everything surrounding a system that is not viewed as part of that system. The distinction between a system and its environment is a choice based on the purpose, ease of description, and minimization of interconnections. 2. In an interpreter, the state on which the interpreter should perform the actions directed by program instructions.

environment reference | The component of an interpreter that tells the interpreter where to find its environment.

erasure | An error in a string of bits, bytes, or groups of bits in which an identified bit, byte, or group of bits is missing or has indeterminate value.

ergodic | A property of some time-dependent probabilistic processes: that the (usually easier to measure) ensemble average of some parameter measured over a set of elements subject to the process is the same as the time average of that parameter of any single element of the ensemble.

error | Informally, a label for an incorrect data value or control signal caused by an active fault. If there is a complete formal specification for the internal design of a module, an error is a violation of some assertion or invariant of the specification. An error in a module is not identical to a failure of that module, but if an error is not masked, it may lead to a failure of the module.

error containment | Limiting how far the effects of an error propagate. A module is normally designed to contain errors in such a way that the effects of an error appear in a predictable way at the module's interface.

error correction | A scheme to set to the correct value a data value or control signal that is in error. Compare with error detection.

error detection | A scheme to discover that a data value or control signal is in error. Compare with error correction.

error-correction code | a method of encoding stored or transmitted data with a modest amount of redundancy, in such a way that any errors during storage or transmission will, with high probability, lead to a decoding that is identical to the original data. See also the general definition of error correction. Compare with error-detection code.

error-detection code | a method of encoding stored or transmitted data with a small amount of redundancy, in such a way that any errors during storage or transmission will, with high probability, lead to a decoding that is obviously wrong. Compare with error correction code and checksum. See also the general definition of error detection. Compare with error-correction code and checksum.

Ethernet | A widely used broadcast network in which all participants share a common wire and can hear one another transmit. Ethernet is characterized by a transmit protocol in which a station wishing to send data first listens to ensure that no one else is sending, and then continues to monitor the network during its own transmission to see if some other station has tried to transmit at the same time, an error known as a collision. This protocol is named Carrier Sense Multiple Access with Collision Detection, abbreviated CSMA/CD.

eventcount | A special type of shared variable used for sequence coordination. It supports two primary operations: AWAIT and ADVANCE. An eventcount is a counter that is incremented atomically, using ADVANCE, while other threads wait for the counter to reach a certain value using AWAIT. Eventcounts are often used in combination with sequencers.

eventual consistency | A requirement that at some unspecified time following an update to a collection of data, if there are no more updates, the memory model for that collection will hold.

exactly-once | A protocol assurance that the intended operation or message delivery was performed both at-least-once and at-most-once.

exception | An interrupt event that pertains to the thread that a processor is currently running.

explicit context reference | For a name or an object, an associated reference to the context in which that name, or all names contained in that object, are to be resolved. Compare with default context reference.

explicitness | A property of a message in a security protocol: if a message is explicit, then the message contains all the information necessary for a receiver to reliably determine that the message is part of a particular run of the protocol with a specific function and set of participants.

exponential backoff | An adaptive procedure used to set a timer, for example, to wait for congestion to dissipate. Each time the timer setting proves to be too small, the action doubles (or, more generally, multiplies by a constant greater than one) the length of its next timer setting. The intent is obtain a suitable timer value as quickly as possible. See also exponential random backoff.

exponential random backoff | A form of exponential backoff in which an action that repeatedly encounters interference repeatedly doubles (or, more generally, multiplies by a constant greater than one) the size of an interval from which it randomly chooses its next delay before retrying. The intent is that by randomly changing the timing relative to other, interfering actions, the interference will not recur.

export | In naming, to provide a name for an object that other objects can use.

fail-fast | Describes a system or module design that contains detected errors by reporting at its interface that its output may be incorrect. Compare with fail-stop.

fail-safe | Describes a system design that detects incorrect data values or control signals and forces them to values that, even if not correct, are known to allow the system to continue operating safely.

fail-secure | Describes an application of fail-safe design to information protection: a failure is guaranteed not to allow unauthorized access to protected information. In early work on fault tolerance, this term was also occasionally used as a synonym for fail-fast.

fail-soft | Describes a design in which the system specification allows errors to be masked by degrading performance or disabling some functions in a predictable manner.

fail-stop | Describes a system or module design that contains detected errors by stopping the system or module as soon as possible. Compare with fail-fast, which does not require other modules to take additional action, such as setting a timer, to detect the failure.

fail-vote | Describes an N-modular redundancy system with a majority voter.

failure | The outcome when a component or system does not produce the intended result at its interface. Compare with fault.

failure tolerance | A measure of the ability of a system to mask active faults and continue operating correctly. A typical measure counts the number of contained components that can fail without causing the system to fail.

fault | A defect in materials, design, or implementation that may (or may not) cause an error and lead to a failure. (Compare with failure.)

fault avoidance | A strategy to design and implement a component with a probability of faults that is so low that it can be neglected. When applied to software, fault avoidance is sometimes called valid construction.

fault tolerance | A set of techniques that involve noticing active faults and lower-level subsystem failures and masking them, rather than allowing the resulting errors to propagate.

file | A popular memory abstraction to durably store and retrieve data. A typical interface for a file consists of procedures to OPEN the file, to READ and WRITE regions of the file, and to CLOSE the file.

fingerprint | Another term for a witness.

first-come | A scheduling policy in which requests are processed in the order in which they arrive.

first-in | A particular page-removal policy for a multilevel memory system. FIFO chooses to remove the page that has been in the primary device the longest.

flow control | 1. In networks, an end-to-end protocol between a fast sender and a slow recipient, a mechanism that limits the sender's data rate so that the recipient does not receive data faster than it can handle. 2. In security, a system that allows untrusted programs to work with sensitive data but confines all program outputs to prevent unauthorized disclosure.

force | (v.) When output may be buffered, to ensure that a previous output value has actually been written to durable storage or sent as a message. Caches that are not writethrough usually have a feature that allows the invoker to force some or all of their contents to the secondary storage medium.

forward error correction | A technique for controlling errors in which enough redundancy to correct anticipated errors is applied before an error occurs. Forward error correction is particularly applicable when the original source of the data value or control signal will not be available to recalculate or resend it. Compare with backward error correction.

forward secrecy | A property of a security protocol. A protocol has forward secrecy if information, such as an encryption key, deduced from a previous transcript doesn't allow an adversary to decrypt future messages.

forwarding table | A table that tells the network layer which link to use to forward a packet, based on its destination address.

fragment | 1. (v.) In network protocols, to divide the payload of a packet so that it can fit into smaller packets for carriage across a link with a small maximum transmission unit. 2. (n.) The resulting pieces of payload.

frame | 1. (n.) The unit of transmission in the link layer. Compare with packet, segment, and message. 2. (v.) To delimit the beginning and end of a bit, byte, frame (n.), packet, segment, or message within a stream.

freshness | A property of a message in a security protocol: if the message is fresh, it is assured not to be a replay.

full-duplex | Describes a duplex link or connection between two stations that can be used in both directions at the same time. Compare with simplex, duplex, and half-duplex.

gate | A predefined protected entry point into a domain.

generated name | A name created algorithmically, rather than chosen by a person.

global name | In a layered naming scheme, a name that is bound only in the outermost context layer, and thus has the same meaning to all users.

half-duplex | Describes a duplex link or connection between two stations that can be used in only one direction at a time. Compare with simplex, duplex, and full-duplex.

Hamming distance | in an encoding system, the number of bits in an element of a code that would have to change to transform it into a different element of the code. The Hamming distance of a code is the minimum Hamming distance between any pair of elements of the code.

hard real-time scheduling policy | A real-time scheduler in which missing a deadline may result in a disaster.

hash function—A function that algorithmically derives a relatively short | length string of bits from an arbitrarily-large block of data. The resulting short string is known as a hash. See also cryptographic hash function.

header | Information that a protocol layer adds to the front of a packet.

hierarchical routing | A routing system that takes advantage of hierarchically assigned network destination addresses to reduce the size of its routing tables.

hierarchy | A technique of organizing systems that contain many components: group small numbers of components into self-contained and stable subsystems that then become components of larger self-contained and stable subsystems, and so on.

hit ratio | In a multilevel memory, the fraction of references satisfied by the primary memory device.

hop limit | A network-layer protocol field that acts as a safety net to prevent packets from endlessly circulating in a network that has inconsistent forwarding tables.

hot swap | To replace modules in a system while the system continues to provide service.

idempotent | Describes an action that can be interrupted and restarted from the beginning any number of times and still produce the same result as if the action had run to completion without interruption. The essential feature of an idempotent action is that if there is any question about whether or not it completed, it is safe to do it again. "Idempotent" is correctly pronounced with the accent on the second syllable, not on the first and third.

identifier | A synonym for name, sometimes used to avoid an implication that the name might be meaningful to a person rather than to a machine.

illegal instruction | An instruction that an interpreter is not equipped to execute because it is not in the interpreter's instruction repertoire or it has an out-of-range operand (for example, an attempt to divide by zero). An illegal instruction typically causes an interrupt.

incommensurate scaling | A property of most systems, that as the system grows (or shrinks) in size, not all parts grow (or shrink) at the same rate, thus stressing the system design.

incremental backup | A backup copy that contains only data that has changed since making the previous backup copy.

indirect name | A name that is bound to another name in the same name space. “Symbolic link”, “soft link”, and “shortcut” are other words used for this concept. Some operating systems also define the term alias to have this meaning rather than its more general meaning of synonym.

indirection | Decoupling a connection from one object to another by interposing a name with the goal of delaying the choice of (or allowing a later change about) which object the name refers to. Indirection makes it possible to delay the choice of or change which object is used without the need to change the object that uses it. Using a name is sometimes described as “inserting a level of indirection”.

install | In a system that uses logs to achieve all-or-nothing atomicity, to write data to cell storage.

instruction reference | A characteristic component of an interpreter: the place from which it will take its next instruction.

intended load | The amount of a shared resource that a set of users would attempt to utilize if the resource had unlimited capacity. In systems that have no provision for congestion control, the intended load is equal to the offered load. The goal of congestion control is to make the offered load smaller than the intended load. Compare with offered load.

interleaving | A technique to improve performance by distributing apparently sequential requests to several instances of a device, so that the requests may actually be processed concurrently.

intermittent fault | A persistent fault that is active only occasionally. Compare with transient fault.

International Organization for Standardization (ISO) | An international nongovernmental body that sets many technical and manufacturing standards including the (frequently ignored) Open Systems Interconnect (OSI) reference model for data communication networks. The short name ISO is not an acronym, it is the Greek word for “equal”, chosen to be the same in all languages and always spelled in all capital letters.

interpreter | The abstraction that models the active mechanism performing computations. An interpreter comprises three components: an instruction reference, a context reference, and an instruction repertoire.

interrupt | An event that causes an interpreter to transfer control to the first instruction of a different procedure, an interrupt handler, instead of executing the next instruction.

invalidate | In a cache, to mark “do not use” or completely remove a cache entry because some event has occurred that may make the value associated with that entry incorrect.

isochronous (From Greek roots meaning “equal” and “time”) | Describes a communication link over which data is sent in frames whose length is fixed in advance and whose timing relative to other frames is precisely predictable. Compare with asynchronous.

jitter | In real-time applications, variability in the delivery times of successive data elements.

job | The unit of granularity on which threads are scheduled. A job corresponds to the burst of activity of a thread between two idle periods.

journal storage | Storage in which a WRITE or PUT appends a new value, rather than overwriting a previously stored value. Compare with cell storage.

kernel | A trusted intermediary that virtualizes resources for mutually distrustful modules running on the same computer. Kernel modules typically run with kernel mode enabled.

kernel mode | A feature of a processor that when set allows threads to use special processor features (e.g., the page-map address register) that are disallowed to threads that run with kernel mode disabled. Compare with user mode.

key distribution center (KDC) | A principal that authenticates other principals to one another and also provides one or more temporary cryptographic keys for communication between other principals.

key-based cryptographic transformation | A cryptographic transformation for which successfully meeting the cryptographic goals depends on the secrecy of some component of the transformation. That component is called a cryptographic key, and a usual design is to make that key a small, modular, separable, and easily changeable component.

latency | The delay between a change at the input to a system and the corresponding change at its output. As used in reliability, the time between when a fault becomes active and when the module in which the fault occurred either fails or detects the resulting error.

latent fault | A fault that is not currently causing an error. Compare with active fault.

layering | A technique of organizing systems in which the designer builds on an interface that is already complete (a lower layer), to create a different complete interface (an upper layer).

least-recently-used (LRU) policy | A popular page-removal policy for a multilevel memory system. LRU chooses to remove the page that has not been used the longest.

lexical scope | Another term for static scope.

limited name space | A name space in which a limited number of names can be expressed and therefore names must be allocated, deallocated, and reused.

link | 1 (n.) Another term for a synonym (usually called a hard link) or an indirect name (usually called a soft or symbolic link). 2 (v.) Another term for bind. 3. (n.) In data communication, a communication path between two points.

link layer | The communication system layer that moves data directly from one physical point to another.

list system | A design for an access control mechanism in which each protected object is associated with a list of authorized principals.

livelock | An undesirable interaction among a group of threads in which each thread begins a sequence of actions, discovers that it cannot complete the sequence because actions of other threads have interfered, and begins again, endlessly.

locality of reference | A property of most programs that memory references tend to be clustered in both time and address space.

lock | A flag associated with a data object, set by a thread to warn concurrent threads that the object is in use and that it may be a mistake for other threads to read or write it. Locks are one technique used to achieve before-or-after atomicity.

lock point | In a system that provides before-or-after atomicity by locking, the first instant in a before-or-after action when every lock that will ever be in its lock set has been acquired.

lock set | The collection of all locks acquired during the execution of a before-or-after action.

lock-step protocol | In networking, any transport protocol that requires acknowledgment of the previously sent message, segment, packet, or frame before sending another message, segment, packet, or frame to the same destination. Sometimes called a stop and wait protocol. Compare with pipeline.

log | 1. (n.) A specialized use of journal storage to maintain an append-only record of some application activity. Logs are used to implement all-or-nothing actions, for performance enhancement, for archiving, and for reconciliation. 2. (v.) To append a record to a log.

logical copy | A replica that is organized in a form determined by a higher layer. An example is a replica of a file system that is made by copying one file at a time. Analogous to logical locking. Compare with physical copy.

logical locking | Locking of higher-layer data objects such as records or fields of a database. Compare with physical locking.

Manchester code | A particular type of phase encoding in which each bit is represented by two bits of opposite value.

margin | The amount by which a specification is better than necessary for correct operation. The purpose of designing with margins is to mask some errors.

mark point | 1. (adj.) An atomicity-assuring discipline in which each newly created action n must wait to begin reading shared data objects until action (n – 1) has marked all of the variables it intends to modify. 2. (n.) The instant at which an action has marked all of the variables it intends to modify.

marshal/unmarshal | To marshal is to transform the internal representation of one or more pieces of data into a form that is more suitable for transmission or storage. The opposite action, to unmarshal, is to parse marshaled data into its constituent data pieces and transform those pieces into a suitable internal representation.

maskable error | An error or class of errors that is detectable and for which a systematic recovery strategy can in principle be devised. Compare with detectable error and tolerated error.

masking | As used in reliability, containing an error within a module in such a way that the module meets its specifications as if the error had not occurred.

master | In a multiple-site replication scheme, the site to which updates are directed. Compare with slave.

maximum transmission unit (MTU) | A limit on the size of a packet, imposed to control the time commitment involved in transmitting the packet, to control the amount of loss if congestion causes the packet to be discarded, and to keep low the probability of a transmission error.

mean time between failures (MTBF) | The sum of MTTF and MTTR for the same component or system.

mean time to failure (MTTF) | The expected time that a component or system will operate continuously without failing. “Time” is sometimes measured in cycles of operation.

mean time to repair (MTTR) | The expected time to replace or repair a component or system that has failed. The term is sometimes written as “mean time to restore service”, but it is still abbreviated MTTR.

mediation | Before a service performs a requested operation, determining which principal is associated with the request and whether the principal is authorized to request the operation.

memory | The abstraction for remembering data values, using READ and WRITE operations. The WRITE operation specifies a value to be remembered and a name by which that value can be recalled in the future. See also storage.

memory manager | A device located between a processor and memory that translates virtual to physical addresses and checks that memory references by the thread running on the processor are in the thread's domain(s).

memory-mapped I/O | An interface that allows an interpreter to communicate with an I/O module using LOAD and STORE instructions that have ordinary memory addresses.

memoryless | A property of some time-dependent probabilistic processes, that the probability of what happens next does not depend on what has happened before.

message | The unit of communication at the application level. The length of a message is determined by the application that sends it. Since a network may have a maximum size for its unit of transmission, the end-to-end layer divides a message into one or more segments, each of which is carried in a separate packet. Compare with frame (n.), segment, and packet.

message authentication | The verification of the integrity of the origin and the data of a message.

message authentication code (MAC) | An authentication tag computed with sharedsecret cryptography. MAC is sometimes used as a verb in security jargon, as in "Just to be careful, let's MAC the address field of that message."

metadata | Information about an object that is not part of the object itself. Examples are the name of the object, the identity of its owner, the date it was last modified, and the location in which it is stored.

microkernel | A kernel organization in which most operating system comonents run in separate, user-mode address spaces.

mirror | (n.) One of a set of replicas that is created or updated synchronously. Compare with primary copy and backup copy. Sometimes used as a verb, as in "Let's mirror that data by making three replicas."

missing-page exception | The event when an addressed page is not present in the primary device and the virtual memory manager has to move the page in from a secondary device. The literature also uses the term page fault.

modular sharing | Sharing of an object without the need to know details of the implementation of the shared object. With respect to naming, modular sharing is sharing without the need to know the names that the shared object uses to refer to its components.

module | A system component that can be separately designed, implemented, managed, and replaced.

monolithic kernel | A kernel organization in which most operating system components run in a single, kernel-mode address space.

most-recently-used (MRU) policy | A page-removal policy for a multilevel memory system. MRU chooses for removal the most recently used page in the primary device.

MTU discovery | A procedure that systematically discovers the smallest maximum transmission unit along the path between two network attachment points.

multihomed | Describes a single physical interface between the network layer and the endto-end layer that is associated with more than one network attachment point, each with its own network-layer address.

multilevel memory | Memory built out of two or more different memory devices that have significantly different latencies and cost per bit.

multiple lookup | A name-mapping algorithm that tries several contexts in sequence, looking for the first one that can successfully resolve a presented name.

multiplexing | Sharing a communication link among several, usually independent, simultaneous communications. The term is also used in layered protocol design when several different higher-layer protocols share the same lower-layer protocol.

multipoint | Describes communication that involves more than two parties. A multipoint link is a single physical medium that connects several parties. A multipoint protocol coordinates the activities of three or more participants.

N + 1 redundancy | When a load can be handled by sharing it among N equivalent modules, the technique of installing N + 1 or more of the modules, so that if one fails the remaining modules can continue to handle the full load while the one that failed is being repaired.

N-modular redundancy (NMR) | A redundancy technique that involves supplying identical inputs to N equivalent modules and connecting the outputs to one or more voters.

N-version programming | The software version of N-modular redundancy. N different teams each independently write a program from its specifications. The programs then run in parallel, and voters compare their outputs.

name | A designator or an identifier of an object or value. A name is an element of a name space.

name conflict | An occurrence when, for some reason, it seems necessary to bind the same name to two different values at the same time in the same context. Usually, a result of encountering a preexisting name in a naming scheme that does not provide modular sharing. When names are algorithmically generated, name conflicts are called collisions.

name space | The set of all possible names of a particular naming scheme. A name space is defined by a set of symbols from some alphabet together with a set of syntax rules that define which names are members of the name space.

name-mapping algorithm | See naming scheme.

name-to-key binding | A binding between a principal identifier and a cryptographic key.

naming hierarchy | A naming network that is constrained to a tree-structured form. The root used for interpretation of absolute path names (which in a naming hierarchy are sometimes called "tree names") is normally the base of the tree.

naming network | A naming scheme in which contexts are named objects and any context may contain a binding for any other context, as well as for any non-context object. An object in a naming network is identified by a multicomponent path name that traces a path through the naming network from some starting point, which may be either a default context or a root.

naming scheme | A particular combination of a name space, a universe of values (which may include physical objects) that can be named, and a name-mapping algorithm that provides a partial mapping from the name space to the universe of values.

negative acknowledgment (NAK or NACK) | A status report from a recipient to a sender asserting that some previous communication was not received or was received incorrectly. The usual reason for sending a negative acknowledgment is to avoid the delay that would be incurred by waiting for a timer to expire. Compare with acknowledgment.

network | A communication system that interconnects more than two things.

network address | In a network, the identifier of the source or destination of a packet.

network attachment point | The place at which the network layer accepts or delivers payload data to and from the end-to-end layer. Each network attachment point has an identifier, its address, that is unique within that network. A network attachment point is sometimes called an access point, and in ISO terminology, a Network Services Access Point (NSAP).

network layer | The communication system layer that forwards data through intermediate links to carry it to its intended destination.

non-discretionary access control | A property of an access control system. In a nondiscretionary access control system, some principal other than the owner has the authority to decide which principals have to access the object. Compare with discretionary access control.

non-preemptive scheduling | A scheduling policy in which threads run until they explicitly yield or wait.

non-volatile memory | A kind of memory that does not require a continuous source of power, so it retains its content when its power supply is off. The phrase "stable storage" is a common synonym. Compare with volatile memory.

nonce | A unique identifier that should never be reused.

object | As used in naming, any software or hardware structure that can have a distinct name.

offered load | The amount of a shared service that a set of users attempt to utilize. Presented load is an occasionally encountered synonym.

opaque name | In a modular system, a name that, from the point of view of the current module, carries no overloading that the module knows how to interpret.

operating system | A collection of programs that provide services such as abstraction and management of hardware devices and features such as libraries of commonly needed procedures, all of which are intended to make it easier to write application programs.

optimal (OPT) page-removal policy | An unrealizable page-removal policy for a multilevel memory system. The optimal policy removes from primary memory the page that will not be used for the longest time. Because identifying that page requires knowing the future, the optimal policy is not implementable in practice. Its utility is that after any particular reference string has been observed, one can then simulate the operation of that reference string with the optimal policy, to compare the number of missing-page exceptions with the number obtained when using other, realizable policies.

optimistic concurrency control | A concurrency control scheme that allows concurrent threads to proceed even though a risk exists that they will interfere with each other, with the plan of detecting whether there actually is interference and, if necessary, forcing one of the threads to abort and retry. Optimistic concurrency control is an effective technique in situations where interference is possible but not likely. Compare with pessimistic concurrency control.

origin authenticity | Authenticity of the claimed origin of a message. Compare with data integrity.

overload | When offered load exceeds the capacity of a service for a specified period of time.

overloaded name | A name that does more than simply identify an object; it also carries other information, such as the type of the object, the date it was modified, or how to locate it. Overloading is commonly encountered when a system has not made suitable provision to handle metadata. Contrast with pure name.

packet | The unit of transmission of the network layer. A packet consists of a segment of payload data, accompanied by guidance information that allows the network to forward it to the network attachment point that is intended to receive the data carried in the packet. Compare with frame (n.), segment, and message.

packet forwarding | In the network layer, upon receiving a packet that is not destined for the local end layer, to send it out again along some link with the intention of moving the packet closer to its destination.

packet switch | A specialized computer that forwards packets in a data communication network. Sometimes called a packet forwarder or, if it also implements an adaptive routing algorithm, a router.

page | In a page-based virtual memory system, the unit of translation between virtual addresses and physical addresses.

page fault | See missing-page exception. page map —Data structure employed by the virtual memory manager to map virtual addresses to physical addresses.

page table | A particular form of a page map, in which the map is organized as an array indexed by page number.

page-map address register | A processor register maintained by the thread manager. It contains a pointer to the page map used by the currently active thread, and it can be changed only when the processor is in kernel mode.

page-removal policy | A policy for deciding which page to move from the primary to the secondary device to make a space to bring in a new page.

pair | and-spare—See pair-and-compare. parallel transmission—A scheme for increasing the data rate between two modules by sending data over several parallel lines that are coordinated by the same clock.

pair-and-compare | A method for constructing fail-fast modules from modules that do not have that property, by connecting the inputs of two replicas of the module together and connecting their outputs to a comparator. When one repairs a failed pair-andcompare module by replacing the entire two-replica module with a spare, rather than identifying and replacing the replica that failed, the method is called pair-and-spare.

partition | To divide a job up and assign it to different physical devices, with the intent that a failure of one device does not prevent the entire job from being done.

password | A secret character string used to authenticate the claimed identity of an individual.

path name | A name with internal structure that traces a path through a naming network. Any prefix of a path name can be thought of as the explicit context reference to use for resolution of the remainder of the path name. See also absolute path name and relative path name.

path selection | In a network-layer routing protocol, when a participant updates its own routing information with new information learned from an exchange with its neighbors.

payload | In a layered description of a communication system, the data that a higher layer has asked a lower layer to send; used to distinguish that data from the headers and trailers that the lower layer adds. (This term seems to have been borrowed from the transportation industry, where it is used frequently in aerospace applications.)

pending | A state of an all-or-nothing action, when that action has not yet either committed or aborted. Also used to describe the value of a variable that was set or changed by a still-pending all-or-nothing action.

persistence | A property of an active agent such as an interpreter that, when it detects it has failed, it keeps trying until it succeeds. Compare with stability and durability, terms that have different technical definitions as explained in Sidebar 2.7. The adjective "persistent" is used in some contexts as a synonym for stable and sometimes also in the sense of immutable.

persistent fault | A fault that cannot be masked by retry. Compare with transient fault and intermittent fault.

persistent sender | A transport protocol participant that, by sending the same message repeatedly, tries to ensure that at least one copy of the message gets delivered.

pessimistic concurrency control | A concurrency control scheme that forces a thread to wait if there is any chance that by proceeding it may interfere with another, concurrent, thread. Pessimistic concurrency control is an effective technique in situations where interference between concurrent threads has a high probability. Compare with optimistic concurrency control.

phase encoding | A method of encoding data for digital transmission in which at least one level transition is associated with each transmitted bit, to simplify framing and recovery of the sender's clock.

physical address | An address that is translated geometrically to read or write data stored on a device. Compare with virtual address.

physical copy | A replica that is organized in a form determined by a lower layer. An example is a replica of a disk that is made by copying it sector by sector. Analogous to physical locking. Compare with logical copy.

physical locking | Locking of lower-layer data objects, typically chunks of data whose extent is determined by the physical layout of a storage medium. Examples of such chunks are disk sectors or even an entire disk. Compare with logical locking.

piggybacking | In an end-to-end protocol, a technique for reducing the number of packets sent back and forth by including acknowledgments and other protocol state information in the header of the next packet that goes to the other end.

pipeline | In networking, a transport protocol design that allows sending a packet before receiving an acknowledgment of the packet previously sent to the same destination. Contrast with lock-step protocol.

plaintext | The result of decryption. Also sometimes used to describe data that has not been encrypted, as in "The mistake was sending that message as plaintext." Compare with ciphertext.

point-to-point | Describes a communication link between two stations, as contrasted with a broadcast or multipoint link.

polling | A style of interaction between threads or between a processor and a device in which one periodically checks whether the other needs attention.

port | In an end-to-end transport protocol, the multiplexing identifier that tells which of several end-to-end applications or application instances should receive the payload.

preemptive scheduling | A scheduling policy in which a thread manager can interrupt and reschedule a running thread at any time.

prepaging | An optimization for a multilevel memory manager in which the manager predicts which pages might be needed and brings them into the primary memory before the application demands them. Compare with demand algorithm.

prepared | In a layered or multiple-site all-or-nothing action, a state of a component action that has announced that it can, on command, either commit or abort. Having reached this state, it awaits a decision from the higher-layer coordinator of the action.

presentation protocol | A protocol that translates semantics and data of the network to match those of the local programming environment.

presented load | See offered load. preventive maintenance—Active intervention intended to increase the mean time to failure of a module or system and thus improve its reliability and availability.

primary copy | Of a set of replicas that are not written or updated synchronously, the one that is considered authoritative and, usually, written or updated first. (Compare with mirror and backup copy.)

primary device | In a multilevel memory system, the memory device that is faster and usually more expensive and thus smaller. Compare with secondary device.

principal | The representation inside a computer system of an agent (a person, a computer, a thread) that makes requests to the security system. A principal is the entity in a computer system to which authorizations are granted; thus, it is the unit of accountability and responsibility in a computer system.

priority scheduling policy | A scheduling policy in which some jobs have priority over other jobs.

privacy | A socially defined ability of an individual (or organization) to determine if, when, and to whom personal (or organizational) information is to be released and also what limitations should apply to use of released information.

private key | In public-key cryptography, the cryptographic key that must be kept secret. Compare with public key.

processing delay | In a communication network, that component of the overall delay contributed by computation that takes place in various protocol layers.

program counter | A processor register that holds the reference to the current or next instruction that the processor is to execute.

progress | A desirable guarantee provided by an atomicity-assuring mechanism: that despite potential interference from concurrency some useful work will be done. An example of such a guarantee is that the atomicity-assuring mechanism will not abort at least one member of the set of concurrent actions. In practice, lack of a progress guarantee can sometimes be repaired by using exponential random backoff. In formal analysis of systems, progress is one component of a property known as "liveness". Progress is an assurance that the system will move toward some specified goal, whereas liveness is an assurance that the system will eventually reach that goal.

propagation delay | In a communication network, the component of overall delay contributed by the velocity of propagation of the physical medium used for communication.

propagation of effects | A property of most systems: a change in one part of the system causes effects in areas of the system that are far removed from the changed part. A good system design tends to minimize propagation of effects.

protection | 1. Synonym for security. 2. Sometimes used in a narrower sense to denote mechanisms and techniques that control the access of executing programs to information.

protection group | A principal that is shared by more than one user.

protocol | An agreement between two communicating parties, for example on the messasges and format of data that they intend to exchange.

public key | In public-key cryptography, the key that can be published (i.e., the one that doesn't have to be kept secret). Compare with private key.

public-key cryptography | A key-based cryptographic transformation that can provide both confidentiality and authenticity of messages without the need to share a secret between sender and recipient. Public-key systems use two cryptographic keys, one of which must be kept secret but does not need to be shared.

publish/subscribe | A communication style using a trusted intermediary. Clients push or pull messages to or from an intermediary. The intermediary determines who actually receives a message and if a message should be fanned out to multiple recipients.

pure name | A name that is not overloaded in any way. The only operations that apply to a pure name are COMPARE, RESOLVE, BIND, and UNBIND. Contrast with overloaded name.

purging | A technique used in some N-modular redundancy designs, in which the voter ignores the output of any replica that, at some time in the past, disagreed with several others.

qualified name | A name that includes an explicit context reference.

quench | (n.) An administrative message sent by a packet forwarder to another forwarder or to an end-to-end-layer sender asking that the forwarder or sender stop sending data or reduce its rate of sending data.

queuing delay | In a communication network, the component of overall delay that is caused by waiting for a resource such as a link to become available.

quorum | A partial set of replicas intended to improve availability. One defines a read quorum and a write quorum that intersect, with the goal that for correctness it is sufficient to read from a read quorum and write to a write quorum.

race condition | A timing-dependent error in thread coordination that may result in threads computing incorrect results (for example, multiple threads simultaneously try to update a shared variable that they should have updated one at a time).

RAID | An acronym for Redundant Array of Independent (or Inexpensive) Disks, a set of techniques that use a controller and multiple disk drives configured to improve some combination of storage performance or durability. A RAID system usually has an interface that is electrically and programmatically identical to a single disk, thus allowing it to transparently replace a single disk.

random access memory | A memory device for which the latency for memory cells chosen at random is approximately the same as the latency obtained by choosing cells in the pattern best suited for that memory device.

random drop | A strategy for managing an overloaded resource: the system refuses service to a queue member chosen at random.

random early detection (RED) | A combination of random drop and early drop.

rate monotonic scheduling policy | A policy that schedules periodic jobs for a real-time system. Each job receives in advance a priority that is proportional to the frequency of the occurrence of that job. The scheduler always runs the highest priority job, preempting a running job, if necessary.

Read and Set Memory (RSM) | A hardware or software function used primarily for implementing locks. RSM loads a value from a memory location into a register and stores another value in the same memory location. The important property of RSM is that no other loads and stores by concurrent threads can come between the load and the store of an RSM. RSM is nearly always implemented as a hardware instruction.

read/write coherence | A property of a memory, that a READ always returns the result of the most recent WRITE.

ready/acknowledge protocol | A data transmission protocol in which each transmission is framed by a ready signal from the sender and an acknowledge signal from the receiver.

real time | 1. (adj.) Describes a system that requires delivery of results before some deadline. 2. (n.) The wall-clock sequence that an all-seeing observer would associate with a series of actions.

real-time scheduling policy | A scheduler that attempts to schedule jobs in such a way that all jobs complete before their deadlines.

reassembly | Reconstructing a message by arranging, in correct order, the segments it was divided into for transmission.

reconciliation | A procedure that compares replicas that are intended to be identical and repairs any differences.

recursive name resolution | A method of resolving path names. The least significant component of the path name is looked up in the context named by the remainder of the path name, which must thus be resolved first.

redo action | An application-specified action that, when executed during failure recovery, produces the effect of some committed component action whose effect may have been lost in the failure. (Some systems call this a "do action". Compare with undo action.)

redundancy | Extra information added to detect or correct errors in data or control signals.

reference | (n.) Use of a name by an object to refer to another object. In grammatical English, the corresponding verb is "to refer to". In computer jargon, the non-standard verb "to reference" appears frequently, and the coined verb "dereference" is a synonym for resolve.

reference string | The string of addresses issued by a thread during its execution (typically the string of the virtual addresses issued by a thread's execution of LOAD and STORE instructions; it may also include the addresses of the instructions themselves).

relative path name | A path name that the name resolver resolves in a default context provided by the environment.

reliability | A statistical measure, the probability that a system is still operating at time t, given that it was operating at some earlier time t0.

reliable delivery | A transport protocol assurance: it provides both at-least-once delivery and data integrity.

remote procedure call (RPC) | A stylized form of client/service interaction in which each request is followed by a response. Usually, remote procedure call systems also provide marshaling and unmarshaling of the request and the response data. The word "procedure" in "remote procedure call" is misleading, since RPC semantics are different from those of an ordinary procedure call: for example, RPC specifically allows for clients and the service to fail independently.

repair | An active intervention to fix or replace a module that has been identified as failing, preferably before the system of which it is a part fails.

repertoire | The set of operations or actions an interpreter is prepared to perform. The repertoire of a general-purpose processor is its instruction set.

replica | 1. One of several identical modules that, when presented with the same inputs, is expected to produce the same output. 2. One of several identical copies of a set of data.

replicated state machine | A method of performing an update to a set of replicas that involves sending the update request to each replica and performing it independently at each replica.

replication | The technique of using multiple replicas to achieve fault tolerance.

repudiate | To disown an apparently authenticated message.

request | The message sent from a client to a service.

resolve | To perform a name-mapping algorithm from a name to the corresponding value.

response | The message sent from a service to a client in response to a previous request.

roll-forward recovery | A write-ahead log protocol with the additional requirement that the application log its outcome record before it performs any install actions. If there is a failure before the all-or-nothing action passes its commit point, the recovery procedure does not need to undo anything; if there is a failure after commit, the recovery procedure can use the log record to ensure that cell storage installs are not lost. Also known as redo logging. Compare with rollback recovery.

rollback recovery | A write-ahead log protocol with the additional requirement that the application perform all install actions before logging an outcome record. If there is a failure before the all-or-nothing action commits, a recovery procedure can use the log record to undo the partially completed all-or-nothing action. Also known as undo logging. Compare with roll-forward recovery.

root | The context used for the interpretation of absolute path names. The name for the root is usually bound to a constant value (typically, a well-known name of a lower layer) and that binding is normally built in to the name resolver at design time.

round-robin scheduling | A preemptive scheduling policy in which a thread runs for some maximum time before the next one is scheduled. When all threads have run, the scheduler starts again with the first thread.

round-trip time | In a network, the time between sending a packet and receiving the corresponding response or acknowledgment. Round-trip time comprises two (possibly different) network transit times and the time required for the correspondent to process the packet and prepare a response.

router | A packet forwarder that also participates in a routing algorithm.

routing algorithm | An algorithm intended to construct consistent, efficient forwarding tables. A routing algorithm can be either centralized, which means that one node calculates the forwarding tables for the entire network, or decentralized, which means that many participants perform the algorithm concurrently.

scheduler | The part of the thread manager that implements the policy for deciding which thread to run. Policies can be preemptive or non-preemptive.

scope | In a layered naming scheme, the set of contexts in which a particular name is bound to the same value.

search | As used in naming, a synonym for multiple lookup. This usage of the term is a highly constrained form of the more general definition of search as used in information retrieval and full-text search systems: to locate all instances of records that match a given query.

search path | A default context reference that consists of the identifiers of the contexts to be used in a multiple lookup name resolution. The word "path" as used here has no connection with its use in path name, and the word "search" has only a distant connection with the concept of key word search.

secondary device | In a multilevel memory system, the memory device that is larger but also usually slower. Compare with primary device.

secrecy | Synonym for confidentiality.

secure area | A physical space or a virtual address space in which confidential information can be safely confined.

secure channel | A communication channel that can safely send information from one secure area to another. The channel may provide confidentiality or authenticity or, more commonly, both.

security | The protection of information and information systems against unauthorized access or modification of information, whether in storage, processing, or transit, and against denial of service to authorized users.

security protocol | A message protocol designed to achieve some security objective (e.g., authenticating a sender). Designers of security protocols must assume that some of the communicating parties are adversaries.

segment | 1. A numbered block of contiguously addressed virtual memory, the block having a range of memory addresses starting with address zero and ending at some specified size. Programs written for a segment-based virtual memory issue addresses that are really two numbers: the first identifies the segment number, and the second identifies the address within that segment. The memory manager must translate the segment number to determine where in real memory the segment is located. The second address may also require translation using a page map. 2. In a communication network, the data that the end to-end layer gives to the network layer for forwarding across the network. A segment is the payload of a packet. Compare with frame (n.), message, and packet.

self-pacing | A property of some transmission protocols. A self-pacing protocol automatically adjusts its transmission rate to match the bottleneck data rate of the network over which it is operating.

semaphore | A special type of shared variable for sequence coordination among several concurrent threads. A semaphore supports two atomic operations: DOWN and UP. If the semaphore's value is larger than zero, DOWN decrements the semaphore and returns to its caller; otherwise, DOWN releases its processor until another thread increases the semaphore using UP. When control returns to the thread that originally issued the DOWN operation, that thread retries the DOWN operation.

sequence coordination | A coordination constraint among threads: for correctness, a certain event in one thread must precede some other certain event in another thread.

sequencer | A special type of shared variable used for sequence coordination. The primary operation on a sequencer is TICKET, which operates likes the "take a number" machine in a bakery or post office: two threads concurrently calling TICKET on the same sequencer receive different values, and the ordering of the values returned corresponds to the time ordering of the execution of TICKET.

serial transmission | A scheme for increasing the data rate between two modules by sending a series of self-clocking bits over a single transmission line with infrequent or no acknowledgments.

serializable | A property of before-or-after actions, that even if several operate concurrently, the result is the same as if they had acted one at a time, in some sequential (in other words, serial) order.

server | A module that implements a service. More than one server might implement the same service, or collaborate to implement a fault tolerant version of the service such that even if a server fails, the service is still available.

service | A module that responds to actions initiated by clients.

set up | The steps required to allocate storage space for and initialize the state of a connection.

shadow copy | A working copy of an object that an all-or-nothing action creates so that it can make several changes to the object while the original remains unmodified. When the all-or-nothing action has made all of the changes, it then carefully exchanges the working copy with the original, thus preserving the appearance that all of the changes occurred atomically. Depending on the implementation, either the original or the working copy may be identified as the "shadow" copy, but the technique is the same in either case.

shared-secret cryptography | A key-based cryptographic transformation in which the cryptographic key for transforming can be easily determined from the key for the reverse transformation, and vice versa. In most shared-secret systems, the keys for a transformation and its reverse transformation are identical.

shared-secret key | The key used by a shared-secret cryptography system.

sharing | Allowing an object to be used by more than one other object without requiring multiple copies of the first object.

sign | To generate an authentication tag by transforming a message so that a receiver can use the tag to verify that the message is authentic. The word "sign" is usually restricted to public-key authentication systems. The corresponding description for shared-secret authentication systems is "generate a MAC".

simple locking | A locking protocol for creating before-or-after actions requiring that no data be read or written before reaching the lock point. For the atomic action to also be all-or-nothing, a further requirement is that no locks be released before commit (or abort). Compare with two-phase locking.

simple serialization | An atomicity protocol requiring that each newly created atomic action must wait to begin execution until all previously started atomic actions are no longer pending.

simplex | Describes a link between two stations that can be used in only one direction. Compare with duplex, half-duplex, and full-duplex.

single-acquire protocol | A simple protocol for locking: a thread can acquire a lock only if some other thread has not already acquired it.

single-event upset | A synonym for transient fault.

slave | In a multiple-site replication scheme, a site that takes update requests from only the master site. Compare with master.

sliding window | In flow control, a technique in which the receiver sends an additional window allocation before it has fully consumed the data from the previous allocation, intending that the new allocation arrive at the sender in time to keep data flowing smoothly, taking into account the transit time of the network.

snoopy cache | In a multiprocessor system with a bus and a cache in each processor, a cache design in which the cache actively monitors traffic on the bus to watch for events that invalidate cache entries.

soft modularity | Modularity defined by convention but not enforced by physical constraints. Compare with enforced modularity.

soft real-time scheduler | A real-time scheduler in which missing a deadline occasionally is acceptable.

soft state | State of a running program that the program can easily reconstruct if it becomes necessary to abruptly terminate and restart the program.

source | The network attachment point that originated the payload of a packet. Sometimes used as shorthand for source address.

source address | An identifier of the source of a packet, usually carried as a field in the header of the packet.

spatial locality | A kind of locality of reference in which the reference string contains clusters of references to adjacent or nearby addresses.

speaks for | A phrase used to express delegation relationships between principals. "A speaks for B" means that B has delegated some authority to A.

speculation | A technique to improve performance by performing an operation in advance of receiving a request on the chance that it will be requested. The hope is that the result can be delivered with less latency and with less setup overhead. Examples include demand paging with larger pages than strictly necessary, prepaging, prefetching, and writing dirty pages before the primary device space is needed.

spin loop | A situation in which a thread waits for an event to happen without releasing the processor.

stability | A property of an object that, once it has a value, it maintains that value indefinitely. Compare with durability and persistence, terms that have different technical definitions, as explained in Sidebar 2.7.

stable binding | A binding that is guaranteed to map a name to the same value for the lifetime of the name space. One of the features of a unique identifier name space.

stack algorithm | A class of page-removal algorithms in which the set of pages in a primary device of size m is always a subset of the set of pages in a primary device of size n, if m is smaller than n. Stack algorithms have the property that increasing the size of the memory is guaranteed not to result in increased numbers of missing-page exceptions.

starvation | An undesirable situation in which several threads are competing for a shared resource and because of adverse scheduling one or more of the threads never receives a share of the resource.

static routing | A method for setting up forwarding tables in which, once calculated, they do not automatically change in response to changes in network topology and load. Compare with adaptive routing.

static scope | An example of an explicit context, used to resolve names of program variables in some programming languages. The name resolver searches for a binding starting with the procedure that used the name, then in the procedure in which the first procedure was defined, and so on. Sometimes called lexical scope. Compare with dynamic scope.

station | A device that can send or receive data over a communication link.

stop and wait | A synonym for lock step.

storage | Another term for memory. Memory devices that are non-volatile and are read and written in large blocks are traditionally called storage devices, but there are enough exceptions that in practice the words "memory" and "storage" should be treated as synonyms.

store and forward | A forwarding network organization in which transport-layer messages are buffered in a non-volatile memory such as magnetic disk, with the goal that they never be lost. Many authors use this term for any forwarding network.

stream | A sequence of data bits or messages that an application intends to flow between two attachment points of a network. It also usually intends that the data of a stream be delivered in the order in which it was sent, and that there be no duplication or omission of data.

strict consistency | An interface requirement that temporary violation of a data invariant during an update never be visible outside of the action doing the update. One feature of the read/write coherence memory model is strict consistency. Sometimes called strong consistency.

stub | A procedure that hides from the caller that the callee is not invoked with the ordinary procedure call conventions. The stub may marshal the arguments into a message and send the message to a service, where another stub unmarshals the message and invokes the callee.

supermodule | A set of replicated modules interconnected in such a way that it acts like a single module.

supervisor call instruction (SVC) | A processor instruction issued by user modules to pass control of the processor to the kernel.

swapping | A feature of some virtual memory systems in which a multilevel memory manager removes a complete address space from a primary device and moves in a complete new one.

synonym | One of multiple names that map to the same value. Compare with alias, a term that usually, but not always, has the same meaning.

system | A set of interconnected components that has an expected behavior observed at the interface with its environment. Contrast with environment.

tail drop | A strategy for managing an overloaded resource: the system refuses service to the queue entry that arrived most recently.

tear down | The steps required to reset the state of a connection and deallocate the space that was used for storage of that state.

temporal locality—A kind of locality of reference in which the reference string contains closely | spaced references to the same address.

thrashing | An undesirable situation in which the primary device is too small to run a thread or a group of threads, leading to frequent missing-page exceptions.

thread | An abstraction that encapsulates the state of a running module. This abstraction encapsulates enough of the state of the interpreter that executes the module so that one can stop a thread at any point in time and later resume it. The ability to stop a thread and resume it later allows virtualization of the interpreter.

thread manager | A module that implements the thread abstraction. It typically provides calls for creating a thread, destroying it, allowing the thread to yield, and coordinating with other threads.

threat | A potential security violation from either a planned attack by an adversary or an unintended mistake by a legitimate user.

throughput | a measure of the rate of useful work done by a service for a given workload.

ticket system | A security system in which each principal maintains a list of capabilities, one for each object to which the principal is authorized to have access.

tolerated error | An error or class of errors that is both detectable and maskable, and for which a systematic recovery procedure has been implemented. Compare with detectable error, maskable error, and untolerated error.

tombstone | A piece of data that will probably never be used again but cannot be discarded because there is still a small chance that it will be needed.

trailer | Information that a protocol layer adds to the end of a packet.

transaction | A multistep action that is both atomic in the face of failure and atomic in the face of concurrency. That is, it is both all-or-nothing and before-or-after.

transactional memory | A memory model in which multiple references to primary memory are both all-or-nothing and before-or-after.

transient fault | A fault that is temporary and for which retry of the putatively failed component has a high probability of finding that it is okay. Sometimes called a singleevent upset. Compare with persistent fault and intermittent fault.

transit time | In a forwarding network, the total delay time required for a packet to go from its source to its destination. In other contexts, this kind of delay is sometimes called latency.

transmission delay | In a communication network, the component of overall delay contributed by the time spent sending a frame at the available data rate.

transport protocol | An end-to-end protocol that moves data between two attachment points of a network while providing a particular set of specified assurances. It can be thought of as a prepackaged set of improvements on the best-effort specification of the network layer.

triple | modular redundancy (TMR)—N-modular redundancy with N = 3.

trusted computing base (TCB) | That part of a system that must work properly to make the overall system secure.

trusted intermediary | A service that acts as the trusted third party on behalf of multiple, perhaps distrustful, clients. It enforces modularity, thereby allowing multiple distrustful clients to share resources in a controlled manner.

two generals dilemma | An intrinsic problem that no finite protocol can guarantee to simultaneously coordinate state values at two places that are linked by an unreliable communication network.

two-phase commit | A protocol that creates a higher-layer transaction out of separate, lower-layer transactions. The protocol first goes through a preparation (sometimes called voting) phase, at the end of which each lower-layer transaction reports either that it cannot perform its part or that it is prepared to either commit or abort. It then enters a commitment phase in which the higher-layer transaction, acting as a coordinator, makes a final decision—thus the name two-phase. Two-phase commit has no connection with the similar-sounding term two-phase locking.

two-phase locking | A locking protocol for before-or-after atomicity that requires that no locks be released until all locks have been acquired (that is, there must be a lock point). For the atomic action to also be all-or-nothing, a further requirement is that no locks for objects to be written be released until the action commits. Compare with simple locking. Two-phase locking has no connection with the similar-sounding term two-phase commit.

undo action | An application-specified action that, when executed during failure recovery or an abort procedure, reverses the effect of some previously performed, but not yet committed, component action. The goal is that neither the original action nor its reversal be visible above the layer that implements the action. Compare with redo and compensate.

unique identifier name space | A name space in which each name, once it is bound to a value, can never be reused for a different value. A unique identifier name space thus provides a stable binding. In a billing system, customer account numbers usually constitute a unique identifier name space.

universal name space | A name space of a naming scheme that has only one context. A universal name space has the property that no matter who uses a name it has the same binding. Computer file systems typically provide a universal name space for absolute path names.

universe of values | The set of all possible values that can be named by a particular naming scheme.

unlimited name space | A name space in which names never have to be reused.

untolerated error | An error or class of errors that is undetectable, unmaskable, or unmasked and therefore can be expected to lead to a failure. Compare with detectable error, maskable error, and tolerated error.

user mode | A feature of a processor that, when set, disallows the use of certain processor features (e.g., changing the page-map address register). Compare with kernel mode.

user-dependent binding | A binding for which a name used by a shared object resolves to different values, depending on the identity of the user of the shared object.

utilization | The percentage of capacity used for a given workload.

valid construction | The term used by software designers for fault avoidance.

value | The thing to which a name is bound. A value may be a real, physical object, or it may be another name either from the original name space or from a different name space.

version history | The set of all values for an object or variable that have ever existed, stored in journal storage.

virtual address | An address that must be translated to a physical address before using it to refer to memory. Compare with physical address.

virtual circuit | A connection intended to carry a stream through a forwarding network, in some ways simulating an electrical circuit.

virtual machine | A method of emulation in which, to maximize performance, a physical processor is used as much as possible to implement virtual instances of itself.

virtual machine monitor | The software that implements virtual machines.

virtual memory manager | A memory manager that implements virtual addresses, resolving them to physical addresses by using, for example, a page map.

virtualization | A technique that simulates the interface of a physical object, in some cases creating several virtual objects using one physical instance, in others creating one large virtual object by aggregating several smaller physical instances, and in yet other cases creating a virtual object from a different kind of physical object.

volatile memory | A kind of memory in which the mechanism of retaining information actively consumes energy. When one disconnects the power source it forgets its information content. Compare with non-volatile memory.

voter | A device used in some NMR designs to compare the output of several nominally identical replicas that all have the same input.

well-known name (or address) | A name or address that has been advertised so widely that one can depend on it not changing for the lifetime of the value to which it is bound. In the United States, the emergency telephone number "911" is a well-known name. In some file system designs, sector or block number 1 of every storage device is reserved as a place to store device data, making "1" a well-known address in that context.

window | In flow control, the quantity of data that the receiving side of a transport protocol is prepared to accept from the sending side.

witness | A (usually cryptographically strong) hash value that attests to the content of a file. Another widely used term for this concept is fingerprint.

working directory | In a file system, a directory used as a default context, for resolution of relative path names.

working set | The set of all addresses to which a thread refers in the interval Δt. If the application exhibits locality of reference, this set of addresses will be small compared to the maximum number of possible addresses during Δt.

write tearing | See atomic storage. write-through—A property of a cache: a write operation updates the value in both the primary device and the secondary device before acknowledging completion of the write. (A cache without the write-through property is sometimes called a write-behind cache.)

write-ahead-log (WAL) protocol | A recovery protocol that requires appending a log record in journal storage before installing the corresponding data in cell storage.

Detailed Licensing

Overview

Title: Principles of Computer System Design (Saltzer and Kaashoek)

Webpages: 76

Applicable Restrictions: Noncommercial

All licenses found:

- CC BY-NC-SA 4.0: 86.8% (66 pages)
- Undeclared: 13.2% (10 pages)

By Page

- Principles of Computer System Design (Saltzer and Kaashoek) - *CC BY-NC-SA 4.0*
  - Front Matter - *CC BY-NC-SA 4.0*
    - TitlePage - *CC BY-NC-SA 4.0*
    - InfoPage - *CC BY-NC-SA 4.0*
    - Acknowledgments - *CC BY-NC-SA 4.0*
    - Table of Contents - *Undeclared*
    - Computer System Design Principles - *CC BY-NC-SA 4.0*
    - Licensing - *Undeclared*
  - 1: The Network as a System and as a System Component - *CC BY-NC-SA 4.0*
    - 1.1: Overview - *CC BY-NC-SA 4.0*
    - 1.2: Interesting Properties of Networks - *CC BY-NC-SA 4.0*
    - 1.3: Getting Organized - Layers - *CC BY-NC-SA 4.0*
    - 1.4: The Link Layer - *CC BY-NC-SA 4.0*
    - 1.5: The Network Layer - *CC BY-NC-SA 4.0*
    - 1.6: The End-to-End Layer - *CC BY-NC-SA 4.0*
    - 1.7: A Network System Design Issue - Congestion Control - *CC BY-NC-SA 4.0*
    - 1.8: Wrapping Up Networks - *CC BY-NC-SA 4.0*
    - 1.9: Case Study - Mapping the Internet to the Ethernet - *CC BY-NC-SA 4.0*
    - 1.10: War Stories - Surprises in Protocol Design - *CC BY-NC-SA 4.0*
    - 1.11: Exercises - *CC BY-NC-SA 4.0*
  - 2: Fault Tolerance - Reliable Systems from Unreliable Components - *CC BY-NC-SA 4.0*
    - 2.1: Overview - *CC BY-NC-SA 4.0*
    - 2.2: Faults, Failures, and Fault-Tolerant Design - *CC BY-NC-SA 4.0*
    - 2.3: Measures of Reliability and Failure Tolerance - *CC BY-NC-SA 4.0*
    - 2.4: Tolerating Active Faults - *CC BY-NC-SA 4.0*
    - 2.5: Systematically Applying Redundancy - *CC BY-NC-SA 4.0*
    - 2.6: Applying Redundancy to Software and Data - *CC BY-NC-SA 4.0*
    - 2.7: Wrapping Up Reliability - *CC BY-NC-SA 4.0*
    - 2.8: Application - A Fault Tolerance Model for CMOS RAM - *CC BY-NC-SA 4.0*
    - 2.9: War Stories - Fault-Tolerant Systems That Failed - *CC BY-NC-SA 4.0*
    - 2.10: Exercises - *CC BY-NC-SA 4.0*
  - 3: Atomicity - All-or-nothing and Before-or-after - *CC BY-NC-SA 4.0*
    - 3.1: Overview - *CC BY-NC-SA 4.0*
    - 3.2: Atomicity - *CC BY-NC-SA 4.0*
    - 3.3: All-Or-Nothing Atomicity I - Concepts - *CC BY-NC-SA 4.0*
    - 3.4: All-Or-Nothing Atomicity II - Pragmatics - *CC BY-NC-SA 4.0*
    - 3.5: Before-Or-After Atomicity I - Concepts - *CC BY-NC-SA 4.0*
    - 3.6: Before-Or-After Atomicity II - Pragmatics - *CC BY-NC-SA 4.0*
    - 3.7: Atomicity Across Layers and Multiple Sites - *CC BY-NC-SA 4.0*
    - 3.8: A More Complete Model of Disk Failure (Advanced Topic) - *CC BY-NC-SA 4.0*
    - 3.9: Case Studies - Machine Language Atomicity - *CC BY-NC-SA 4.0*
    - 3.10: Exercises - *CC BY-NC-SA 4.0*
  - 4: Consistency - *CC BY-NC-SA 4.0*
    - 4.1: Overview - *CC BY-NC-SA 4.0*
    - 4.2: Constraints and Interface Consistency - *CC BY-NC-SA 4.0*
    - 4.3: Cache Coherence - *CC BY-NC-SA 4.0*
    - 4.4: Durable Storage Revisited - Geographically Separated Replicas - *CC BY-NC-SA 4.0*
    - 4.5: Reconciliation - *CC BY-NC-SA 4.0*
    - 4.6: Perspectives - *CC BY-NC-SA 4.0*
    - 4.7: Exercises - *CC BY-NC-SA 4.0*

- 5: Information Security - *CC BY-NC-SA 4.0*
  - 5.1: Overview - *Undeclared*
  - 5.2: Introduction to Secure Systems - *CC BY-NC-SA 4.0*
  - 5.3: Authenticating Principals - *CC BY-NC-SA 4.0*
  - 5.4: Authenticating Messages - *CC BY-NC-SA 4.0*
  - 5.5: Message Confidentiality - *CC BY-NC-SA 4.0*
  - 5.6: Security Protocols - *CC BY-NC-SA 4.0*
  - 5.7: Authorization-Controlled Sharing - *CC BY-NC-SA 4.0*
  - 5.8: Reasoning About Authentication (Advanced Topic) - *CC BY-NC-SA 4.0*
  - 5.9: Cryptography as a Building Block (Advanced Topic) - *CC BY-NC-SA 4.0*
  - 5.10: Summary - *CC BY-NC-SA 4.0*
  - 5.11: Case Study - Transport Layer Security (TLS) for the Web - *CC BY-NC-SA 4.0*
  - 5.12: War Stories - Security System Breaches - *CC BY-NC-SA 4.0*
  - 5.13: Exercises - *CC BY-NC-SA 4.0*
- 6: Ancillary Materials - *CC BY-NC-SA 4.0*
  - 6.1: Powerpoint Presentations - *CC BY-NC-SA 4.0*
  - 6.2: Problem Sets - *Undeclared*
    - 6.2.1: Problem Sets 1-10 - *Undeclared*
    - 6.2.2: Problem Sets 11-20 - *Undeclared*
    - 6.2.3: Problem Sets 21-30 - *Undeclared*
    - 6.2.4: Problem Sets 31-32 - *Undeclared*
  - 6.3: Suggestions for Further Reading - *Undeclared*
- Back Matter - *CC BY-NC-SA 4.0*
  - Index - *CC BY-NC-SA 4.0*
  - Glossary - *CC BY-NC-SA 4.0*
  - Detailed Licensing - *Undeclared*