Jerome H. Saltzer
M. Frans Kaashoek

# Principles of Computer System Design

## AN INTRODUCTION

- SYSTEMS
- ORGANIZATION
- NAMING
- CLIENTS AND SERVICES
- VIRTUALIZATION
- PERFORMANCE
- NETWORKS
- ATOMICITY
- FAULT TOLERANCE
- CONSISTENCY
- SECURITY

MORGAN KAUFMANN

Morgan Kaufmann Publishers is an imprint of Elsevier.
30 Corporate Drive, Suite 400,
Burlington, MA 01803, USA

This book is printed on acid-free paper. ♾

Copyright © 2009 by Jerome H. Saltzer and M. Frans Kaashoek. All rights reserved.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. All trademarks that appear or are otherwise referred to in this work belong to their respective owners. Morgan Kaufmann Publishers does not have any relationship or affiliation with such trademark owners nor do such trademark owners confirm, endorse or approve the contents of this work. Readers, however, should contact the appropriate companies for more information regarding trademarks and any related registrations.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, scanning, or otherwise—without prior written permission of the publisher.

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, E-mail: permissions@elsevier.com. You may also complete your request online via the Elsevier homepage (*http://elsevier.com*), by selecting "Support & Contact" then "Copyright and Permission" and then "Obtaining Permissions."

**Suggestions, comments, and corrections:** Please send correspondence by e-mail to Saltzer@mit.edu and kaashoek@mit.edu

**Library of Congress Cataloging-in-Publication Data**
Application Submitted

ISBN: 978-0-12-374957-4

For information on all Morgan Kaufmann publications,
visit our Web site at *www.mkp.com* or *www.elsevierdirect.com*

Printed in the United States of America
09 10 11 12 13 10 9 8 7 6 5 4 3 2 1

*Typeset by*: diacriTech, Chennai, India

Working together to grow
libraries in developing countries
www.elsevier.com | www.bookaid.org | www.sabre.org
ELSEVIER BOOK AID International Sabre Foundation

*To Marlys and Mathilda*

# List of Sidebars

## Part I


**CHAPTER 1 Systems**

**Sidebar 1.1:** Stopping a Supertanker ..... 6
**Sidebar 1.2:** Why Airplanes can't Fly ..... 6
**Sidebar 1.3:** Terminology: Words Used to Describe System Composition ..... 9
**Sidebar 1.4:** The Cast of Characters and Organizations..... 14
**Sidebar 1.5:** How Modularity Reshaped the Computer Industry ..... 21
**Sidebar 1.6:** Why Computer Technology has Improved Exponentially with Time ..... 32

**CHAPTER 2 Elements of Computer System Organization**

**Sidebar 2.1:** Terminology: Durability, Stability, and Persistence ..... 46
**Sidebar 2.2:** How Magnetic Disks Work..... 49
**Sidebar 2.3:** Representation: Pseudocode and Messages ..... 54
**Sidebar 2.4:** What is an Operating System? ..... 79
**Sidebar 2.5:** Human Engineering and the Principle of Least Astonishment ..... 85

**CHAPTER 3 The Design of Naming Schemes**

**Sidebar 3.1:** Generating a Unique Name from a Timestamp..... 125
**Sidebar 3.2:** Hypertext Links in the Shakespeare Electronic Archive..... 129

**CHAPTER 4 Enforcing Modularity with Clients and Services**

**Sidebar 4.1:** Enforcing Modularity with a High-Level Languages..... 154
**Sidebar 4.2:** Representation: Timing Diagrams ..... 156
**Sidebar 4.3:** Representation: Big-Endian or Little-Endian?..... 158
**Sidebar 4.4:** The X Window System..... 162
**Sidebar 4.5:** Peer-to-peer: Computing without Trusted Intermediaries..... 164

**CHAPTER 5 Enforcing Modularity with Virtualization**

**Sidebar 5.1:** RSM, Test-and-Set and Avoiding Locks..... 224
**Sidebar 5.2:** Constructing a Before-or-After Action without Special Instructions ..... 226
**Sidebar 5.3:** Bootstrapping an Operating System ..... 239
**Sidebar 5.4:** Process, Thread, and Address Space ..... 249
**Sidebar 5.5:** Position-Independent Programs..... 251

**Sidebar 5.6:** Interrupts, Exceptions, Faults, Traps, and Signals ..... 259
**Sidebar 5.7:** Avoiding the Lost Notification Problem with Semaphores ..... 277

**CHAPTER 6 Performance**
**Sidebar 6.1:** Design Hint: When in Doubt Use Brute Force ..... 301
**Sidebar 6.2:** Design Hint: Optimize for the Common Case ..... 307
**Sidebar 6.3:** Design Hint: Instead of Reducing Latency, Hide It ..... 310
**Sidebar 6.4:** RAM Latency ..... 323
**Sidebar 6.5:** Design Hint: Separate Mechanism from Policy ..... 330
**Sidebar 6.6:** OPT is a Stack Algorithm and Optimal ..... 343
**Sidebar 6.7:** Receive Livelock ..... 350
**Sidebar 6.8:** Priority Inversion ..... 358

## PART II [ON-LINE]

**CHAPTER 7 The Network as a System and a System Component**
**Sidebar 7.1:** Error Detection, Checksums, and Witnesses
**Sidebar 7.2:** The Internet
**Sidebar 7.3:** Framing Phase-Encoded Bits
**Sidebar 7.4:** Shannon's Capacity Theorem
**Sidebar 7.5:** Other End-to-End Transport Protocol Interfaces
**Sidebar 7.6:** Exponentially Weighted Moving Averages
**Sidebar 7.7:** What does an Acknowledgment Really Mean?
**Sidebar 7.8:** The Tragedy of the Commons
**Sidebar 7.9:** Retrofitting TCP
**Sidebar 7.10:** The Invisible Hand

**CHAPTER 8 Fault Tolerance: Reliable Systems from Unreliable Components**
**Sidebar 8.1:** Reliability Functions
**Sidebar 8.2:** Risks of Manipulating MTTFs
**Sidebar 8.3:** Are Disk System Checksums a Wasted Effort?
**Sidebar 8.4:** Detecting Failures with Heartbeats.

**CHAPTER 9 Atomicity: All-or-nothing and Before-or-after**
**Sidebar 9.1:** Actions and Transactions
**Sidebar 9.2:** Events that Might Lead to Invoking an Exception Handler
**Sidebar 9.3:** Cascaded Aborts
**Sidebar 9.4:** The Many Uses of Logs

**CHAPTER 10 Consistency**

**CHAPTER 11 Information Security**

**Sidebar 11.1:** Privacy
**Sidebar 11.2:** Should Designs and Vulnerabilities be Public?
**Sidebar 11.3:** Malware: Viruses, Worms, Trojan Horses, Logic Bombs, Bots, Drive-by Downloads, etc.
**Sidebar 11.4:** Why are Buffer Overrun Bugs so Common?
**Sidebar 11.5:** Authenticating Personal Devices: the Resurrecting Duckling Policy
**Sidebar 11.6:** The Kerberos Authentication System
**Sidebar 11.7:** Economics of Computer Security
**Sidebar 11.8:** Secure Hash Algorithm (SHA)

# Preface

To the best of our knowledge this textbook is unique in its scope and approach. It provides a broad and in-depth introduction to the main principles and abstractions for engineering computer systems, be it an operating system, a client/service application, a database system, a secure Web site, or a fault-tolerant disk cluster. These principles and abstractions are timeless and are of value to any student or professional reader, whether or not specializing in computer systems. The principles and abstractions derive from insights that have proven to work over generations of computer systems, the authors' own experience with building computer systems, and teaching about them for several decades.

The book teaches a broad set of principles and abstractions, yet it explores them in depth. It captures the core of a concept using pseudocode so that readers can test their understanding of a concrete instance of the concept. Using pseudocode, the book carefully documents the essence of client/service computing, remote procedure calls, files, threads, address spaces, best-effort networks, atomicity, authenticated messages, and so on. This approach continues in the problem sets, where readers can explore the design of a wide range of systems by studying their pseudocode.

This printed textbook is Part I of a two-part publication, containing just the first six chapters. Part II, consisting of Chapters 7–11 and additional supporting materials, is posted on-line as an open educational resource. For details of how and where to find Part II on-line, see "Where to find Part II and other on-line materials" on page xxix.

## WHY THIS TEXTBOOK?

Many fundamental ideas concerning computer systems, such as design principles, modularity, naming, abstraction, concurrency, communications, fault tolerance, and atomicity, are common to several of the upper-division electives of the Computer Science and Engineering (CSE) curriculum. A typical CSE curriculum starts with two beginning courses, one on programming and one on hardware. It then branches out, with one of the main branches consisting of systems-oriented electives that carry labels such as

- Operating systems
- Networks
- Database systems
- Distributed systems
- Programming languages
- Software engineering
- Security
- Fault tolerance
- Concurrency
- Architecture

The primary problem with this list is that it has grown over the last three decades, and most students interested in systems do not have the time to take all or even

several of those courses. The typical response is for the CSE curriculum to require either "choose three" or "take Operating Systems plus two more". The result is that most students end up with no background at all in the remaining topics. In addition, none of the electives can assume that any of the other electives have preceded it, so common material ends up being repeated several times. Finally, students who are not planning to specialize in systems but want to have some background have little choice but to go into depth in one or two specialized areas.

This book cuts across all of these courses, identifying common mechanisms and design principles, and explaining in depth a carefully chosen set of cross-cutting ideas. This approach provides an opportunity to teach a core undergraduate course that is accessible to all Computer Science and Engineering students, whether or not they intend to specialize in systems. On the one hand, students who will just be users of systems will take away a solid grounding, while on the other hand those who plan to plan to make a career out of designing systems can learn more advanced material more effectively through electives that have the same names as in the list above but with more depth and less duplication. Both groups will acquire a broad base of what the authors hope are timeless concepts rather than current and possibly short-lived techniques. We have found this course structure to be effective at M.I.T.

The book achieves its extensive range of coverage without sacrificing intellectual depth by focusing on underlying and timeless concepts that will serve the student over an entire professional career, rather than providing detailed expositions of the mechanics of operation of current systems that will soon become obsolete. A pervading philosophy of the book is that pedagogy takes precedence over job training. For example, the text does not teach a particular operating system or rely on a single computer architecture. Instead it introduces models that exhibit the main ideas found in contemporary systems, but in forms less cluttered with evolutionary vestiges. The pedagogical model is that for someone who understands the concepts, the detailed mechanics of operation of any particular system can easily and quickly be acquired from other books or from the documentation of the system itself. At the same time, the text makes concepts concrete using pseudocode fragments, so that students have something specific to examine and to test their understanding of the concepts.

## FOR WHOM IS THIS BOOK INTENDED?

The authors intend the book for students and professionals who will

- Design computer systems.
- Supervise the design of computer systems.
- Engineer applications of computer systems to information management.
- Direct the integration of computer systems within an organization.
- Evaluate performance of computer systems.

- Keep computer systems technologically up to date.
- Go on to study individual topics such as networks, security, or transaction management in greater depth.
- Work in other areas of computer science and engineering, but would like to have a basic understanding of the main ideas about computer systems.

***Level:*** This book provides an *introduction* to computer systems. It does not attempt to explore every issue or get to the bottom of those issues it does explore. Instead, its goal is for the reader to acquire insight into the complexities of the systems he or she will be depending on for the remainder of a career as well as the concepts needed to interact with system designers. It provides a solid foundation about the mechanisms that underlie operating systems, database systems, data networks, computer security, distributed systems, fault tolerant computing, and concurrency. By the end of the book, the reader should in principle be able to follow the detailed engineering of many aspects of computer systems, be prepared to read and understand current professional literature about systems, and know what questions to ask and where to find the answers.

The book can be used in several ways. It can be the basis for a one-semester, two-quarter, or three-quarter series on computer systems. Or one or two selected chapters can be an introduction of a traditional undergraduate elective or a graduate course in operating systems, networks, database systems, distributed systems, security, fault tolerance, or concurrency. Used in this way, a single book can serve a student several times.Another possibility is that the text can be the basis for a graduate course in systems in which students review those areas they learned as undergraduates and fill in the areas they missed.

***Prerequisites:*** The book carefully limits its prerequisites. When used as a textbook, it is intended for juniors and seniors who have taken introductory courses on software design and on computer hardware organization, but it does not require any more advanced computer science or engineering background. It defines new terms as it goes, and it avoids jargon, but nevertheless it also assumes that the reader has acquired some practical experience with computer systems from a summer job or two or from laboratory work in the prerequisite courses. It does not require that the reader be fluent in any particular computer language, but rather be able to transfer general knowledge about computer programming languages to the varied and sometimes *ad hoc* programming language used in pseudocode examples.

***Other Readers:*** Professionals should also find this book useful. It provides a modern and forward-looking perspective on computer system design, based on enforcing modularity. This perspective recognizes that over the last decade or two, the primary design challenge has become that of keeping complexity under control rather than fighting resource constraints. In addition, professionals who in college took only a subset of the classes in computer systems or an operating systems class that focused on resource management will find that this text refreshes them with a modern and broader perspective.

## HOW TO USE THIS BOOK

***Exercises and Problem Sets:*** Each chapter of the textbook ends with a few short-answer exercises intended to test understanding of some of the concepts in that chapter. At the end of the book is a much longer collection of problem sets that challenge the reader to apply the concepts to new and different problems similar to those that might be encountered in the real world. In most cases, the problem sets require concepts from several chapters. Each problem set identifies the chapter or chapters on which it is focused, but later problem sets typically draw concepts from all earlier chapters. Answers to the exercises and solutions for the problem sets are available from the publisher in a separate book for instructors.

The exercises and problem sets can be used in several ways:

- *As tools for learning.* In this mode, the answers and solutions are available to the student, who is encouraged to work the exercises and problem sets and come up with answers and solutions on his or her own. By comparing those answers and solutions with the expected ones, the student receives immediate feedback that can correct misconceptions and can raise questions about ambiguities or misunderstandings. One technique to encourage study of the exercises and solutions is to announce that questions identical to or based on one or more of the problem sets will appear on a forthcoming examination.
- *As homework or examination material.* In this mode, exercises and problem sets are assigned as homework, and the student hands in answers that are evaluated and handed back together with copies of the answers and solutions.
- *As the source of ideas for new exercises and problem sets.*

***Case Studies and Readings:*** To complement the text, the reader should supplement it with readings from the professional technical literature and with case studies. Following the last chapter is a selected bibliography of books and papers that offer wisdom, system design principles, and case studies surrounding the study of systems. By varying the pace of introduction and the number and intellectual depth of the readings, the text can be the basis for a one-term undergraduate core course, a two-term or three-quarter undergraduate sequence, or a graduate-level introduction to computer systems.

***Projects:*** Our experience is that for a course that touches many aspects of computer systems, a combination of several lightweight hands-on assignments (for example, experimentally determine the size of the caches of a personal computer or trace asymmetrical routes through the Internet), plus one or two larger paper projects that involve having a small team do a high-level system design (for example, in a 10-page report design a reliable digital storage system for the Library of Congress), make an excellent adjunct to the text. On the other hand, substantial programming projects that require learning the insides of a particular system take so much homework time that when combined with a broad concepts course they create an overload. Courses with programming projects do work well in follow-on specialized electives, for example,

on operating systems, networks, databases, or distributed systems. For this reason, at M.I.T. we assign programming projects in several advanced electives but not in the systems course that is based on this textbook.

***Support:*** Several on-line resources provide support for this textbook. The first of these resources is a set of course syllabi, reading lists, problem sets, videotaped lectures, quizzes, and quiz solutions. A second resource is a Web site of the publisher that is devoted to collecting resources and links of interest to students, professional readers, and instructors. A third resource is a mostly open Web site for communication between instructors of M.I.T. course 6.033, which uses this text, and their current students. It contains announcements, readings, and problem assignments for the current or most recent teaching term. In addition to current class communications, this Web site also holds an archive going back to 1995 that includes

- Design project assignments
- Hands-on assignments
- Examinations and solutions (These overlap the exercises and problem sets of the textbook but they also include exam questions and answers about the outside readings.)
- Lecture and recitation schedules
- Reading assignments and essay questions about the readings

Instructions for finding all of these on-line resources are in the section "Where to find Part II and other on-line materials".

## HOW THE BOOK IS ORGANIZED

Because not every instructor may want to use every chapter of the textbook, it is presented in what, at least at the time of publication, may be viewed as a somewhat novel way: The first six chapters, which the authors consider to be the core materials for almost any course about computer systems, appear in this printed book. The remaining five chapters are available on-line from the authors and M.I.T. under a Creative Commons license that permits free, unlimited non-commercial use and remixing. The on-line chapters are also available on the Web site of the publisher of this textbook. There are many forward cross-references from the core chapters to the later chapters. Those cross-references are identified as in this example: "This topic is explored in more detail in Section 7.4.1 [on-line]".

***Themes:*** Three themes run through this textbook. First, as suggested by its title, the text emphasizes the importance of systematic design principles. As each design principle is encountered for the first time, it appears in display form with a label and a mnemonic catchphrase. When that design principle is encountered again, it is identified by its name and highlighted with a distinctive print format as a reminder of its wide applicability. The design principles are also summarized on the inside front cover of this book. A second theme is that the text is network-centered, introducing communication and networks in the beginning chapters and building on that base in the

succeeding chapters. A third theme is that it is security-centered, introducing enforced modularity in early chapters and adding successively more stringent enforcement methods in succeeding chapters. The security chapter ends the book, not because it is an afterthought, but because it is the logical culmination of a development based on enforced modularity. Traditional texts and courses teach about threads and virtual memory primarily as a resource allocation problem. This text approaches those topics primarily as ways of providing and enforcing modularity, while at the same time taking advantage of multiple processors and large address spaces.

***Terminology and examples:*** The text identifies and develops concepts and design principles that are common to several specialty fields: software engineering, programming languages, operating systems, distributed systems, networking, database systems, and machine architecture. Experienced computer professionals are likely to find that at least some parts of this text use examples, ways of thinking, and terminology that seem unusual, even foreign to their traditional ways of explaining their favorite topics. But workers from these different specialties will compile different lists of what seems foreign. The reason is that, historically, workers within these specialties have identified what turn out to be identical underlying concepts and design principles, but they have used different language, different perspectives, different examples, and different terminology to explain them.

This text chooses, for each concept, what the authors believe is the most pedagogically effective explanation and examples, adopting widely used terminology wherever possible. In cases where different specialty areas use conflicting terms, glossaries and sidebars provide bridges and discuss terminology collisions. The result is a novel, but in our experience effective, way of teaching new generations of Computer Science and Engineering students what is fundamental about computer system design. With this starting point, when the student reads an advanced book or paper or takes an advanced elective course, he or she should be able to immediately recognize familiar concepts cloaked in the terminology of the specialty. A scientist would explain this approach by saying "The physics is independent of the units of measurement." A similar principle applies to the engineering of computer systems: "The concepts are independent of the terminology".

***Citations:*** The text does not use citations as a scholarly method of identifying the originators of each concept or idea; if it did, the book would be twice as thick. Instead the citations that do appear are pointers to related materials that the authors think are worth knowing about. There is one exception: certain sections are devoted to war stories, which may have been distorted by generations of retelling. These stories include citations intended to identify the known sources of each story, so that the reader has a way to assess their validity.

## CHAPTER CONTENT

***Relation to ACM/IEEE recommendations:*** The ACM/IEEE Computer Science and Engineering recommendations of 2001 and 2004 describe two layers. The first layer is a set of modules that constitute an appropriate CSE education. The second layer consists of

several suggested packagings of those modules into term-sized courses. This book may be best viewed as a distinct, modern packaging of the modules, somewhat resembling the ACM/IEEE Computer Science 2001 recommendation CS226c, Operating Systems and Networking (compressed), but with the additional scope of naming, fault tolerance, atomicity, and both system and network security. It also somewhat resembles the ACM/IEEE Computer engineering 2004 recommendation $CPE_D203$, Operating Systems and Net-Centric computing, with the additional scope of naming, fault tolerance, atomicity, and cryptographic protocols.

***Chapter 1: Systems.*** This chapter lays out the general philosophy of the authors on ways to think about systems, with examples illustrating how computer systems are similar to, and different from, other engineering systems. It also introduces three main ideas: (1) the importance of systematic design principles, (2) the role of modularity in controlling complexity of large systems, and (3) methods of enforcing modularity.

***Chapter 2: Elements of Computer System Organization.*** This chapter introduces three key methods of achieving and taking advantage of modularity in computer systems: abstraction, naming, and layers. The discussion of abstraction lightly reviews computer architecture from a systems perspective, creating a platform on which the rest of the book builds, but without simple repetition of material that readers probably already know. The naming model is fundamental to how computer systems are modularized, yet it is a subject usually left to advanced texts on programming language design. The chapter ends with a case study of the way in which naming, layering, and abstraction are applied in the UNIX file system. Because the case study develops as a series of pseudocode fragments, it provides both a concrete example of the concepts of the chapter and a basis for reference in later chapters.

***Chapter 3: Design of Naming Schemes.*** This chapter continues the discussion of naming in system design by introducing pragmatic engineering considerations and reinforcing the role that names play in organizing a system as a collection of modules. The chapter ends with a case study and a collection of war stories. The case study uses the Uniform Resource Locator (URL) of the World Wide Web to show an example of nearly every naming scheme design consideration. The war stories are examples of failures of real-world naming systems, illustrating what goes wrong when a designer ignores or is unaware of design considerations.

***Chapter 4: Enforcing Modularity with Clients and Services.*** The first three chapters developed the importance of modularity in system design. This chapter begins the theme of enforcing that modularity by introducing the client/service model, which is a powerful and widely used method of allowing modules to interact without interfering with one another. This chapter also begins the network-centric perspective that pervades the rest of the book. At this point, we view the network only as an abstract communication system that provides a strong boundary between client and service. Two case studies again help nail down the concepts. The first is of the Internet Domain Name System (DNS), which provides a concrete illustration of the concepts of both Chapters 3 and 4. The second case study, that of the Sun Network

File System (NFS), builds on the case study of the UNIX file system in Chapter 2 and illustrates the impact of remote service on the semantics of application programming interfaces.

***Chapter 5: Enforcing Modularity with Virtualization.*** This chapter switches attention to enforcing modularity within a computer by introducing virtual memory and virtual processors, commonly called threads. For both memory and threads, the discussion begins with an environment that has unlimited resources. The virtual memory discussion starts with an assumption of many threads operating in an unlimited address space and then adds mechanisms to prevent threads from unintentionally interfering with one another's data—addressing domains and the user/kernel mode distinction. Finally, the text examines limited address spaces, which require introducing virtual addresses and address translation, along with the inter-address-space communication problems that they create.

Similarly, the discussion of threads starts with the assumption that there are as many processors as threads, and concentrates on coordinating their concurrent activities. It then moves to the case where a limited number of real processors are available, so thread management is also required. The discussion of thread coordination uses eventcounts and sequencers, a set of mechanisms that are not often seen in practice but that fit the examples in a natural way. Traditionally, thread coordination is among the hardest concepts for the first-time reader to absorb. Problem sets then invite readers to test their understanding of the principles with semaphores and condition variables.

The chapter explains the concepts of virtual memory and threads both in words and in pseudocode that help clarify how the abstract ideas actually work, using familiar real-world problems. In addition, the discussion of thread coordination is viewed as the first step in understanding atomicity, which is the subject of Chapter 9 [on-line].

The chapter ends with a case study and an application. The case study explores how enforced modularity has evolved over the years in the Intel x86 processor family. The application is the use of virtualization to create virtual machines. The overall perspective of this chapter is to focus on enforcing modularity rather than on resource management, taking maximum advantage of contemporary hardware technology, in which processor chips are multicore, address spaces are 64 bits wide, and the amount of directly addressable memory is measured in gigabytes.

***Chapter 6: Performance.*** This chapter focuses on intrinsic performance bottlenecks that are found in common across many kinds of computer systems, including operating systems, databases, networks, and large applications. It explores two of the traditional topics of operating systems books—resource scheduling and multilevel memory management—but in a context that emphasizes the importance of maintaining perspective on performance optimization in a world where each decade brings a thousand-fold improvement in some underlying hardware capabilities while barely affecting other performance metrics. As an indication of this different perspective, scheduling is illustrated with a disk arm scheduling problem rather than the usual time-sharing processor scheduler.

***Chapters 7 through 11 are on-line, in Part II of the book. Their contents are described in the section titled "About Part II" on page 369, and information on how to locate them can be found in "Where to find Part II and other on-line materials".***

***Suggestions for Further Reading.*** A selected reading list includes commentary on why each selection is worth reading. The selection emphasis is on books and papers that provide insight rather than materials that provide details.

***Problem Sets.*** The authors use examinations not just as a method of assessment, but also as a method of teaching. Therefore, some of the exercises at the end of each chapter and the problem sets at the end of the book (all of which are derived from examinations administered over the years while teaching the material of this textbook) go well beyond simple practice with the concepts. In working the problems out, the student explores alternative designs, learns about variations of techniques seen in the textbook, and becomes familiar with interesting, sometimes exotic, ideas and methods that have been proposed for or used in real system designs. The problem sets generally have significant setup, and they ask questions that require applying concepts creatively, with the goal of understanding the trade-offs that arise in using these methods.

***Glossary.*** As mentioned earlier, the literature of computer systems derives from several different specialties that have each developed their own dictionaries of system-related concepts. This textbook adopts a uniform terminology throughout, and the Glossary offers definitions of each significant term of art, indicates which chapter introduces the term, and in many cases explains different terms used by different workers in different specialties. For completeness and for easy reference, the Glossary in this book includes terms introduced in Part II.

***Index of Concepts.*** The index tells where to find the defining discussion of every concept. In addition, it lists every application of each of the design principles. (For completeness, it includes concepts that are introduced in Part II, listing just the chapter number.)

# Where to find Part II and other On-line Materials

1. Professors Saltzer and Kaashoek and MIT OpenCourseWare* provide, free of charge, on-line versions of Chapters 7 through 11, additional problem sets, a copy of the glossary, and a comprehensive index in the form of one Portable Document Format (PDF) file per chapter or section and also a single PDF file containing the entire set. Those materials can be found at

   http://ocw.mit.edu/Saltzer-Kaashoek

2. The publisher of this printed book also maintains a set of on-line resources at

   www.ElsevierDirect.com/9780123749574

   Click on the link "Companion Materials" where you will find Part II of the book as well as other resources, including figures from the text in several formats. Additional materials for instructors (registration required) can be found by clicking the "Manual" link.

3. Teaching and support materials can be found at

   http://ocw.mit.edu/6-033

4. The Web site for the current MIT class that uses this textbook, including the archives of older teaching materials, is at

   http://mit.edu/6.033

   (Some copyrighted or privacy-sensitive materials on that Web site are restricted to current MIT students.)

*The M.I.T. OpenCourseWare initiative places on-line, for non-commercial free access, teaching materials from many M.I.T. courses, and thus is helping set a standard for curricula in science and engineering. In addition to Chapters 7 through 11, OpenCourseWare publishes on-line materials for the M.I.T. course that uses these materials, 6.033. Thus, an instructor interested in making use of the textbook can find in one place course syllabi, reading lists, problem sets, videotaped lectures, quizzes, and solutions.

# 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 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)

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)

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 [on-line])

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 [on-line])

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 [on-line] and 10 [on-line])

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 [on-line])

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. The 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.

*Jerome H. Saltzer*
*M. Frans Kaashoek*
*2009*

CHAPTER

# Systems

# 1

## CHAPTER CONTENTS

**Overview** ..... **2**
**1.1 Systems and Complexity** ..... **3**
1.1.1 Common Problems of Systems in Many Fields ..... 3
1.1.2 Systems, Components, Interfaces, and Environments ..... 8
1.1.3 Complexity ..... 10
**1.2 Sources of Complexity** ..... **13**
1.2.1 Cascading and Interacting Requirements ..... 13
1.2.2 Maintaining High Utilization ..... 17
**1.3 Coping with Complexity I** ..... **19**
1.3.1 Modularity ..... 19
1.3.2 Abstraction ..... 20
1.3.3 Layering ..... 24
1.3.4 Hierarchy ..... 25
1.3.5 Putting it Back Together: Names make Connections ..... 26
**1.4 Computer Systems are the Same but Different** ..... **27**
1.4.1 Computer Systems have no Nearby Bounds on Composition ..... 28
1.4.2 d(technology)/dt is Unprecedented ..... 31
**1.5 Coping With Complexity II** ..... **35**
1.5.1 Why Modularity, Abstraction, Layering, and Hierarchy aren't Enough ..... 36
1.5.2 Iteration ..... 36
1.5.3 Keep it Simple ..... 39
**What The Rest of this Book is About** ..... **40**
**Exercises** ..... **41**

**Principles of Computer System Design: An Introduction**
Copyright © 2009 by Jerome H. Saltzer and M. Frans Kaashoek. All rights of reproduction in any form reserved.
DOI: 10.1016/B978-0-12-374957-4.00009-8

## OVERVIEW

This book is about computer systems, and this chapter introduces some of the vocabulary and concepts used in designing computer systems. It also introduces "systems perspective", a way of thinking about systems that is global and encompassing rather than focused on particular issues. A full appreciation of this way of thinking can't really be captured in a short summary, so this chapter is actually just a preview of ideas that will be developed in depth in succeeding chapters.

The usual course of study of computer science and engineering begins with linguistic constructs for describing computations (software) and physical constructs for realizing computations (hardware). It then branches, focusing, for example, on the theory of computation, artificial intelligence, or the design of systems, which itself is usually divided into specialities: operating systems, transaction and database systems, computer architecture, software engineering, compilers, computer networks, security, and reliability. Rather than immediately tackling one of those specialties, we assume that the reader has completed the introductory courses on software and hardware, and we begin a broad study of computer systems that supports the entire range of systems specialties.

Many interesting applications of computers require

- fault tolerance
- coordination of concurrent activities
- geographically separated but linked data
- vast quantities of stored information
- protection from mistakes and intentional attacks
- interactions with many people

To develop applications that have these requirements, the designer must look beyond the software and hardware and view the computer system as a whole. In doing so, the designer encounters many new problems—so many that the limit on the scope of computer systems generally arises neither from laws of physics nor from theoretical impossibility, but rather from limitations of human understanding.

Some of these same problems have counterparts, or at least analogs, in other systems that have, at most, only incidental involvement of computers. The study of systems is one place where computer engineering can take advantage of knowledge from other engineering areas: civil engineering (bridges and skyscrapers), urban planning (the

Much wisdom about systems that has accumulated over the centuries is passed along in the form of folklore, maxims, aphorisms and quotations. Some of that wisdom is captured in the boxes at the bottom of these pages.

Everything should be made as simple as possible, but no simpler.

**— commonly attributed to Albert Einstein; it is actually a paraphrase of a comment he made in a 1933 lecture at Oxford.**

design of cities), mechanical engineering (automobiles and air conditioning), aviation and space flight, electrical engineering, and even ecology and political science. We start by looking at some of those common problems. Then we will examine two ways in which computer systems pose problems that are quite different. Don't worry if some of the examples are of things you have never encountered or are only dimly aware of. The sole purpose of the examples is to illustrate the range of considerations and similarities across different kinds of systems.

As we proceed in this chapter and throughout the book, we shall point out a series of *system design principles*, which are rules of thumb that usually apply to a diverse range of situations. Design principles are not immutable laws, but rather guidelines that capture wisdom and experience and that can help a designer avoid making mistakes. The astute reader will quickly realize that sometimes a tension, even to the point of contradiction, exists between different design principles. Nevertheless, if a designer finds that he or she is violating a design principle, it is a good idea to review the situation carefully.

At the first encounter of a design principle, the text displays it prominently. Here is an example, found on page 16.

---

**Avoid excessive generality**

*If it's good for everything, it's good for nothing.*

---

Each design principle thus has a formal title ("Avoid excessive generality") and a brief informal description ("If it's good for . . ."), which are intended to help recall the principle. Most design principles will show up several times, in different contexts, which is one reason why they are useful. The text highlights later encounters of a principle such as: *avoid excessive generality*. A list of all of the design principles in the book can be found on the inside front cover and also in the index, under "Design principles".

The remaining sections of this chapter discuss common problems of systems, the sources of those problems, and techniques for coping with them.

## 1.1 SYSTEMS AND COMPLEXITY

### 1.1.1 Common Problems of Systems in Many Fields

The problems one encounters in these many kinds of systems can usefully be divided into four categories: *emergent properties, propagation of effects, incommensurate scaling*, and *trade-offs*.

> Seek simplicity and distrust it.
>
> — **Alfred North Whitehead, *The Concept of Nature* (1920)**

#### 1.1.1.1 Emergent Properties

*Emergent properties* are properties that are not evident in the individual components of a system, but show up when combining those components, so they might also be called surprises. Emergent properties abound in most systems, although there can always be a (fruitless) argument about whether or not careful enough prior analysis of the components might have allowed prediction of the surprise. It is wise to avoid this argument and instead focus on an unalterable fact of life: some things turn up only when a system is built.

Some examples of emergent properties are well known. The behavior of a committee or a jury often surprises outside observers. The group develops a way of thinking that could not have been predicted from knowledge about the individuals. (The concept of—and the label for—emergent properties originated in sociology.) When the Millennium Bridge for pedestrians over the River Thames in London opened, its designers had to close it after only a few days. They were surprised to discover that pedestrians synchronize their footsteps when the bridge sways, causing it to sway even more. Interconnection of several electric power companies to allow load sharing helps reduce the frequency of power failures, but when a failure finally occurs it may take down the entire interconnected structure. The political surprise is that the number of customers affected may be large enough to attract the unwanted attention of government regulators.

#### 1.1.1.2 Propagation of Effects

The electric power inter-tie also illustrates the second category of system problems—*propagation of effects*—when a tree falling on a power line in Oregon leads to the lights going out in New Mexico, 1000 miles away. What looks at first to be a small disruption or a local change can have effects that reach from one end of a system to the other. An important requirement in most system designs is to limit the impact of failures. As another example of propagation of effects, consider an automobile designer's decision to change the tire size on a production model car from 13 to 15 inches. The reason for making the change might have been to improve the ride. On further analysis, this change leads to many other changes: redesigning the wheel wells, enlarging the spare tire space, rearranging the trunk that holds the spare tire, and moving the back seat forward slightly to accommodate the trunk redesign. The seat change makes knee room in the back seat too small, so the backs of the seats must be made thinner, which in turn reduces the comfort that was the original reason for changing the tire size, and it may also reduce safety in a collision. The extra weight of the trunk and rear seat design means that stiffer rear springs are now needed. The rear axle ratio must be modified to keep the force delivered to the road by the wheels correct, and the speedometer gearing must be changed to agree with the new tire size and axle ratio.

Those effects are the obvious ones. In complicated systems, as the analysis continues, more distant and subtle effects normally appear. As a typical example, the

> Our life is frittered away by detail . . . simplicity, simplicity, simplicity!
>
> **— Henry David Thoreau, *Walden; or, Life in the Woods* (1854)**

automobile manufacturer may find that the statewide purchasing office for Texas does not currently have a certified supplier for replacement tires of the larger size. Thus there will probably be no sales of cars to the Texas government for two years, which is the length of time it takes to add a supplier onto the certified list. Folk wisdom characterizes propagation of effects as: "There are no small changes in a large system".

#### *1.1.1.3 Incommensurate Scaling*

The third characteristic problem encountered in the study of systems is *incommensurate scaling*: as a system increases in size or speed, not all parts of it follow the same scaling rules, so things stop working. The mathematical description of this problem is that different parts of the system exhibit different orders of growth. Some examples:

- Galileo observed that "nature cannot produce a . . . giant ten times taller than an ordinary man unless by . . . greatly altering the proportions of his limbs and especially of his bones, which would have to be considerably enlarged over the ordinary" [*Discourses and Mathematical Demonstrations on Two New Sciences*, second day, Leiden, 1638]. In a classic 1928 paper, "On being the right size" [see Suggestions for Further Reading 1.4.1], J. B. S. Haldane uses the example of a mouse, which, if scaled up to the size of an elephant, would collapse of its own weight. For both examples, the reason is that weight grows with volume, which is proportional to the cube of linear size, but bone strength, which depends primarily on cross-sectional area, grows only with the square of linear size. Thus a real elephant requires a skeletal arrangement that is quite different from that of a scaled-up mouse.
- The Egyptian architect Sneferu tried to build larger and larger pyramids. Unfortunately, the facing fell off the pyramid at Meidum, and the ceiling of the burial chamber of the pyramid at Dashur cracked. He later figured out that he could escalate a pyramid to the size of the pyramids at Giza by lowering the ratio of the pyramid's height to its width. The reason this solution worked has apparently never been completely analyzed, but it seems likely that incommensurate scaling was involved—the weight of a pyramid increases with the cube of its linear size, while the strength of the rock used to create the ceiling of a burial chamber increases only with the area of its cross-section, which grows with the square.
- The captain of a modern oil supertanker finds that the ship is so massive that when underway at full speed it takes 12 miles to bring it to a straight line stop—but 12 miles is beyond the horizon as viewed from the ship's bridge (see Sidebar 1.1 for the details).
- The height of a skyscraper is limited by the area of lower floors that must be devoted to providing access to the floors above. The amount of access area

> By undue profundity we perplex and enfeeble thought.
>
> **— Edgar Allan Poe, "The Murders in the Rue Morgue" (1841)**

**Sidebar 1.1 Stopping a Supertanker** A little geometry reveals that the distance to the visual horizon is proportional to the square root of the height of the bridge. That height (presumably) grows with the first power of the supertanker's linear dimension. The energy required to stop or turn a supertanker is proportional to its mass, which grows with the third power of its linear dimensions. The time required to deliver the stopping or turning energy is less clear, but pushing on the rudder and reversing the propellers are the only tools available, and both of those have surface area that grows with the square of the linear dimension.

Here is the bottom line: if we double the tanker's linear dimensions, the momentum goes up by a factor of 8, and the ability to deliver stopping or turning energy goes up by only a factor of 4, so we need to see twice as far ahead. Unfortunately, the horizon will be only 1.414 times as far away. Inevitably, there is some size for which visual navigation must fail.

required (for example, for elevators and stairs) is proportional to the number of people who have offices on higher floors. That number is in turn proportional to the number of higher floors multiplied by the usable area of each floor. If all floors have the same area, and the number of floors increases, at some point the bottom floor will be completely used up providing access to higher floors, so the bottom floor provides no added value (apart from being able to brag about the building's height). In practice, the economics of office real estate dictate that no more than 25% of the lowest floor be devoted to access.

Incommensurate scaling shows up in most systems. It is usually the factor that limits the size or speed range that a single system design can handle. On the other hand, one must be cautious with scaling arguments. They were used at the beginning of the twentieth century to support the claim that it was a waste of time to build airplanes (see Sidebar 1.2).

#### *1.1.1.4 Trade-offs*

The fourth problem of system design is that many constraints present themselves as *trade-offs*. The general model of a trade-off begins with the observation that there is a limited amount of some form of goodness in the universe, and the design challenge is first to maximize that goodness, second to avoid wasting it, and third to allocate it to the places where it will help the most. One common form of trade-off is sometimes called the *waterbed effect*: pushing down on a problem at one point causes another problem to pop up somewhere else. For example, one can typically push a hardware circuit to run at a higher clock rate, but that change increases both power consumption and the risk of timing errors. It may be possible to reduce the risk of timing errors by making the circuit physically smaller, but then less area will be available to dissipate the heat caused

KISS: Keep It Simple, Stupid.

**— traditional management folklore; source lost in the mists of time**

**Sidebar 1.2 Why Airplanes can't Fly** The weight of an airplane grows with the third power of its linear dimension, but the lift, which is proportional to surface area, can grow only with the second power. Even if a small plane can be built, a larger one will never get off the ground.

This line of reasoning was used around 1900 by both physicists and engineers to argue that it was a waste of time to build heavier-than-air machines. Alexander Graham Bell proved that this argument wasn't the whole story by flying box kites in Maine in the summer of 1902. In his experiments he attached two box kites side by side, a configuration that doubled the lifting surface area, but also allowed removal of the redundant material and supports where the two kites touched. Thus, the lift-to-weight ratio actually improved as the scale increased. Bell published his results in "The tetrahedral principle in kite structure" [see Suggestions for Further Reading 1.4.2].

by the increased power consumption. Another common form of trade-off appears in *binary classification*, which arises, for example, in the design of smoke detectors, spam (unwanted commercial e-mail message) filters, database queries, and authentication devices. The general model of binary classification is that we wish to classify a set of things into two categories based on the presence or absence of some property, but we lack a direct measure of that property. We therefore instead identify and use some indirect measure, known as a *proxy.* Occasionally, this scheme misclassifies something. By adjusting parameters of the proxy, the designer may be able to reduce one class of mistakes (in the case of a smoke detector, unnoticed fires; for a spam filter, legitimate messages marked as spam), but only at the cost of increasing some other class of mistakes (for the smoke detector, false alarms; for the spam filter, spam marked as legitimate messages). Appendix A explores the binary classification trade-off in more detail. Much of a system designer's intellectual effort goes into evaluating various kinds of trade-offs.

Emergent properties, propagation of effects, incommensurate scaling, and trade-offs are issues that the designer must deal with in every system. The question is how to build useful computer systems in the face of such problems. Ideally, we would like to describe a constructive theory, one that allows the designer systematically to synthesize a system from its specifications and to make necessary trade-offs with precision, just as there are constructive theories in such fields as communications systems, linear control systems, and (to a certain extent) the design of bridges and skyscrapers. Unfortunately, in the case of computer systems, we find that we were apparently born too soon. Although our early arrival on the scene offers the challenge to develop the missing theory, the problem is quickly apparent—we work almost entirely by analyzing *ad hoc* examples rather than by synthesizing.

Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.

**— Alan J. Perlis, "Epigrams in Programming" (1982)**

So, in place of a well-organized theory, we use case studies. For each subtopic in this book, we shall begin by identifying requirements with the apparent intent of deriving the system structure from the requirements. Then, almost immediately we switch to case studies and work backwards to see how real, in-the-field systems meet the requirements that we have set. Along the way we point out where systematic approaches to synthesizing a system from its requirements are beginning to emerge, and we introduce representations, abstractions, and design principles that have proven useful in describing and building systems. The intended result of this study is insight into how designers create real systems.

### 1.1.2 Systems, Components, Interfaces, and Environments

*Webster's Third New International Dictionary, Unabridged*, defines a system as "a complex unity formed of many often diverse parts subject to a common plan or serving a common purpose." Although this definition will do for casual use of the word, engineers usually prefer something a bit more concrete. We identify the "many often diverse parts" by naming them *components*. We identify the "unity" and "common plan" with the *interconnections* of the components, and we perceive the "common purpose" of a system to be to exhibit a certain behavior across its *interface* to an *environment*. Thus we formulate our technical definition: **A *system* is a set of interconnected components that has an expected behavior observed at the interface with its environment**.

The underlying idea of the concept of system is to divide all the things in the world into two groups: those under discussion and those not under discussion. Those things under discussion are part of the system—those that are not are part of the *environment*. For example, we might define the solar system as consisting of the sun, planets, asteroids, and comets. The environment of the solar system is the rest of the universe. (Indeed, the word "universe" is a synonym for environment.)

There are always interactions between a system and its environment; these interactions are the *interface* between the system and the environment. The interface between the solar system and the rest of the universe includes gravitational attraction for the nearest stars and the exchange of electromagnetic radiation. The primary interfaces of a personal computer typically include things such as a display, keyboard, speaker, network connection, and power cord, but there are also less obvious interfaces such as the atmospheric pressure, ambient temperature and humidity, and the electromagnetic noise environment.

One studies a system to predict its overall behavior, based on information about its components, their interconnections, and their individual behaviors. Identifying the components, however, depends on one's point of view, which has two aspects, *purpose* and *granularity*. One may, with different purposes in mind, look at a system quite

> And simplicity is the unavoidable price we must pay for reliability.
>
> **— Charles Anthony Richard Hoare, "Data Reliability" (1975)**

**Sidebar 1.3 Terminology: Words used to Describe System Composition** Since systems can contain component subsystems that are themselves systems from a different point of view, decomposition of systems is recursive. To avoid recursion in their writing, authors and designers have come up with a long list of synonyms, all trying to capture this same concept: *systems*, *subsystems*, *components*, *elements*, *constituents*, *objects*, *modules*, *submodules*, *assemblies*, *subassemblies*, and so on.

differently. One may also choose any of several different granularities. These choices affect one's identification of the components of the system in important ways.

To see how point of view can depend on purpose, consider two points of view of a jet aircraft as a system. The first looks at the aircraft as a flying object, in which the components of the system include the body, wings, control surfaces, and engines. The environment is the atmosphere and the earth, with interfaces consisting of gravity, engine thrust, and air drag. A second point of view looks at the aircraft as a passenger-handling system. Now, the components include seats, flight attendants, the air conditioning system, and the galley. The environment is the set of passengers, and the interfaces are the softness of the seats, the meals, and the air flowing from the air conditioning system.

In the first point of view, the aircraft as a flying object, the seats, flight attendants, and galley were present, but the designer considers them primarily as contributors of weight. Conversely, in the second point of view, as a passenger-handling system, the designer considers the engine as a source of noise and perhaps also exhaust fumes, and probably ignores the control surfaces on the wings. Thus, depending on point of view, we may choose to ignore or consolidate certain system components or interfaces.

The ability to choose granularity means that a component in one context may be an entire system in another. From an aircraft designer's point of view, a jet engine is a component that contributes weight, thrust, and perhaps drag. On the other hand, the manufacturer of the engine views it as a system in its own right, with many components—turbines, hydraulic pumps, bearings, afterburners, all of which interact in diverse ways to produce thrust—one interface with the environment of the engine. The airplane wing that supports the engine is a component of the aircraft system, but it is part of the environment of the engine system.

When a system in one context is a component in another, it is usually called a *subsystem* (but see Sidebar 1.3). The composition of systems from subsystems or decomposition of systems into subsystems can be carried on to as many levels as is useful.

In summary, then, to analyze a system one must establish a point of view to determine which things to consider as components, what the granularity of those

Pluralitas non est ponenda sine neccesitate. (Plurality should not be assumed without necessity.)

**— William of Ockham (14th century. Popularly known as "Occam's razor," though the idea itself is said to appear in writings of greater antiquity.)**

components should be, where the boundary of the system lies, and which interfaces between the system and its environment are of interest.

As we use the term, a *computer system* or an *information system* is a system intended to store, process, or communicate information under automatic control. Further, we are interested in systems that are predominantly digital. Here are some examples:

- a personal computer
- the onboard engine controller of an automobile
- the telephone system
- the Internet
- an airline ticket reservation system
- the space shuttle ground control system
- a World Wide Web site

At the same time we will sometimes find it useful to look at examples of nondigital and nonautomated information handling systems, such as the post office or library, for ideas and guidance.

### 1.1.3 Complexity

Webster's definition of "system" used the word "complex". Looking up that term, we find that *complex* means "difficult to understand". Lack of systematic understanding is the underlying feature of complexity. It follows that complexity is both a subjective and a relative concept. That is, one can argue that one system is more complex than another, but even though one can count up various things that seem to contribute to complexity, there is no unified measure. Even the argument that one system is more complex than another can be difficult to make compelling—again because of the lack of a unified measure. In place of such a measure, we can borrow a technique from medicine: describe a set of *signs* of complexity that can help confirm a diagnosis. As a corollary, we abandon hope of producing a definitive description of complexity. We must instead look for its signs, and if enough appear, argue that complexity is present. To that end, here are five signs of complexity:

1. **Large number of components.** Sheer size certainly affects our view of whether or not a system rates the description "complex".

2. **Large number of interconnections.** Even a few components may be interconnected in an unmanageably large number of ways. For example, the Sun and the known planets comprise only a few components, but every one has gravitational

> Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher. (It is as if perfection be attained not when there is nothing more to add, but when there is nothing more to take away.)
>
> **— Antoine de Saint-Exupéry, *Terre des Hommes* (1939)**

attraction for every other, which leads to a set of equations that are unsolvable (in closed form) with present mathematical techniques. Worse, a small disturbance can, after a while, lead to dramatically different orbits. Because of this sensitivity to disturbance, the solar system is technically *chaotic*. Although there is no formal definition of chaos for computer systems, that term is often informally applied.

3. **Many irregularities.** By themselves, a large number of components and interconnections may still represent a simple system, if the components are repetitive and the interconnections are regular. However, a lack of regularity, as shown by the number of exceptions or by non-repetitive interconnection arrangements, strongly suggests complexity. Put another way, exceptions complicate understanding.

4. **A long description.** Looking at the best available description of the system one finds that it consists of a long laundry list of properties rather than a short, systematic specification that explains every aspect. Theoreticians formalize this idea by measuring what they call the "Kolmogorov complexity" of a computational object as the length of its shortest specification. To a certain extent, this sign may be merely a reflection of the previous three, although it emphasizes an important aspect of complexity: it is relative to understanding. On the other hand, lack of a methodical description may also indicate that the system is constructed of ill-fitting components, is poorly organized, or may have unpredictable behavior, any of which add complexity to both design and use.

5. **A team of designers, implementers, or maintainers.** Several people are required to understand, construct, or maintain the system. A fundamental issue in any system is whether or not it is simple enough for a single person to understand all of it. If not, it is a complex system because its description, construction, or maintenance will require not just technical expertise but also coordination and communication across a team.

Again, an example can illustrate: contrast a small-town library with a large university library. There is obviously a difference in scale: the university has more books, so the first sign is present. The second sign is more subtle: where the small library may have a catalog to guide the user, the university library may have not only a catalog,

'Tis the gift to be simple, 'tis the gift to be free,
'Tis the gift to come down where we ought to be;
And when we find ourselves in the place just right,
'Twill be in the valley of love and delight.
When true simplicity is gained
To bow and to bend we shan't be ashamed;
To turn, turn will be our delight,
Till by turning, turning we come round right.

— ***Simple Gifts*, traditional Shaker hymn**

but also finding aids, readers' guides, abstracting services, journal indexes, and so on. Although these elaborations make the large library more useful (at least to the experienced user), they also complicate the task of adding a new item to the library: someone must add many interconnections (in this case, cross-references) so that the new item can be found in all the intended ways. The third sign, a large number of exceptions, is also apparent. Where the small library has only a few classifications (fiction, biography, nonfiction, and magazines) and a few exceptions (oversized books are kept over the newspaper rack), the university library is plagued with exceptions. Some books are oversized, others come on microfilm or on digital media, some books are rare or valuable and must be protected, the books that explain how to build a hydrogen bomb can be loaned only to certain patrons, some defy cataloging in any standard classification system. As for the fourth sign, any user of a large university library will confirm that there are no methodical rules for locating a piece of information and that library usage is an art, not a science.

Finally, the fifth sign of complexity, a staff of more than one person, is evident in the university library. Where many small towns do in fact have just one librarian—typically an energetic person who knows each book because at one time or another he or she has had occasion to touch it—the university library has not only many personnel, but even specialists who are familiar with only one facet of library operations, such as the microform collection.

The university library exhibits all five signs of complexity, but unanimity is not essential. On the other hand, the presence of only one or two of the signs may not make a compelling case for complexity. Systems considered in thermodynamics contain an unthinkably large number of components (elementary particles) and interactions, yet from the right point of view they do not qualify as complex because there is a simple, methodical description of their behavior. It is exactly when we lack such a simple, methodical description that we have complexity.

One objection to conceiving complexity as being based on the five signs is that all systems are indefinitely, perhaps infinitely, complex because the deeper one digs the more signs of complexity turn up. Thus, even the simplest digital computer is made of gates, which are made with transistors, which are made of silicon, which is composed of protons, neutrons, and electrons, which are composed of quarks, which some physicists suggest are describable as vibrating strings, and so on. We shall address this objection in a moment by limiting the depth of digging, a technique known as *abstraction*. The complexity that we are interested in and worried about is the complexity that remains despite the use of abstraction.

> Whatever man builds . . . all of man's . . . efforts . . . invariably culminate in . . . a thing whose sole and guiding principle is . . . simplicity . . . perfection of invention touches hands with absence of invention, as if . . . [there] were a line that had not been invented but . . . [was] in the beginning . . . hidden by nature and in the end . . . found by the engineer.
>
> **— Antoine de Saint-Exupéry, *Terre des Hommes* (1939)**

## 1.2 SOURCES OF COMPLEXITY

There are many sources of complexity, but two merit special mention. The first is in the number of requirements that the designer expects a system to meet. The second is one particular requirement: maintaining high utilization.

### 1.2.1 Cascading and Interacting Requirements

A primary source of complexity is just the list of requirements for a system. Each requirement, viewed by itself, may seem straightforward. Any particular requirement may even appear to add only easily tolerable complexity to an existing list of requirements. The problem is that the accumulation of many requirements adds not only their individual complexities but also complexities from their interactions. This interaction complexity arises from pressure for generality and exceptions that add complications, and it is made worse by change in individual requirements over time.

Most users of a personal computer have by now encountered some version of the following scenario: The vendor announces a new release of the program you use to manage your checkbook, and the new release has some feature that seems important or useful (e.g., it handles the latest on-line banking systems), so you order the program. Upon trying to install it, you discover that this new release requires a newer version of some shared library package. You track down that newer version and install it, only to find that the library package requires a newer version of the operating system, which you had not previously had any reason to install. Biting the bullet, you install the latest release of the operating system, and now the checkbook program works, but your add-on hard disk begins to act flaky. On investigation it turns out that the disk vendor's proprietary software is incompatible with the new operating system release. Unfortunately, the disk vendor is still debugging an update for the disk software, and the best thing available is a beta test version that will expire at the end of the month.

The underlying cause of this scenario is that the personal computer has been designed to meet many requirements: a well-organized file system, expandability of storage, ability to attach a variety of I/O devices, connection to a network, protection from malevolent persons elsewhere in the network, usability, reliability, low cost—the list goes on and on. Each of these requirements adds complexity of its own, and the interactions among them add still more complexity.

Similarly, the telephone system has, over the years, acquired a large number of line customizing features—call waiting, call return, call forwarding, originating and terminating call blocking, reverse billing, caller ID, caller ID blocking, anonymous call

> When in doubt, make it stout, and of things you know about.
> When in doubt, leave it out.
>
> **— folklore sayings from the automobile industry**

**Sidebar 1.4 The Cast of Characters and Organizations** In concrete examples throughout this book, the reader will encounter a standard cast of characters named Alice, Bob, Charles, Dawn, Ella, and Felipe. Alice is usually the sender of a message, and Bob is its recipient. Charles is sometimes a mutual acquaintance of Alice and Bob. The others play various supporting roles, depending on the example. When we come to security, an adversarial character named Lucifer will appear. Lucifer's role is to crack the security measures and perhaps interfere with the presumably useful work of the other characters.

The book also introduces a few fictional organizations. There are two universities: Pedantic University, on the Internet at Pedantic.edu, and The Institute of Scholarly Studies, at Scholarly.edu. There are also four mythical commercial organizations on the Internet at TrustUs.com, ShopWithUs.com, Awesome.net, and Awful.net.

M.I.T. Professor Ronald Rivest introduced Alice and Bob to the literature of computer science in Suggestions for Further Reading 11.5.1. Any other resemblance to persons living or dead or organizations real or imaginary is purely coincidental.

rejection, do not disturb, vacation protection—again, the list goes on and on. These features interact in so many ways that there is a whole field of study of "feature interaction" in telephone systems. The study begins with debates over what *should* happen. For example, so-called 900 numbers have the feature called reverse billing—the called party can place a charge on the caller's bill. Alice (Alice is the first character we have encountered in our cast of characters, described in Sidebar 1.4) has a feature that blocks outgoing calls to reverse billing numbers. Alice calls Bob, whose phone is forwarded to a 900 number. Should the call go through, and if so, which party should pay for it, Bob or Alice? There are three interacting features, and at least four different possibilities: block the call, allow the call and charge it to Bob, ring Bob's phone, or add yet another feature that (for a monthly fee) lets Bob choose the outcome.

The examples suggest that there is an underlying principle at work. We call it the:

**Principle of escalating complexity**

*Adding a requirement increases complexity out of proportion.*

The principle is subjective because complexity itself is subjective—its magnitude is in the mind of the beholder. Figure 1.1 provides a graphical interpretation of the

Perfection must be reached by degrees; she requires the slow hand of time.

**— attributed to François-Marie Arouet (Voltaire)**

**FIGURE 1.1**

The principle of escalating complexity.

principle. Perhaps the most important thing to recognize in studying this figure is that the complexity barrier is soft: as you add features and requirements, you don't hit a solid roadblock to warn you to stop adding. It just gets worse.

As the number of requirements grows, so can the number of exceptions and thus the complications. It is the incredible number of special cases in the United States tax code that makes filling out an income tax return a complex job. The impact of any one exception may be minor, but the cumulative impact of many interacting exceptions can make a system so complex that no one can understand it. Complications also can arise from outside requirements such as insistence that a certain component must come from a particular supplier. That component may be less durable, heavier, or not as available as one from another supplier. Those properties may not prevent its use, but they add complexity to other parts of the system that have to be designed to compensate.

Meeting many requirements with a single design is sometimes expressed as a need for *generality.* Generality may be loosely defined as "applying to a variety of circumstances." Unfortunately, generality contributes to complexity, so it comes with a tradeoff, and the designer must use good judgment to decide how much of the generality is actually *wanted*. As an extreme example, an automobile with four independent steering wheels, each controlling one tire, offers some kind of ultimate in generality, almost all of which is unwanted. Here, both the aspect of unwantedness and the resulting complexity of guidance of the auto are obvious enough, but in many cases both of these aspects are more difficult to assess: How much does a proposed form of generality complicate the system, and to what extent is that generality really useful? Unwanted generality also contributes to complexity indirectly: users of a system with excessive generality will adopt styles of usage that simplify and suppress generality that they do not need. Different users may adopt different styles and then discover that they cannot easily exchange ideas with one another. Anyone who tries to use a personal computer customized by someone else will notice this problem.

Periodically, someone tries to design a vehicle that one can drive on the highway, fly, and use as a boat, but the result of such a general design does not seem to work

> The best is the enemy of the good.
>
> **— François-Marie Arouet (Voltaire), *Dictionnaire Philosophique* (1764)**

well in any of the intended modes of transport. To help counter excessive generality, experience suggests another design principle:*

**Avoid excessive generality**

*If it is good for everything, it is good for nothing.*

There is a tension between exceptions and generality. Part of the art of designing a subsystem is to make its features general enough to minimize the number of exceptions that must be handled as special cases. This area is one where the judgment of the system designer is most evident.

Counteracting the effects of incommensurate scaling can be an additional source of complexity. Haldane, in his essay "On being the right size", points out that small organisms such as insects absorb enough oxygen to survive through their skins, but larger organisms, which require an amount of oxygen proportional to the cube of their linear size, don't have enough surface area. To compensate for this incommensurate scaling, they add complexity in the form of lungs and blood vessels to absorb and deliver oxygen throughout their bodies. In the case of computers, the programmer of a 4-bit microprocessor to control a toaster can in a few days successfully write the needed code entirely with binary numbers, while the programmer of a video game with a 64-bit processor and 40 gigabytes of supporting data requires an extensive array of tools—compilers, image or video editors, special effects generators, and the like, as well as an operating system, to be able to get the job done within a lifetime. Incommensurate scaling has required employment of a far more complex set of tools.

Finally, a major source of complexity is that requirements *change*. System designs that are successful usually remain in use for a long time, during which the environment of the system changes. Improvements in hardware technology may lead the system maintainers to want to upgrade to faster, cheaper, or more reliable equipment. Meanwhile, knowledge of how to maintain the older equipment (and the supply of spare parts) may be disappearing. As users accumulate experience with the system, it becomes clearer that some additional requirements should have been part of the design and that some of the original requirements were less important than originally thought. Often a system will expand in scale, sometimes far beyond the vision of its original designers.

In each of these cases, the ground rules and assumptions that the original designers used to develop the system begin to lose their relevance. The system designers

*Computer industry consultant (and erstwhile instructor of the course for which this textbook was written) Michael Hammer suggested the informal version of this design principle.

A complex system that works is invariably found to have evolved from a simple system that works.

**— John Gall, *Systemantics* (1975)**

may have foreseen some environmental changes, but there were other changes they probably did not anticipate. As changes to meet unforeseen requirements occur, they usually add complexity. Because it can be difficult to change the architecture of a deployed system (Section 1.3 explains why), there is a powerful incentive to make changes within the existing architecture, whether or not that is the best thing to do. Propagation of effects can amplify the problems caused by change because more distant effects of a change may not be noticed until someone invokes some rarely used feature. When those distant effects finally do surface, the maintainer may again find it easiest to deal with them locally, perhaps by adding exceptions. Incommensurate scaling effects begin to dominate behavior when a later maintainer scales a system up in size or replaces the underpinnings with faster hardware. Again, the first response to these effects is usually to make local changes (sometimes called *patches*) to counteract them rather than to make fundamental changes in design that would require changing several modules or changing interfaces between modules.

A closely related problem is that as systems grow in complexity with the passage of time, even the simplest change, such as to repair a bug, has an increasing risk of introducing another bug because complexity tends to obscure the full impact of the repair. A common phenomenon in older systems is that the number of bugs introduced by a bug fix release may exceed the number of bugs fixed by that release.*

The bottom line is that as systems age, they tend to accumulate changes that make them more complex. The lifetime of a system is usually limited by the complexity that accumulates as it evolves farther and farther from its original design.

### 1.2.2 Maintaining High Utilization

One requirement by itself is frequently a specific source of complexity. It starts with a desire for high performance or high efficiency. Whenever a scarce resource is involved, an effort arises to keep its *utilization* high.

Consider, for example, a single-track railroad line running through a long, narrow canyon.† To improve the utilization of the single track, and push more traffic through, one might allow trains to run both ways at the same time by installing a switch and a short side track in a wide spot about halfway through the canyon. Then, if one is careful in scheduling, trains going in opposite directions will meet at the side track, where

*This phenomenon was documented by Laszlo A. Belady and Meir M. Lehman in "A model of large program development", *IBM Systems Journal 15*, 3 (1976), pages 225–252.

†Michael D. Schroeder suggested this example of a railroad line in a canyon.

Een schip op't droogh gezeylt, dat is een seeker baken. (A ship, sailed on to dry land, that is a certain beacon. Learn from the mistakes of others.)

**— Jacob Cats, *Mirror on Old and New Times* (1632), based on a Dutch proverb**

they can pass each other, effectively doubling the number of trains that the track can carry each day. However, the train operations are now much more complex than they used to be. If either train is delayed, the schedules of both are disrupted. A signaling system needs to be installed because human schedulers or operators may make mistakes. And—an emergent property—the trains now have a limit on their length. If two trains are to pass in the middle, at least one of them must be short enough to pull completely onto the side track.

The train in the canyon is a good illustration of how efforts to increase utilization can increase complexity. When striving for higher utilization, one usually encounters a general design principle that economists call

**The law of diminishing returns**

*The more one improves some measure of goodness, the more effort the next improvement will require.*

This phenomenon is particularly noticeable in attempts to use resources more efficiently: the more completely one tries to use a scarce resource, the greater the complexity of the strategies for use, allocation, and distribution. Thus a rarely used street intersection requires no traffic control beyond a rule that the car on the right has the right-of-way. As usage increases, one must apply progressively more complex measures: stop signs, then traffic lights, then marked turning lanes with multiphase lights, then vehicle sensors to control the lights. As traffic in and out of an airport nears the airport's capacity, measures such as stacking planes, holding them on the ground at distant airports, or coordinated scheduling among several airlines must be taken. As a general rule, the more one tries to increase utilization of a limited resource, the greater the complexity (see Figure 1.2).

**FIGURE 1.2**

An example of diminishing returns: complexity grows with increasing utilization.

The perceptive reader will notice that Figures 1.1 and 1.2 are identical. It would

> It is impossible to foresee the consequences of being clever.
>
> **— Christopher Strachey, as reported by Roger Needham**

be useful to memorize this figure because some version of it can be used to describe many different things about systems.

## 1.3 COPING WITH COMPLEXITY I

As one might expect, with many fields contributing examples of systems with common problems and sources of complexity, some common techniques for coping with complexity have emerged. These techniques can be loosely divided into four general categories: *modularity, abstraction, layering,* and *hierarchy.* The following sections sketch the general method of each of the techniques. In later chapters many examples of each technique will emerge. It is only by studying those examples that their value will become clear.

### 1.3.1 Modularity

The simplest, most important tool for reducing complexity is the divide-and-conquer technique: analyze or design the system as a collection of interacting subsystems, called *modules*. The power of this technique lies primarily in being able to consider interactions among the components within a module without simultaneously thinking about the components that are inside other modules.

To see the impact of reducing interactions, consider the debugging of a large program with, say, $N$ statements. Assume that the number of bugs in the program is proportional to its size and the bugs are randomly distributed throughout the code. The programmer compiles the program, runs it, notices a bug, finds and fixes the bug, and recompiles before looking for the next bug. Assume also that the time it takes to find a bug in a program is roughly proportional to the size of the program. We can then model the time spent debugging:

$$\begin{aligned} BugCount &\sim N \\ DebugTime &\sim N \times BugCount \\ &\sim N^2 \end{aligned}$$

Unfortunately, the debugging time grows proportional to the square of the program size.

Now suppose that the programmer divides the program into $K$ modules, each of roughly equal size, so that each module contains $N/K$ statements. To the extent that the modules implement independent features, one hopes that discovery of a bug usually will require examining only one module. The time required to debug any one module is thus reduced in two ways: the smaller module can be debugged faster, and

> Plan to throw one away; you will, anyhow.
>
> **— Frederick P. Brooks, *The Mythical Man Month* (1974)**

since there are fewer bugs in smaller programs, any one module will not need to be debugged as many times. These two effects are partially offset by the need to debug all $K$ modules. Thus our model of the time required to debug the system of $K$ modules becomes

$$\begin{aligned} DebugTime &\sim \left(\frac{N}{K}\right)^2 \times K \\ &\sim \frac{N^2}{K} \end{aligned}$$

Modularization into $K$ components thus reduces debugging time by a factor of $K$. Although the detailed mechanism by which modularity reduces effort differs from system to system, this property of modularity is universal. For this reason, one finds modularity in every large system.

The feature of modularity that we are taking advantage of here is that it is easy to replace an inferior module with an improved one, thus allowing incremental improvement of a system without completely rebuilding it. Modularity thus helps control the complexity caused by change. This feature applies not only to debugging but to all aspects of system improvement and evolution. At the same time, it is important to recognize a design principle associated with modularity, which we may call

**The unyielding foundations rule**

*It is easier to change a module than to change the modularity.*

The reason is that once an interface has been used by another module, changing the interface requires replacing at least two modules. If an interface is used by many modules, changing it requires replacing all of those modules simultaneously. For this reason, it is particularly important to get the modularity right.

Whole books have been written about modularity and the good things it brings. Sidebar 1.5 describes one of those books.

### 1.3.2 Abstraction

An important assumption in the numerical example of the effect of modularity on debugging time may not hold up in practice: that discovery of a bug should usually lead to examining just one module. For that assumption to hold true, there is a further requirement: there must be little or no propagation of effects from one module to

> The purpose of computing is insight, not numbers.
>
> — **Richard W. Hamming, *Numerical Methods for Scientists and Engineers* (1962)**

**Sidebar 1.5 How Modularity Reshaped the Computer Industry** Two Harvard Business School professors, Carliss Baldwin and Kim Clark, have written a whole book about modularity.* It discusses many things, but one of the most interesting is its explanation of a major transition in the computer business. In the 1960s, computer systems were a vertically integrated industry. That is, IBM, Burroughs, Honeywell, and several others each provided top-to-bottom systems and support, offering processors, memory, storage, operating systems, applications, sales, and maintenance; IBM even manufactured its own chips. By the 1990s, the industry had transformed into a horizontally organized one in which Intel sells processors, Micron sells memory, Seagate sells disks, Microsoft sells operating systems, Adobe sells text and image applications, Oracle sells database systems, and Gateway and Dell assemble boxes called "computers" out of components provided by the other players.

Carliss Baldwin and Kim Clark explain this transition as an example of modularity in action. The companies that created vertically integrated product lines immediately found complexity running amok, and they concluded that the only effective way to control it was to modularize their products. After a few experiments with wrong modularities (IBM originally designed different computers for business and for scientific applications), they eventually hit on effective ways of splitting things up and thereby keeping their development costs and delivery schedules under control:

- IBM developed the System/360 architecture specification, which could apply to machines of widely ranging performance. This modularity allowed any software to run on any size processor. IBM also developed a standard I/O bus and disk interface, so that any I/O device or disk manufactured by IBM could be attached to any IBM computer.
- Digital Equipment Corporation developed the PDP-11 family, which, with improving technology, could simultaneously be driven down in price toward the PDP-11/03 and up in function toward the PDP-11/70. A hardware-assisted emulation strategy for missing hardware instructions on the smaller machines allowed applications written for any machine to run on any other machine in the family. Digital also developed an I/O architecture, the UNIBUS®, that allowed any I/O device to attach to any PDP-11 model.

The long-range result was that once this modularity was defined and proven to be effective, other vendors were able to jump in and turn each module into a distinct business. The result is the computer industry since the 1990s, which is remarkably horizontal, especially considering its rather different shape only 20 years earlier.

(*Sidebar continues*)

*Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity* [see Suggestions for Further Reading 1.3.7]. Warning: the authors use the word "modularity" to mean all of modularity, abstraction, layering, and hierarchy.

Carliss Baldwin and Kim Clark also observe, more generally, that a market economy is characterized by modularity. Rather than having a self-supporting farm family that does everything for itself, a market economy has coopers, tinkers, blacksmiths, stables, dressmakers, and so on, each being more productive in a modular specialty, all selling things to one another using a universal interface—money.

another. Although there are lots of ways of dividing a system up into modules, some of these ways will prove to be better than others—"according to the natural formation, where the joint is, not breaking any part as a bad carver might" (Plato, *Phaedrus* 265e, Benjamin Jowett translation).

Thus the best divisions usually follow natural or effective boundaries. They are characterized by fewer interactions among modules and by less propagation of effects from one module to another. More generally, they are characterized by the ability of any module to treat all the others entirely on the basis of their external specifications, without need for knowledge about what goes on inside. This additional requirement on modularity is called *abstraction*. Abstraction is separation of interface from internals, of specification from implementation. Because abstraction nearly always accompanies modularity, some authors do not make any distinction between the two ideas. One sometimes sees the term *functional modularity* used to mean modularity with abstraction.

Thus one purchases a DVD player planning to view it as a device with a dozen or so buttons on the front panel and hoping never to look inside. If one had to know the details of the internal design of a television set in order to choose a compatible DVD player, no one would ever buy the player. Similarly, one turns a package over to an overnight delivery service without feeling a need to know anything about the particular kinds of vehicles or routes the service will use. Confidence that the package will be delivered tomorrow is the only concern.

In the computer world, abstraction appears in countless ways. The general ability of sequential circuits to remember state is abstracted into particular, easy-to-describe modules called *registers*. Programs are designed to hide details of their representation of complex data structures and details of which other programs they call. Users expect easy-to-use, button-pushing application interfaces such as computer

It must be remembered that there is nothing more difficult to plan, more doubtful of success, nor more dangerous to manage than the creation of a new system. For the initiator has the enmity of all who would profit by the preservation of the old institutions and merely lukewarm defenders in those who would gain by the new ones.

**— Niccolò Machiavelli, *The Prince* (1513, published 1532; Tr. by Thomas G. Bergin, Appleton-Century-Crofts, 1947)**

games, spreadsheet programs, or Web browsers that abstract incredibly complex underpinnings of memory, processor, communication, and display management.

The goal of minimizing interconnections among modules may be defeated if unintentional or accidental interconnections occur as a result of implementation errors or even well-meaning design attempts to sneak past modular boundaries in order to improve performance or meet some other requirement. Software is particularly subject to this problem because the modular boundaries provided by separately compiled subprograms are somewhat soft and easily penetrated by errors in using pointers, filling buffers, or calculating array indices. For this reason, system designers prefer techniques that enforce modularity by interposing impenetrable walls between modules. These techniques ensure that there can be no unintentional or hidden interconnections. Chapters 4 and 5 develop some of these techniques for enforcing modularity.

Well-designed and properly enforced modular abstractions are especially important in limiting the impact of faults because they control propagation of effects. As we shall see when we study fault tolerance in Chapter 8 [on-line], modules are the units of fault containment, and the definition of a failure is that a module does not meet its abstract interface specifications.

Closely related to abstraction is an important design rule that makes modularity work in practice:

**The robustness principle**

*Be tolerant of inputs and strict on outputs.*

This principle means that a module should be designed to be liberal in its interpretation of its input values, accepting them even if they are not within specified ranges, if it is still apparent how to sensibly interpret them. On the other hand, the module should construct its outputs conservatively in accordance with its specification—if possible making them even more accurate or more constrained than the specification requires. The effect of the robustness principle is to tend to suppress, rather than propagate or even amplify, noise or errors that show up in the interfaces between modules.

The robustness principle is one of the key ideas underlying modern mass production. Historically, machinists made components that were intended to mate by machining one of the components and then machining a second component to exactly fit against or into the first one, a technique known as *fitting*. The breakthrough came with the realization that if one specified *tolerances* for components and designed

We are faced with an insurmountable opportunity.

**— Pogo (Walt Kelley)**

each component to mate with any other component that was within its specified tolerance, then it would be possible to modularize and speed up manufacturing by having interchangeable parts. Apparently, this concept was first successfully applied in an 1822 contract to deliver rifles to the United States Army. By the time production lines for the Model T automobile were created, Henry Ford captured the concept in the aphorism, "In mass production there are no fitters."

The robustness principle plays a major role in computer systems. It is particularly important in human interfaces, network protocols, and fault tolerance, and, as Section 1.4 of this chapter explains, it forms the basis for digital logic. At the same time, a tension exists between the robustness principle and another important design principle:

**The safety margin principle**

*Keep track of the distance to the cliff, or you may fall over the edge.*

When inputs are not close to their specified values, that is usually an indication that something is starting to go wrong. The sooner that something going wrong can be noticed, the sooner it can be fixed. For this reason, it is important to track and report out-of-tolerance inputs, even if the robustness principle would allow them to be interpreted successfully.

Some systems implement the safety margin principle by providing two modes of operation, which might be called "shake-out" and "production". In shake-out mode, modules check every input carefully and refuse to accept anything that is even slightly out of specification, thus allowing immediate discovery of problems and of programming errors near their source. In production mode, modules accept any input that they can reasonably interpret, in accordance with the robustness principle. Carefully designed systems blend the two ideas: accept any reasonable input but report any input that is beginning to drift out of tolerance so that it may be repaired before it becomes completely unusable.

### 1.3.3 Layering

Systems that are designed using good abstractions tend to minimize the number of interconnections among their component modules. One powerful way to reduce module interconnections is to employ a particular method of module organization known as *layering*. In designing with layers, one builds on a set of mechanisms that is already complete (a lower layer) and uses them to create a different complete set of mechanisms (an upper layer). A layer may itself be implemented as several modules,

> There is no such thing as a small change to a large system.
>
> **— systems folklore, source lost in the mists of time**

but as a general rule, a module of a given layer interacts only with its peers in the same layer and with the modules of the next higher and next lower layers. That restriction can significantly reduce the number of potential intermodule interactions in a big system.

Some of the best examples of this approach are found in computer systems: an interpreter for a high-level language is implemented using a lower-level, more machine-oriented, language. Although the higher-level language doesn't allow any new programs to be expressed, it is easier to use, at least for the application for which it was designed.

Thus, nearly every computer system comprises several layers. The lowest layer consists of gates and memory cells, upon which is built a layer consisting of a processor and memory. On top of this layer is built an operating system layer, which acts as an augmentation of the processor and memory layer. Finally, an application program executes on this augmented processor and memory layer. In each layer, the functions provided by the layer below are rearranged, repackaged, reabstracted, and reinterpreted as appropriate for the convenience of the layer above. As will be seen in Chapter 7 [on-line], layers are also the primary organizing technique of data communication networks.

Layered design is not unique to computer systems and communications. A house has an inner structural layer of studs, joists, and rafters to provide shape and strength, a layer of sheathing and drywall to keep the wind out, a layer of siding, flooring and roof tiles to make it watertight, and a cosmetic layer of paint to make it look good. Much of mathematics, particularly algebra, is elegantly organized in layers (in the case of algebra, integers, rationals, complex numbers, polynomials, and polynomials with polynomial coefficients), and that organization provides a key to deep understanding.

### 1.3.4 Hierarchy

The final major technique for coping with complexity also reduces interconnections among modules but in a different, specialized way. Start with a small group of modules, and assemble them into a stable, self-contained subsystem that has a well-defined interface. Next, assemble a small group of subsystems to produce a larger subsystem. This process continues until the final system has been constructed from a small number of relatively large subsystems. The result is a tree-like structure known as a *hierarchy*. Large organizations such as corporations are nearly always set up this way, with a manager responsible for only five to ten employees, a higher-level manager responsible for five to ten managers, on up to the president of the company, who may

> The first 80 percent of a project takes 80 percent of the effort.
> The last 20 percent takes another 80.
>
> **— source unknown**

supervise five to ten vice presidents. The same thinking applies to armies. Even layers can be thought of as a kind of degenerate one-dimensional hierarchy.

There are many other striking examples of hierarchy, ranging from microscopic biological systems to the assembly of Alexander's empire. A classic paper by Herbert Simon, "The architecture of complexity" [Suggestions for Further Reading 1.4.3], contains an amazing range of such examples and offers compelling arguments that, under evolution, hierarchical designs have a better chance of survival. The reason is that hierarchy constrains interactions by permitting them only among the components of a subsystem. Hierarchy constrains a system of $N$ components, which in the worst case might exhibit $N \times (N - 1)$ interactions, so that each component can interact only with members of its own subsystem, except for an interface component that also interacts with other members of the subsystem at the next higher level of hierarchy. (The interface component in a corporation is called a "manager"; in an army it is called the "commanding officer"; for a program it is called the "application programming interface".) If subsystems have a limit of, say, 10 components, this number remains constant no matter how large the system grows. There will be $N/10$ lowest level subsystems, $N/100$ next higher level subsystems, and so on, but the total number of subsystems, and thus the number of interactions, remains proportional to $N$. Analogous to the way that modularity reduces the effort of debugging, hierarchy reduces the number of potential interactions among modules from square-law to linear.

This effect is most strongly noticed by the designer of an individual module. If there are no constraints, each module should in principle be prepared to interact with every other module of the system. The advantage of a hierarchy is that the module designer can focus just on interactions with the interfaces of other members of its immediate subsystem.

### 1.3.5 Putting it Back Together: Names Make Connections

The four techniques for coping with complexity—modularity, abstraction, layering, and hierarchy—provide ways of dividing things up and placing the resulting modules in suitable relation one to another. However, we still need a way of connecting those modules. In digital systems, the primary connection method is that one module *names* another module that it intends to use. Names allow postponing of decisions, easy replacement of one module with a better one, and sharing of modules. Software uses names in an obvious way. Less obviously, hardware modules connected to a bus also use names for interconnection—addresses, including bus addresses, are a kind of name.

> Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.
>
> **— Douglas Hofstadter: *Gödel, Escher, Bach: An Eternal Golden Braid* (1979)**

In a modular system, one can usually find several ways to combine modules to implement a desired feature. The designer must at some point choose a specific implementation from among many that are available. Making this choice is called *binding*. Recalling that the power of modularity comes from the ability to replace an implementation with a better one, the designer usually tries to maintain maximum flexibility by delaying binding until the last possible instant, perhaps even until the first instant that the feature is actually needed.

One way to delay binding is just to name a feature rather than implementing it. Using a name allows one to design a module as if a feature of another module exists, even if that feature has not yet been implemented, and it also makes it mechanically easy to later choose a different implementation. By the time the feature is actually invoked, the name must, of course, be bound to a real implementation of the other module. Using a name to delay or allow changing a binding is called *indirection*, and it is the basis of a design principle:

**Decouple modules with indirection**

*Indirection supports replaceability.*

A folk wisdom version of this principle, attributed to computer scientist David Wheeler of the University of Cambridge, exaggerates the power of indirection by suggesting that "any problem in a computer system can be solved by adding a layer of indirection." A somewhat more plausible counterpart of this folk wisdom is the observation that any computer system can be made faster by removing a layer of indirection.

When a module has a name, several other modules can make use of it by name, thereby sharing the design effort, cost, or information contained in the first module. Because names are a cornerstone element of modularity in digital systems, Chapters 2 and 3 are largely about the design of naming schemes.

## 1.4 COMPUTER SYSTEMS ARE THE SAME BUT DIFFERENT

As we have repeatedly suggested, there is an important lesson to be drawn from the wide range of examples used up to this point to illustrate system problems. Certain common problems show up in all complex systems, whatever their field. Emergent properties, propagation of effects, incommensurate scaling, and trade-offs are considerations in activities as diverse as space station design, management of the economy,

> A system is never finished being developed until it ceases to be used.
>
> **— attributed to Gerald M. Weinberg**

the building of skyscrapers, gene-splicing, petroleum refineries, communication satellite networks, and the governing of India, as well as in the design of computer systems. Furthermore, the techniques that have been devised for coping with complexity are universal. Modularity, abstraction, layering, and hierarchy are used as tools in most fields that deal with complex systems. It is therefore useful for the computer system designer to investigate systems from other fields, both to gain additional perspective on how system problems arise and to discover specific techniques from other fields that may also apply to computer systems. Stated briefly, we conclude that *computer systems are the same as all other systems*.

But there is one problem with that conclusion: it is wrong. There are at least two significant ways in which computer systems differ from every other kind of system with which designers have experience:

- *The complexity of a computer system is not limited by physical laws.*
- *The rate of change of computer system technology is unprecedented.*

These two differences have an enormous impact on complexity and on ways of coping with it.

### 1.4.1 Computer Systems have no Nearby Bounds on Composition

Computer systems are mostly digital, and they are controlled by software. Each of these two properties separately leads to relaxations of what, in other systems, would be limits on complexity arising from physical laws.

Consider first the difference between analog and digital systems. All analog systems have the engineering limitation that each component of the system contributes noise. This noise may come from the environment in the form of, for example, vibration or electromagnetic radiation. Noise may also appear because the component's physical behavior does not precisely follow any tractable model of operation: the pile of rocks that a civil engineer specifies to go under a bridge abutment does not obey a simple deformation model; a resistor in an electronic circuit generates random noise whose level depends on the temperature. When analog components are composed into systems, the noise from individual components accumulates (if the noise sources are statistically independent, the noise may accumulate only slowly but it still accumulates). As the number of components increases, noise will at some point dominate the behavior of the system. (This analysis applies to systems designed by human engineers.

> I was to learn later in life that we tend to meet any new situation by reorganisation; and what a wonderful method it can be for creating the illusion of progress while producing confusion, inefficiency and demoralisation.
>
> — **shortened version of an observation by Charlton Ogburn, "Merrill's Marauders: The truth about an incredible adventure", *Harper's Magazine* (January 1957). Widely but improbably misattributed to Petronius Arbiter (ca. A.D. 60)**

**FIGURE 1.3**

How gain and non-linearity of a digital component restore levels. The input level and output level span the same range of values, but the range of accepted inputs is much wider than the range of generated outputs.

Natural biological, thermodynamic, and macroeconomic systems, composed of billions of analog components, somehow use hierarchy, layering, abstraction, and modularity to operate despite noise, but they are so complex that we do not understand them well enough to adopt the same techniques.)

Noise thus provides a limit on the number of analog components that a designer can usefully compose or on the number of stages that a designer can usefully cascade. This argument applies to any engineered analog system: a bridge across a river, a stereo, or an airliner. It is the reason a photocopy of a photocopy is harder to read than the original. There may also be other limits on size (arising from the strength of materials, for example), but noise is always a limit on the complexity of analog systems.

In contrast, digital systems are noise-free; complexity can therefore grow without any constraint of a bound arising from noise. The designers of digital logic use a version of the ***robustness principle*** known as the *static discipline*. This discipline is the primary source of the magic that seems to surround digital systems. The static discipline requires that the range of analog values that a device accepts as meaning the digital value `ONE` (or `ZERO`) be wider than the range of analog values that the device puts out when it means digital `ONE` (or `ZERO`). This discipline is an example of being tolerant of inputs and strict on outputs.

Digital systems are, at some lower level, constructed of analog components. The analog components chosen for this purpose are non-linear, and they have gain between input and output. When used appropriately, non-linearity allows inputs to have a wide tolerance, and gain ensures that outputs stay within narrow specifications, as shown in Figure 1.3. Together they produce the property of digital circuits called level restoration or regeneration. Regenerated signal levels appear at the output of every digital component, whatever their level of granularity: a gate, a flip-flop, a memory chip, a processor, or a complete computer system. Regenerated levels create clean interfaces that allow one subsystem to be connected to the next with

> The probability of failure of a system tends to be proportional to the confidence that its designer has in its reliability.
>
> **— systems folklore, source lost**

confidence. Unlike the civil engineer's pile of rocks, a logic gate performs exactly as its designer intends.

The static discipline and level restoration do *not* guarantee that devices with digital inputs and outputs never make mistakes. Any component can fail. Or an input signal that is intended to be a ONE may be so far out of tolerance that the receiving component accepts it as a ZERO. When that happens, the output of the component that accepted that value incorrectly is likely to be wrong, too. The important consequence is that digital components make big mistakes, not little ones, and as we shall see when we reach the chapter on fault tolerance, big mistakes are relatively easy to detect and handle.

If a signal does not accumulate noise as it goes through a string of devices, then noise does not limit the number of devices one can string together. In other words, noise does not constrain the maximum depth of composition for digital systems. Unlike analog systems, digital systems can grow in complexity until they exceed the ability of their designers to understand them. As of 2009, processor chips contain over two billion transistors, far more than any analog chip. No airliner has nearly that many components—except in its on-board computers.

The second reason composition has no nearby bounds is that computer systems are controlled by software. Bad as the contribution to complexity from the static discipline may be, the contribution from software turns out to be worse. Hardware is at least subject to *some* physical limits—the speed of light, the rate of settling of signals in real semiconductor materials, unwanted electrical coupling between adjacent components, the rate at which heat can be removed, and the space that it occupies. Software appears to have no physical limits whatever beyond the availability of memory to store it and processors to execute it. As a result, composition of software can go on as fast as people can create it. Thus one routinely hears of operating systems, database systems, and even word processors consisting of more than 10 million program statements.

In principle, abstraction can help control software composition by hiding implementation beneath module interfaces. The problem is that most abstractions are, in reality, slightly "leaky" in that they don't perfectly conceal the underlying implementation. A simple example of leakiness is addition of integers: in most implementations, the addition operation perfectly matches the mathematical specification as long as the result fits in the available word size, but if the result is larger than that, the resulting overflow becomes a complication for the programmer. Leakiness, like noise in analog systems, accumulates as the number of software modules grows. Unlike noise, it accumulates in the form of complexity, so the lack of physical constraints on

> The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair.
>
> **— Douglas Adams, *Mostly Harmless (Hitchhiker's Guide to the Galaxy V)* (1993)**

software composition remains a fundamental problem. It is, therefore, mechanically easy to create a system with complexity that is far beyond the ability of its designers to understand. And since it is easy, it happens often, and sometimes with disastrous results.*

Between the absence of a noise-imposed limit on composition of digital hardware and very distant physical limits on composition of software, it is too easy for an unwary designer to misuse the tools of modularity, abstraction, layering, and hierarchy to include still more complexity. This phenomenon is quite unknown in the design of bridges and airliners. *In contrast with other systems, computer systems allow composition to a depth whose first limit is the designer's ability to understand.* Unfortunately, this lack of nearby natural, physical bounds on depth of composition tempts designers to build more complex systems. If nature does not impose a nearby limit on composition, the designer must self-impose a limit. Since it can be hard to say no to a reasonable-sounding feature, features keep getting added. Therein lies the fate of too many computer system designs.

### 1.4.2 d(technology)/dt is Unprecedented

For reasons partly explained by Sidebar 1.6, during the last 35 years the cost of the digital hardware used for computation and communication has dropped an average of about 30% each year. This rate of change means that just two years' passage of time has been enough to allow technology to cut prices in half, and in seven or eight years it has led to a drop in prices by a factor of 10. Some components have experienced even greater rates of improvement. Figure 1.4 shows the cost of magnetic disk storage over a 25-year span. During that time, disk prices have actually dropped by a factor of 10 roughly every five years, so disk prices have dropped nearly 60% each year. Disk experts project a similar rate of improvement for at least another few years. Their projection seems relatively safe, since no major roadblocks have been reported by development laboratories that are already working on the next rounds of magnetic recording technology. Similar charts apply to random access memory, processor cost, and the speed of optical fiber transmission.

This rapid change of technology has created a substantial difference between computer systems and other engineering systems. Since complex systems can take several years to build, by the time a computer system is ready for delivery, the

*The terminology "leaky" is apparently due to software developer Joel Spolsky.

> Structural engineering is the art of modeling materials we do not wholly understand, into shapes we cannot precisely analyse so as to withstand forces we cannot properly assess, in such a way that the public has no reason to suspect the extent of our ignorance.
>
> **— A. R. Dykes, Scottish Branch, Institution of Structural Engineers (1946)**

**Sidebar 1.6 Why Computer Technology has Improved Exponentially with Time** Popular media frequently use the term "exponential" to describe the explosive rate of improvement of computer technology. Stephen Ward has pointed out that there is a good reason this adjective is appropriate: computer technology appears to be the rare engineering discipline in which the technology being improved is routinely employed to improve the technology. People building airplanes, bridges, skyscrapers, and chemical plants rarely, if ever, have this opportunity.

For example, the performance of a microprocessor is determined at least in part by the cleverness of its layout, which in turn is limited by the time available to use computer-assisted layout tools that can take advantage of lithography advances. If Intel, through improved layout, makes a version of the Pentium that is twice as fast, as soon as that new Pentium is available, it will be used as the processor to make the layout tools for the next Pentium run twice as fast; the next design can benefit from twice as much computation in its layout. This effect is probably one of the drivers of Moore's law, which predicts an exponential increase in component count on chips with a doubling time of 18 months [Suggestions for Further Reading 1.6.1].

If indeed the rate at which we can improve our technology is proportional to the quality of the technology itself, we can express this idea as

$$\frac{d(technology)}{dt} = K \times technology$$

which has an exponential solution,

$$technology = e^{K \cdot t}$$

The actual situation is, of course, far more complicated than that equation suggests, but all equations that even remotely resemble that form, in which technology's rate of growth is some positive function of its current state, have growing exponentials in their solution.

In the real world, exponentials must eventually hit some limit. In hardware there are fairly clear fundamental physical limits to exponential growth, such as the uncertainty principle, the minimum energy required to switch a gate, and the rate at which heat can be removed from a device. The interesting part is that it isn't obvious which one is going to become the roadblock, or when. Thus far, engineering ingenuity in exploiting trade-offs has postponed the day of reckoning. For software, similar limits on exponential growth must exist, but their nature is not at all clear.

More to the immediate point, virtually every improvement in computer and communications technology—whether faster chips, better Internet routing algorithms, more effective prototyping languages, better browser interfaces, faster compilers, bigger disks, or larger RAM—is immediately put to work by everyone who is working on faster chips, better Internet routing algorithms, more effective prototyping languages, better browser interfaces, faster compilers, bigger disks, or larger RAM. Computer system designers live inside a giant feedback system that, at least for the moment, is enjoying exponential solutions.

**FIGURE 1.4**

Magnetic disk price history and projection, 1983–2007.

ground rules under which it was originally designed have shifted. Incommensurate scaling typically means that the designer must adjust for strains when any system parameter changes by a factor of 2, because not all of the components scale up (or down) by the same proportion. More to the point, a whole new design is usually needed when any system parameter changes by a (decimal) order of magnitude. This rule of thumb about strains caused by parameter changes gives us our next design principle:

**The incommensurate scaling rule**

*Changing any system parameter by a factor of 10 usually requires a new design.*

If you design it so that it can be assembled wrong, someone will assemble it wrong.

**— Edward A. Murphy, Jr. (paraphrase of the original Murphy's law, 1949; see sidebar 2.5)**

This rule, when combined with the observed rate of change of technology, means that by the time a newly designed computer system is ready for delivery it may have already needed two rounds of adjustment and be ready for a complete redesign. Even if the designer has tried to predict the impact of technology change, crystal balls are at best cloudy. Worse, during the development of the system, things may run an order of magnitude slower than they will when the system is finished, the code and data don't fit in the available address space, or perhaps the data has to be partitioned across several hard disks instead of nicely fitting on one. One can compensate for each of these problems, but each such compensation absorbs intellectual resources and contributes complexity to the development process.

Even without those adjustments or redesign, the original plan was probably already a new design. A bridge (or airplane) may have a modest number of things that are different from the previous one, but a civil (or aeronautical) engineer almost always ends up designing something that is only a little different from some previous bridge (or airplane). In the case of computer systems, ideas that were completely unrealistic a year or two ago can become mainstream in no time, so the computer system designer almost always ends up designing something that is significantly different from the previous computer system. This difference makes deep analysis of previous designs more rewarding for civil and aeronautical engineers than for computer system designers, and also usually means that in computer systems there hasn't been time to discover and iron out most of the mistakes of the previous design before going on to the next major revision. Those mistakes can contribute strongly to complexity.

Because technology has improved so rapidly, the field of computer system design tends to place much less emphasis on detailed performance analysis and fine-tuning than do most other engineering endeavors. Where an electric power generation system may benefit dramatically from a new steam turbine that improves energy transfer by 1%, a needed 20% improvement in performance of a computer system can usually be obtained just by waiting four months for the next round of hardware product announcements. If a proposal to rewrite an application to obtain that same improvement would require a year of work, it is probably more cost-effective to just wait for technology change to solve the problem. Put another way, rapidly improving technology means that brute-force solutions (buy more memory, wait for a faster processor, use a simpler algorithm) are often the right approach in computer systems, whereas in other systems they may be unthinkable. The owner of the railroad through the canyon probably would not view as economically reasonable a proposal to blast the canyon wider and install a second track. Even if the resources were available, the environmental impact would be a deterrent.

> This "telephone" has too many shortcomings to be seriously considered as a means of communication. The device is inherently of no value to us.
>
> **— frequently attributed to an 1876 Western Union internal memo, but there is no evidence of this memo and it is probably a myth.**

A second major consequence of the rapid rate of change of technology in computer systems is that usability, and related qualities that go under the label "human engineering", of computer systems is always ragged. It takes years of trial and error to make systems usable, friendly, and forgiving, but by the time one level of computer technology has been tamed, a new level of computer technology opens the possibilities of many new features at the same cost, or of providing the previous features more cheaply to a vast new audience of unprepared users.

Similarly, legal and judicial processes take decades to come to grips with new issues, as people debate the wisdom of various policies, discover abuses, and explore alternative remedies. In the face of rapidly changing computer system technology, these processes fall far behind, delaying resolution of such concerns as how to reward innovative software ideas, or what rules should protect information stored in computers, and adding uncertainty of requirements to the burden of the computer system designer.*

Finally, modern high-speed communications with global reach have greatly accelerated the rate at which people discover that a new technology is useful and adopt it. Where it took several decades for electricity and the telephone to move from curiosities to widespread use, recent innovations such as digital cameras and DVDs have swept their markets in less than a decade, and a single mention of a previously obscure World Wide Web site on CNN or in *Newsweek* magazine can cause that site to be suddenly overwhelmed with millions of hits per day. More generally, newly viable applications, such as peer-to-peer file sharing, can change the shape of the workload on existing systems practically overnight.

Thus, the study of computer systems involves telescoping of the usual processes of planning, examining requirements, tailoring details, and integrating with users and society. This telescoping leads to the delivery of systems that have rough edges and without the benefit of the cleverest thought. People who build airplanes and bridges do not have to face these problems. Such problems can be viewed either as a frustrating difficulty or as an exciting challenge, depending on one's perspective.

## 1.5 COPING WITH COMPLEXITY II

Modest physical limits in hardware and very distant physical limits in software together give us the opportunity to create systems of unimaginable—and unmanageable—complexity, and the rapid pace of technology change tempts designers to deliver

*Lawrence Lessig provides a good analysis of the interactions of law, society, and computer technology in *Code: and Other Laws of Cyberspace* [Suggestions for Further Reading 1.1.4].

> Books will soon be obsolete in the public schools. . . . It is possible to teach every branch of human knowledge with the motion picture. Our school system will be completely changed inside of ten years.
>
> **— Thomas A. Edison, as quoted in the *New York Dramatic Mirror* (July 9, 1913)**

systems using new and untested ground rules. These two effects amplify the complexity of computer systems when compared with systems from other engineering areas. Thus, computer system designers need some additional tools to cope with complexity.

### 1.5.1 Why Modularity, Abstraction, Layering, and Hierarchy aren't Enough

Modularity, abstraction, layering, and hierarchy are a major help, but by themselves they aren't enough to keep the resulting complexity under control. The reason is that all four of those techniques assume that the designer understands the system being designed. In the real, fast-changing world of computer systems, it is hard to choose

- the *right* modularity from a sea of plausible alternative modularities.
- the *right* abstraction from a sea of plausible alternative abstractions.
- the *right* layering from a sea of plausible alternative layerings.
- the *right* hierarchy from a sea of plausible alternative hierarchies.

Although some design principles are available, they are far too few, and the only real guidance comes from experience with previous systems.

As might be expected, designers of computer systems have developed and refined at least one additional technique to cope with complexity. Designers of other kinds of systems use this technique as well, but they usually do not consider it to be so fundamental to success as it is for computer systems, probably because the technique is particularly feasible with software. It is a development process called *iteration*.

### 1.5.2 Iteration

The essence of iteration is to start by building a simple, working system that meets only a modest subset of the requirements and then evolve that system in small steps to gradually encompass more and more of the full set of requirements. The idea is that small steps can help reduce the risk that complexity will overwhelm a system design. Having a working system available at all times helps provide assurance that something can be built and provides on-going experience with the current technology ground rules as well as an opportunity to discover and fix bugs. Finally, adjustments for technology changes that arrive during the system development are easier to incorporate as part of one or more of the iterations. When you see a piece of software identified as "release 5.4", that is usually an indication that the vendor is using iteration.

Successful iteration requires considerable foresight. That foresight involves several elements, two of which we identify as design principles:

> I think there is a world market for maybe five computers.
>
> **— Frequently claimed to be said by Thomas J. Watson, Sr., chairman of IBM, in a 1943 talk, but there is little evidence that it is anything but a legend.**

- First of all,

**Design for iteration**

*You won't get it right the first time, so make it easy to change.*

Document the assumptions behind the design so that when the time comes to change the design you can more easily figure out what else has to change. Expect not only to modify and replace modules, but also to remodularize as the system and its requirements become better understood.

- *Take small steps. T*he purpose is to allow discovery of both design mistakes and bad ideas quickly, so that they can be changed or removed with small effort and before other parts of the system in later iterations start to depend on them and they effectively become unchangeable. Systems under active development may be subjected to a complete system rebuild every day because the rebuilding process invokes a large number of checks and tests that can reveal implementation mistakes, while the changes that caused the mistakes are fresh in the minds of the implementers.
- *Don't rush*. Even though individual steps may be small, they must still be well planned. In most projects, the temptation is to rush to implementation. With iterative design, that temptation can be stronger, and the designer must make sure that the design is ready for the next step.
- *Plan for feedback.* Include as part of the design both feedback paths and positive incentives to provide feedback. Testers, installers, maintainers, and users of the system can provide much of the information needed to refine it. Alpha testing ("we're not at all sure this even works") and beta testing ("seems to work, use at your own risk") are common examples, and many vendors encourage users to report details of problems and transcripts of failures by e-mail. A well-designed system will provide many such feedback schemes at all levels.
- *Study failures.* An important goal is to learn from failures rather than assign blame for them. Incentives must be carefully designed to ensure that feedback about failures is not ignored or even suppressed by people fearful of being blamed. Then, having found the apparent cause of a failure,

**Keep digging**

*Complex systems fail for complex reasons.*

Computers in the future may weigh no more than 1.5 tons.

— ***Popular Mechanics*** **(March 1949)**

> Continue looking for other contributing or more basic causes. Working systems often work for reasons that aren't well understood. It is common to find that a new release of a system reveals a bug that has actually been in the system for a long time but has never mattered until now. Much can be learned by figuring out why it never mattered. It can also be useful to explore the mindset of the designers to understand what allowed them to design a system that could fail in this way.* Similarly, don't ignore unexplained behavior. If the feedback reports something that now seems not to be a problem or to have gone away, it is probably a sign that something is wrong rather than that the system magically fixed itself.

Iteration sounds like a straightforward technique, but several obstacles tend to interfere with it. The main obstacle is that as a design evolves through a series of iterations, a risk of losing conceptual integrity arises. That risk suggests that the overall plan for the initial, simplest version of the system must accommodate all of the iterations needed to reach the final version (thus the need for foresight). Someone must constantly be on guard to make sure that the overall design rationale remains clear despite changes made during iteration.

In most organizations, good news (e.g., a major piece of the system is working ahead of schedule) flows rapidly throughout the organization, but bad news (e.g., an important module isn't working yet) often gets confined to the part of the organization that discovers it, at least until it can fix the problem and report good news. This phenomenon, the *bad-news diode*, can prevent realization that changing a different part of the system is more appropriate.

A related problem is that when someone finally realizes that the modularity is wrong, it can be hard to change, for two reasons. First, the *unyielding foundations rule* (see page 20) comes into play. Changing modularity by definition involves changing more than one module, and sometimes several. Second, designers who have invested time and effort in developing a module that, from their point of view, is doing what was intended can be reluctant to see this time and effort lost in a rework. Simply put, to change modularity one must deal with both committed components and committed designers.

---

*The idea of learning from failure and the observation that complex systems fail for complex reasons are the themes of a fascinating book by Henry Petroski, *Design Paradigms: Case Histories of Error and Judgment in Engineering* [Suggestions for Further Reading 1.2.3].

> Based on extensive financial and market analysis, it's projected that no more than five thousand of the new Haloid machines will sell. . . . Model 914, has no future in the office copying market.
>
> **— Consulting firm Arthur D. Little's report to IBM on the prospects for xerographic copying machines (1959)**

A longer-term risk of iteration sometimes shows up when the initial design is both simple and successful. Success can lead designers to be overconfident and to be too ambitious on a later iteration. Technology has improved in the time since deployment of the initial version of the system and feedback has suggested lots of new features. Each suggested feature looks straightforward by itself, and it is difficult to judge how they might interact. The result is often a disastrous overreaching and consequent failure that is so common that it has a name: the *second-system effect*.

Iteration can be thought of as applying modularity to the management of the system design and implementation process. It thus takes us into the realm of management techniques, which are not directly addressed in this book.*

### 1.5.3 Keep it Simple

Remarkably, one of the most effective techniques in coping with complexity is also one that is most difficult to apply: *simplicity*. As Section 1.4.1 explained, computer systems lack natural physical limits to curb their complexity, so the designer must impose limits; otherwise the designer risks being overwhelmed.

The problem with the apparently obvious advice to keep it simple is that

- previous systems give a taste of how great things could be if more features were added.
- the technology has improved so much that cost and performance are not constraints.
- each of the suggested new features has been successfully demonstrated somewhere.
- none of the exceptions or other complications seems by itself to be especially hard to deal with.
- there is fear that a competitor will market a system that has even more features.
- among system designers, arrogance, pride, and overconfidence are more common than clear awareness of the dangers of complexity.

These considerations make it hard to say "no" to any one requirement, feature, exception, or complication. It is their cumulative impact that produces the complexity explosion illustrated in Figure 1.1. The system designer must keep this cumulative

*An excellent book on the subject of system development, by a veteran designer, is Frederick P. Brooks Jr., *The Mythical Man-Month* [Suggestions for Further Reading 1.1.3]. Another highly recommended reading is the Alan Turing Award lecture by Fernando J. Corbató, "On building systems that will fail" [Suggestions for Further Reading 1.5.3].

> There is no reason anyone would want a computer in their home.
>
> **— Kenneth Olsen, president of Digital Equipment Corporation (1977)**

impact in mind at all times. The bottom line is that a computer system designer's most potent weapon against complexity is the ability to say, "No. This will make it too complicated."

As we proceed to study specific computer system engineering topics, we shall make much use of a particular kind of simplicity, to the extent that it is yet another design principle:

**Adopt sweeping simplifications**

*So you can see what you are doing.*

Each topic area will explicitly introduce one or more sweeping simplifications. The reason is that they allow the designer to make compelling arguments for correctness, they make detail irrelevant, and they make clear to all participants exactly what is going on. They will turn out to be one of our best hopes for keeping control of complexity.

## WHAT THE REST OF THIS BOOK IS ABOUT

This chapter has introduced some basic ideas that underlie the study of computer systems. In the course of building on these basic ideas, the ensuing chapters explore a series of system engineering topics in the light of three recurring themes:

- The pervasive importance of modularity
- Principle-based system design
- Making systems robust and resilient

Modularity appears in each engineering topic either as one of the goals of that topic or as one of its design cornerstones. Words from chapter titles suggest this theme. *Abstractions and layering* are particular ways to build on modularity. *Naming* is a fundamental mechanism for interconnecting and replacing modules. *Clients and services* and *virtualization* are two ways of enforcing modularity. *Networks* are built on a foundation of modularity. In *fault tolerance*, the module is the unit that limits the extent of failure. A*tomicity* is an exceptionally robust form of modularity that the designer can exploit to obtain *consistency*. Finally, *protection of information* involves further strengthening of modular walls.

The second theme, principle-based system design, has already emerged, both in explicit mention of several principles and in the list of *design principles* on the inside front cover. These principles capture, in brief phrases, widely applicable nuggets of wisdom that have been developed by generations of computer system designers. Later chapters apply these general principles and also introduce additional design principles that are more specific to particular engineering areas. Even with these principles in mind, it is often difficult to offer a precise recipe for design. Therefore throughout the text the reader will find a second form of captured wisdom in the form of several design *hints* that encode

rationales for making trade-offs.* Together, the principles and hints suggest that computer system design, though for the most part not based on mathematical theories, is also not completely *ad hoc*: it is actually based on sound principles derived from experience and analysis of both successful and failed systems. The reader who understands and absorbs these principles and hints will have learned much of what this book has to say.

The third theme, making systems robust and resilient, has also already emerged, both in the statement of the *robustness principle* and with the idea that modularity, by limiting interconnections, can help control propagation of effects. The terms *robustness* and *resilience* are informal and overlapping descriptions of a general goal of design: that a system should not be sensitive to modest, long-term shifts in its environment (usually called robustness) and that it should continue operating correctly in the face of transient adversity (usually called resilience). Each succeeding chapter introduces at least one progressively stronger way to make a system more robust and resilient. Thus, the chapter on naming shows how indirection of names can make systems less fragile. Then, the chapters on clients and services and on virtualization demonstrate how to enforce modularity to limit the effects of mistakes and accidents. The chapter on networks introduces techniques that provide reliable communications despite communication failures. The chapter on fault tolerance then generalizes those techniques to make entire systems resilient, even though they contain faulty components. The chapters on atomicity and consistency apply fault tolerance techniques to the particular problem of maintaining the integrity of stored data, despite concurrent activity and in the face of software and hardware failures. Finally, the chapter on protecting information introduces techniques to limit the impact of malicious adversaries who would deliberately steal, modify, or deny access to information.

## EXERCISES

**1.1** True or false? Explain: modularity reduces complexity because

**A.** It reduces the effect of incommensurate scaling.

**B.** It helps control propagation of effects.

*1994-1-3d and 1995-1-1e*

**1.2** True or false? Explain: hierarchy reduces complexity because

**A.** It reduces the size of individual modules.

**B.** It cuts down on the number of interconnections between elements.

**C.** It assembles a number of smaller elements into a single larger element.

**D.** It enforces a structure on the interconnections between elements.

**E.** All of the above.

*1994-1-3c and 1999-1-02*

*Many, if not all, of the hints were originally described by Butler Lampson in his paper "Hints for computer system design" [Suggestions for Further Reading 1.5.4].

**1.3** If one created a graph of personal friendships, one would have a hierarchy. True or false?

*1995-1-1b*

**1.4** Which of the following is usually observed in a complex computer system?

**A.** The underlying technology has a high rate of change.

**B.** It is easy to write a succinct description of the behavior of the system.

**C.** It has a large number of interacting features.

**D.** It exhibits emergent properties that make the system perform better than envisioned by the system's designers.

*2005-1-1*

**1.5** Ben Bitdiddle has written a program with 16 major modules of code. Each module contains several procedures. In the first implementation of his program, he finds that each module contains at least one call to every other module. Each module contains 100 lines of code.

**1.5a** How long is Ben's program in lines of code?

**1.5b** How many module interconnections are there in his implementation? (Each call from one module to another is an interconnection.)

Ben decides to change the implementation. Now there are four main modules, each containing four submodules in a one-level hierarchy. The four main modules each have calls to all the other main modules, and within each main module, the four submodules each have calls to one another. There are still 100 lines of code per submodule, but each main module needs 100 lines of management code.

**1.5c** How long is Ben's program now?

**1.5d** How many interconnections are there now? Include module-to-module and submodule-to-submodule interconnections.

**1.5e** Was using hierarchy a good decision? Why or why not?

*1996-1-2a…e*

**Additional exercises relating to Chapter 1 can be found in the problem sets beginning on page 425.**

CHAPTER

# Elements of Computer System Organization

# 2

## CHAPTER CONTENTS


**Overview** ..... **44**
**2.1 The Three Fundamental Abstractions** ..... **45**
2.1.1 Memory ..... 45
2.1.2 Interpreters ..... 53
2.1.3 Communication Links ..... 59
**2.2 Naming in Computer Systems** ..... **60**
2.2.1 The Naming Model ..... 61
2.2.2 Default and Explicit Context References ..... 66
2.2.3 Path Names, Naming Networks, and Recursive Name Resolution ..... 71
2.2.4 Multiple Lookup: Searching Through Layered Contexts ..... 73
2.2.5 Comparing Names ..... 75
2.2.6 Name Discovery ..... 76
**2.3 Organizing Computer Systems with Names and Layers** ..... **78**
2.3.1 A Hardware Layer: The Bus ..... 80
2.3.2 A Software Layer: The File Abstraction ..... 87
**2.4 Looking Back and Ahead** ..... **90**
**2.5 Case Study: UNIX® File System Layering and Naming** ..... **91**
2.5.1 Application Programming Interface for the UNIX File System ..... 91
2.5.2 The Block Layer ..... 93
2.5.3 The File Layer ..... 95
2.5.4 The Inode Number Layer ..... 96
2.5.5 The File Name Layer ..... 96
2.5.6 The Path Name Layer ..... 98
2.5.7 Links ..... 99
2.5.8 Renaming ..... 101
2.5.9 The Absolute Path Name Layer ..... 102
2.5.10 The Symbolic Link Layer ..... 104
2.5.11 Implementing the File System API ..... 106



**Principles of Computer System Design: An Introduction**


Copyright © 2009 by Jerome H. Saltzer and M. Frans Kaashoek. All rights of reproduction in any form reserved.


DOI: 10.1016/B978-0-12-374957-4.00010-4

2.5.12 The Shell and Implied Contexts, Search Paths, and Name Discovery .....110
2.5.13 Suggestions for Further Reading .....112
**Exercises .....112**

## OVERVIEW

Although the number of potential abstractions for computer system components is unlimited, remarkably the vast majority that actually appear in practice fall into one of three well-defined classes: *the memory, the interpreter*, and *the communication link*. These three abstractions are so fundamental that theoreticians compare computer algorithms in terms of the number of data items they must remember, the number of steps their interpreter must execute, and the number of messages they must communicate.

Designers use these three abstractions to organize physical hardware structures, not because they are the only ways to interconnect gates, but rather because

- they supply fundamental functions of recall, processing, and communication,
- so far, these are the only hardware abstractions that have proven both to be widely useful and to have understandably simple interface semantics.

To meet the many requirements of different applications, system designers build layers on this fundamental base, but in doing so they do not routinely create completely different abstractions. Instead, they elaborate the same three abstractions, rearranging and repackaging them to create features that are useful and interfaces that are convenient for each application. Thus, for example, the designer of a general-purpose system such as a personal computer or a network server develops interfaces that exhibit highly refined forms of the same three abstractions. The user, in turn, may see the memory in the form of an organized file or database system, the interpreter in the form of a word processor, a game-playing system, or a high-level programming language, and the communication link in the form of instant messaging or the World Wide Web. On examination, underneath each of these abstractions is a series of layers built on the basic hardware versions of those same abstractions.

A primary method by which the abstract components of a computer system interact is *reference*. What that means is that the usual way for one component to connect to another is by *name*. Names appear in the interfaces of all three of the fundamental abstractions as well as the interfaces of their more elaborate higher-layer counterparts. The memory stores and retrieves objects by name, the interpreter manipulates named objects, and names identify communication links. Names are thus the glue that interconnects the abstractions. Named interconnections can, with proper design, be easy to change. Names also allow the sharing of objects, and they permit finding previously created objects at a later time.

This chapter briefly reviews the architecture and organization of computer systems in the light of abstraction, naming, and layering. Some parts of this review will be familiar to the reader with a background in computer software or hardware, but the systems perspective may provide some new insights into those familiar concepts and

it lays the foundation for coming chapters. Section 2.1 describes the three fundamental abstractions, Section 2.2 presents a model for naming and explains how names are used in computer systems, and Section 2.3 discusses how a designer combines the abstractions, using names and layers, to create a typical computer system, presenting the file system as a concrete example of the use of naming and layering for the memory abstraction. Section 2.4 looks at how the rest of this book will consist of designing some higher-level version of one or more of the three fundamental abstractions, using names for interconnection and built up in layers. Section 2.5 is a case study showing how abstractions, naming, and layering are applied in a real file system.

## 2.1 THE THREE FUNDAMENTAL ABSTRACTIONS

We begin by examining, for each of the three fundamental abstractions, what the abstraction does, how it does it, its interfaces, and the ways it uses names for interconnection.

### 2.1.1 Memory

*Memory*, sometimes called *storage*, is the system component that remembers data values for use in computation. Although memory technology is wide-ranging, as suggested by the list of examples in Figure 2.1, all memory devices fit a simple abstract model that has two operations, named WRITE and READ:

WRITE (*name, value*)
*value* ← READ (*name*)

The WRITE operation specifies in *value* a value to be remembered and in *name* a name by which one can recall that value in the future. The READ operation specifies in *name* the name of some previously remembered value, and the memory device returns that value. A later call to WRITE that specifies the same name updates the value associated with that name.

Memories can be either volatile or non-volatile. A *volatile* memory is one whose mechanism of retaining information consumes energy; if its power supply is interrupted for some reason, it forgets its information content. When one turns off the power to a *non-volatile* memory (sometimes called "stable storage"), it retains its content, and when power is again available, READ operations return the same values as before. By connecting a volatile memory to a battery or an

Hardware memory devices:
- RAM chip
- Flash memory
- Magnetic tape
- Magnetic disk
- CD-R and DVD-R

Higher level memory systems:
- RAID
- File system
- Database management system

**FIGURE 2.1**

Some examples of memory devices that may be familiar.

**Sidebar 2.1 Terminology: Durability, Stability, and Persistence** Both in common English usage and in the professional literature, the terms *durability, stability,* and *persistence* overlap in various ways and are sometimes used almost interchangeably. In this text, we define and use them in a way that emphasizes certain distinctions.

*Durability* A property of a storage medium: the length of time it remembers.
*Stability* A property of an object: it is unchanging.
*Persistence* A property of an active agent: it keeps trying.

Thus, the current chapter suggests that files be placed in a durable storage medium—that is, they should survive system shutdown and remain intact for as long as they are needed. Chapter 8 [on-line] revisits durability specifications and classifies applications according to their durability requirements.

This chapter introduces the concept of stable bindings for names, which, once determined, never again change.

Chapter 7 [on-line] introduces the concept of a persistent sender, a participant in a message exchange who keeps retransmitting a message until it gets confirmation that the message was successfully received, and Chapter 8 [on-line] describes persistent faults, which keep causing a system to fail.

uninterruptible power supply, it can be made *durable*, which means that it is designed to remember things for at least some specified period, known as its *durability*. Even non-volatile memory devices are subject to eventual deterioration, known as *decay*, so they usually also have a specified durability, perhaps measured in years. We will revisit durability in Chapters 8 [on-line] and 10 [on-line], where we will see methods of obtaining different levels of durability. Sidebar 2.1 compares the meaning of durability with two other, related words.

At the physical level, a memory system does not normally name, READ, or WRITE values of arbitrary size. Instead, hardware layer memory devices READ and WRITE contiguous arrays of bits, usually fixed in length, known by various terms such as *bytes* (usually 8 bits, but one sometimes encounters architectures with 6-, 7-, or 9-bit bytes), *words* (a small integer number of bytes, typically 2, 4, or 8), *lines* (several words), and *blocks* (a number of bytes, usually a power of 2, that can measure in the thousands). Whatever the size of the array, the unit of physical layer memory written or read is known as a memory (or storage) *cell*. In most cases, the *name* argument in the READ and WRITE calls is actually the name of a cell. Higher-layer memory systems also READ and WRITE contiguous arrays of bits, but these arrays usually can be of any convenient length, and are called by terms such as *record*, *segment*, or *file*.

#### 2.1.1.1 Read/Write Coherence and Atomicity

Two useful properties for a memory are *read/write coherence* and *before-or-after atomicity*. Read/write coherence means that the result of the READ of a named cell is always the same as the most recent WRITE to that cell. Before-or-after atomicity

means that the result of every READ or WRITE is as if that READ or WRITE occurred either completely before or completely after any other READ or WRITE. Although it might seem that a designer should be able simply to assume these two properties, that assumption is risky and often wrong. There are a surprising number of threats to read/write coherence and before-or-after atomicity:

- *Concurrency.* In systems where different actors can perform READ and WRITE operations concurrently, they may initiate two such operations on the same named cell at about the same time. There needs to be some kind of arbitration that decides which one goes first and to ensure that one operation completes before the other begins.

- *Remote storage.* When the memory device is physically distant, the same concerns arise, but they are amplified by delays, which make the question of "which WRITE was most recent?" problematic and by additional forms of failure introduced by communication links. Section 4.5 introduces remote storage, and Chapter 10 [on-line] explores solutions to before-or-after atomicity and read/write coherence problems that arise with remote storage systems.

- *Performance enhancements.* Optimizing compilers and high-performance processors may rearrange the order of memory operations, possibly changing the very meaning of "the most recent WRITE to that cell" and thereby destroying read/write coherence for concurrent READ and WRITE operations. For example, a compiler might delay the WRITE operation implied by an assignment statement until the register holding the value to be written is needed for some other purpose. If someone else performs a READ of that variable, they may receive an old value. Some programming languages and high-performance processor architectures provide special programming directives to allow a programmer to restore read/write coherence on a case-by-case basis. For example, the Java language has a SYNCHRONIZED declaration that protects a block of code from read/write incoherence, and Hewlett-Packard's Alpha processor architecture (among others) includes a *memory barrier* (MB) instruction that forces all preceding READs and WRITEs to complete before going on to the next instruction. Unfortunately, both of these constructs create opportunities for programmers to make subtle mistakes.

- *Cell size incommensurate with value size.* A large value may occupy multiple memory cells, in which case before-or-after atomicity requires special attention. The problem is that both reading and writing of a multiple-cell value is usually done one cell at a time. A reader running concurrently with a writer that is updating the same multiple-cell value may end up with a mixed bag of cells, only some of which have been updated. Computer architects call this hazard *write tearing.* Failures that occur in the middle of writing multiple-cell values can further complicate the situation. To restore before-or-after atomicity, concurrent readers and writers must somehow be coordinated, and a failure in the middle of an update must leave either all or none of the intended update intact. When these conditions are met, the READ or WRITE is said to be *atomic.* A closely related

risk arises when a small value shares a memory cell with other small values. The risk is that if two writers concurrently update different values that share the same cell, one may overwrite the other's update. Atomicity can also solve this problem. Chapter 5 begins the study of atomicity by exploring methods of coordinating concurrent activities. Chapter 9 [on-line] expands the study of atomicity to also encompass failures.

- *Replicated storage.* As Chapter 8 [on-line] will explore in detail, reliability of storage can be increased by making multiple copies of values and placing those copies in distinct storage cells. Storage may also be replicated for increased performance, so that several readers can operate concurrently. But replication increases the number of ways in which concurrent READ and WRITE operations can interact and possibly lose either read/write coherence or before-or-after atomicity. During the time it takes a writer to update several replicas, readers of an updated replica can get different answers from readers of a replica that the writer hasn't gotten to yet. Chapter 10 [on-line] discusses techniques to ensure read/write coherence and before-or-after atomicity for replicated storage.

Often, the designer of a system must cope with not just one but several of these threats simultaneously. The combination of replication and remoteness is particularly challenging. It can be surprisingly difficult to design memories that are both efficient and also read/write coherent and atomic. To simplify the design or achieve higher performance, designers sometimes build memory systems that have weaker coherence specifications. For example, a multiple processor system might specify: "The result of a READ will be the value of the latest WRITE if that WRITE was performed by the same processor." There is an entire literature of "data consistency models" that explores the detailed properties of different memory coherence specifications. In a layered memory system, it is essential that the designer of a layer know precisely the coherence and atomicity specifications of any lower layer memory that it uses. In turn, if the layer being designed provides memory for higher layers, the designer must specify precisely these two properties that higher layers can expect and depend on. Unless otherwise mentioned, we will assume that physical memory devices provide read/write coherence for individual cells, but that before-or-after atomicity for multicell values (for example, files) is separately provided by the layer that implements them.

#### 2.1.1.2 Memory Latency

An important property of a memory is the time it takes for a READ or a WRITE to complete, which is known as its *latency* (often called *access time*, though that term has a more precise definition that will be explained in Sidebar 6.4). In the magnetic disk memory (described in Sidebar 2.2) the latency of a particular sector depends on the mechanical state of the device at the instant the user requests access. Having read a sector, one may measure the time required to also read a different but nearby sector in microseconds—but only if the user anticipates the second read and requests it before the disk rotates past that second sector. A request just a few microseconds late may encounter

**Sidebar 2.2 How Magnetic Disks Work** Magnetic disks consist of rotating circular platters coated on both sides with a magnetic material such as ferric oxide. An electromagnet called a *disk head* records information by aligning the magnetic field of the particles in a small region on the platter's surface. The same disk head reads the data by sensing the polarity of the aligned particles as the platter spins by. The disk spins continuously at a constant rate, and the disk head actually floats just a few nanometers above the disk surface on an air cushion created by the rotation of the platter.

From a single position above a platter, a disk head can read or write a set of bits, called a *track*, located a constant distance from the center. In the top view below, the shaded region identifies a track. Tracks are formatted into equal-sized blocks, called *sectors*, by writing separation marks periodically around the track. Because all sectors are the same size, the outer tracks have more sectors than the inner ones.

A typical modern disk module, known as a "hard drive" because its platters are made of a rigid material, contains several platters spinning on a common axis called a *spindle*, as in the side view above. One disk head per platter surface is mounted on a comb-like structure that moves the heads in unison across the platters. Movement to a specific track is called *seeking*, and the comb-like structure is known as a *seek arm*. The set of tracks that can be read or written when the seek arm is in one position (for example, the shaded regions of the side view) is called a *cylinder*. Tracks, platters, and sectors are each numbered. A sector is thus addressed by geometric coordinates: track number, platter number, and rotational position. Modern disk controllers typically do the geometric mapping internally and present their clients with an address space consisting of consecutively numbered sectors.

To read or write a particular sector, the disk controller first seeks the desired track. Once the seek arm is in position, the controller waits for the beginning of the desired sector to rotate under the disk head, and then it activates the head on the desired platter. Physically encoding digital data in analog magnetic domains usually requires that the controller write complete sectors.

The time required for disk access is called *latency*, a term defined more precisely in Chapter 6. Moving a seek arm takes time. Vendors quote seek times of 5 to 10 milliseconds, but that is an average over all possible seek arm moves. A move from one

(*Sidebar continues*)

cylinder to the next may require only 1/20 of the time of a move from the innermost to the outermost track. It also takes time for a particular sector to rotate under the disk head. A typical disk rotation rate is 7200 rpm, for which the platter rotates once in 8.3 milliseconds. The time to transfer the data depends on the magnetic recording density, the rotation rate, the cylinder number (outer cylinders may transfer at higher rates), and the number of bits read or written. A platter that holds 40 gigabytes transfers data at rates between 300 and 600 megabits per second; thus a 1-kilobyte sector transfers in a microsecond or two. Seek time and rotation delay are limited by mechanical engineering considerations and tend to improve only slowly, but magnetic recording density depends on materials technology, which has improved both steadily and rapidly for many years.

Early disk systems stored between 20 and 80 megabytes. In the 1970s Kenneth Haughton, an IBM inventor, described a new technique of placing disk platters in a sealed enclosure to avoid contamination. The initial implementation stored 30 megabytes on each of two spindles, in a configuration known as a 30–30 drive. Haughton nicknamed it the "Winchester", after the Winchester 30–30 rifle. The code name stuck, and for many years hard drives were known as Winchester drives. Over the years, Winchester drives have gotten physically smaller while simultaneously evolving to larger capacities.

a delay that is a thousand times longer, waiting for that second sector to again rotate under the read head. Thus the maximum rate at which one can transfer data to or from a disk is dramatically larger than the rate one would achieve when choosing sectors at random. A *random access memory (RAM)* is one for which the latency for memory cells chosen at random is approximately the same as the latency for cells chosen in the pattern best suited for that memory device. An electronic memory chip is usually configured for random access. Memory devices that involve mechanical movement, such as optical disks (CDs and DVDs) and magnetic tapes and disks, are not.

For devices that do not provide random access, it is usually a good idea, having paid the cost in delay of moving the mechanical components into position, to READ or WRITE a large block of data. Large-block READ and WRITE operations are sometimes relabeled GET and PUT, respectively, and this book uses that convention. Traditionally, the unqualified term *memory* meant random-access volatile memory and the term *storage* was used for non-volatile memory that is read and written in large blocks with GET and PUT. In practice, there are enough exceptions to this naming rule that the words "memory" and "storage" have become almost interchangeable.

### 2.1.1.3 Memory Names and Addresses

Physical implementations of memory devices nearly always name a memory cell by the geometric coordinates of its physical storage location. Thus, for example, an electronic memory chip is organized as a two-dimensional array of flip-flops, each holding one named bit. The access mechanism splits the bit name into two parts, which in

turn go to a pair of multiplexers. One multiplexer selects an x-coordinate, the other a y-coordinate, and the two coordinates in turn select the particular flip-flop that holds that bit. Similarly, in a magnetic disk memory, one component of the name electrically selects one of the recording platters, while a distinct component of the name selects the position of the seek arm, thereby choosing a specific track on that platter. A third name component selects a particular sector on that track, which may be identified by counting sectors as they pass under the read head, starting from an index mark that identifies the first sector.

It is easy to design hardware that maps geometric coordinates to and from sets of names consisting of consecutive integers (0, 1, 2, etc.). These consecutive integer names are called *addresses*, and they form the *address space* of the memory device. A memory system that uses names that are sets of consecutive integers is called a *location-addressed memory*. Because the addresses are consecutive, the size of the memory cell that is named does not have to be the same as the size of the cell that is read or written. In some memory architectures each byte has a distinct address, but reads and writes can (and in some cases must always) occur in larger units, such as a word or a line.

For most applications, consecutive integers are not exactly the names that one would choose for recalling data. One would usually prefer to be allowed to choose less constrained names. A memory system that accepts unconstrained names is called an *associative memory*. Since physical memories are generally location-addressed, a designer creates an associative memory by interposing an associativity layer, which may be implemented either with hardware or software, that maps unconstrained higher-level names to the constrained integer names of an underlying location-addressed memory, as in Figure 2.2. Examples of software associative memories, constructed on top of one or more underlying location-addressed memories, include personal telephone directories, file systems, and corporate database systems. A *cache*, a device that remembers the result of an expensive computation in the hope of not redoing that computation if it is needed again soon, is sometimes implemented as an

**FIGURE 2.2**

An associative memory implemented in two layers. The associativity layer maps the unconstrained names of its arguments to the consecutive integer addresses required by the physical layer location-addressed memory.

associative memory, either in software or hardware. (The design of caches is discussed in Section 6.2.)

Layers that provide associativity and name mapping figure strongly in the design of all memory and storage systems. For example, Table 2.2 on page 93 lists the layers of the UNIX file system. For another example of layering of memory abstractions, Chapter 5 explains how memory can be virtualized by adding a name-mapping layer.

### 2.1.1.4 Exploiting the Memory Abstraction: RAID

Returning to the subject of abstraction, a system known as RAID provides an illustration of the power of modularity and of how the storage abstraction can be applied to good effect. RAID is an acronym for Redundant Array of Independent (or Inexpensive) Disks. A RAID system consists of a set of disk drives and a controller configured with an electrical and programming interface that is identical to the interface of a single disk drive, as shown in Figure 2.3. The RAID controller intercepts READ and WRITE requests coming across its interface, and it directs them to one or more of the disks. RAID has two distinct goals:

- Improved performance, by reading or writing disks concurrently
- Improved durability, by writing information on more than one disk

**FIGURE 2.3**

Abstraction in RAID. The READ/WRITE electrical and programming interface of the RAID system, represented by the solid arrow, is identical to that of a single disk.

Different RAID configurations offer different trade-offs between these goals. Whatever trade-off the designer chooses, because the interface abstraction is that of a single disk, the programmer can take advantage of the improvements in performance and durability without reprogramming.

Certain useful RAID configurations are traditionally identified by (somewhat arbitrary) numbers. In later chapters, we will encounter several of these numbered configurations. The configuration known as RAID 0 (in Section 6.1.5) provides increased performance by allowing concurrent reading and writing. The configuration known as RAID 4 (shown in Figure 8.6 [on-line]) improves disk reliability by applying error-correction codes. Yet another configuration known as RAID 1 (in Section 8.5.4.6 [on-line]) provides high durability by

making identical copies of the data on different disks. Exercise 8.8 [on-line] explores a simple but elegant performance optimization known as RAID 5. These and several other RAID configurations were originally described in depth in a paper by Randy Katz, Garth Gibson, and David Patterson, who also assigned the traditional numbers to the different configurations [see Suggestions for Further Reading 10.2.2].

### 2.1.2 Interpreters

*Interpreters* are the active elements of a computer system; they perform the *actions* that constitute computations. Figure 2.4 lists some examples of interpreters that may be familiar. As with memory, interpreters also come in a wide range of physical manifestations. However, they too can be described with a simple abstraction, consisting of just three components:

1. An *instruction reference*, which tells the interpreter where to find its next instruction
2. A *repertoire*, which defines the set of actions the interpreter is prepared to perform when it retrieves an instruction from the location named by the instruction reference
3. An *environment reference*, which tells the interpreter where to find its *environment*, the current state on which the interpreter should perform the action of the current instruction

The normal operation of an interpreter is to proceed sequentially through some program, as suggested by the diagram and pseudocode of Figure 2.5. Using the environment reference to find the current environment, the interpreter retrieves from that environment the program instruction indicated in the instruction reference. Again using the environment reference, the interpreter performs the action directed by the program instruction. That action typically involves using and perhaps changing data in the environment, and also an appropriate update of the instruction reference. When it finishes performing the instruction, the interpreter moves on, taking as its next instruction the one now named by the instruction reference. Certain events, called *interrupts*, may catch the attention of the interpreter, causing it, rather than the program, to supply the next instruction. The original program no longer controls the interpreter; instead, a different program, the interrupt handler, takes control and handles the event. The interpreter may also change the environment reference to one that is appropriate for the interrupt handler.

Hardware:
- Pentium 4, PowerPC 970, UltraSPARC T1
- disk controller
- display controller

Software:
- Alice, AppleScript, Perl, Tcl, Scheme
- LISP, Python, Forth, Java bytecode
- JavaScript, Smalltalk
- TeX, LaTeX
- Safari, Internet Explorer, Firefox

**FIGURE 2.4**

Some common examples of interpreters. The disk controller example is explained in Section 2.3 and the Web browser examples are the subject of Exercise 4.5.

```
procedure INTERPRET()
    do forever
        instruction ← READ (instruction_reference)
        perform instruction in the context of environment_reference
        if interrupt_signal = TRUE then
            instruction_reference ← entry point of INTERRUPT_HANDLER
            environment_reference ← environment ref of INTERRUPT_HANDLER
```

**FIGURE 2.5**

Structure of, and pseudocode for, an abstract interpreter. Solid arrows show control flow, and dashed arrows suggest information flow. Sidebar 2.3 describes this book's conventions for expressing pseudocode.

**Sidebar 2.3 Representation: Pseudocode and Messages** This book presents many examples of program fragments. Most of them are represented in pseudocode, an imaginary programming language that adopts familiar features from different existing programming languages as needed and that occasionally intersperses English text to characterize some step whose exact detail is unimportant. The pseudocode has some standard features, several of which this brief example shows.

```
procedure SUM (a, b)     // Add two numbers.
    total ← a + b
    return total
```

The line numbers on the left are not part of the pseudocode; they are there simply to allow the text to refer to lines in the program. Procedures are explicitly declared

(*Sidebar continues*)

(as in line *1*), and indentation groups blocks of statements together. Program variables are set in *italic*, program key words in **bold**, and literals such as the names of procedures and built-in constants in SMALL CAPS. The left arrow denotes substitution or assignment (line *2*) and the symbol "=" denotes equality in conditional expressions. The double slash precedes comments that are not part of the pseudocode. Various forms of iteration (**while**, **until**, **for each**, **do occasionally**), conditionals (**if**), set operations (**is in**), and case statements (**do case**) appear when they are helpful in expressing an example. The construction **for** *j* **from** 0 **to** 3 iterates four times; array indices start at 0 unless otherwise mentioned. The construction *y.x* means the element named *x* in the structure named *y*. To minimize clutter, the pseudocode omits declarations wherever the meaning is reasonably apparent from the context. Procedure parameters are passed by value unless the declaration **reference** appears. Section 2.2.1 of this chapter discusses the distinction between use by value and use by reference. When more than one variable uses the same structure, the declaration *structure_name* **instance** *variable_name* may be used.

The notation *a*(11. . .15) denotes extraction of bits 11 through 15 from the string *a* (or from the variable *a* considered as a string). Bits are numbered left to right starting with zero, with the most significant bit of integers first (using big-endian notation, as described in Sidebar 4.3). The + operator, when applied to strings, concatenates the strings.

Some examples are represented in the instruction repertoire of an imaginary reduced instruction set computer (RISC). Because such programs are cumbersome, they appear only when it is essential to show how software interacts with hardware.

In describing and using communication links, the notation

$$x \Rightarrow y\text{: } \{M\}$$

represents a message with contents $M$ from sender $x$ to recipient $y$. The notation {*a, b, c*} represents a message that contains the three named fields marshaled in some way that the recipient presumably understands how to unmarshal.

Many systems have more than one interpreter. Multiple interpreters are usually *asynchronous*, which means that they run on separate, uncoordinated, clocks. As a result, they may progress at different rates, even if they are nominally identical and running the same program. In designing algorithms that coordinate the work of multiple interpreters, one usually assumes that there is no fixed relation among their progress rates and therefore that there is no way to predict the relative timing, for example, of the LOAD and STORE instructions that they issue. The assumption of interpreter asynchrony is one of the reasons memory read/write coherence and before-or-after atomicity can be challenging design problems.

#### 2.1.2.1 Processors

A general-purpose processor is an implementation of an interpreter. For purposes of concrete discussion throughout this book, we use a typical reduced instruction set processor. The processor's instruction reference is a *program counter*, stored in a fast memory register inside the processor. The program counter contains the address of the memory location that stores the next instruction of the current program. The environment reference of the processor consists in part of a small amount of built-in location-addressed memory in the form of named (by number) registers for fast access to temporary results of computations.

Our general-purpose processor may be directly wired to a memory, which is also part of its environment. The addresses in the program counter and in instructions are then names in the address space of that memory, so this part of the environment reference is wired in and unchangeable. When we discuss virtualization in Chapter 5, we will extend the processor to refer to memory indirectly via one or more registers. With that change, the environment reference is maintained in those registers, thus allowing addresses issued by the processor to map to different names in the address space of the memory.

The repertoire of our general-purpose processor includes instructions for expressing computations such as adding two numbers (`ADD`), subtracting one number from another (`SUB`), comparing two numbers (`CMP`), and changing the program counter to the address of another instruction (`JMP`). These instructions operate on values stored in the named registers of the processor, which is why they are colloquially called "op-codes".

The repertoire also includes instructions to move data between processor registers and memory. To distinguish program instructions from memory operations, we use the name `LOAD` for the instruction that `READ`s a value from a named memory cell into a register of the processor and `STORE` for the instruction that `WRITE`s the value from a register into a named memory cell. These instructions take two integer arguments, the name of a memory cell and the name of a processor register.

The general-purpose processor provides a *stack*, a push-down data structure that is stored in memory and used to implement procedure calls. When calling a procedure, the caller pushes arguments of the called procedure (the callee) on the stack. When the callee returns, the caller pops the stack back to its previous size. This implementation of procedures supports recursive calls because every invocation of a procedure always finds its arguments at the top of the stack. We dedicate one register for implementing stack operations efficiently. This register, known as the *stack pointer*, holds the memory address of the top of the stack.

As part of interpreting an instruction, the processor increments the program counter so that, when that instruction is complete, the program counter contains the address of the next instruction of the program. If the instruction being interpreted is a `JMP`, that instruction loads a new value into the program counter. In both cases, the flow of instruction interpretation is under control of the running program.

The processor also implements interrupts. An interrupt can occur because the processor has detected some problem with the running program (e.g., the program attempted to execute an instruction that the interpreter does not or cannot

implement, such as dividing by zero). An interrupt can also occur because a signal arrives from outside the processor, indicating that some external device needs attention (e.g., the keyboard signals that a key press is available). In the first case, the interrupt mechanism may transfer control to an *exception* handler elsewhere in the program. In the second case, the interrupt handler may do some work and then return control to the original program. We shall return to the subject of interrupts and the distinction between interrupt handlers and exception handlers in the discussion of threads in Chapter 5.

In addition to general-purpose processors, computer systems typically also have special-purpose processors, which have a limited repertoire. For example, a clock chip is a simple, hard-wired interpreter that just counts: at some specified frequency, it executes an `ADD` instruction, which adds 1 to the contents of a register or memory location that corresponds to the clock. All processors, whether general-purpose or specialized, are examples of interpreters. However, they may differ substantially in the repertoire they provide. One must consult the device manufacturer's manual to learn the repertoire.

### 2.1.2.2 Interpreter Layers

Interpreters are nearly always organized in layers. The lowest layer is usually a hardware engine that has a fairly primitive repertoire of instructions, and successive layers provide an increasingly rich or specialized repertoire. A full-blown application system may involve four or five distinct layers of interpretation. Across any given layer interface, the lower layer presents some repertoire of possible instructions to the upper layer. Figure 2.6 illustrates this model.

**FIGURE 2.6**

The model for a layered interpreter. Each layer interface, shown as a dashed line, represents an abstraction barrier, across which an upper layer procedure requests execution of instructions from the repertoire of the lower layer. The lower layer procedure typically implements an instruction by performing several instructions from the repertoire of a next lower layer interface.

Consider, for example, a calendar management program. The person making requests by moving and clicking a mouse views the calendar program as an interpreter of the mouse gestures. The instruction reference tells the interpreter to obtain its next instruction from the keyboard and mouse. The repertoire of instructions is the set of available requests—to add a new event, to insert some descriptive text, to change the hour, or to print a list of the day's events. The environment is a set of files that remembers the calendar from day to day.

The calendar program implements each action requested by the user by invoking statements in some programming language such as Java. These statements—such as iteration statements, conditional statements, substitution statements, procedure calls—constitute the instruction repertoire of the next lower layer. The instruction reference keeps track of which statement is to be executed next, and the environment is the collection of named variables used by the program. (We are assuming here that the Java language program has not been compiled directly to machine language. If a compiler is used, there would be one less layer.)

The actions of the programming language are in turn implemented by hardware machine language instructions of some general-purpose processor, with its own instruction reference, repertoire, and environment reference.

Figure 2.7 illustrates the three layers just described. In practice, the layered structure may be deeper—the calendar program is likely to be organized with an internal upper layer that interprets the graphical gestures and a lower layer that manipulates the calendar data, the Java interpreter may have an intermediate byte-code interpreter layer, and some machine languages are implemented with a microcode interpreter layer on top of a layer of hardware gates.

**FIGURE 2.7**

An application system that has three layers of interpretation, each with its own repertoire of instructions.

One goal in the design of a layered interpreter is to ensure that the designer of each layer can be confident that the layer below either completes each instruction successfully or does nothing at all. Half-finished instructions should never be a concern, even if there is a catastrophic failure. That goal is another example of atomicity, and achieving it is relatively difficult. For the moment, we simply assume that interpreters are atomic, and we defer the discussion of how to achieve atomicity to Chapter 9 [on-line].

### 2.1.3 Communication Links

A *communication link* provides a way for information to move between physically separated components. Communication links, of which a few examples are listed in Figure 2.8, come in a wide range of technologies, but, like memories and interpreters, they can be described with a simple abstraction. The communication link abstraction has two operations:

SEND (*link_name*, *outgoing_message_buffer*)
RECEIVE (*link_name*, *incoming_message_buffer*)

The SEND operation specifies an array of bits, called a *message,* to be sent over the communication link identified by *link_name* (for example, a wire). The argument *outgoing_message_buffer* identifies the message to be sent, usually by giving the address and size of a buffer in memory that contains the message. The RECEIVE operation accepts an incoming message, again usually by designating the address and size of a buffer in memory to hold the incoming message. Once the lowest layer of a system has received a message, higher layers may acquire the message by calling a RECEIVE interface of the lower layer, or the lower layer may "upcall" to the higher layer, in which case the interface might be better characterized as DELIVER (*incoming_message*).

Names connect systems to communication links in two different ways. First, the *link_name* arguments of SEND and RECEIVE identify one of possibly several available communication links attached to the system. Second, some communication links are actually multiply-attached networks of links, and some additional method is needed to name which of several possible recipients should receive the message. The name of the intended recipient is typically one of the components of the message.

At first glance, it might appear that sending and receiving a message is just an example of copying an array of bits from one memory to another memory over a wire using a sequence of READ and WRITE operations,

Hardware technology:
- twisted pair
- coaxial cable
- optical fiber

Higher level
- Ethernet
- Universal Serial Bus (USB)
- the Internet
- the telephone system
- a UNIX pipe

**FIGURE 2.8**

Some examples of communication links.

so there is no need for a third abstraction. However, communication links involve more than simple copying—they have many complications, such as a wide range of operating parameters that makes the time to complete a SEND or RECEIVE operation unpredictable, a hostile environment that threatens integrity of the data transfer, asynchronous operation that leads to the arrival of messages whose size and time of delivery can not be known in advance, and most significant, the message may not even be delivered. Because of these complications, the semantics of SEND and RECEIVE are typically quite different from those associated with READ and WRITE. Programs that invoke SEND and RECEIVE must take these different semantics explicitly into account. On the other hand, some communication link implementations do provide a layer that does its best to hide a SEND/RECEIVE interface behind a READ/WRITE interface.

Just as with memory and interpreters, designers organize and implement communication links in layers. Rather than continuing a detailed discussion of communication links here, we defer that discussion to Section 7.2 [on-line], which describes a three-layer model that organizes communication links into systems called *networks*. Figure 7.18 [on-line] illustrates this three-layer network model, which comprises a link layer, a network layer, and an end-to-end layer.

## 2.2 NAMING IN COMPUTER SYSTEMS

Computer systems use names in many ways in their construction, configuration, and operation. The previous section mentioned memory addresses, processor registers, and link names, and Figure 2.9 lists several additional examples, some of which are probably familiar, others of which will turn up in later chapters. Some system names resemble those of a programming language, whereas others are quite different. When building systems out of subsystems, it is essential to be able to use a subsystem without having to know details of how that subsystem refers to its components. Names are thus used to achieve modularity, and at the same time, modularity must sometimes hide names.

We approach names from an object point of view: the computer system manipulates *objects*. An interpreter performs the manipulation under control of a program or perhaps under the direction of a human user. An object may be structured, which means that it uses other objects as components. In a direct analogy with two ways in which procedures can pass arguments, there are two ways to arrange for one object to use another as a component:

- create a copy of the component object and include the copy in the using object (use by *value*), or
- choose a name for the component object and include just that name in the using object (use by *reference*). The component object is said to *export* the name.

When passing arguments to procedures, use by value enhances modularity, because if the callee accidentally modifies the argument it does not affect the original. But use by value can be problematic because it does not easily permit two or more objects to *share* a component object whose value changes. If both object A

| | |
|---|---|
| R5 | (processor register) |
| $174FFF_{hex}$ | (memory address) |
| `pedantic.edu` | (network attachment point name) |
| 18.72.0.151 | (network attachment point address) |
| alice | (user name) |
| `alice@pedantic.edu` | (e-mail address) |
| /u/alice/startup_plan.doc | (file name) |
| `http://pedantic.edu/alice/home.html` | (WWW URL) |

**FIGURE 2.9**
Examples of names used in systems.

and B use object C by value, then changing the value of C is a concept that is either meaningless or difficult to implement—it could require tracking down the two copies of C included in A and B to update them. Similarly, in procedure calls it is sometimes useful to give the callee the ability to modify the original object, so most programming languages provide some way to pass the name (pseudocode in this text uses the **reference** declaration for that purpose) rather than the value. One purpose of names, then, is to allow use by reference and thus simplify the sharing of changeable objects.

Sharing illustrates one fundamental purpose for names: as a communication and an organizing tool. Because two uses of the same name can refer to the same object, whether those uses are by different users or by the same user at different times, names are invaluable both for communication and for organization of things so that one can find them later.

A second fundamental purpose for a name is to allow a system designer to defer to a later time an important decision: to which object should this name refer? A name also makes it easy to change that decision later. For example, an application program may refer to a table of data by name. There may be several versions of that table, and the decision about which version to use can wait until the program actually needs the table.

Decoupling one object from another by using a name as an intermediary is known as *indirection*. Deciding on the correspondence between a name and an object is an example of *binding*. Changing a binding is a mechanically easy way to replace one object with another. Modules are objects, so naming is a cornerstone of modularity.

This section introduces a general model for the use of names in computer systems. Some parts of this model should be familiar; the discussion of the three fundamental abstractions in the previous section introduced names and some naming terminology. The model is only one part of the story. Chapter 3 discusses in more depth the many decisions that arise in the design of naming schemes.

### 2.2.1 The Naming Model

It is helpful to have a model of how names are associated with specific objects. A system designer creates a *naming scheme*, which consists of three elements. The first element is a *name space*, which comprises an alphabet of symbols together with syntax rules that specify which names are acceptable. The second element is

a *name-mapping algorithm*, which associates some (not necessarily all) names of the name space with some (again, not necessarily all) values in a *universe of values*, which is the third and final element of the naming scheme. A *value* may be an object, or it may be another name from either the original name space or from a different name space. A name-to-value mapping is an example of a *binding*, and when such a mapping exists, the name is said to be *bound* to the value. Figure 2.10 illustrates.

In most systems, typically several distinct naming schemes are in operation simultaneously. For example, a system may be using one naming scheme for e-mail mailbox names, a second naming scheme for Internet hosts, a third for files, and a fourth for virtual memory addresses. When a program interpreter encounters a name, it must know which naming scheme to invoke. The environment surrounding use of the name usually provides enough information to identify the naming scheme. For example, in an application program, the author of that program knows that the program should expect file names to be interpreted only by the file system and Internet host names to be interpreted only by some network service.

The interpreter that encounters the name runs the name-mapping algorithm of the appropriate naming scheme. The name-mapping algorithm *resolves* the name, which means that it discovers and returns the associated value (for this reason, the name-mapping algorithm is also called a *resolver*). The name-mapping algorithm is usually controlled by an additional parameter, known as a *context*. For a given naming scheme, there can be many different contexts, and a single name of the name space may map to different values when the resolver uses different contexts. For example, in ordinary discourse when a person refers to the names "you", "here", or "Alice", the meaning of each of those names depends on the context in which the person utters it. On the other hand, some naming schemes have only one context. Such naming schemes provide what are called *universal name spaces*, and they have the nice property that a name always has the same meaning within that naming scheme, no matter who uses it. For example, in the United States, social security numbers, which identify government pension and tax accounts, constitute a universal name space.

**FIGURE 2.10**

General model of the operation of a naming scheme. The name-mapping algorithm takes in a name and a context, and it returns an element from the universe of values. The arrows indicate that, using context "A", the algorithm resolves the name "N4" to the value "13".

When there is more than one context, the interpreter may tell the resolver which one it should use or the resolver may use a default context.

We can summarize the naming model by defining the following conceptual operation on names:

*value* ← RESOLVE (*name*, *context*)

When an interpreter encounters a name in an object, it first figures out what naming scheme is involved and thus which version of RESOLVE it should invoke. It then identifies an appropriate context, resolves the name in that context, and replaces the name with the resolved value as it continues interpretation. The variable *context* tells RESOLVE which context to use. That variable contains a name known as a ***context reference***.

In a processor, register numbers are names. In a simple processor, the set of register names, and the registers those names are bound to, are both fixed at design time. In most other systems that use names (including the register naming scheme of some high-performance processors), it is possible to create new bindings and delete old ones, ***enumerate*** the name space to obtain a list of existing bindings, and compare two names. For these purposes we define four more conceptual operations:

*status* ← BIND (*name*, *value*, *context*)
*status* ← UNBIND (*name*, *context*)
*list* ← ENUMERATE (*context*)
*result* ← COMPARE (*name1*, *name2*)

The first operation changes *context* by adding a new binding; the *status* result reports whether or not the change succeeded (it might fail if the proposed *name* violates the syntax rules of the name space). After a successful call to BIND, RESOLVE will return the new *value* for *name*.* The second operation, UNBIND, removes an existing binding from *context*, with *status* again reporting success or failure (perhaps because there was no such existing binding). After a successful call to UNBIND, RESOLVE will no longer return that *value* for *name*. The BIND and UNBIND operations allow the use of names to make connections between objects and change those connections later. A designer of an object can, by using a name to refer to a component object, choose the object to which that name is bound either then or at a later time by invoking BIND, and eliminate a binding that is no longer appropriate by invoking UNBIND, all without modifying the object that uses the name. This ability to delay and change bindings is a powerful tool used in the design of nearly all systems. Some naming implementations provide an ENUMERATE operation, which returns a list of all the names that can be resolved in *context*. Some implementations of ENUMERATE can also return a list of all values currently bound in *context*. Finally, the COMPARE operation reports (TRUE or FALSE) whether or not *name1* is the same as *name2*. The meaning of "same" is an interesting question addressed in Section 2.2.5, and it may require supplying additional context arguments.

*The WRITE operation of the memory abstraction creates a name-value association, so it can be viewed as a specialized instance of BIND. Similarly, the READ operation can be viewed as a specialized instance of RESOLVE.

Different naming schemes have different rules about the uniqueness of name-to-value mappings. Some naming schemes have a rule that a name must map to exactly one value in a given context and a value must have only one name, while in other naming schemes one name may map to several values, or one value may have several names, even in the same context. Another kind of uniqueness rule is that of a *unique identifier name space*, which provides a set of names that will never be reused for the lifetime of the name space and, once bound, will always remain bound to the same value. Such a name is said to have a *stable binding.* If a unique identifier name space also has the rule that a value can have only one name, the unique names become useful for keeping track of objects over a long period of time, for comparing references to see if they are to the same object, and for coordination of multiple copies in systems where objects are replicated for performance or reliability. For example, the customer account number of most billing systems constitutes a unique identifier name space. The account number will always refer to the same customer's account as long as that account exists, despite changes in the customer's address, telephone number, or even personal name. If a customer's account is deleted, that customer's account number will not someday be reused for a different customer's account. Named fields within the account, such as the balance due, may change from time to time, but the binding between the customer account number and the account itself is stable.

The name-mapping algorithm plus a single context do not necessarily map all names of the name space to values. Thus, a possible outcome of performing RESOLVE can be a *not-found* result, which RESOLVE may communicate to the caller either as a reserved value or as an exception. On the other hand, if the naming scheme allows one name to map to several values, a possible outcome can be a list of values. In that case, the UNBIND operation may require an additional argument that specifies which value to unbind. Finally, some naming schemes provide *reverse lookup*, which means that a caller can supply a value as an argument to the name-mapping algorithm, and find out what name or names are bound to that value.

Figure 2.10 illustrates the naming model, showing a name space, the corresponding universe of values, a name-mapping algorithm, and a context that controls the name-mapping algorithm.

In practice, one encounters three frequently used name-mapping algorithms:

- Table lookup
- Recursive lookup
- Multiple lookup

The most common implementation of a context is a table of {*name*, *value*} pairs. When the implementation of a context is a table, the name-mapping algorithm is just a lookup of the name in that table. The table itself may be complex, involving hashing or B-trees, but the basic idea is still the same. Binding a new name to a value consists of adding that {*name*, *value*} pair to the table. Figure 2.11 illustrates this common implementation of the naming model. There is one such table for each context, and different contexts may contain different bindings for the same name.

**FIGURE 2.11**

A system that uses table lookup as the name-mapping algorithm. As in the example of Figure 2.10, this system also resolves the name "N4" to the value "13".

Real-world examples of both the general naming model and the table-lookup implementation abound:

1. A telephone book is a table-lookup context that binds names of people and organizations to telephone numbers. As in the data communication network example, telephone numbers are themselves names that the telephone company resolves into physical line appearances, using a name-mapping algorithm that involves area codes, exchanges, and physical switchgear. The telephone books for Boston and for San Francisco are two contexts of the same naming scheme; any particular name may appear in both telephone books, but if so, it is probably bound to different telephone numbers.

2. Small integers name the registers of a processor. The value is the register itself, and the mapping from name to value is accomplished by wiring.

3. Memory cells are similarly named with the numbers called addresses, and the name-to-value mapping is again accomplished by wiring. Chapter 5 describes an address-renaming mechanism known as virtual memory, which binds blocks of virtual addresses to blocks of contiguous memory cells. When a system implements multiple virtual memories, each virtual memory is a distinct context; a given address can refer to a different memory cell in each virtual memory. Memory cells can also be shared among virtual memories, in which case the same memory cell may have the same (or different) addresses in different virtual memories, as determined by the bindings.

4. A typical computer file system uses several layers of names and contexts: disk sectors, disk partitions, files, and directories are all named objects. Directories are examples of table-lookup contexts. A particular file name may appear in several different directories, bound to either the same or different files. Section 2.5 presents a case study of naming in the UNIX file system.

5. Computers connect to data communication networks at places known as *network attachment points.* Network attachment points are usually named with two distinct naming schemes. The first one, used inside the network, involves a name space consisting of numbers in a fixed-length field. These names are bound, sometimes permanently and sometimes only briefly, to physical entrance

and exit points of the network. A second naming scheme, used by clients of the network, maps a more user-friendly universal name space of character strings to names of the first name space. Section 4.4 is a case study of the Domain Name System, which provides user-friendly attachment point naming for the Internet.

6. A programmer identifies procedure variables by names, and each activation of the procedure provides a distinct context in which most such names are resolved. Some names, identified as "static" or "global names", may instead be resolved in a context that is shared among activations or among different procedures. When a procedure is compiled, some of the original user-friendly names of variables may be replaced with integer identifiers that are more convenient for a machine to manipulate, but the naming model still holds.

7. A Uniform Resource Locator (URL) of the World Wide Web is mapped to a specific Web page by a relatively complicated algorithm that breaks the URL up into several constituent parts and resolves the parts using different naming schemes; the result eventually identifies a particular Web page. Section 3.2 is a case study of this naming scheme.

8. A customer billing system typically maintains at least two kinds of names for each customer account. The account number names the account in a unique identifier name space, but there is also a distinct name space of personal names that can also be used to identify the account. Both of these names are typically mapped to account records by a database system, so that accounts can be retrieved either by account number or by personal name.

These examples also highlight a distinction between "naming" and binding. Some, but not all, contexts "name" things, in the sense that they map a name to an object that is commonly thought of as having that name. Thus, the telephone directory does not "name" either people or telephone lines. Somewhere else there are contexts that bind names to people and that bind telephone numbers to particular physical phones. The telephone directory binds the names of people to the names of telephones.

For each of these examples a context reference must identify the context in which the name-mapping algorithm should resolve the name. Next, we explore where context references come from.

### 2.2.2 Default and Explicit Context References

When a program interpreter encounters a name in an object, someone must supply a context reference so that the name-mapping algorithm can know which context it should use to resolve the name. Many apparently puzzling problems in naming can be simply diagnosed: the name-mapping algorithm, for whatever reason, used the wrong context reference.

There are two ways to come up with a context with which to resolve the names found in an object: default and explicit. A *default context reference* is one that the resolver supplies, whereas an *explicit context reference* is one that comes packaged

Context references for names found in an object

- Default: supplied by the resolver
  - Constant built in to the resolver
  - Variable from the current environment
- Explicit: supplied by the object
  - Per object
  - Per name (qualified name)

**FIGURE 2.12**

Taxonomy of context references.

with the name-using object. Sometimes a naming scheme allows for use of both explicit and default methods: it uses an explicit context reference if the object or name provides one; if not, it uses a default context. Figure 2.12 outlines the taxonomy of context references described in the next two paragraphs.

A default context reference can be a constant that is built in to the resolver as part of its design. Since a constant allows for just one context, the resulting name space is universal. Alternatively, a default context reference can be a variable that the resolver obtains from its current execution environment. That variable may be set by some context assignment rule. For example, in most multiple-user systems, each user's execution environment contains a state variable called the *working directory*. The working directory acts as the default context for resolving file names. Similarly, the system may assign a default context for each distinct activity of a user or even, as will be seen in Chapter 3 (Figures 3.2 and 3.3), for each major subsystem of a system.

In contrast, an explicit context reference commonly comes in one of two forms: a single-context reference intended to be used for all the names that an object uses, or a distinct context reference associated with each name in the object. The second form, in which each name is packaged with its own context reference, is known as a *qualified name*.

A context reference is itself a name (it names the context), which leads some writers to describe it as a *base name*. The name resolver must thus resolve the name represented by the context reference before it can proceed with the original name resolution. This recursion may be repeated several times, but it must terminate somewhere, with the invocation of a name resolver that has a single built-in context. This built-in context contains the bindings that permit the recursion to be unraveled.

That description is quite abstract. To make it concrete, let's revisit the previous real-world examples of names, in each case looking for the context reference the resolver uses:

1. When looking up a number in a telephone book, you must provide the context reference: you need to know whether to pick up the Boston or the San Francisco telephone book. If you call Directory Assistance to ask for a number, the operator will immediately ask you for the context reference by saying, "What city, please?" If you got the name from a personal letter, that letter may mention the city—an example of an explicit context reference. If not, you may have to guess, or undertake a search of the directories of several different cities.

2. In a processor, there is usually only one set of numbered registers; they comprise a default context that is built-in using wires. Some processors have multiple

register sets, in which case there is an additional register, usually hidden from the application programmer, that determines which register set is currently in use. The processor uses the contents of that register, which is a component of the current interpretation environment, as a default context reference. It resolves that number with a built-in context that binds register set numbers to physical register sets by interpreting the register set number as an address that locates the correct bank of registers.

3. In a system that implements multiple virtual memories, the interpretation environment includes a processor register (the page-map address register of Chapter 5) that names the currently active page table; that register contains a reference to the default context. Some virtual memory systems provide a feature known as *segments*. In those systems, a program may issue addresses that contain an explicit context reference known as a *segment number*. Segments are discussed in Section 5.4.5.

4. In a file system with many directories, when a program refers to a file using an unqualified or incompletely qualified file name, the file system uses the working directory as a default context reference. Alternatively, a program may use an *absolute path name*, an example of a fully qualified name that we will discuss in depth in just a moment. The path name contains its own explicit context reference. In both the working directory and the absolute path name, the context reference is itself a name that the resolver must resolve before it can proceed with the original name resolution. This need leads to recursive name resolution, which is discussed in Section 2.2.3.

5. In the Internet, names of network attachment points may be qualified (e.g., `ginger.pedantic.edu`) or unqualified (e.g., `ginger`). When the network name resolver encounters an unqualified name, it qualifies that name with a default context reference, sometimes called the default domain. However it materializes, a qualified name is an absolute path name that still needs to be resolved. A different default—usually a configuration parameter of the name resolver—supplies the context for resolution of that absolute path name in the universal name space of Internet domain names. Section 4.4 describes in detail the rather elaborate mechanism that resolves Internet domain names.

6. The programming language community uses its own terminology to describe default and explicit context references. When implementing *dynamic scope*, the resolver uses the current naming environment as a default context for resolving names. When implementing *static* (also called *lexical*) *scope*, the creator of an object (usually a procedure object) associates the object with an explicit context reference—the naming environment at that instant. The language community calls this combination of an object and its context reference a *closure*.

7. For resolution of a URL for the World Wide Web, the name resolver is distributed, and different contexts are used for different components of the URL. Section 3.2 provides details.

8. Database systems provide the contexts for resolution of both account numbers and personal names in a billing system. If the billing system has a graphical user interface, it may offer a lookup form with blank fields for both account number and personal name. A customer service representative chooses the context reference by typing in one of the two fields and hitting a "find" button, which invokes the resolver. Each of the fields corresponds to a different context.

A context reference can be dynamic, meaning that it changes from time to time. An example is when the user clicks on a menu button labeled "Help". Although the button may always appear in the same place on the screen, the context in which the name "Help" is resolved (and thus the particular help screen that appears in response) is likely to depend on which application program, or even which part of that program, is running at the instant that the user clicks on the button.

A common problem is that the object that uses a name does not provide an explicit context, and the name resolver chooses the wrong default context. For example, a file system typically resolves a file name relative to a current working directory, even though this working directory may be unrelated to the identity of the program or data object making the reference. Compared with the name resolution environment of a programming system, most file systems provide a rather primitive name resolution mechanism.

An electronic mail system provides an example of the problem of making sure that names are interpreted in the intended context. Consider the e-mail message of Figure 2.13, which originated at Pedantic University. In this message, `Alice`, `Bob`, and `Dawn` are names from the local name space of e-mailboxes at Pedantic University, and `Charles@cse.Scholarly.edu` is a qualified name of an e-mailbox managed by a mail service named `cse.Scholarly.edu` at the Institute of Scholarly Studies. The name `Charles` is of a particular mailbox at that mail service, and the @-sign is conventionally used to separate the name of the mailbox from the name of the mail service.

As it stands, if user `Charles` tries to reply to the sender of this message, the response will be addressed to `Bob`. Since the first name resolver to encounter the reply message is probably inside the system named `cse.Scholarly.edu`, that resolver would in the normal course of events use as a default context reference the name of the local mail service. That is, it would try to send the message to `Bob@cse.Scholarly.edu`. That isn't the mailbox address of the user who sent the original message. Worse, it might be someone else's mailbox address.

```
To: Bob
Cc: Charles@cse.Scholarly.edu
From: Alice
------
Based on Dawn's suggestions, this chapter has
experienced a major overhaul this year. If you
like it, send your compliments to Dawn (her e-mail
address is "Dawn"); if you do not like it, send your
complaints to me.
```

**FIGURE 2.13**

An e-mail message that uses default contexts.

```
To: Bob@Pedantic.edu
cc: Charles@cse.Scholarly.edu
From: Alice@Pedantic.edu
------
Based on Dawn's suggestions, this chapter has
experienced a major overhaul this year. If you
like it, send your compliments to Dawn (her e-mail
address is "Dawn"); if you do not like it, send
your complaints to me.
```

**FIGURE 2.14**

The e-mail message of Figure 2.13 after the mail system expands every unqualified address in the headers to include an explicit context reference.

When constructing the e-mail message, Alice intended local names such as `Bob` to be resolved in her own context. Most mail sending systems know that a local name is not useful to anyone outside the local context, so it is conventional for the mail system to tinker with unqualified names found in the address fields by automatically rewriting them as qualified names, thus adding an explicit context reference to the names `Bob` and `Alice`, as shown in Figure 2.14.

Unfortunately, the mail system can perform this address rewriting only for the headers because that is the only part of the message format it fully understands. If an e-mail address is embedded in the text of the message (as in the example, the mailbox name `Dawn`), the mail system has no way to distinguish it from the other text. If the recipient of the message wishes to make use of an e-mail address found in the text of the message, that recipient is going to have to figure out what context reference is needed. Sometimes it is easy to figure out what to do, but if a message has been forwarded a few times, or the recipient is unaware of the problem, a mistake is likely.

A partial solution could be to tag the e-mail message with an explicit context reference, using an extra header, as in Figure 2.15. With this addition, a recipient of this message could select either `Alice` in the header or `Dawn` in the text and ask the mail system to send a reply. The mail system could, by examining the `Context:` header, determine how to resolve any unqualified e-mail address associated with this message, whether found in the original headers or extracted from the text of the message. This scheme is quite *ad hoc*; if user `Bob` forwards the message of Figure 2.15 with an added note to someone in yet another naming context, any unqualified addresses in the added note would need a different explicit context reference. Although this scheme is not actually used in any e-mail system that the authors are aware of, it has been used in other naming systems. An example is the base element of HTML, the display language of the World Wide Web, described briefly in Section 3.2.2.

A closely related problem is that different contexts may bind different names for the same object. For example, to call a certain telephone, it may be that a person in the same organization dials 2–7104, a second person across the city dials 312–7104, a third who is a little farther away dials (517) 312–7104, and a person in another country may have to dial 001 (517) 312–7104. When the same object has different names in different contexts, passing a name from one user to another is awkward because, as with the e-mail message example, someone must translate the name before the other user can use it. As with the e-mail address, if someone hands you a scrap of paper on

```
To: Bob
cc: Charlies@cse.Scholarly.edu
From: Alice
Context: Pedantic.edu
------
Based on Dawn's suggestions, this chapter has
experienced a major overhaul this year. If you
like it, send your compliments to Dawn (her e-mail
address is "Dawn"); if you do not like it, send your
complaints to me.
```

**FIGURE 2.15**

An e-mail message that provides an explicit context reference as one of its headers.

which is written the telephone number 312–7104, simply dialing that number may or may not ring the intended telephone. Even though the several names are related, some effort may be required to figure out just what translation is required.

### 2.2.3 Path Names, Naming Networks, and Recursive Name Resolution

The second of the three common name-mapping algorithms listed on page 64 is *recursive name resolution*. A path name can be thought of as a name that explicitly includes a reference to the context in which it should be resolved. In some naming schemes, path names are written with the context reference first, in others with the context reference last. Some examples of path names are:

```
ginger.pedantic.edu.
/usr/bin/emacs
Macintosh HD:projects:CSE 496:problem set 1
```
Chapter 2, section 2, part 3, first paragraph
Paragraph 1 of part 3 of section 2 of chapter 2

As these examples suggest, a path name involves multiple components and some syntax that permits a name resolver to parse the components. The last two examples illustrate that different naming schemes place the component names in opposite orders, and indeed the other examples also demonstrate both orders. The order of the components must be known to the user of the name and to the name resolver, but either way interpretation of the path name is most easily explained recursively by borrowing terminology from the representation of numbers: all but the least significant component of a path name is an explicit context reference that identifies the context to be used to resolve that least significant component. In the above examples, the least significant components and their explicit context references are, respectively,

| Least significant component | Explicit context reference |
|---|---|
| `ginger` | `pedantic.edu.` |
| `emacs` | `/usr/bin` |
| `problem set 1` | `Macintosh hd:projects:CSE 491` |
| first paragraph | Chapter 2, section 2, part 3 |
| Paragraph 1 | part 3 of section 2 of chapter 2 |

The recursive aspect of this description is that the explicit context reference is itself a path name that must be resolved. So we repeat the analysis as many times as needed until what was originally the most significant component of the path name is also the least significant component, at which point the resolver can do an ordinary table lookup using some context. In the choice of this context, the previous discussion of default and explicit context references again applies. In a typical design, the resolver uses one of two default context references:

- A special context reference, known as the *root,* that is built in to the resolver. The root is an example of a universal name space. A path name that the resolver can resolve with recursion that ends at the root context is known as an *absolute path name*.
- The path name of yet another default context. To avoid circularity, this path name must be an absolute path name. A path name that is resolved by looking up its most significant component in yet another context is known as a *relative path name*. (In a file system, the path name of this default context is what example 4 on page 68 identified as the working directory.) Thus in the UNIX file system, for example, if the working directory is `/usr/Alice`, the relative path name `plans/Monday` would resolve to the same file as the absolute path name `/usr/Alice/plans/Monday`.

If a single name resolver is prepared to resolve both relative and absolute path names, some scheme such as a syntactic flag (e.g., the initial `"/"` in `/usr/bin/emacs` and the terminal `"."` in `ginger.pedantic.edu.`) may distinguish one from the other, or perhaps the name resolver will try both ways in some order, using the first one that seems to work. Trying two schemes in order is a simple form of multiple name lookup, about which we will have more to say in the next subsection.

Path names can also be thought of as identifying objects that are organized in what is called a *naming network*. In a naming network, contexts are treated as objects, and any context may contain a name-to-object binding for any other object, including another context. The name resolver somehow chooses one context to use as the root (perhaps by having a lower-level name for that context wired into the resolver), and it then resolves all absolute path names by tracing a path from the chosen root to the first named context in the path name, then the next, continuing until it reaches the object that was named by the original path name. It similarly resolves relative path names starting with a default context found in a variable in its environment. That variable contains the absolute path name of the default context. Since there can be many paths from one place to another, there can be many different path names for the same object or context. Multiple names for the same object are known as *synonyms* or *aliases*. (This text avoids the word "alias" because different systems use it in quite different ways.) On the other hand, since the root provides a universal name space, every object that uses the same absolute path name is referring to the same exporting object.

Sharing names of a naming network can be a problem because each user may express path names relative to a different starting point. As a result, it may not be

obvious how to translate a path name when passing it from one user to another. One standard solution to this problem is to require that users share only absolute path names, all of which begin with the root.

The file system of a computer operating system is usually organized as a naming network, with directories acting as contexts. It is common in file systems to encounter implementation-driven restrictions on the shape of the naming network, for example, requiring that the contexts be organized in a *naming hierarchy* with the root acting as the base of the tree. A true naming hierarchy is so constraining that it is rarely found in practice; real systems, even if superficially hierarchical, usually provide some way of adding cross-hierarchy *links.* The simplest kind of link is just a synonym: a single object may be bound in more than one context. Some systems allow a more sophisticated kind of link, known as an *indirect name*. An indirect name is one that a context binds to another name in the same name space rather than to an object. Because many designers have independently realized that indirect names are useful, they have come to be called by many different labels, including *symbolic link*, *soft link, alias,* and *shortcut*. The UNIX file system described in Section 2.5 includes a naming hierarchy, links, and indirect names called soft links.

A path name has internal structure, so a naming scheme that supports path names usually has rules regarding construction of allowable path names. Path names may have a maximum length, and certain symbols may be restricted for use only as structural separators.

### 2.2.4 Multiple Lookup: Searching through Layered Contexts

Returning to the topic of default contexts (in the taxonomy of Figure 2.12), context assignment rules are a blunt tool. For example, a directory containing library programs may need to be shared among different users; no single assignment rule can suffice. This inflexibility leads to the third, more elaborate name resolution scheme, *multiple lookup.** The idea of multiple lookup is to abandon the notion of a single, default context and instead resolve the name by systematically trying several different contexts. Since a name may be bound in more than one context, multiple lookup can produce multiple resolutions, so some scheme is needed to decide which resolution to use.

A common such scheme is called the *search path*, which is nothing more than a specific list of contexts to be tried, in order. The name resolver tries to resolve the name using the first context in the list. If it gets a not-found result, it tries the next context, and so on. If the name is bound in more than one of the listed contexts, the one earliest in the list wins and the resolver returns the value associated with that binding.

A search path is often used in programming systems that have libraries. Suppose, for example, a library procedure that calculates the square root math function exports

*The operating system community traditionally uses the word "search" for multiple lookup, but the advent of "search engines" on both the Internet and the desktop has rendered that usage ambiguous. The last paragraph of Section 2.2.4, on page 75, discusses this topic.

a procedure interface named SQRT. After compiling this function, the writer places a copy of the binary program in a math library. A prospective user of the square root function writes the statement

$x \leftarrow \text{SQRT}(y)$

in a program, and the compiler generates code that uses the procedure named SQRT. The next step is that the compiler (or in some systems a later loader) undertakes a series of lookups in various public and private libraries that it knows about. Each library is a context, and the search path is a list of the library contexts. Each step of the multiple lookup involves an invocation of a simpler, single-context name resolver. Some of these attempted resolutions will probably return a not-found result. The first resolution attempt that finds a program named SQRT will return that program as the result of the lookup.

A search path is usually implemented as a per-user list, some or all of whose elements the user can set. By placing a library that contains personally supplied programs early in the search path, an individual user can effectively replace a library program with another that has the same name, thereby providing a *user-dependent binding*. This replace-by-name feature can be useful, but it can also be hazardous because one may unintentionally choose a name for a program that is also exported by some completely unrelated library program. When some other application tries to call that unrelated program, the ensuing multiple lookup may find the wrong one. As the number of libraries and names in the search path increases, the chance increases that two libraries will accidentally contain two unrelated programs that happen to export the same name.

Despite the hazards, search paths are a widely used mechanism. In addition to loaders using search paths to locate library procedures, user interfaces use search paths to locate commands whose names the user typed, compilers use search paths to locate interfaces, documentation systems use search paths to find cited documents, and word processing systems use search paths to locate text fragments to be included in the current document.

Some naming schemes use a more restricted multiple lookup method. For example, rather than allowing an arbitrary list of contexts, a naming scheme may require that contexts be arranged in nested layers. Whenever a resolution returns not-found in some layer, the resolver retries in the enclosing layer. Layered contexts were at one time popular in programming languages, where programs define and call on subprograms, because it can be convenient (to the point of being undisciplined, which is why it is no longer so popular) to allow a subprogram access by name to the variables of the defining or calling program. For another example, the scheme for numbering Internet network attachment points has an outer public layer and an inner private layer. Certain Internet address ranges (e.g., all addresses with a first byte of 10) are reserved for use in private networks; those address ranges constitute an inner private layer. These network addresses may be bound to different network attachment points in different private contexts without risk of conflict. Internet addresses that are outside the ranges reserved for private contexts should not be bound in any private context; they are instead resolved in the public context.

In a set of layered contexts, the *scope* of a name is the range of layers in which the name is bound to the same object. A name that is bound only in the outermost layer, and is always bound to the same object, independent of the current context layer, is known as a *global name*. The outermost layer that resolves global names is an example of a universal name space.

Incidentally, we have now used the term *path* as both an adjective qualifier and a noun, but with quite different meanings. A *path name* is a name that carries its own explicit context, while a *search path* is a context that consists of a list of contexts. Thus each element of a search path may be a path name.

The word "search" also has another, related but somewhat different, meaning. Internet search engines such as Google and AltaVista take as input a query consisting of one or more key words, and they return a list of World Wide Web pages that contain those key words. Multiple results (known as "hits") are the common case, and Google, for example, implements a sophisticated system for ranking the hits. Google also offers the user the choice of receiving just the highest-ranked hit ("I'm feeling lucky") or receiving a rank-ordered list of hits. Most modern desktop computer systems also provide some form of key word search for local files. When one encounters the unqualified word "search", it is a good idea to pause and figure out whether it refers to multiple lookup or to key word query.

### 2.2.5 Comparing Names

As mentioned earlier, one more operation is sometimes applied to names:

*result* ← COMPARE(*name1*, *name2*)

where *result* is a binary value, TRUE or FALSE. The meaning of name comparison requires some thought because the invoker might have one of three different questions in mind:

1. Are the two names the same?
2. Are the two names bound to the same value?
3. If the value or values are actually the identifiers of storage containers, such as memory cells or disk sectors, are the contents of the storage containers the same?

The first question is mechanically easiest to answer because it simply involves comparing the representations of the two names ("Is Jim Smith the same as Jim Smith?"), and it is exactly what a name resolver does when it looks things up in a table-lookup context: look through the context for a name that is the same as the one being resolved. On the other hand, in many situations the answer is not useful, since the same name may be bound to different values in different contexts and two different names may be synonyms that are bound to the same value. All one learns from the first question is whether or not the name strings have the same bit pattern.

For that reason, the answer to the second question is often more interesting. ("Is the Jim Smith who just received the Nobel prize the same Jim Smith I knew in high school?") Getting that answer requires supplying the contexts for the two names as

additional arguments to COMPARE, so that it can resolve the names and compare the results. Thus, for example, resolving the variable name *A* and the variable name *B* may reveal that they are both bound to the same storage cell address. Even this answer may still not reveal as much as expected because the two names may resolve to two names of a different, lower-layer naming scheme, in which case the same questions need to be asked recursively about the lower-layer names. For example, variable names *A* and *B* may be bound to different storage cell addresses, but if a virtual memory is in use those different virtual storage cell addresses might map to the same physical cell address. (This example will make more sense when we reach Chapter 5.)

Even after reaching the bottom of that recursion, the result may be the names of two different physical storage containers that contain identical copies of data, or it may be two different lower-layer names (that is, synonyms) for the same storage container. ("This biography file on Jim Smith is identical to that biography file on Jim Smith. Are there one or two biography files?" "This biography about Edwin Aldrin is identical to that biography about Buzz Aldrin. Are those two names for the same person?") Thus the third question arises, along with a need to understand what it means to be the "same". Unless one has some specific understanding of the underlying physical representation, the only way to distinguish the two cases may be to change the contents of one of the named storage containers and see if that causes the contents of the other one to change. ("Kick this one and see if that one squeals.")

In practice, systems (and some programming languages) typically provide several COMPARE operators that have different semantics designed to help answer these different questions, and the programmer or user must understand which COMPARE operation is appropriate for the task at hand. For example, the LISP language provides three comparison operators, named EQ (which compares the bindings of its named arguments), EQU (which compares the values of its named arguments), and EQUALS (which recursively compares entire data structures.)

### 2.2.6 Name Discovery

Underlying all name reference is a recursive protocol that answers the question, "How did you know to use this name?" This *name discovery protocol* informs an object's prospective user of the name that the object exports. Name discovery involves two basic elements: the exporter *advertises* the existence of the name, while the prospective user *searches* for an appropriate advertisement. The thing that makes name discovery recursive is that the name user must first know the name of a place to search for the advertisement. This recursion must terminate somewhere, perhaps in a direct, outside-the-computer communication between some name user and some name exporter.

The simplest case is a programmer who writes a program consisting of two procedures, one of which refers to the other by name. Since the same programmer wrote both, name discovery is explicit and no recursion is necessary. Next, suppose the two programs are written by two different programmers. The programmer who wants to use a procedure by name must somehow discover the exported name. One possibility is that the second programmer performs the advertisement by shouting the

procedure's name down the hall. Another possibility is that the using programmer looks in a shared directory in which everyone agrees to place shared procedures. How does that programmer know the name of that shared directory? Perhaps someone shouted that name down the hall. Or perhaps it is a standard library directory whose name is listed in the programmers' reference manual, in which case that manual terminates the recursive protocol. Although program library names don't usually appear in magazine advertisements or on billboards, it has become commonplace to discover the name of a World Wide Web site in such places. Name discovery can take several forms:

- *Well-known name:* A name (such as "Google" or "Yahoo!") that has been advertised so widely that one can depend on it being stable for at least as long as the thing it names. Running across a well-known name is a method of name discovery.
- *Broadcast:* A way of advertising a name, for example by wearing a badge that says "Hello, my name is . . . ", posting the name on a bulletin board, or sending it to a mailing list. Broadcast is used by automatic configuration protocols sometimes called "plug-and-play" or "zero configuration". It may even be used on a point-to-point communication link in the hope that there is someone listening at the other end who will reply. Listening for broadcasts is a method of name discovery.
- *Query* (also called *search*): Present one or more key words to, for example, a search engine such as Google. Query is a widely used method of name discovery.
- *Broadcast query:* A generalized form of key word query. Ask everyone within hearing distance "does anyone know a name for . . . ?" (sometimes confusingly called "reverse broadcast").
- *Resolving a name of one name space to a name of a different name space:* Looking up a name in the telephone book leads to discovery of a telephone number. The Internet Domain Name System, described in Section 4.4, performs a similar service, looking up a domain name and returning a network attachment point address.
- *Introduction:* What happens at parties and in on-line dating services. Some entity that you already know knows a name and gives that name to you. In a computer system, a friend may send you an e-mail message that mentions the name of an interesting Web site or the e-mail address of another friend. For another example, each World Wide Web page typically contains introductions (technically known as hypertext links) to other Web pages.
- *Physical rendezvous:* A meeting held outside the computer. It requires somehow making prior arrangements concerning time and place, which implies prior communication, which implies prior knowledge of some names. Once set up, physical rendezvous can be used for discovering other names as well as for verifying authenticity. Many organizations require that setting up a new account on a company computer system must involve a physical rendezvous with the system administrator to exchange names and choose a password.

Any of the above methods of name discovery may require first discovering some other name, such as the name of the reference source for well-known names, the name of the bulletin board on which broadcasts are placed, the name of the name resolver, the name of the party host, and so on. The method of discovering this other name may be the same as the method first invoked, or it may be different. The important thing the designer must keep in mind is that the recursion must terminate somewhere—it can't be circular.

Some method of name discovery is required wherever a name is needed. An interesting exercise is to analyze some of the examples of names mentioned in earlier parts of this chapter, tracing the name discovery recursion to see how it terminates, because in many cases that termination is so distant from the event of name usage and resolution that it has long since been forgotten. Many additional examples of name discovery will show up in later chapters: names used for clients and services, where a client needs to discover the name of an appropriate service; data communication networks, where routing provides a particularly explicit example of name discovery; and security, where it is critical to establish the integrity of the terminating step.

## 2.3 ORGANIZING COMPUTER SYSTEMS WITH NAMES AND LAYERS

Section 2.1 demonstrated how computer system designers use layers to implement more elaborate versions of the three fundamental abstractions, and Section 2.2 explained how names are used to connect system components. Designers also use layers and names in many other ways in computer systems. Figure 2.16 shows the typical organization of a computer system as three distinct layers. The bottom layer consists of hardware components, such as processors, memories, and communication links. The middle layer consists of a collection of software modules, called the *operating system* (see Sidebar 2.4), that abstract these hardware resources into a convenient *application programming interface (API)*. The top layer consists of software that implements application-specific functions, such as a word processor, payroll program, computer game, or Web

**FIGURE 2.16**

A typical computer system organized in three layers. The operating system layer allows bypass, so the application layer can directly invoke many features of the hardware layer. However, the operating system layer hides certain dangerous features of the hardware layer.

**Sidebar 2.4 What is an Operating System?** An *operating system* is a set of programs and libraries that make it easy for computer users and programmers to do their job. In the early days of computers, operating systems were simple programs that assisted operators of computers (at that time the only users who interacted with a computer directly), which is why they are called operating systems.

Today operating systems come in many flavors and differ in the functions they provide. The operating system for the simplest computers, such as that for a microwave oven, may comprise just a library that hides hardware details in order to make it easier for application programmers to develop applications. Personal computers, on the other hand, ship with operating systems that contain tens of millions of lines of code. These operating systems allow several people to use the same computer; permit users to control which information is shared and with whom; can run many programs at the same time while keeping them from interfering with one another; provide sophisticated user interfaces, Internet access, file systems, backup and archive applications, device drivers for the many possible hardware gadgets on a personal computer, and a wide range of abstractions to simplify the job of application programmers, and so on.

Operating systems also offer an interesting case study of system design. They are evolving rapidly because of new requirements. Their designers face a continuous struggle to control their complexity. Some modern operating systems have interfaces consisting of thousands of procedures, and their implementations are so complex that it is a challenge to make them work reliably.

This book has much more to say about operating systems, starting in Section 5.1.1, where it begins development of a minimal model operating system.

browser. If we examine each layer in detail, we are likely to find that it is itself organized in layers. For example, the hardware layer may comprise a lower layer of gates, flip-flops, and wires, and an upper layer of registers, memory cells, and finite-state machines.

The exact division of labor between the hardware layer and the software layers is an engineering trade-off and a topic of considerable debate between hardware and software designers. In principle, every software module can be implemented in hardware. Similarly, most hardware modules can also be implemented in software, except for a few foundational components such as transistors and wires. It is surprisingly difficult to state a generic principle for how to decide between an implementation in hardware or software. Cost, performance, flexibility, convenience, and usage patterns are among the factors that are part of the trade-off, but for each individual function they may be weighted differently. Rather than trying to invent a principle, we discuss the trade-off between hardware and software in the context of specific functions as they come up.

The operating system layer usually exhibits an interesting phenomenon that we might call *layer bypass*. Rather than completely hiding the lower, hardware layer, an operating system usually hides only a few features of the hardware layer, such as

particularly dangerous instructions. The remaining features of the hardware layer (in particular, most of the instruction repertoire of the underlying processor) pass through the operating system layer for use directly by the application layer, as in Figure 2.16. Thus, the dangerous instructions can be used only by the operating system layer, while all of the remaining instructions can be used by both the operating system and application layers. Conceptually, a designer could set things up so that the operating system layer intercepts every invocation of the hardware layer by the application layer and then explicitly invokes the hardware layer. That design would slow a heavily used interface down unacceptably, so in the usual implementation the application layer directly invokes the hardware layer, completely bypassing the operating system layer. Operating systems provide bypass for performance reasons, but bypass is not unique to operating systems, nor is it used only to gain performance. For example, the Internet is a layered communication system that permits bypass of most features of most of its layers, to achieve flexibility.

In this section we examine two examples of layered computer system organization: the hardware layer at the bottom of a typical computer system and one part of the operating system layer that creates the typical application programming interface known as the file system.

### 2.3.1 A Hardware Layer: The Bus

The hardware layer of a typical computer is constructed of modules that directly implement low-level versions of the three fundamental abstractions. In the example of Figure 2.17, the processor modules interpret programs, the random access memory modules store both programs and data, and the input/output (I/O) modules implement communication links to the world outside the computer.

There may be several examples of each kind of hardware module—multiple processors (perhaps several on one chip, an organization that goes by the buzzword name *multicore*), multiple memories, and several kinds of I/O modules. On closer inspection the I/O modules turn out to be specialized interpreters that implement I/O programs. Thus, the disk controller is an interpreter of disk I/O programs. Among its duties are mapping disk addresses to track and sector numbers and moving data from the disk to the memory. The network controller is an interpreter that talks on its other side to one or more real communication links. The display controller interprets display lists that it finds in memory, lighting pixels on the display as it goes. The keyboard controller interprets keystrokes and places the result in memory. The clock may be nothing but a minuscule interpreter that continually updates a single register with the time of day.

The various modules plug into the shared *bus*, which is a highly specialized communication link used to SEND messages to other modules. There are numerous bus designs, but they have some common features. One such common feature is a set of wires*

*This description in terms of several parallel wires is of a structure called a *parallel bus*. A more thorough discussion of link communication protocols in Section 7.3 [on-line] shows how a bus can also be implemented by sending coded signals down just a few wires, a scheme called a *serial bus.*

**FIGURE 2.17**

A computer with several modules connected by a shared bus. The numbers are the bus addresses to which the attached module responds.

comprising address, data, and control lines that connect to a *bus interface* on each module. Because the bus is shared, a second common feature is a set of rules, called the *bus arbitration protocol,* for deciding which module may send or receive a message at any particular time. Some buses have an additional module, the *bus arbiter*, a circuit or a tiny interpreter that chooses which of several competing modules can use the bus. In other designs, bus arbitration is a function distributed among the bus interfaces. Just as there are many bus designs, there are also many bus arbitration protocols. A particularly influential example of a bus is the UNIBUS®, introduced in the 1970s by Digital Equipment Corporation. The modularity provided by a shared bus with a standard arbitration protocol helped to reshape the computer industry, as was described in Sidebar 1.5.

A third common feature of bus designs is that a bus is a *broadcast* link, which means that every module attached to the bus hears every message. Since most messages are actually intended for just one module, a field of the message called the *bus address* identifies the intended recipient. The bus interface of each module is configured to respond to a particular set of bus addresses. Each module examines the bus address field (which in a parallel bus is usually carried on a set of wires separate from the rest of the message) of every message and ignores any message not intended for it. The bus addresses thus define an address space. Figure 2.17 shows that the two processors might accept messages at bus addresses 101 and 102, respectively; the display controller at bus address 103; the disk controller at bus addresses 104 and 105 (using two addresses makes it convenient to distinguish requests for its two disks); the network at bus address 106; the keyboard at bus address 107; and the clock at bus address 109. For speed, memory modules typically are configured with a range of bus addresses, one bus address per memory address. Thus, if in Figure 2.17 the two memory modules each implement an address space of 1,024 memory addresses, they might be configured with bus addresses 1024–2047 and 3072–4095, respectively.*

*These bus addresses are chosen for convenience of the illustration. In practice, a memory module is more likely to be configured with enough bus addresses to accommodate several gigabytes.

Any bus module that wishes to send a message over the bus must know a bus address that the intended recipient is configured to accept. Name discovery in some buses is quite simple: whoever sets up the system explicitly configures the knowledge of bus addresses into the processor software, and that software passes this knowledge along to other modules in messages it sends over the bus. Other bus designs dynamically assign bus addresses to modules as they are plugged in to the bus and announce their presence.

A common bus design is known as *split-transaction*. In this design, when one module wants to communicate with another, the first module uses the bus arbitration protocol on the control wires to request exclusive use of the bus for a message. Once it has that exclusive use, the module places a bus address of the destination module on the address wires and the remainder of the message on the data wires. Assuming a design in which the bus and the modules attached to it run on uncoordinated clocks (that is, they are asynchronous), it then signals on one of the control wires (called READY) to alert the other modules that there is a message on the bus. When the receiving module notices that one of its addresses is on the address lines of the bus, it copies that address and the rest of the message on the data wires into its local registers and signals on another control line (called ACKNOWLEDGE) to tell the sender that it is safe to release the bus so that other modules can use it. (If the bus and the modules are all running with a common clock, the READY and ACKNOWLEDGE lines are not needed; instead, each module checks the address lines on each clock cycle.) Then, the receiver inspects the address and message and performs the requested operation, which may involve sending one or more messages back to the original requesting module or, in some cases, even to other modules.

For example, suppose that processor #2, while interpreting a running application program, encounters the instruction

```
LOAD        1742, R1
```

which means "load the contents of memory address 1742 into processor register R1". In the simplest scheme, the processor just translates addresses it finds in instructions directly to bus addresses without change. It thus sends this message across the bus:

*processor* #2 ⇒ *all bus modules*: {1742, READ, 102}

The message contains three fields. The first message field (1742) is one of the bus addresses to which memory #1 responds; the second message field requests the recipient to perform a READ operation; and the third indicates that the recipient should send the resulting value back across the bus, using the bus address 102. The memory addresses recognized by each memory module are based on powers of two, so the memory modules can recognize all of the addresses in their own range by examining just a few high-order address bits. In this case, the bus address is within the range recognized by memory module 1, so that module responds by copying the message into its own registers. It acknowledges the request, the processor releases the bus, and the memory module then performs the internal operation

*value* ← READ (1742)

With *value* in hand, the memory module now itself acquires the bus and sends the result back to processor #2 by performing the bus operation

*memory* #1 ⇒ *all bus modules*: {102, *value*}

where 102 is the bus address of the processor as supplied in the original READ request message. The processor, which is probably waiting for this result, notices that the bus address lines now contain its own bus address 102. It therefore copies the value from the data lines into its register R1, as the original program instruction requested. It acknowledges receipt of the message, and the memory module releases the bus for use by other modules.

Simple I/O devices, such as keyboards, operate in a similar fashion. At system initialization time, one of the processors SENDs a message to the keyboard controller telling it to SEND all keystrokes to that processor. Each time that the user depresses a key, the keyboard controller SENDs a message to the processor containing as data the name of the key that was depressed. In this case, the processor is probably *not* waiting for this message, but its bus interface (which is in effect a separate interpreter running concurrently with the processor) notices that a message with its bus address has appeared. The bus interface copies the data from the bus into a temporary register, acknowledges the message, and sends a signal to the processor that will cause the processor to perform an interrupt on its next instruction cycle. The interrupt handler then transfers the data from the temporary register to some place that holds keyboard input, perhaps by SENDing yet another message over the bus to one of the memory modules.

One potential problem of this design is that the interrupt handler must respond and read the keystroke data from the temporary register before the keyboard handler SENDs another keystroke message. Since keyboard typing is slow compared with computer speeds, there is a good chance that the interrupt handler will be there in time to read the data before the next keystroke overwrites it. However, faster devices such as a hard disk might overwrite the temporary register. One solution would be to write a processor program that runs in a tight loop, waiting for data that the disk controller sends over the bus and immediately SENDing that data again over the bus to a memory module.

Some low-end computer designs do exactly that, but a designer can obtain substantially higher performance by upgrading the disk controller to use a technique called *direct memory access*, or DMA. With this technique, when a processor SENDs a request to a disk controller to READ a block of data from the disk, it includes the address of a buffer in memory as a field of the request message. Then, as data streams in from the disk, the disk controller SENDs it directly to the memory module, incrementing the memory address appropriately between SENDs. In addition to relieving the load on the processor, DMA also reduces the load on the shared bus because it transfers each piece of data across the bus just once (from the disk controller to the memory) rather than twice (first from the disk controller to the processor and then from the processor to the memory). Also, if the bus allows long messages, the DMA controller may be able to take better advantage of that feature than the processor, which is usually designed to SEND and RECEIVE bus data in units that are the same size as its own registers. By SENDing longer messages, the DMA controller increases performance

because it amortizes the overhead of the bus arbitration protocol, which it must perform once per message. Finally, DMA allows the processor to execute some other program at the same time that the disk controller is transferring data. Because concurrent operation can hide the latency of the disk transfer, it can provide an additional performance enhancement. The idea of enhancing performance by hiding latency is discussed further in Chapter 6.

A convenient interface to I/O and other bus-attached modules is to assign bus addresses to the control registers and buffers of the module. Since each processor maps bus addresses directly into its own memory address space, LOAD and STORE instructions executed in the processor can in effect address the registers and buffers of the I/O module as if they were locations in memory. The technique is known as *memory-mapped I/O*.

Memory-mapped I/O can be combined with DMA. For example, suppose that a disk controller designed for memory-mapped I/O assigns bus addresses to four of its control registers as follows:

| bus address | control register |
|---|---|
| 121 | *sector_number* |
| 122 | *DMA_start_address* |
| 123 | *DMA_count* |
| 124 | *control* |

To do disk I/O, the processor uses STORE instructions to SEND appropriate initialization values to the first three disk controller registers and a final STORE instruction to SEND a value that sets a bit in the *control* register that the disk controller interprets as the signal to start. A program to GET a 256-byte disk sector currently stored at sector number 11742 and transfer the data into memory starting at location 3328 starts by loading four registers with these values and then issuing STOREs of the registers to the appropriate bus addresses:

```
R1 ← 11742; R2 ← 3328; R3 ← 256; R4 ← 1;
STORE 121,R1                    // set sector number
STORE 122,R2                    // set memory address register
STORE 123,R3                    // set byte count
STORE 124,R4                    // start disk controller running
```

Upon completion of the bus SEND generated by the last STORE instruction, the disk controller, which was previously idle, leaps into action, reads the requested sector from the disk into an internal buffer, and begins using DMA to transfer the contents of the buffer to memory one block at a time. If the bus can handle blocks that are 8 bytes long, the disk controller would SEND a series of bus messages such as

*disk controller* #1 ⇒ *all bus modules*: {3328, *block*[1]}
*disk controller* #1 ⇒ *all bus modules*: {3336, *block*[2]}
etc . . .

Memory-mapped I/O is a popular interface because it provides a uniform memory-like LOAD and STORE interface to every bus module that implements it. On the other hand,

the designer must be cautious in trying to extend the memory-mapped model too far. For example, trying to arrange so the processor can directly address individual bytes or words on a magnetic disk could be problematic in a system with a 32-bit address space because a disk as small as 4 gigabytes would use up the entire address space. More important, the latency of a disk is extremely large compared with the cycle time of a processor. For the STORE instruction to sometimes operate in a few nanoseconds (when the address is in electronic memory) and other times require 10 milliseconds to complete (when the address is on the disk) would be quite unexpected and would make it difficult to write programs that have predictable performance. In addition, it would violate a fundamental rule of human engineering, the ***principle of least astonishment*** (see Sidebar 2.5). The bottom line is that the physical properties of the magnetic disk make the DMA access model more appropriate than the memory-mapped I/O model.

**Sidebar 2.5 Human Engineering and the Principle of Least Astonishment** An important principle of human engineering for usability, which for computer systems means designing to make them easy to set up, easy to use, easy to program, and easy to maintain, is the principle of least astonishment.

**The principle of least astonishment**

*People are part of the system. The design should match the user's experience, expectations, and mental models.*

Human beings make mental models of the behavior of everything they encounter: components, interfaces, and systems. If the actual component, interface, or system follows that mental model, there is a better chance that it will be used as intended and less chance that misuse or misunderstanding will lead to a mistake or disappointment. Since complexity is relative to understanding, the principle also tends to help reduce complexity.

For this reason, when choosing among design alternatives, it is usually better to choose one that is most likely to match the expectations of those who will have to use, apply, or maintain the system. The principle should also be a factor when evaluating tradeoffs. It applies to all aspects of system design, especially to the design of human interfaces and to computer security.

Some corollaries are to be noted: Be consistent. Be predictable. Minimize side-effects. Use names that describe. Do the obvious thing. Provide sensible interpretations for all reasonable inputs. Avoid unnecessary variations.

Some authors prefer the words "principle of least surprise" to "principle of least astonishment". When Bayesian statisticians invoke the principle of least surprise, they usually mean "choose the mostly likely explanation", a version of the closely related Occam's razor. (See the aphorism at the bottom of page 9.)

(*Sidebar continues*)

**Human Engineering and the Original Murphy's Law.** If you ask a group of people "What is Murphy's law?" most responses will be some variation of "If anything can go wrong, it will", followed by innumerable equivalents, such as the toast always falls butter side down.

In fact, Murphy originally said something quite different. Rather than a comment on the innate perversity of inanimate objects (sometimes known as *Finagle's law*, from a science fiction story), Murphy was commenting on a property of human nature that one must take into account when designing complex systems: *If you design it so that it can be assembled wrong, someone will assemble it wrong*. Murphy was pointing out the wisdom of good human engineering of things that are to be assembled: design them so that the only way to assemble them is the right way.

Edward A. Murphy, Jr., was an engineer working on United States Air Force rocket sled experiments at Edwards Air Force Base in 1949, in which Major John Paul Stapp volunteered to be subjected to extreme decelerations (40 Gs) to determine the limits of human tolerance for ejection seat design. On one of the experiments, someone wired up all of the strain gauges incorrectly, so at the end of Stapp's (painful) ride there was no usable data. Murphy said, in exasperation at the technician who wired up the strain gauges, "if that guy can find a way to do it wrong, he will." Stapp, who as a hobby made up laws at every opportunity, christened this observation "Murphy's law," and almost immediately began telling it to others in the different and now widely known form "If anything can go wrong, it will."

A good example of Murphy's original observation in action showed up in an incident on a Convair 580 cargo plane in 1997. Two identical control cables ran from a cockpit control to the elevator trim tab, a small movable surface on the rear stabilizing wing that, when adjusted up or down, forces the nose of the plane to rise or drop, respectively. Upon take-off on the first flight after maintenance, the pilots found that the plane was pitching nose-up. They tried adjusting the trim tab to maximum nose-down position, but the problem just got worse. With much effort they managed to land the plane safely. When mechanics examined the plane, they discovered that the two cables to the trim tab had been interchanged, so that moving the control up caused the trim tab to go down and vice versa*.

A similar series of incidents in 1988 and 1989 involved crossed connections in cargo area smoke alarm signal wires and fire extinguisher control wires in the Boeing 737, 757, and 767 aircraft†.

---

*Transportation Safety Board of Canada, *Report A97O0077*, January 13, 2000, updated October 6, 2002.

†Karen Fitzgerald, "Boeing's crossed connections", *IEEE Spectrum 26*, 5 (May 1989), pages 30–35.

### 2.3.2 A Software Layer: The File Abstraction

The middle and higher layers of a computer system are usually implemented as software modules. To make this layered organization concrete, consider the *file*, a high-level version of the memory abstraction. A file holds an array of bits or bytes, the number of which the application chooses. A file has two key properties:

- *It is durable.* Information, once stored, will remain intact through system shutdowns and can be retrieved later, perhaps weeks or months later. Applications use files to durably store documents, payroll data, e-mail messages, programs, and anything else they do not want to be lost.
- *It has a name.* The name of a file allows users and programs to store information in such a way that they can find and use it again at a later time. File names also make it possible for users to share information. One can WRITE a named file and tell a friend the file name, and then the friend can use the name to READ the file.

Taken together, these two features mean that if, for example, Alice creates a new file named "strategic plan", WRITEs some information in it, shuts down the computer, and the next day turns it on again, she will then be able to READ the file named "strategic plan" and get back its content. Furthermore, she can tell Bob to look at the file named "strategic plan". When Bob asks the system to READ a file with that name, he will read the file that she created. Most file systems also provide other additional properties for files, such as timestamps to determine when they were created, last modified, or last used, assurances about their durability (a topic that Chapter 10 [on-line] revisits), and the ability to control who may share them (one of the topics of Chapter 11 [on-line]).

The system layer implements files using modules from the hardware layer. Figure 2.18 shows the pseudocode of a simple application that reads input from a keyboard device, writes that input to a file, and also displays it on the display device.

```
character buf                                   // buffer for input character
file ← OPEN ("strategic plan", READWRITE)       // open file for reading and writing
input ← OPEN ("keyboard", READONLY)             // open keyboard device for reading
display ← OPEN ("display", WRITEONLY)           // open display device for writing

while not END_OF_FILE (input) do
    READ (input, buf, 1)                        // read 1 character from keyboard
    WRITE (file, buf, 1)                        // store input into file
    WRITE (display, buf, 1)                     // display input

CLOSE (file)
CLOSE (input)
CLOSE (display)
```

**FIGURE 2.18**

Using the file abstraction to implement a display program, which also writes the keyboard input in a file. For clarity, this program ignores the possibility that any of the abstract file primitives may return an error status.

A typical API for the file abstraction contains calls to OPEN a file, to READ and WRITE parts of the file, and to CLOSE the file. The OPEN call translates the file name into a temporary name in a local name space to be used by the READ and WRITE operations. Also, OPEN usually checks whether this user is permitted access to the file. As its last step, OPEN sets a *cursor*, sometimes called a *file pointer*, to zero. The cursor records an offset from the beginning of the file to be used as the starting point for READs and WRITEs. Some file system designs provide a separate cursor for READs and WRITEs, in which case OPEN may initialize the WRITE cursor to the number of bytes in the file.

A call to READ delivers to the caller a specified number of bytes from the file, starting from the READ cursor. It also adds to the READ cursor the number of bytes read so that the next READ proceeds where the previous READ left off. If the program asks to read bytes beyond the end of the file, READ returns some kind of end-of-file status indicator.

Similarly, the WRITE operation takes as arguments a buffer with bytes and a length, stores those bytes in the file starting at the offset indicated by the WRITE cursor (if the WRITE cursor starts at or reaches the end of the file, WRITE usually implies extending the size of the file), and adds to the WRITE cursor the number of bytes written so that the next WRITE can continue from there. If there is not enough space on the device to write that many bytes, the WRITE procedure fails by returning some kind of device-full error status or exception.

Finally, when the program is finished reading and writing, it calls the CLOSE procedure. CLOSE frees up any internal state that the file system maintains for the file (for example, the cursors and the record of the temporary file name, which is no longer meaningful). Some file systems also ensure that, when CLOSE returns, all parts of the modified file have been stored durably on a non-volatile memory device. Other file systems perform this operation in the background after CLOSE returns.

The file system module implements the file API by mapping bytes of the file to disk sectors. For each file the file system creates a record of the name of the file and the disk sectors in which it has stored the file. The file system also stores this record on the disk. When the computer restarts, the file system must somehow discover the place where it left these records so that it can again find the files. A typical procedure for name discovery is for the file system to reserve one, well-known, disk sector such as sector number 1, and use that well-known disk sector as a toehold to locate the sectors where it left the rest of the file system information. A detailed description of the UNIX file system API and its implementation is in Section 2.5.

One might wonder why the file API supports OPEN and CLOSE in addition to READ and WRITE; after all, one could ask the programmer to pass the file name and a file position offset on each READ and WRITE call. The reason is that the OPEN and CLOSE procedures mark the beginning and the end of a sequence of related READ and WRITE operations so that the file system knows which reads and writes belong together as a group. There are several good reasons for grouping and for the use of a temporary file name within the grouping. Originally, performance and resource management concerns motivated the introduction of OPEN and CLOSE, but later implementations of the interface exploited the existence of OPEN and CLOSE to provide clean semantics under concurrent file access and failures.

Early file systems introduced OPEN to amortize the cost of resolving a file name. A file name is a path name that may contain several components. By resolving the file

name once on OPEN and giving the result a simple name, READ and WRITE avoid having to resolve the name on each invocation. Similarly, OPEN amortizes the cost of checking whether the user has the appropriate permissions to use the file.

CLOSE was introduced to simplify resource management: when an application invokes CLOSE, the file system knows that the application doesn't need the resources (e.g., the cursor) that the file system maintains internally. Even if a second application removes a file before a first application is finished reading and writing the file, the file system can implement READ and WRITE procedures for the first application sensibly (for example, discard the contents of the file only after everyone that OPENed the file has called CLOSE).

More recent file systems use OPEN and CLOSE to mark the beginning and end of an *atomic* action. The file system can treat all intervening READ and WRITE calls as a single indivisible operation, even in the face of concurrent access to the file or a system crash after some but not all of the WRITEs have completed. Two opportunities ensue:

1. The file system can use the OPEN and CLOSE operations to coordinate concurrent access to a file: if one program has a file open and another program tries to OPEN that same file, the file system can make the second program wait until the first one has CLOSEd the file. This coordination is an example of *before-or-after atomicity*, a topic that Section 5.2.4 explores in depth.

2. If the file system crashes (for example, because of a power failure) before the application CLOSEs the file, none of the WRITEs will be in the file when the system comes back up. If it crashes after the application CLOSEd the file, all of the WRITEs will be in the file. Not all file systems provide this guarantee, known as *all-or-nothing atomicity*, since it is not easy to implement correctly and efficiently, as Chapter 9 [on-line] explains.

There is a cost to the OPEN/CLOSE model: the file system must maintain per-client state in the form of the resolved file name and the cursor(s). It is possible to design a completely stateless file interface. An example is the Network File System, described in Section 4.5.

The file is such a convenient memory abstraction that in some systems (for example, the UNIX system and its derivatives) *every* input/output device in a computer system provides a file interface (see Figure 2.19). In such systems, files not only are an abstraction for non-volatile memories (e.g., magnetic disks), but they are also a convenient interface to the keyboard device, the display, communication links, and so on. In such systems, each I/O device has a name in the file naming scheme. A program OPENs the keyboard device, READs bytes from the keyboard device, and then CLOSEs the keyboard device, without having to know any details about the keyboard management procedure, what type of keyboard it is, and the like. Similarly, to interact with the display, a program can OPEN the display device, WRITE to it, and CLOSE it. The program need not know any details about the display. In accordance with the ***principle of least astonishment***, each device management procedure provides some reasonable interpretation for every file system method. The pseudocode of Figure 2.18 exemplifies the benefit of this kind of design uniformity.

**FIGURE 2.19**

Using the file abstraction and layering to integrate different kinds of input and output devices. The file system acts as an intermediary that provides a uniform, abstract interface, and the various device managers are programs that translate that abstract interface into the operational requirements for different devices.

One feature of such a uniform interface is that in many situations one can, by simply rebinding the name, replace an I/O device with a file, or vice versa, without modifying the application program in any way. This use of naming in support of modularity is especially helpful when debugging an application program. For example, one can easily test a program that expects keyboard input by slipping a file filled with text in the place of the keyboard device. Because of such examples, the file system abstraction has proven to be very successful.

## 2.4 LOOKING BACK AND AHEAD

This chapter has developed several ideas and concepts that provide useful background for the study of computer system design. First, it described the three major abstractions used in designing computer systems—memory, interpreters, and communication links. Then it presented a model of how names are used to glue together modules based on those abstractions to create useful systems. Finally, it described some parts of a typical modern layered computer system in terms of the three major abstractions. With this background, we are now prepared to undertake a series of more in-depth discussions of specific computer system design topics. The first such in-depth discussion, in Chapter 3, is of the several engineering problems surrounding the use of names. Each of the remaining chapters undertakes a similar in-depth discussion of a different system design topic.

Before moving on to those in-depth discussions, the last section of this chapter is a case study of how abstraction, naming, and layers appear in practice. The case study uses those three concepts to describe the UNIX system.

## 2.5 CASE STUDY: UNIX® FILE SYSTEM LAYERING AND NAMING

The UNIX family of operating systems can trace its lineage back to the UNIX operating system that was developed by Bell Telephone Laboratories for the Digital Equipment Corporation PDP line of minicomputers in the late 1960s and early 1970s [Suggestions for Further Reading 2.2], and before that to the Multics* operating system in the early 1960s [Suggestions for Further Reading 1.7.5 and 3.1.4]. Today there are many flavors of UNIX systems with complex historical relationships; a few examples include GNU/Linux, versions of GNU/Linux distributed by different organizations (e.g., Red Hat, Ubuntu), Darwin (a UNIX operating system that is part of Apple's operating system Mac OS X), and several flavors of BSD operating systems. Some of these are directly derived from the early UNIX operating system; others provide similar interfaces but have been implemented from scratch. Some are the result of an effort by a small group of programmers, and others are the result of an effort by many. In the latter case, it is even unclear how to exactly name the operating system because substantial parts come from different teams.† The collective result of all these efforts is that operating systems of the UNIX family run on a wide range of computers, including personal computers, server computers, parallel computers, and embedded computers. Most of the UNIX interface is an official standard,‡ and non-UNIX operating systems often support this standard too. Because the source code of some versions is available to the public, one can easily study the UNIX system.

This case study examines the various ways in which the UNIX file system uses names in its design. In the course of examining how it implements its naming scheme, we will also incidentally get a first-level overview of how the UNIX file system is organized.

### 2.5.1 Application Programming Interface for the UNIX File System

A program can create a file with a user-chosen name, read and write the file's content, and set and get a file's metadata. Example metadata include the time of last modification, the user ID of the file's owner, and access permissions for other users. (For a full discussion of metadata see Section 3.1.2.) To organize their files, users can group them in directories with user-chosen names, creating a naming network. Users can also graft a naming network stored on a storage device onto an existing naming network, allowing naming networks for different devices to be incorporated into a single large nam-

*The name UNIX evolved from Unics, which was a word joke on Multics.

†We use "Linux" for the Linux kernel, while we use "GNU/Linux" for the complete system, recognizing that this naming convention is not perfect either because there are pieces of the system that are neither GNU software nor part of the kernel (e.g., the X Window System; see Sidebar 4.4).

‡POSIX® (Portable Operating System Interface), Federal Information Processing Standards (FIPS) 151-2. FIPS 151-2 adopts ISO/IEC 9945-1: 2003 (IEEE Std. 1003.1: 2001) Information Technology-Portable Operating System Interface (POSIX)-Part 1: System Application: Program Interface (API) [C Language].

**Table 2.1** UNIX File System Application Programming Interface

| Procedure | Brief Description |
|---|---|
| OPEN (*name, flags, mode*) | Open file *name*. If the file doesn't exist and *flags* is set, create file with permissions *mode*. Set the file cursor to 0. Returns a file descriptor. |
| READ (*fd, buf, n*) | Read *n* bytes from the file at the current cursor and increase the cursor by the number of bytes read. |
| WRITE (*fd, buf, n*) | Write *n* bytes at the current cursor and increase the cursor by the bytes written. |
| SEEK (*fd, offset, whence*) | Set the cursor to *offset* bytes from beginning, end, or current position. |
| CLOSE (*fd*) | Delete file descriptor. If this is the last reference to the file, delete the file. |
| FSYNC (*fd*) | Make all changes to the file durable. |
| STAT (*name*) | Read metadata of file. |
| CHMOD, CHOWN, etc. | Various procedures to set specific metadata. |
| RENAME (*from_name, to_name*) | Change name from *from_name* to *to_name* |
| LINK (*name, link_name*) | Create a hard link *link_name* to the file *name*. |
| UNLINK (*name*) | Remove *name* from its directory. If *name* is the last name for a file, remove file. |
| SYMLINK (*name, link_name*) | Create a symbolic name *link_name* for the file *name*. |
| MKDIR (*name*) | Create a new directory named *name*. |
| CHDIR (*name*) | Change current working directory to *name*. |
| CHROOT (*name*) | Change the default root directory to *name*. |
| MOUNT (*name, device*) | Graft the file system on *device* onto the name space at *name*. |
| UNMOUNT (*name*) | Unmount the file system at *name*. |

ing network. To support these operations, the UNIX file system provides the application programming interface (API) shown in Table 2.1.

To tackle the problem of implementing this API, the UNIX file system employs a divide-and-conquer strategy. The UNIX file system makes use of several hidden layers of machine-oriented names (that is, addresses), one on top of another, to implement files. It then applies the UNIX durable object naming scheme to map user-friendly names to these files. Table 2.2 illustrates this structure.

In the rest of this section we work our way up from the bottom layer of Table 2.2 to the top layer, proceeding from the lowest layer of the system up toward

**Table 2.2** The Naming Layers of the UNIX File System

| Layer | Purpose | |
|---|---|---|
| Symbolic link layer | Integrate multiple file systems with symbolic links. | ↑ |
| Absolute path name layer | Provide a root for the naming hierarchies. | user-oriented names |
| Path name layer | Organize files into naming hierarchies. | ↓ |
| File name layer | Provide human-oriented names for files. | machine-user interface |
| Inode number layer | Provide machine-oriented names for files. | ↑ |
| File layer | Organize blocks into files. | machine-oriented names |
| Block layer | Identify disk blocks. | ↓ |

the user. This description corresponds closely to the implementation of Version 6 of the UNIX system, which dates back to the early 1970s. Version 6 is well documented [Suggestions for Further Reading 2.2.2] and captures the important ideas that are found in many modern UNIX file systems, but modern versions are more complex; they provide better robustness and handle large files, many files, and so on, more efficiently. In a few places we will point out some of these differences, but the reader is encouraged to consult papers in the file system literature to find out how modern UNIX file systems work and are evolving.

### 2.5.2 The Block Layer

At the bottom layer the UNIX file system names some physical device such as a magnetic disk, flash disk, or magnetic tape that can store data durably. The storage on such a device is divided into fixed-size units, called *blocks*. For a magnetic disk (see Sidebar 2.2), a block corresponds to a small number of disk sectors. A block is the smallest allocation unit of disk space, and its size is a trade-off between several goals. A small block reduces the amount of disk wasted for small files; if many files are smaller than 4 kilobytes, a 16-kilobyte block size wastes space. On the other hand, a very small block size may incur large data structures to keep track of free and allocated blocks. In addition, there are performance considerations that impact the block size, some of which we discuss in Chapter 6. In version 6, the UNIX file system used 512-byte blocks, but modern UNIX file systems often use 8-kilobyte blocks.

The names of these blocks are numbers, which typically correspond to the offset of the block from the beginning of the device. In the bottom naming layer, a storage

device can be viewed as a context that binds block numbers to physical blocks. The name-mapping algorithm for a block device is simple: it takes as input a block number and returns the block. Actually, we don't really want the block itself—that would be a pile of iron oxide. What we want is the *contents* of the block, so the algorithm actually implements a fixed mapping between block name and block contents. If we represent the storage device as a linear array of blocks, then the following code fragment implements the name-mapping algorithm:

```
procedure BLOCK_NUMBER_TO_BLOCK (integer b) returns block
    return device[b]
```

In this simple algorithm the variable name *device* refers to some particular physical device. In many devices the mapping is more complicated. For example, a hard drive might keep a set of spare blocks at the end and rebind the block numbers of any blocks that go bad to spares. The hard drive may itself be implemented in layers, as will be seen in Section 8.5.4 [on-line]. The value returned by BLOCK_NUMBER_TO_BLOCK is the contents of block *b*.

*Name discovery*: The names of blocks are integers from a compact set, but the block layer must keep track of which blocks are in use and which are available for assignment. As we will see, the file system in general has a need for a description of the layout of the file system on disk. As an anchor for this information, the UNIX file system starts with a ***super block***, which has a well-known name (e.g., 1). The super block contains, for example, the size of the file system's disk in blocks. (Block 0 typically stores a small program that starts the operating system; see Sidebar 5.3.)

Different implementations of the UNIX file system use different representations for the list of free blocks. The version 6 implementation keeps a list of block numbers of unused blocks in a linked list that is stored in some of the unused blocks. The block number of the first block of this list is stored in the super block. A call to allocate a block leads to a procedure in the block layer that searches the list array for a free block, removes it from the list, and returns that block's block number.

Modern UNIX file systems often use a bitmap for keeping track of free blocks. Bit *i* in the bitmap records whether block *i* is free or allocated. The bitmap itself is stored at a well-known location on the disk (e.g., right after the super block). Figure 2.20 shows a possible disk layout for a simple file system. It starts with the super block, followed by a bitmap that records which disk blocks are in use. After the bitmap comes the inode table, which has one entry for each file (as explained next), followed by blocks that are either free or allocated to some file. The super block contains the size of the bitmap and inode table in blocks.

**FIGURE 2.20**
Possible disk layout for a simple file system.

### 2.5.3 The File Layer

Users need to store items that are larger than one block in size and that may grow or shrink over time. To support such items, the UNIX file system introduces a next naming layer for *files*. A file is a linear array of bytes of arbitrary length. The file system needs to record in some way which blocks belong to each file. To support this requirement, the UNIX file system creates an index node, or *inode* for short, as a container for metadata about the file. Our initial declaration of an inode is:

```
structure inode
    integer block_numbers[N]    // the numbers of the blocks that constitute the file
    integer size                // the size of the file in bytes
```

The inode for a file is thus a context in which the various blocks of the file are named by integer block numbers. With this structure, a simplified name-mapping algorithm for resolving the name of a block in a file is as follows:

```
procedure INDEX_TO_BLOCK_NUMBER (inode instance i, integer index) returns integer
    return i.block_numbers[index]
```

The version 6 UNIX file system uses this algorithm for small files, which are limited to $N = 8$ blocks. For large files, version 6 uses a more sophisticated algorithm for mapping the *index*-th block of an inode to a block number. The first seven entries in *i.block_numbers* are ***indirect blocks***. Indirect blocks do not contain data, but block numbers. For example, with a block size of 512 bytes and an *index* of 2 bytes (as in Version 6), an indirect block can contain 256 2-byte block numbers. The eighth entry is a doubly indirect block (blocks that contain block numbers of indirect blocks). This design with indirect and doubly indirect blocks allows for $(N - 1) \times 256 + 1 \times 256 \times 256 = 67{,}329$ blocks when $N = 8$, about 32 megabytes.* Problem set *1* explores some design trade-offs to allow the file system to support large files. Some modern UNIX file systems use different representations or more sophisticated data structures, such as B+ trees, to implement files.

The UNIX file system allows users to name any particular byte in a file by layering the previous two naming schemes and specifying the byte number as an offset from the beginning of the file:

```
procedure INODE_TO_BLOCK (integer offset, inode instance i) returns block
    o ← offset / BLOCKSIZE
    b ← INDEX_TO_BLOCK_NUMBER (i, o)
    return BLOCK_NUMBER_TO_BLOCK (b)
```

The value returned is the entire block that holds the value of the byte at *offset*. Version 6 used for *offset* a 3-byte number, which limits the maximum file size to $2^{24}$ bytes. Modern UNIX file systems use a 64-bit number. The procedure returns the

*The implementation of Version 6, however, restricts the maximum number of blocks per file to $2^{15}$.

entire block that contains the named byte. As we will see in Section 2.5.11, READ uses this procedure to return the requested bytes.

### 2.5.4 The Inode Number Layer

Instead of passing inodes themselves around, it would be more convenient to name them and pass their names around. To support this feature, the UNIX file system provides another naming layer that names inodes by an inode number. A convenient way to implement this naming layer is to employ a table that directly contains all inodes, indexed by inode number. Here is the naming algorithm:

```
procedure INODE_NUMBER_TO_INODE (integer inode_number) returns inode
    return inode_table[inode_number]
```

where *inode_table* is an object that is stored at a fixed location on the storage device (e.g., at the beginning). The name-mapping algorithm for *inode_table* just returns the starting block number of the table.

*Name discovery*: inode numbers, like disk block numbers, are a compact set of integers, and again the inode number layer must keep track of which inode numbers are in use and which are free to be assigned. As with block number assignment, different implementations use various representations for a list of free inodes and provide calls to allocate and deallocate inodes. In the simplest implementation, the inode contains a field recording whether or not it is free.

By putting these three layers together, we obtain the following procedure:

```
procedure INODE_NUMBER_TO_BLOCK (integer offset, integer inode_number)
                                                    returns block
    inode instance i ← INODE_NUMBER_TO_INODE (inode_number)
    o ← offset / BLOCKSIZE
    b ← INDEX_TO_BLOCK_NUMBER (i, o)
    return BLOCK_NUMBER_TO_BLOCK (b)
```

This procedure returns the block that contains the byte at *offset* in the file named by *inode_number*. This procedure traverses three layers of naming. There are numbers for storage blocks, numbered indexes for blocks belonging to an inode, and numbers for inodes.

### 2.5.5 The File Name Layer

Numbers are convenient names for use by a computer (numbers can be stored in fixed-length fields that simplify storage allocation) but are inconvenient names for use by people (numbers have little mnemonic value). In addition, block and inode numbers specify a location, so if it becomes necessary to rearrange the physical storage, the numbers must change, which is again inconvenient for people. The UNIX file system deals with this problem by inserting a naming layer whose sole purpose is

| File name | Inode number |
|---|---|
| program | 10 |
| paper | 12 |

**FIGURE 2.21**

A directory.

to hide the metadata of file management. Above this layer is a user-friendly naming scheme for durable objects—files and input/output devices. This naming scheme again has several layers. The most visible component of the durable object naming scheme is the *directory*. In the UNIX file system, a directory is a context containing a set of bindings between character-string names and inode numbers.

To create a file, the UNIX file system allocates an inode, initializes its metadata, and binds the proposed name to that inode in some directory. As the file is written, the file system allocates blocks to the inode.

By default, the UNIX file system adds the file to the current working directory. The current working directory is a context reference to the directory in which the active application is working. The form of the context reference is just another inode number. If *wd* is the name of the state variable that contains the working directory for a running program (called a *process* in the UNIX system), one can look up the inode number of the just-created file by supplying *wd* as the second argument to a procedure such as:

```
procedure NAME_TO_INODE_NUMBER (character string filename, integer dir) returns integer
    return LOOKUP (filename, dir)
```

The procedure CHDIR, whose implementation we describe later, allows a process to set *wd*.

To represent a directory, the UNIX file system reuses the mechanisms developed so far: it represents directories as files. By convention, a file that represents a directory contains a table that maps file names to inode numbers. For example, Figure 2.21 is a directory with two file names ("program" and "paper"), which are mapped to inode numbers 10 and 12, respectively. In Version 6, the maximum length of a name is 14 bytes, and the entries in the table have a fixed length of 16 bytes (14 for the name and 2 for the inode number). Modern UNIX file systems allow for variable-length names, and the table representation is more sophisticated.

To record whether an inode is for a directory or a file, the UNIX file system extends the inode with a type field:

```
structure inode
    integer block_numbers[N]   // the numbers of the blocks that constitute the file
    integer size               // the size of the file in bytes
    integer type               // type of file: regular file, directory, . . .
```

MKDIR creates a zero-length file (directory) and sets *type* to DIRECTORY. Extensions introduced later will add additional values for *type*.

With this representation of directories and inodes, LOOKUP is as follows:

```
1   procedure LOOKUP (character string filename, integer dir) returns integer
2       block instance b
3       inode instance i ← INODE_NUMBER_TO_INODE (dir)
4       if i.type ≠ DIRECTORY then return FAILURE
5       for offset from 0 to i.size − 1 do
6           b ← INODE_NUMBER_TO_BLOCK (offset, dir)
7           if STRING_MATCH (filename, b) then
8               return INODE_NUMBER (filename, b)
9           offset ← offset + BLOCKSIZE
10      return FAILURE
```

LOOKUP reads the blocks that contain the data for the directory *dir* and searches for the string *filename* in the directory's data. It computes the block number for the first block of the directory (line *6*) and the procedure STRING_MATCH (no code shown) searches that block for an entry for the name *filename*. If there is an entry, INODE_NUMBER (no code shown) returns the inode number in the entry (line *8*). If there is no entry, LOOKUP computes the block number for the second block, and so on, until all blocks of the directory have been searched. If none of the blocks contain an entry for *filename*, LOOKUP returns an error (line *10*). As an example, an invocation of LOOKUP ("program", *dir*), where *dir* is the inode number for the directory of Figure 2.21, would return the inode number 10.

### 2.5.6 The Path Name Layer

Having all files in a single directory makes it hard for users to keep track of large numbers of files. Enumerating the contents of a large directory would generate a long list that is organized simply (e.g., alphabetically) at best. To allow arbitrary groupings of user files, the UNIX file system permits users to create named directories.

A directory can be named just like a file, but the user also needs a way of naming the files in that directory. The solution is to add some structure to file names: for example, "projects/paper", in which "projects" names a directory and "paper" names a file in that directory. Structured names such as these are examples of path names. The UNIX file system uses a virgule (forward slash) as a separator of the components of a path name; other systems choose different separator characters such as period, back slash, or colon. With these tools, users can create a hierarchy of directories and files.

The name-resolving algorithm for path names can be implemented by layering a recursive procedure over the previous directory lookup procedure:

```
procedure PATH_TO_INODE_NUMBER (character string path, integer dir) returns integer
    if (PLAIN_NAME (path)) return NAME_TO_INODE_NUMBER (path, dir)
    else
        dir ← LOOKUP (FIRST (path), dir)
        path ← REST (path)
        return PATH_TO_INODE_NUMBER (path, dir)
```

The function PLAIN_NAME (*path*) scans its argument for the UNIX standard path name separator (forward slash) and returns TRUE if it does not find one. If there is no

separator, the program resolves the simple name to an inode number in the requested directory (line *2*). If there is a separator in *path*, the program takes it to be a path name and goes to work on it (lines *4* through *6*). The function FIRST peels off the first component name from the path, and REST returns the remainder of the path name. Thus, for example, the call PATH_TO_NAME ("projects/paper", *wd*) results in the recursive call PATH_TO_NAME ("paper", *dir*), where *dir* is the inode number for the directory "projects".

With path names, one often has to type names with many components. To address this annoyance, the UNIX file system supports a change directory procedure, CHDIR, allowing a process to set its working directory:

```
procedure CHDIR (path character string)
    wd ← PATH_TO_INODE_NUMBER (path, wd)
```

When a process starts, it inherits the working directory from the parent process that created this process.

### 2.5.7 Links

To refer to files in directories other than the current working directory still requires typing long names. For example, while we are working in the directory "projects"—after calling CHDIR ("projects")—we might have to refer often to the file "Mail/inbox/new-assignment". To address this annoyance, the UNIX file system supports synonyms known as *links*. In the example, we might want to create a link for this file in the current working directory, "projects". Invoking the LINK procedure with the following arguments:

```
LINK ("Mail/inbox/new-assignment", "assignment")
```

makes "assignment" a synonym for "Mail/inbox/new-assignment" in "projects", if "assignment" doesn't exist yet. (If it does, LINK will return an error saying "assignment" already exists.) With links, the directory hierarchy turns from a strict hierarchy into a directed graph. (The UNIX file system allows links only to files, not to directories, so the graph is not only directed but acyclic. We will see why in a moment.)

The UNIX file system implements links simply as bindings in different contexts that map different file names to the same inode number; thus, links don't require any extension to the naming scheme developed so far. For example, if the inode number for "new-assignment" is 481, then the directory "Mail/inbox" contains an entry {"new-assignment", 481} and after the above command is executed the directory "projects" contains an entry {"assignment", 481}. In UNIX system jargon, "projects/assignment" is now linked to "Mail/inbox/new-assignment".

When a file is no longer needed, a process can remove a file using UNLINK (*filename*), indicating to the file system that the name *filename* is no longer in use. UNLINK removes the binding of *filename* to its inode number from the directory that contains *filename*. The file system also puts *filename*'s inode and the blocks of *filename*'s inode on the free list if this binding is the last one containing the inode's number.

Before we added links, a file was bound to a name in only one directory, so if a process asks to delete the name from that directory, the file system can also delete the file. But now that links have been added, when a process asks to delete a name, there may still be names in other directories bound to the file, in which case the file shouldn't be deleted. This raises the question, when should a file be deleted? The UNIX file system deletes a file when a process removes the last binding for a file. The UNIX file system implements this policy by keeping a reference count in the inode:

```
structure inode
    integer block_numbers[N]
    integer size
    integer type
    integer refcnt
```

Whenever it makes a binding to an inode, the file system increases the reference count of that inode. To delete a file, the UNIX file system provides an UNLINK(*filename*) procedure, which deletes the binding specified by *filename*. At the same time the file system decreases the reference count in the corresponding inode by one. If the decrease causes the reference count to go to zero, that means there are no more bindings to this inode, so the file system can free the inode and its corresponding blocks. For example, UNLINK ("Mail/inbox/new-assignment") removes the directory entry "new-assignment" in the directory "Mail/inbox", but not "assignment", because after the unlink the *refcnt* in inode 481 will be 1. Only after calling UNLINK ("assignment") will the inode 481 and its blocks be freed.

Using reference counts works only if there are no cycles in the naming graph. To ensure that the UNIX naming network is a directed graph without cycles, the UNIX file system forbids links to directories. To see why cycles are avoided, consider a directory "a", which contains a directory "b". If a program invokes LINK ("a/b/c", "a") in the directory that contains "a", then the system would return an error and not perform the operation. If the system had performed this operation, it would have created a cycle from "c" to "a" and would have increased the reference count in the inode of "a" by one. If a program then invokes UNLINK ("a"), the name "a" is removed, but the inode and the blocks of "a" wouldn't be removed because the reference count in the inode of "a" is still positive (because of the link from "c" to "a"). But once the name "a" would be removed, a user would no longer be able to name the directory "a" and wouldn't be able to remove it either. In that case, the directory "a" and its subdirectories would be disconnected from the naming graph, but the system would not remove it because the reference count in the inode of "a" is still positive. It is possible to detect this situation, for example by using garbage collection, but it is expensive to do so. Instead, the designers chose a simpler solution: don't allow links to directories, which rules out the possibility of cycles.

There are two special cases, however. First, by default each directory contains a link to itself; the UNIX file system reserves the string "." (a single dot) for this purpose. The name "." thus allows a process to name the current directory without knowing

which the directory it is. When a directory is created, the directory's inode has a reference count of two: one for the inode of the directory and one for the link ".", because it points to itself. Because "." introduces a cycle of length 0, there is no risk that part of the naming network will become disconnected when removing a directory. When unlinking a directory, the file system just decreases the reference count of the directory's inode by 2.

Second, by default, each directory also contains a link to a parent directory; the file system reserves the string ".." (two consecutive dots) for this purpose. The name ".." allows a process to name a parent directory and, for example, move up the file hierarchy by invoking CHDIR ("..."). The link doesn't create problems. Only when a directory has no other entries than "." and ".." can it be removed. If a user wants to remove a directory "a", which contains a directory "b", then the file system refuses to do so until the user first has removed "b". This rule ensures that the naming network cannot become disconnected.

### 2.5.8 Renaming

Using LINK and UNLINK, Version 6 implemented RENAME (*from_name*, *to_name*) as follows:

1 UNLINK (*to_name*)
2 LINK (*from_name*, *to_name*)
3 UNLINK (*from_name*)

This implementation, however, has an undesirable property. Programs often use RENAME to change a working copy of a file into the official version; for example, a user may be editing a file "x". The text editor actually makes all changes to a temporary file "#x". When the user saves the file, the editor renames the temporary file "#x" to "x".

The problem with implementing RENAME using LINK and UNLINK is that if the computer fails between steps 1 and 2 and then restarts, the name *to_name* ("x" in this case) will be lost, which is likely to surprise the user, who is unlikely to know that the file still exists but under the name "#x". What is really needed is that "#x" be renamed to "x" in a single, atomic operation, but that requires atomic actions, which are the topic of Chapter 9 [on-line].

Without atomic actions, it is possible to implement the following slightly weaker specification for RENAME: if *to_name* already exists, an instance of *to_name* will always exist, even if the system should fail in the middle of RENAME. This specification is good enough for the editor to do the right thing and is what modern versions provide.

Modern versions implement this specification in essence as follows:

1 LINK (*from_name*, *to_name*)
2 UNLINK (*from_name*)

Because one cannot link to a name that already exists, RENAME implements the effects of these two calls by manipulating the file system structures directly. RENAME first changes the inode number in the directory entry for *to_name* to the inode number for

*from_name* on disk. Then, RENAME removes the directory entry for *from_name*. If the file system fails between these two steps, then on recovery the file system must increase the reference count in *from_name*'s inode because both *from_name* and *to_name* are pointing to the inode. This implementation ensures that if *to_name* exists before the call to RENAME, it will continue to exist, even if the computer fails during RENAME.

### 2.5.9 The Absolute Path Name Layer

The UNIX system provides each user with a personal directory, called a user's *home directory*. When a user logs on to a UNIX system, it starts a command interpreter (known as the *shell*) through which a user can interact with the system. The shell starts with the working directory (*wd*) set to the inode number of the user's home directory. With the above procedures, users can create personal directory trees to organize the files in their home directory.

But having several personal directory trees does not allow one user to share files with another. To do that, one user needs a way of referring to the names of files that belong to another user. The easiest way to accomplish that is to bind a name for each user to that user's top-level directory, in some context that is available to every user. But then there is a requirement to name this systemwide context. Typically, there are needs for other systemwide contexts, such as a directory containing shared program libraries. To address these needs with a minimum of additional mechanisms, the file system provides a universal context, known as the *root* directory. The root directory contains bindings for the directory of users, the directory containing program libraries, and any other widely shared directories. The result is that all files of the system are integrated into a single directory tree (with restricted cross-links) based on the root.

This design leaves a name discovery question: how can a user name the root directory? Recall that name lookup requires a context reference—the name of a directory inode—and until now that directory inode has been supplied by the working directory state variable. To implement the root, the file system simply declares inode number 1 to be the inode for the root directory. This *well-known name* can then be used by any user as the starting context in which to look up the name of a shared context, or another user (or even to look up one's own name, to set the working directory when logging in).

The file system actually provides two ways to refer to things in the root directory. Starting from any directory in the system, one can use the name ".." to name that directory's parent, "../.." to name the directory above that, and so on until the root directory is reached. A user can tell that the root directory is reached, because ".." in the root directory names the root directory. That is, in the root directory, both "." and ".." are links to the root directory. The other way is with absolute path names, which in the UNIX file system are names that start with a "/", for example, "/Alice/Mail/inbox/new-assignment".

To support absolute path names as well as relative path names, we need one more layer in the naming scheme:

```
procedure GENERALPATH_TO_INODE_NUMBER (character string path) returns integer
    if (path[0] = "/") return PATH_TO_INODE_NUMBER(path, 1)
    else return PATH_TO_INODE_NUMBER(path, wd)
```

At this point we have completed a naming scheme that allows us to name and share durable storage on a single disk. For example, to find the blocks corresponding to the file "/programs/pong.c" with the information in Figure 2.22, we start by finding the inode table, which starts at a block number (block 4 in our example) stored in the super block (not shown in this figure, but see Figure 2.20). From there we locate the root inode (which is known to be inode number 1). The root inode contains the block numbers that in turn contain the blocks of the root directory; in the figure the root starts in block number 14. Block 14 lists the entries in the root directory: "programs" is named by inode number 7. The inode table says that data for inode number 7 starts in block number 23, which contains the contents of the "programs" directory. The file "pong.c" is named by inode number 9. Referring once more to the inode table, to see where inode 9 is stored, we see that the data corresponding to inode 9 starts in block number 61. In short, directories and files are carefully laid out so that all information can be found by starting from the well-known location of the root inode.

The default root directory in Version 6 is inode 1. Version 7 added a call, CHROOT, to change the root directory for a process. For example, a Web server can be run in the corner of the UNIX name space by changing its root directory to, for example, "/tmp". After this call, the root directory for the Web server corresponds to the inode number of the directory "/tmp" and ".." in "/tmp" is a link to "/tmp". Thus, the server can name only directories and files below "/tmp".

**FIGURE 2.22**

Example disk layout for a UNIX file system, refining Figure 2.20 by focusing on the inode table and data blocks. The inode table is a group of contiguous blocks starting at a well-known address, found in the super block (not shown). In this example, blocks 4, 5, and 6 contain the inode table, while blocks 7–61 contain directories and files. The root inode is by convention the well-known inode #1. Typically, inodes are smaller than a block, so in this example there are four inodes in each block. Blocks #14, #37, and #16 constitute the root directory, while block #23 is the first of four blocks of the directory named "/programs", and block #61 is the first block of the three-block file "/programs/pong.c".

### 2.5.10 The Symbolic Link Layer

To allow users to name files on other disks, the UNIX file system supports an operation to attach new disks to the name space. A user can choose the name under which each device is attached: for example, the procedure

```
MOUNT ("/dev/fd1", "/flash")
```

grafts the directory tree stored on the physical device named "/dev/fd1" onto the directory "/flash". (This command demonstrates that each device also has a name in the same object name space we have been describing; the file corresponding to a device typically contains information about the device itself.) Typically mounts do not survive a shutdown: after a reboot, the user has to explicitly remount the devices. It is interesting to contrast the elegant UNIX approach with the DOS approach, in which devices are named by fixed one-character names (e.g.,"C:").

The UNIX file system implements MOUNT by recording in the in-memory inode for "flash" that a file system has been mounted on it and keeps this inode in memory until at least the corresponding UNMOUNT. In memory, the system also records the device and the root inode number of the file system that has been mounted on it. In addition, it records in the in-memory version of the inode for "/dev/fd1" what its parent inode is.

The information for mount points is all recorded in volatile memory instead of on disk and doesn't survive a computer failure. After a failure, the system administrator or a program must invoke MOUNT again. Supporting MOUNT also requires a change to the file name layer: if LOOKUP runs into an inode on which a file system is mounted, it uses the root inode of the mounted file system for the lookup.

UNMOUNT undoes the mount.

With mounted file systems, synonyms become a more difficult problem because per mounted file system there is an address space of inode numbers. Every inode number has a default context: the disk on which it is located. Thus, there is no way for a directory entry on one disk to bind to an inode number on a different disk. This problem can be approached in several ways, two of which are: (1) make inodes unique across all disks or (2) create synonyms for files on other disks in a different way. The UNIX system chooses the second approach by using indirect names called *symbolic* or *soft* links, which bind a file name to another file name. Most systems use method (2) because of the complications that would be involved in trying to keep inode numbers universally unique, small in size, and fast to resolve.

Using the procedure SYMLINK, users can create synonyms for files in the same file system or for files in mounted file systems. The file system implements the procedure SYMLINK by allowing the type field of an inode to be a SYMLINK, which tells whether the blocks associated with the inode contain data or a path name:

```
structure inode
    integer block_numbers[N]
    integer size
    integer type        // Type of inode: regular file, directory, symbolic link, . . .
    integer refcnt
```

If the *type* field has value SYMLINK, then the data in the array *blocks*[*i*] actually contains the characters of a path name rather than a set of inode numbers.

Soft links can be implemented by layering them over GENERALPATH_TO_NODE_NUMBER:

```
procedure PATHNAME_TO_INODE (character string filename) returns inode
    inode instance i
    inode_number ← GENERALPATH_TO_INODE_NUMBER (filename)
    i ← INODE_NUMBER_TO_INODE (inode_number)
    if i.type = SYMBOLIC then
        i = GENERALPATH_TO_INODE_NUMBER (COERCE_TO_STRING (i.block_numbers))
    return i
```

The value returned by PATHNAME_TO_INODE is the contents of the inode for the file named by *filename*. The procedure first looks up the inode number for *filename*. Then, it looks up the the inode using INODE_NUMBER_TO_INODE. If the inode indicates that this file is a symbolic link, the procedure interprets the contents of data of the file as a path name and invokes GENERALPATH_TO_INODE_NUMBER again.

We now have two types of synonyms. A direct binding to an inode number is called a ***hard link***, to distinguish it from a soft link. Continuing an earlier example, a soft link to "Mail/inbox/new-assignment" would contain the string "Mail/inbox/new-assignment", rather than the inode number 481. A soft link is an example of an indirect name: it binds a name to another name in the same name space, while a hard link binds a name to an inode number, which is a name in a lower-layer name space. As a result, the soft link depends on the file name "Mail/inbox/new-assignment"; if the user changes the file's name or deletes the file, then "projects/assignment", the link, will end up as a dangling reference (Section 3.1.6 discusses dangling references). But because it links by name rather than by inode number, a soft link can point to a file on a different disk.

Recall that the UNIX system forbids cycles of hard links, so that it can use reference counts to detect when it is safe to reclaim the disk space for a file. However, you can still form cycles with soft links: a name deep down in the tree can, for example, name a directory high up in the tree. The resulting structure is no longer a directed acyclic graph, but a fully general naming network. Using soft links, a program can even invoke SYMLINK ("cycle", "cycle"), creating a synonym for a file name that doesn't have a file associated with it! If a process opens such a file, it will follow the link chain only a certain number of steps before reporting an error such as "Too many levels of soft links".

Soft links have another interesting behavior. Suppose that the working directory is "/Scholarly/programs/www" and that this working directory contains a symbolic link named "CSE499-web" to "/Scholarly/CSE499/www". The following calls

```
CHDIR ("CSE499-web")
CHDIR ("..")
```

leave the caller in "/Scholarly/CSE499" rather than back where the user started. The reason is that ".." is resolved in the new default context, "/Scholarly/CSE499/www", rather than what might have been the intended context, "/Scholarly/programs/www". This behavior may be desirable or not, but it is a direct consequence of the UNIX

**Table 2.3** The UNIX Naming Layers, with Details of the Naming Scheme of Each Layer

| Layer | Names | Values | Context | Name-Mapping Algorithm | |
|---|---|---|---|---|---|
| Symbolic link | Path names | Path names | The directory hierarchy | PATHNAME_TO_GENERAL_PATH | ↑ |
| Absolute path name | Absolute path names | Inode numbers | The root directory | GENERALPATH_TO_INODE_NUMBER | user-oriented names |
| Path name | Relative path names | Inode numbers | The working directory | PATH_TO_INODE_NUMBER | ↓ |
| File name | File names | Inode numbers | A directory | NAME_TO_INODE_NUMBER | machine-user interface |
| Inode number | Inode numbers | Inodes | The inode table | INODE_NUMBER_TO_INODE | ↑ |
| File | Index numbers | Block numbers | An inode | INDEX_TO_BLOCK_NUMBER | machine-oriented names |
| Block | Block numbers | Blocks | The disk drive | BLOCK_NUMBER_TO_BLOCK | ↓ |

naming semantics; the Plan 9 system has a different plan,* which is also explored in exercises 3.2 and 3.3.

In summary, much of the power of the UNIX object naming scheme comes from its layers of naming. Table 2.3 reprises Table 2.2, this time showing the name, value, context, and pseudocode procedure used at each layer interface. (Although we have examined each of the layers in this table, the algorithms we have demonstrated have in some cases bridged across layers in ways not suggested by the table.) The general design technique has been to introduce for each problem another layer of naming, an application of the principle *decouple modules with indirection*.

### 2.5.11 Implementing the File System API

In the process of describing how the UNIX file system is structured, we saw how it implements CHDIR, MKDIR, LINK, UNLINK, RENAME, SYMLINK, MOUNT, and UNMOUNT. We complete the description of the file system API by describing the implementation of OPEN, READ, WRITE, and CLOSE. Before describing their implementation, we describe what features they must support.

*Rob Pike. Lexical File Names in Plan 9 or Getting Dot-Dot Right. *Proceedings of the 2000 USENIX Technical Conference* (2000), San Diego, pages 85–92.

The file system allows users to control who has access to their files. An owner of a file can specify with what permissions other users can make accesses to the file. For example, the owner may specify that other users have permission only to read a file but not to write it. OPEN must check whether the caller has the appropriate permissions. As a sophistication, a file can be owned by a group of users. Chapter 11 [on-line] discusses security in detail, so we will skip the details here.

The file system records timestamps that capture the date and time of the last access, last modification to a file, and last change to a file's inode. This information is important for programs such as incremental backup, which must determine which files have changed since the last time backup ran. The file system procedures must update these values. For example, READ updates last access time, WRITE updates last modification time and change time, and LINK updates last change time.

OPEN returns a short name for a file, called a file descriptor (*fd*), which READ, WRITE, and CLOSE use to name the file. Each process starts with three open files: "standard in" (file descriptor 0), "standard out" (file descriptor 1), and "standard error" (file descriptor 2). A file descriptor may name a keyboard device, a display device, or a file on disk; a program doesn't need to know. This setup allows a designer to develop a program without having to worry about where the program's input is coming from and where the program's output is going to; the program just reads from file descriptor 0 and writes to file descriptor 1.

Several processes can use a file concurrently (e.g., several processes might write to the display device). If several processes open the same file, their READ and WRITE operations have their own file cursor for that file. If one process opens a file, and then passes the file descriptor for that file to another process, then the two processes share the cursor of the file. This latter case is common because in the UNIX system when one process (the *parent*) starts another process (the *child*), the child inherits all open file descriptors from the parent. This design allows the parent and child, for instance, to share a common output file correctly. If the child writes to the output file, for example, after the parent has written to it, the output of the child appears after the output of the parent because they share the cursor.

If one process has a file open and another process removes the last name pointing to that file, the inode isn't freed until the first process calls CLOSE.

To support these features, the inode is extended as follows:

```
structure inode
    integer block_numbers[N]    // the number of blocks that constitute the file
    integer size                // the size of the file in bytes
    integer type                // type of file: regular file, directory, symbolic link
    integer refcnt              // count of the number of names for this inode
    integer userid              // the user ID that owns this inode
    integer groupid             // the group ID that owns this inode
    integer mode                // inode's permissions
    integer atime               // time of last access (READ, WRITE, . . . )
    integer mtime               // time of last modification
    integer ctime               // time of last change of inode
```

To implement OPEN, READ, WRITE, and CLOSE, the file system keeps in memory several tables: one file table (*file_table*) and for each process a file descriptor table (*fd_table*). The file table records information for the files that processes have open (i.e., files for which OPEN was successful, but for which CLOSE hasn't been called yet). For each open file, this information includes the inode number of the file, its file cursor, and a reference count recording how many processes have the file open. The file descriptor table records for each file descriptor the index into the file table. Because a file's cursor is stored in the *file_table* instead of the *fd_table*, children can share the cursor for an inherited file with their parent.

With this information, OPEN is implemented as follows:

```
1   procedure OPEN (character string filename, flags, mode)
2       inode_number ← PATH_TO_INODE_NUMBER (filename, wd)
3       if inode_number = FAILURE and flags = O_CREATE then        // Create the file?
4           inode_number ← CREATE (filename, mode)                 // Yes, create it.
5       if inode_number = FAILURE then
6           return FAILURE
7       inode ← INODE_NUMBER_TO_INODE (inode_number)
8       if PERMITTED (inode, flags) then   // Does this user have the required permissions
9           file_index ← INSERT (file_table, inode_number)
10          fd ← FIND_UNUSED_ENTRY (fd_table)   // Find entry in file descriptor table
11          fd_table[fd] ← file_index           // Record file index for file descriptor
12          return fd                           // Return fd
13      else return FAILURE                     // No, return a failure
```

Line *2* finds the inode number for the file *filename*. If the file doesn't exist, but the caller wants to create the file as indicated by the flag O_CREATE (line *3*), OPEN calls CREATE, which allocates an inode, initializes it, and returns its inode number (line *4*). If the file doesn't exist (even after trying to create it), OPEN returns a value indicating a failure (line *6*). Line *7* locates the inode. Line *8* uses the information in the inode to check if the caller has permission to open the file; the check is described in detail in Section 11.6.3.4 [on-line]. If so, line *9* creates a new entry for the inode number in the file table and sets the entry's file cursor to zero and reference count to 1. Line *10* finds the first unused file descriptor, records its file index, and returns the file descriptor to the caller (lines *10* through *12*). Otherwise, it returns a value indicating a failure (line *13*).

If a process starts another process, the child process inherits the open file descriptors of the parent. That is, the information in every used entry in the parent's *fd_table* is copied to the same numbered entry in the child's *fd_table*. As a result, the parent and child entries in the *fd_table* will point to the same entry in the *file_table*, resulting in the cursor being shared between parent and child.

READ is implemented as follows:

```
1   procedure READ (fd, character array reference buf, n)
2       file_index ← fd_table[fd]
3       cursor ← file_table[file_index].cursor
4       inode ← INODE_NUMBER_TO_INODE (file_table[file_index].inode_number)
5       m = MINIMUM (inode.size − cursor, n)
6       atime of inode ← NOW ()
7       if m = 0 then return END_OF_FILE
8       for i from 0 to m − 1 do {
9           b ← INODE_NUMBER_TO_BLOCK (i, inode_number)
10          COPY (b, buf, MINIMUM (m − i, BLOCKSIZE))
11          i ← i + MINIMUM (m − i, BLOCKSIZE)
12      file_table[file_index].cursor ← cursor + m
13      return m
```

Lines *2* and *3* use the file index to find the cursor for the file. Line *4* locates the inode. Line *5* and *6* compute how many bytes READ can read and updates the last access time. If there are no bytes left in the file, READ returns a value indicating end of file. Lines *8* through *12* copy the bytes from the file's blocks into the caller's *buf*. Line *13* updates the cursor.

One could design a more sophisticated naming scheme for READ that, for example, allowed naming by keywords rather than by offsets. Database systems typically implement such naming schemes by representing the data as structured records that are indexed by keywords. But in order to keep its design simple, the UNIX file system restricts its representation of a file to a linear array of bytes.

The implementation of WRITE is similar to READ. The major differences are that it copies *buf* into the blocks of the inode, allocating new blocks as necessary, and that it updates the inode's *size* and *mtime*.

CLOSE frees the entry in the file descriptor table and decreases the reference count in entry in the file table. If no other processes are sharing this entry (i.e., the reference count has reached zero), it also frees the entry in the file table. If there are no other entries in the file table using this file and the reference count in the file's inode has reached zero (because another process unlinked it), then CLOSE frees the inode.

Like RENAME, some of these operations require several disk writes to complete. If the file system fails (e.g., because the power goes off) in the middle of one of the operations, then some of the disk writes may have completed and some may not. Such a failure can cause inconsistencies among the on-disk data structures. For example, the on-disk free list may show that a block is allocated, but no on-disk inode records that block in its index. If nothing is done about this inconsistency, then that block is effectively lost. Problem set *8* explores this problem and a simple, special-case solution. Chapter 9 [on-line] explores systematic solutions.

Version 6 (and all modern implementations) maintain an in-memory cache of recently used disk blocks. When the file system needs a block, it first checks the cache for the block. If the block is present, it uses the block from the cache; otherwise, it reads it from the storage device. With the cache, even if the file system needs to read a particular block several times, it reads that block from the storage device only once. Since reading from a disk device is often an expensive operation, the cache can improve the performance of the file system substantially. Chapter 6 discusses the implementation of caches in detail and how they can be used to improve the performance of a file system.

Similarly, to achieve high performance on operations that modify a file (e.g., WRITE), the file system will update the file's blocks in the cache, but will not force the file's modified inode and blocks to the storage device immediately. The file system delays the writes until later so that if a block is updated several times, it will write the block only once. Thus, it can coalesce many updates in one write (see Section 6.1.8).

If a process wants to ensure that the results of a write and inode changes are propagated to the device that stores the file system, it must call FSYNC; the UNIX specification requires that if an invocation of FSYNC for a file returns, all changes to the file must have been written to the storage device.

### 2.5.12 The Shell and Implied Contexts, Search Paths, and Name Discovery

Using the file system API, the UNIX system implements programs for users to manipulate files and name spaces. These programs include text editors (such as `ed`, `vi`, and `emacs`), `rm` (to remove a file), `ls` (to list a directory's content), `mkdir` (to make a new directory), `rmdir` (to remove a directory), `ln` (to make link names), `cd` (to change the working directory), and `find` (to search for a file in a directory tree).

One of the more interesting UNIX programs is its command interpreter, known as the "shell". The shell illustrates a number of other UNIX naming schemes. Say a user wants to compile the C source file named "x.c". The UNIX convention is to overload a file name by appending a suffix indicating the type of the file, such as ".c" for C source files. (A full discussion of overloading can be found in Section 3.1.2.) The user types this command to the shell:

```
cc x.c
```

This command consists of two names: the name of a program (the compiler "cc") and the name of a file containing source code ("x.c") for the compiler to compile. The first thing the shell must do is find the program we want to run, "cc". To do that, the UNIX command interpreter uses a default context reference contained in an environment variable named PATH. That environment variable contains a list of contexts (in this case directories) in which to perform a multiple lookup for the thing named "cc". Assuming the lookup is successful, the shell launches the program, calling it with the argument "x.c".

The first thing the compiler does is try to resolve the name "x.c". This time it uses a different default context reference: the working directory. Once the compilation

is underway, the file "x.c" may contain references to other named files, for example, statements such as

```
#include <stdio.h>
```

This statement tells the compiler to include all definitions in the file "stdio.h" in the file "x.c". To resolve "stdio.h", the compiler needs a context in which to resolve it. For this purpose, the compiler consults another variable (typically passed as an argument when invoking the compiler), which contains a default context to be used as a search path where include files may be found. The variables used by the shell and by the compiler each consist of a series of path names to be used as the basis for an ordered multiple lookup just as was described in Section 2.2.4.

Many other UNIX programs, such as the documentation package, `man`, also do multiple lookups for files using search paths found in environment variables.

The shell resolves names for commands using the PATH variable, but sometimes it is convenient to be able to say "I want to run the program located in the current working directory". For example, a user may be developing a new version of the C compiler, which is also called "cc". If the user types `"cc"`, the shell will look up the C compiler using the PATH variable and find the standard one instead of the new one in the current working directory.

For these cases, users can type the following command:

```
./cc x.c
```

which bypasses the PATH variable and invokes the program named "cc" in the current working directory (".").

Of course, the user could insert "." at the beginning of the PATH variable, so that all programs in the user's working directory will take precedence over the corresponding standard program. That practice, however, may create some surprises. Suppose "." is first entry in the PATH variable, and a user issues the following command sequence to the shell:

```
cd /usr/potluck
ls
```

intending to list the contents of the directory named `potluck`. If that directory contained a program named `ls` that did something different from the standard `ls` command, something surprising might happen (e.g., the program named `ls` could remove all private files)! For this reason, it is not a good idea to include names that are context-dependent, such as "." or ".." in a search path. It is better to include the absolute path name of the desired directory to the front of PATH.

Another command interpreter extension is that names can be descriptive rather than simple names. For example, the descriptive name "*.c", matches all file names that end with ".c". To provide this extension, the command interpreter transforms the single argument into a list of arguments (with the help of a more complicated lookup operation on the entries in the context) before it calls the specified command program. In the UNIX shell, users can use full-blown regular expressions in descriptive names.

As a final note, in practice, the UNIX object naming space has quite a bit of conventional structure. In particular, there are several directories with well-known names. For example, "/bin" names programs, "/etc" names configuration files, "/dev" names input/output devices, and "/usr" (rather than the root itself) names user directories. Over time these conventions have become so ingrained both in programmers' minds and in programs that much UNIX software will not install correctly, and a UNIX wizard will become badly confused, when confronted with a system that does not follow these conventions.

### 2.5.13 Suggestions for Further Reading

For a detailed description of a more modern UNIX operating system, see the book describing the BSD operating system [Suggestions for Further Reading 1.3.4]. A descendant of the original UNIX system is Plan 9 [Suggestions for Further Reading 3.2.2], which contains a number of novel naming abstractions, some of which are finding their way back into newer UNIX implementations. A rich literature exists describing file system implementations and their trade-offs. A good starting point are the papers on FFS [Suggestions for Further Reading 6.3.2], LFS [Suggestions for Further Reading 9.3.1], and soft updates [Suggestions for Further Reading 6.3.3].

## EXERCISES

**2.1** Ben Bitdiddle has accepted a job with the telephone company and has been asked to implement call forwarding. He has been pondering what to do if someone forwards calls to some number and then the owner of that number forwards calls to a third number. So far, Ben has thought of two possibilities for his implementation:

**a.** *Follow me.* Bob is going to a party at Mary's home for the evening, so he forwards his telephone to Mary. Ann is baby-sitting for Bob, so she forwards her telephone to Bob. Jim calls Ann's number, Bob's telephone rings, and Ann answers it.

**b.** *Delegation.* Bob is going to a party at Mary's home for the evening, so he forwards his telephone to Mary. Ann is gone for the week and has forwarded her telephone to Bob so that he can take her calls. Jim calls Ann's number, Mary's telephone rings, and Mary hands the phone to Bob to take the call.

**2.1a** Using the terminology of the naming section of this chapter, explain these two possibilities.

**2.1b** What might go wrong if Bob has already forwarded his telephone to Mary before Ann forwards her telephone to him?

**2.1c** The telephone company usually provides *Delegation* rather than *Follow me*. Why?

**2.2** Consider the part of the file system naming hierarchy illustrated in the following:

You have been handed the following path name:

`/projects/systems/exercises/Ex.2.2`

and you are about to resolve the third component of that path name, the name `exercises`.

**2.2a** In the path name and in the figure, identify the context that you should use for that resolution and the context reference that allows locating that context.

**2.2b** Which of the terms *default*, *explicit*, *built-in*, *per-object*, and *per-name* apply to this context reference?

*1995-2-1a*

**2.3** One way to speed up the resolving of names is to implement a cache that remembers recently looked-up {name, object} pairs.

**2.3a** What problems do synonyms pose for cache designers, as compared with caches that don't support synonyms?

*1994-2-3*

**2.3b** Propose a way of solving the problems if every object has a unique ID.

*1994-2-3a*

**2.4** Louis Reasoner has become concerned about the efficiency of the search rule implementation in the Eunuchs system (an emasculated version of the UNIX system). He proposes to add a *referenced object table* (ROT), which the system

will maintain for each session of each user, set to be empty when the user logs in. Whenever the system resolves a name through of use a search path, it makes an entry in the ROT consisting of the name and the path name of that object. The "already referenced" search rule simply searches the ROT to determine if the name in question appears there. If it finds a match, then the resolver will use the associated path name from the ROT. Louis proposes to always use the "already referenced" rule first, followed by the traditional search path mechanism. He claims that the user will detect no difference, except for faster name resolution. Is Louis right?

*1985-2-2*

**2.5** The last line of Figure 2.4 names three Web browsers as examples of interpreters. Explain how a Web browser is an interpreter by identifying its instruction reference, its repertoire, and its environment reference.

*2009-0-1*

**Additional exercises relating to Chapter 2 can be found in the problem sets beginning on page 425.**

CHAPTER

# The Design of Naming Schemes

# 3

## CHAPTER CONTENTS

**Overview** ..... **115**
**3.1 Considerations in the design of naming schemes** ..... **116**
3.1.1 Modular Sharing ..... 116
3.1.2 Metadata and Name Overloading ..... 120
3.1.3 Addresses: Names that Locate Objects ..... 122
3.1.4 Generating Unique Names ..... 124
3.1.5 Intended Audience and User-Friendly Names ..... 127
3.1.6 Relative Lifetimes of Names, Values, and Bindings ..... 129
3.1.7 Looking Back and Ahead: Names are a Basic System Component ..... 131
**3.2 Case Study: The Uniform Resource Locator (URL)** ..... **132**
3.2.1 Surfing as a Referential Experience; Name Discovery ..... 132
3.2.2 Interpretation of the URL ..... 133
3.2.3 URL Case Sensitivity ..... 134
3.2.4 Wrong Context References for a Partial URL ..... 135
3.2.5 Overloading of Names in URLs ..... 137
**3.3 War stories: Pathologies in the Use of Names** ..... **138**
3.3.1 A Name Collision Eliminates Smiling Faces ..... 139
3.3.2 Fragile Names from Overloading, and a Market Solution ..... 139
3.3.3 More Fragile Names from Overloading, with Market Disruption ..... 140
3.3.4 Case-Sensitivity in User-Friendly Names ..... 141
3.3.5 Running Out of Telephone Numbers ..... 142
**Exercises** ..... **144**

## OVERVIEW

In the previous chapter we developed an abstract model of naming schemes. When the time comes to design a practical naming scheme, many engineering considerations—constraints, additional requirements or desiderata, and environmental pressures—shape

**Principles of Computer System Design: An Introduction**
Copyright © 2009 by Jerome H. Saltzer and M. Frans Kaashoek. All rights of reproduction in any form reserved.
DOI: 10.1016/B978-0-12-374957-4.00011-6

the design. One of the main ways in which users interact with a computer system is through names, and the quality of the user experience can be greatly influenced by the quality of the system's naming schemes. Similarly, since names are the glue that connects modules, the properties of the naming schemes can significantly affect the impact of modularity on a system.

This chapter explores the engineering considerations involved in designing naming schemes. The main text introduces a wide range of naming considerations that affect modularity and usability. A case study of the World Wide Web Uniform Resource Locator (URL) illustrates both the naming model and some problems that arise in the design of naming schemes. Finally, a war stories section explores some pathological problems of real naming schemes.

## 3.1 CONSIDERATIONS IN THE DESIGN OF NAMING SCHEMES

We begin with a discussion of an interaction between naming and modularity.

### 3.1.1 Modular Sharing

Connecting modules by name provides great flexibility, but it introduces a hazard: the designer sometimes has to deal with preexisting names, perhaps chosen by someone else over whom the designer has no control. This hazard can arise whenever modules are designed independently. If, in order to use a module, the designer must know about and avoid the names used within that module for its components, we have failed to achieve one of the primary goals of modularity, called *modular sharing*. Modular sharing means that one can use a shared module by name without knowing the names of the modules it uses.

Lack of modular sharing shows up in the form of *name conflict*, in which for some reason two or more different values compete for the binding of the same name in the same context. Name conflict can arise when integrating two (or more) independently conceived sets of programs, sets of documents, file systems, databases, or indeed any collection of components that use the same naming scheme for internal interconnection as for integration. Name conflict can be a serious problem because fixing it requires changing some of the uses of the conflicting names. Making such changes can be awkward or difficult, for the authors of the original subsystems are not necessarily available to help locate, understand, and change the uses of the conflicting names.

The obvious way to implement modular sharing is to provide each subsystem with its own naming context, and then work out some method of cross-reference between the contexts. Getting the cross-reference to work properly turns out to be the challenge.

Consider, for example, the two sets of programs shown in Figure 3.1—a word processor and a spelling checker—each of which comprises modules linked by name and each of which has a component named INITIALIZE. The designer of the procedure WORD_PROCESSOR wants to use SPELL_CHECK as a component. If the designer tries to combine the two sets of programs by simply binding all of their names in one naming

context, as in the figure (where the arrows show the binding of each name), there are two modules competing for binding of the name INITIALIZE. We have a name conflict.

So the designer instead tries to create a separate context for each set of programs, as in Figure 3.2. That step by itself doesn't completely address the problem because the program interpreter now needs some rule to determine which context to use

**FIGURE 3.1**

Too-simple integration of two independently written sets of programs by just merging their contexts. Procedure WORD_PROCESSOR calls SPELL_CHECK, but SPELL_CHECK has a component that has the same name as a component of WORD_PROCESSOR. No single set of bindings can do the right thing.

**FIGURE 3.2**

Integration of the same two programs but using separate contexts. Having a separate context for SPELL_CHECK eliminates the name conflict, but the program interpreter now needs some basis for choosing one context over the other.

for each use of a name. Suppose, for example, it is running WORD_PROCESSOR, and it encounters the name INITIALIZE. How does it know that it should resolve this name in the context of WORD_PROCESSOR rather than the context of SPELL_CHECK?

Following the naming model of Chapter 2, and the example of the e-mail system, a direct solution to this problem would be to add a binding for SPELL_CHECK in the WORD_PROCESSOR context and attach to every module an explicit context reference, as in Figure 3.3. This addition would require tinkering with the representation of the modules, an alternative that may not be convenient or even not allowed if some of the modules belong to someone else.

Figure 3.4 suggests another possibility: augment the program interpreter to keep track of the context in which it originally found each program. The program interpreter would use that context for resolving all names found in that program. Then, to allow the word processor to call the spell checker by name, place a binding for SPELL_CHECK in the WORD_PROCESSOR context as shown by the solid arrow numbered *1* in that figure. (imagine that the contexts are now file system directories).

That extra binding creates a subtle problem that may produce a later surprise. Because the program interpreter found SPELL_CHECK in the word processor's context, its context selection rule tells it (incorrectly) to use that context for the names it finds inside of SPELL_CHECK, so SPELL_CHECK will call the wrong version of INITIALIZE. A solution is to place an indirect name (the dashed arrow numbered *2* in Figure 3.2) in the word processor's context, bound to the name of SPELL_CHECK in SPELL_CHECK's own context. Then, the interpreter (assuming it keeps track of the context where it actually found each program) will correctly resolve names found in both groups of programs.

**FIGURE 3.3**

Modular sharing with explicit context references. The small circles added to each program module are context references that tell the name interpreter which context to use for names found in that module.

**FIGURE 3.4**

Integration with the help of separate contexts. Having a separate context for spell-check eliminates the name conflict, but the program interpreter still needs some basis for choosing one context over the other. Adding the solid arrow numbered *1* doesn't quite work, but the dashed arrow numbered *2*, an indirect name, does.

Keeping track of contexts and using indirect references (perhaps by using file system directories as contexts) is commonplace, but it is a bit *ad hoc*. Another, more graceful, way of attaching a context reference to an object without modifying its representation is to associate the name of an object not directly with the object itself but instead with a structure that consists of the original object plus its context reference. Some programming languages implement just such a structure for procedure definitions, known as a "closure", which connects each procedure definition with the naming context in which it was defined. Programming languages that use static scope and closures provide a much more systematic scheme for modular sharing of named objects within the different parts of a large application program, but comparable mechanisms are rarely found* in file systems or in merging applications such as the word processing and spell-checking systems of the previous example. One reason for the difference is that a program usually contains many references to lots of named objects, so it is important to be well organized. On the other hand, merging applications involves a small number of large components with only a few cross-references, so *ad hoc* schemes for modular sharing may seem to suffice.

*An ambitious attempt to design a naming architecture with all of these concepts wired into the hardware was undertaken by IBM in the 1970s, documented in a technical report by George Radin and Peter R. Schneider: *An architecture for an extended machine with protected addressing*, IBM Poughkeepsie Laboratory Technical Report TR 00.2757, May, 1976. Although the architecture itself never made it to the market, some of the ideas later appeared in the IBM System/38 and AS/400 computer systems.

### 3.1.2 Metadata and Name Overloading

The name of an object and the context reference that should be associated with it are two examples of a class of information called *metadata*—information that is useful to know about an object but that cannot be found inside the object itself (or if it is inside may not be easy to find). A library bibliographic record is a collection of metadata, including title, author, publisher, publication date, date of acquisition, and shelf location of a book, all in a standard format. Libraries have a lot of experience in dealing with metadata, but failure to systematically organize metadata is a design shortcoming frequently encountered in computer systems.

Some common examples of metadata associated with an object in a computer system are a user-friendly name, a unique identifier, the type of the object (executable program, word processing text, video stream, etc.), the dates it was created, last modified, and last backed up, the location of backup copies, the name of its owner, the program that created it, a cryptographic quality checksum (known as a *witness*—see Sidebar 7.1 [on-line]) to verify its integrity, the list of names of who is permitted to read or update the object, and the physical location of the representation of the object. A common, though not universal, property of metadata is that it is information about an object that may be changed without changing the object itself.

One strategy for maintaining metadata in a file system is to reserve storage for the metadata in the same file system structure that keeps track of the physical location of the file and to provide methods for reading and updating the metadata. This strategy is attractive because it allows applications that do not care about the metadata to easily ignore it. Thus, a compiler can read an input file without having to explicitly identify and ignore the file owner's name or the date on which the file was last backed up, whereas an automatic backup application can use the metadata access method to check those two fields. The UNIX file system, described in Section 2.5.1, uses this strategy by storing metadata in inodes.

Computer file systems nearly always provide for management of specialized metadata about each file such as its physical location, size, and access permissions, but they rarely have any provision for user-supplied metadata other than the file name. Because of this limitation, it is common to discover that file names are *overloaded* with metadata that has little or nothing to do with the use of the name as a reference.* The naming scheme may even impose syntax rules on allowable names to support overloading with metadata. A typical example of name overloading is a file name that ends with an extension that identifies the type of the file, such as text, word processing document, spreadsheet, binary application program, or movie. Other examples are illustrated in Figure 3.5. A physical address is another example of name overloading that is so common that the next section explores its special properties. Names that have no overloading whatever are known as *pure names*. The only operations it makes sense to apply to a pure name are COMPARE, RESOLVE, BIND, and UNBIND; one cannot

*Use of the word "overloading" to describe names that carry metadata is similar to, but distinct from, the use of the same word to describe symbols that stand for several different operators in a programming language.

| Name | Some of the things that overload this name |
|---|---|
| solutions.txt | solutions = file content; txt = file format |
| solutions.txt.backup 2 | backup 2 = this is the second backup copy |
| businessplan 10-26-2007.doc | 10-26-2007 = when file was created |
| executive summary v4 | v4 = version number |
| image079.large.jpg | 079 = where file fits in a sequence; large = image size |
| /disk-07/archives/Alice/ | disk-07 = physical device that holds file; Alice = user id |
| OSX.10.5.2.dmg | OSX = program name; 10.5.2 = program version |
| IPCC_report_TR-4 | IPCC = author; TR-4 = technical report series identifier |
| cse.pedantic.edu | cse = department name; pedantic = university name; edu = registrar name |
| ax539&ttiejh!90rrwl | no (apparent) overloading |

**FIGURE 3.5**

Some examples of overloaded names and a pure name.

extract metadata from it by applying a parsing operation. An overloaded name, on the other hand, can be used in two distinct ways:

1. As an identifier, using COMPARE, RESOLVE, BIND, and UNBIND.
2. As a source from which to extract the overloaded metadata.

Path names are especially susceptible to overloading. Because they describe a path through a series of contexts, the temptation is to overload them with information about the route to the physical location of the object.

Overloading of a name can be harmless, but it can also lead to violation of the principles of modular design and abstraction. The problem usually shows up in the form of a *fragile name*. Name fragility appears, for example, when it is necessary to change the name of a file that moves to a new physical location, even though the identity and content of the file have not changed. For example, suppose that a library program that calculates square roots and that happens to be stored on `disk05` is named `/disk05/library/sqrt`. If `disk05` later becomes too full and that library has to be moved to `disk06`, the path name of the program changes to `/disk06/library/sqrt`, and someone has to track down and modify every use of the old name. Name fragility is one of the reasons that World Wide Web addresses stop working. The case study in Section 3.2 explores that problem in more detail.

The general version of this observation is that overloading creates a tension between the goal of keeping names unchanged and the need to modify the overloaded information. Typically, a module that uses a name needs the name to remain unchanged for at least as long as that module exists. For this reason, overloading must be used with caution and with understanding of how the name will be used.

Finally, in a modular system, an overloaded name may be passed through several modules before reaching the module that actually knows how to interpret the overloading. A name is said to be *opaque* to a module if the name has no overloading

that the module knows how to interpret. A pure name can be thought of as being opaque to all modules except RESOLVE.

There are also more subtle forms of metadata overloading. Overloading can be less obvious if the user's mind, rather than the computer system, performs the metadata extraction. For example, in the Internet host name "CityClerk.Reston.VA.US", the identifier of the context, "Reston.VA.US", is also recognizable as the identifier of a real place, a town named Reston, Virginia, in the United States. Each component of this name is being used to name two different real-world things: the name "Reston" identifies both a town and a table of name/value pairs that acts as a context in which the name of a municipal department may be looked up. Because it has mnemonic value, people find this reuse by overloading helpful—assuming that it is done accurately and consistently. (On the other hand, if someone names a World Wide Web service in Chicago "SaltLakeCity.net" people seeing that name are likely to assume—incorrectly—that it is actually located in Salt Lake City.)

### 3.1.3 Addresses: Names that Locate Objects

In a computer system, an *address* is the name of a physical location or of a virtual location that maps to a physical location. Computer systems are constructed of real physical objects, so they abound in examples of addresses: register numbers, physical and virtual memory addresses, processor numbers, disk sector numbers, removable media volume numbers, I/O channel numbers, communication link identifiers, network attachment point addresses, pixel positions on a display—the list seems endless.

Addresses are not pure names. The thing that characterizes an address is that it is overloaded in such a way that parsing the address provides a guide to the location of the named object in some virtual or real coordinate system. As with other overloaded names, addresses can be used in two ways, in this case:

1. As an identifier with the usual naming operations.
2. As a locator.

Thus, "Leonardo da Vinci" is an identifier that was once bound to a physical person and is now bound to the memory of that Leonardo. This identifier could have been used in comparisons to avoid confusion with Leonardo di Pisa when both of them were visiting Florence.* Today, the identifier helps avoid mixing up their writings. At the same time, "Leonardo da Vinci" is also a locator; it indicates that if you want to examine the birth record of that Leonardo, you should look in the archives of the town named Vinci.

Since access to many physical devices is geometric, addresses are often chosen from compact sets of integers in such a way that address adjacency corresponds to physical adjacency, and arithmetic operations such as "add 1" or subtracting one

*Actually, they could not have both visited Florence at the same time. The mathematician Leonardo di Pisa (also known as Fibonacci) lived three centuries before the artist Leonardo da Vinci.

address from another have a useful, physical meaning. For example, a seek arm finds track #1079 on a magnetic disk by counting the number of tracks it passes, and a disk arm scheduler looks at differences in track addresses to decide the best order in which to perform seeks. For another example, a memory chip contains an array of bits, each of which has a unique integer address. When a read or write request for a particular address arrives at the chip, the chip routes individual bits of that address to selectors that guide the flow of information to and from the intended bit of storage.

Sometimes it is inappropriate to apply arithmetic operations to addresses, even when they are chosen from compact sets of integers. For example, telephone numbers (known technically as "directory numbers") are integers that are overloaded with routing information in their area and exchange codes, but there is no necessary physical adjacency of two area codes that have consecutive addresses. Similarly, there is no necessary physical adjacency of two telephones that have consecutive directory numbers. (In decades past, there was physical adjacency of consecutive directory numbers inside the telephone switching equipment, but that adjacency was so constraining that it was abandoned by introducing a layer of indirection as part of the telephone switch gear.)

The overloaded location information found in addresses can cause name fragility. When an object moves, its address, and thus its name, changes. For this reason, system designers usually follow the example of telephone switching systems: they apply the design principle ***decouple modules with indirection*** to hide addresses. Adding a layer of indirection provides a binding from some externally visible, but stable, name to an address that can easily be changed when the object moves to a new location. Ideally, addresses never need to be exposed above the layer of interpretation that directly manipulates the objects. Thus, for example, the user of a personal computer that has a communication port may be able to write programs using a name such as `COM1` for the port, rather than a hexadecimal address such as $4D7C_{hex}$, which may change to $4D7E_{hex}$ when the port card is replaced.

When a name must be changed because it is being used as an address that is not hidden by a layer of indirection, things become more complicated and they may start to go wrong. At least four alternatives have been used in naming schemes:

- Search for and change all uses of the old address. At best, this alternative is a nuisance. In a large or geographically distributed system, it can be quite painful. The search typically misses some uses of the name, and those users, on their next attempted use of the name, either receive a puzzling not-found response for an object that still exists or, worse, discover that the old address now leads to a different object. For that reason, this scheme may be combined with the next one.

- Plan that users of the name must undertake an attribute-based search for the object if they receive a not-found response or detect that the address has been rebound to a different object. If the search finds the correct object, its new address can replace the old one, at least for that user. A different user will have to do another search.

- If the naming scheme provides either synonyms or indirect names, add bindings so that both the old and new addresses continue to identify the object. If addresses are scarce and must be reused, this alternative is not attractive.
- If the name is bound to an active agent, such as a post office service that accepts mail, place an active intermediary, such as a mail forwarder, at the old address.

None of these alternatives may be attractive. The better method is nearly always for the designer to hide addresses behind a layer of indirection. Section 3.3.2 provides an example of this problem and the solution using indirection. Exercise 2.1 explores some interesting indirection-related naming problems in the telephone system related to the feature known as call forwarding.

One might suggest avoiding the name fragility problem by using only pure names, that is, names with no overloading. The trouble with that approach is that it makes it difficult to locate the object. When the lowest-layer name carries no overloaded addressing metadata, the only way to resolve that name to a physical object is by searching through an enumeration of all the names. If the context is small and local, that technique may be acceptable. If the context is universal and widely distributed, name resolution becomes quite problematic. Consider, for example, the problem of locating a railway car, given only a unique serial number painted on its side. If for some reason you know that the car is on a particular siding, searching may be straightforward, but if the car can be anywhere on the continent, searching is a daunting prospect.

### 3.1.4 Generating Unique Names

In a unique identifier name space, some protocol is needed to ensure that all of the names actually are unique. The usual approach is for the naming scheme to *generate* a name for a newly created object, rather than relying on the creator to propose a unique name. One simple scheme for generating unique names is to dole out consecutive integers or sufficiently fine timestamp values. Sidebar 3.1 shows an example. Another scheme for generating unique names is to choose them at random from a sufficiently large name space. The idea is to make the probability of accidentally choosing the same name twice (a form of name conflict called a *collision*) negligibly small. The trouble with this scheme is that it is hard for a finite-state machine to create genuine randomness, so the chance of accidentally creating a name collision may be much higher than one would predict from the size of the name space. One must apply careful design, for example, by using a high-quality pseudorandom number generator and seeding it with a unique input such as a timestamp that was created when the system started. An example of such a design is the naming system used inside the Apollo DOMAIN operating system, which provided unique identifiers for all objects across a local-area network to provide a high-degree of transparency to users of the system; for more detail, see Suggestions for Further Reading 3.2.1.

Yet another way to avoid generated name collisions, for an object that has a binary representation and that already exists when it is being named, is to choose as its unique name the contents of the object. This approach assigns two objects with the same content the same name. In some applications, however, that may be a feature—it provides

**Sidebar 3.1 Generating a Unique Name from a Timestamp** Some banking systems generate a unique character-string name for each transaction. A typical name generation scheme is to read a digital clock to obtain a timestamp and convert the timestamp to a character string. A typical timestamp might contain the number of microseconds since January 1, 2000. A 50-bit timestamp would repeat after about 35 years, which may be sufficient for the bank's purpose. Suppose the timestamp at 1:35 P.M. on April 1, 2007, is

00010111110110101101001100111001100010111010011001

To convert this string of bits to a character string, divide it into five-bit chunks and interpret each chunk as an index into a table of 32 alphanumeric characters. The five-bit chunks are:

00010–11111–01101–01101–00110–01110–01100–01011–10100–11001

Next, reinterpret the chunks as index numbers:

2 31 13 13 6 16 12 11 20 25

Then look those numbers up in this table of 32 alphanumeric characters:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| B | C | D | F | G | H | J | K | L | M | N | P | Q | R | S | T | U | V | W | X | Y | Z | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 |

The result is the 10-character unique name "D9RRJ-UQTYP". You may have seen similar unique names in transactions performed with an on-line banking system.

a way of discovering the existence of unwanted duplicate copies. That name is likely to be fairly long, so a more practical approach is to use as the name a shorter version of its contents, known as a *hash*. For example, one might run the contents of a stored file through a cryptographic transformation function whose output is a bit string of modest, fixed length, and use that bit string as the name. One version of the Secure Hash Algorithm (SHA, described in Sidebar 11.8 [on-line]) produces, for any size of input, an output that is 160 bits in length. If the transforming function is of sufficiently high quality, two different files will almost certainly end up with different names.

The main problem with any naming scheme that is based on the contents of the named object is that the name is overloaded. When someone modifies an object whose name was constructed from its original contents, the question that arises is whether to change its name. This question does not come up in preservation storage systems that do not allow objects to be modified, so hash-generated unique names are sometimes used in those systems.

Unique identifiers and generated names can also be used in places other than unique identifier name spaces. For example, when a program needs a name for a temporary file, it may assign a generated name and place the file in the user's working directory. In this case, the design challenge for the name generator is to come up with an algorithm that will not collide with the names of already existing names chosen

by people or generated by other automated name generators. Section 3.3.1 gives an example of a system that failed to meet this challenge.

Providing unique names in a large, geographically distributed system requires careful design. One approach is to create a hierarchical naming scheme. This idea takes advantage of an important feature of hierarchy: delegation. For example, a goal of the Internet is to allow creation of several hundred million different, unique names in a universal name space for attachment points for computers. If one tried to meet that goal by having someone at the International Telecommunications Union coordinating name assignment, the immense number of name assignments would almost certainly lead to long delays as well as mistakes in the form of accidental name collisions. Instead, some central authority assigns the name "edu" or "uk" and delegates the responsibility for naming things ending with that suffix to someone else—in the case of "edu", a specialist in assigning university names. That specialist accepts requests from educational institutions and, for example, assigns the name "pedantic" and thereby delegates the responsibility for names ending with the suffix ".pedantic.edu" to the Pedantic University network staff. That staff assigns the name "cse" to the Computer Science and Engineering Department, further delegating responsibility for names ending with the suffix ".cse.pedantic.edu" to someone in that department. The network manager inside the department can, with the help of a list posted on the wall or a small on-line database, assign a name such as "ginger" that is locally unique and at the same time can be confident that the fully qualified name "ginger.cse.pedantic.edu" is also globally unique.

A different example of a unique identifier name space is the addressing plan for the commercial Ethernet. Every Ethernet interface has a unique 48-bit *media access control* (MAC) *address*, typically set into the hardware by the manufacturer. To allow this assignment to be made uniquely, but without a single central registry for the whole world, there is a shallow hierarchy of MAC addresses. A standards-setting authority allocates to each Ethernet interface manufacturer a block of MAC addresses, all of which start with the same prefix. The manufacturer is then free to allocate MAC addresses within that block in any way that is convenient. If a manufacturer uses up all the MAC addresses in a block, it applies to the central authority for another block, which may have a prefix that has no relation to the previous prefix used by that same manufacturer.

One consequence of this strategy, especially noticeable in a large network, is that the MAC address of an Ethernet interface does not provide any overloading information that is useful for physically locating the interface card. Even though the MAC address is assigned hierarchically, the hierarchy is used only to delegate and thus decentralize address assignment, and it has no assured relation to any property (such as the physical place where the card attaches to the network) that would help locate it. Just as in locating a railway car knowing only its unique identifier, resolving a MAC address to the particular physical device that carries it is difficult unless one already has a good idea where to start looking.

People struggling to figure out how to tie a software license to a particular computer sometimes propose to associate the license with the Ethernet MAC address of that computer because that address is globally unique. Apart from the problem that some computers have no Ethernet interface and others have more than one, a trouble

with this approach is that if an Ethernet interface card on the computer fails and needs to be replaced, the new card will have a different MAC address, even though the location of the system, the software, and its owner are unchanged. Furthermore, if the card that failed is later repaired and reinstalled in another system, that other system will now have the MAC address that was previously associated with the first system. The MAC address is thus properly viewed only as the unique name of a specific hardware component, not of the system in which it is embedded.

Deciding what constitutes the unique identity of a system that is constructed of replaceable components is ultimately a convention that requires an arbitrary choice by the designer of the naming scheme. This choice is similar to the question of establishing the identity of wooden ships. If, over the course of 300 years, every piece of wood in the ship has been replaced, is it still the same ship? Apparently, ship registries say "yes". They do not associate the name of the ship with any single component; the name is instead associated with the ship as a whole. Answering this identity question can clarify which of the three meanings of the COMPARE operation that was discussed in Section 2.2.5 is most appropriate for a particular design.

### 3.1.5 Intended Audience and User-Friendly Names

Some naming schemes are intended to be used by people. Names in such a name space are typically user-chosen and user-friendly strings of characters with mnemonic value such as "economics report", "shopping list", or "Joe.Smith" and are widely used as names of files and e-mailboxes. Ambiguity (that is, non-uniqueness) in resolving user-friendly names may be acceptable because in interactive systems the person using the name can be asked to resolve the ambiguity.

Other naming schemes are intended primarily for use by machines. In these schemes, the names need not have mnemonic value, so they are typically integers, often of fixed width designed to fit into a register and to allow fast and unambiguous resolution. Memory addresses and disk sector addresses are examples. Sometimes the term *identifier* is used for a name that is not intended to be intelligible to people, but this usage is by no means universal. Names intended for use by machines are usually chosen mechanically.

When a name is intended to be user-friendly, a tension arises between a need for it to be a unique, easily resolvable identifier and a need to respect other, non-technical values such as being easy to remember or being the same as some existing place or personal name. This tension may be resolved by maintaining a second, machine-oriented identifier, in addition to the user-friendly name—thus billing systems for large companies usually have both an account name and an account number. The second identifier can be unique and thus resolve ambiguities and avoid problems related to overloading of the account name. For example, personal names are usually overloaded with family history metadata (such as the surname, a given middle name that is the same as a mother's surname, or an appended "Jr." or "III"), and they are frequently not unique. Proposals to require that personal names be chosen uniquely always founder on cultural and personal identity objections. To avoid these problems, most systems

that maintain personal records assign distinct unique identifiers to people, and include both the user-friendly name and the unique identifier in their metadata.

Another example of tension in the choice of user-friendly names is found in the use of capital and small letters. Up through the mid-1960s, computer systems used only capital letters, and printed computer output always seemed to be shouting. There were a few terminals and printers that had lower-case letters, but one had to write a device-dependent application to make use of that feature, just as today one has to write a device-dependent application to use a virtual reality helmet. In 1965, the designers of the Multics time-sharing system introduced lower-case alphabetics to names of the file system. This being the first time anyone had tried it, they got it wrong. The designers of the UNIX file system copied the mistake. In turn, many modern file systems copy the UNIX design in order to avoid changing a widely used interface. The mistake is that the names "Court docket 5" and "Court Docket 5" can be bound to different files. The resulting violation of the *principle of least astonishment* can lead to significant confusion, since the computer rigidly enforces a distinction that most people are accustomed to overlooking on paper. Systems that enforce this distinction are called *case-sensitive*.

A more user-friendly way to allow upper- and lower-case letters in names is to permit the user to specify a preferred combination of upper- and lower-case letters for storage and display of a name, but coerce all alphabetic characters to the same case when doing name comparisons. Thus, when another person types the name, the case does not have to precisely match the display form. Systems that operate this way are called *case-preserving*. Both the Internet Domain Name System (described in Section 4.4) and the Macintosh file system provide this more user-friendly naming interface. A less satisfactory way to reduce case confusion is *case-coercing*, in which all names are both coerced to and stored in one case. A case-coercing system constrains the appearance of names in a way that can interfere with good human engineering.

The case studies in Section 3.2 and the war stories in Section 3.3 describe some unusual results when a system design mixes case-sensitive and case-preserving naming systems.

User-friendly names are not always strings of characters. In a graphical user interface (GUI), the shape (and sometimes the position) of an icon on the display is an identifier that acts exactly like a name, even if a character string is not associated with it. What action the system undertakes when the user clicks the mouse depends on where the mouse cursor was at that instant, and in a video game the action may depend on what else is happening at the same time. The identifier is thus bound to a time and a position on the screen, and that combination of values is in turn an identifier that is bound to some action.

Another, similar example of a user-friendly name that does not take the form of a string of characters is the cross-linking system developed by the M.I.T. Shakespeare Project. In that system, hypertext links say where they are coming from rather than where they are going to. Resolution starts by looking up the identifier of the place where the link was found. The principle is identical to that of the GUI/mouse example, and the system is described in Sidebar 3.2.

**Sidebar 3.2 Hypertext Links in the Shakespeare Electronic Archive** There are many representations of all of Shakespeare's plays: a modern text, the sixteenth-century folios, and several movies. In addition, a huge amount of metadata is available about each play: commentaries, stage directions, photographs and sketches of sets, directors' notes, and so on. In the study of a play, it would be helpful if these various representations could be linked together, so that, for example, if one were interested in the line "Alas, poor Yorick! I knew him, Horatio" from *Hamlet*, one could quickly check the wording in the several editions, compare different movie clips of the presentation of that line, and examine commentaries and stage directions that pertain to that line.

The M.I.T. Shakespeare Project has developed a system intended to make this kind of cross-reference easy. The basic scheme is first to assign a line number to every line in the play and then index every representation of the play by line number. A user displays one representation, for example, the text of a modern edition, and selects a line. Because the edition is indexed by line number, that selection is a reference that is bound to the line number. The user then clicks on the selection, causing the system to look up the associated line number in one of several contexts, each context corresponding to one of the other representations. The user selects a context, and the system can immediately resolve the line number in that context and display that representation in a different window on the user's screen.

### 3.1.6 Relative Lifetimes of Names, Values, and Bindings

If names must be chosen from a name space of short, fixed-length strings of bits or characters, they are by nature *limited* in number. The designer may permanently bind the names of a limited name space, as in the case of the registers of a simple processor, which may, for example, run from zero to 31. If the names of a limited name space can be dynamically bound, they must be reused. Therefore, the naming scheme usually replaces the BIND and UNBIND operations with some kind of name allocation/deallocation procedure. In addition, the naming scheme for a limited name space typically assigns the names, rather than letting the user choose them. On the other hand, if the name space is *unlimited*, meaning that it does not significantly constrain name lengths, it is usually possible to allow the user to choose arbitrary names. Thus, the telephone system in North America uses a naming scheme with short, fixed-length names such as 208–555–0175 for telephone numbers, and the telephone company nearly always assigns the numbers. (Section 3.3.5 describes some of the resulting problems.) On the other hand, names in most modern computer file systems are for practical purposes unlimited, and the user gets to choose them.

A naming scheme, a name, the binding of that name to a value, and the value to which the name is bound can all have different lifetimes. Often, both names and values are themselves quite long-lived, but the bindings that relate one to the other are somewhat more transient. Thus, both personal names and telephone numbers are typically long-lived, but when a person moves to a different city, the telephone company will

usually bind that personal name to a new telephone number and, after some delay, bind a new personal name to the old telephone number. In the same way, an application program and the operating system interfaces it uses may both be long-lived, but the binding that connects them may be established anew every time the program runs. Renewing the bindings each time the program is launched makes it possible to update the application program and the operating system independently. For another example, a named network service, such as `PostOffice.gov`, and a network attachment point, such as the Internet address 10.72.43.131, may both be long-lived, but the binding between them may change when the Post Office discovers that it needs to move that service to a different, more reliable computer, and it reassigns the old computer to a less important service.

When a name outlives its binding, any user of that name that still tries to resolve it will encounter a *dangling reference*, which is a use of a previously bound name that resolves either to a not-found result or to an irrelevant value. Thus, an old telephone number that rings in the wrong house or leads to a message saying "that number has been disconnected" is an example of a dangling reference. Dangling references are nearly always a concern when the name space is limited because names from limited name spaces must be reused. An object that incorrectly uses old names may make serious mistakes and even cause damage to an unrelated object that now has that name (for example, if the name is a physical memory address). In some cases, it may be possible to deal with dangling references by considering names to be simply hints that require verification. Thus when looking up the telephone number of a long-lost friend in a distant city, the first question when someone answers the phone at that number is something such as "are you the James Wilson who attended high school in . . . ?"

When a name space is unlimited and names are never reused, dangling references affect only the users of names that have for some reason been unbound from their former values. These dangling references can be less disruptive. For example, in a file system, an indirect name is one that is bound to some other (target) file system name. The indirect name becomes a dangling reference if someone removes the target name. Because an unbound indirect name simply produces a not-found result, it is more likely to be a nuisance than a source of damage. However, if someone accidentally or maliciously reuses the target name for a completely different file, the user of the indirect name could be in for a surprise.

When systems are large or distributed, however, a name, once bound and exported, tends to be discovered and remembered in widely dispersed places. That dispersion creates a need for stable bindings. This effect has been particularly noticed in the World Wide Web, whose design encourages the creation of cross-references to documents whose names are under someone else's control, with the result that cross-references often evolve into dangling references.

There is a converse to the dangling reference: when an object outlives every binding of a name to it, that object becomes what is known as an *orphan* or a *lost object* because no one can ever refer to it by name again. Lost objects can be a serious problem because there may be no good way to reclaim the physical storage they occupy. A system that regularly loses track of objects in this way is said to have a *storage leak*. To avoid lost

objects, some naming schemes keep track of the number of bindings to each object, and, when an UNBIND operation causes that number to reach zero, the system takes the opportunity to reclaim the storage occupied by the object. We saw this reference counting scheme used for links in the case study in Section 2.5. It contrasts with *tracing garbage collection*, an alternative technique used in some programming languages that involves occasional exploration of the named connections among objects to see which objects can and cannot be reached. The UNIX file system, described in Section 2.5, uses reference counting for file objects.

### 3.1.7 Looking Back and Ahead: Names are a Basic System Component

In this and the previous chapter, we have explored both the underlying principles of, and many engineering considerations surrounding, the use of names, but we have only lightly touched on the applications of names in systems. Names are a fundamental building block in all system areas. Looking ahead, almost every chapter will develop techniques and methods that depend on the use of names, name spaces, and binding:

- In modularizing systems with clients and services (Chapter 4), clients need a way to name services.
- In modularizing systems with virtualization (Chapter 5), virtual memory is an address naming system.
- In enhancing performance (Chapter 6), caches are renaming devices.
- Data communication networks (Chapter 7 [on-line]) use names to identify nodes and to route data to them.
- In transactions (Chapter 9 [on-line]) it is frequently necessary to modify several distinct objects "at the same time", meaning that all the changes appear to happen in a single program step, an example of atomicity. One way to obtain this form of atomicity is by temporarily grouping copies of all the objects that are to be changed into a composite object that has a temporary, hidden name, modifying the copies, and then rebinding the composite object to a visible name. In this way, all of the changed components are revealed simultaneously.
- In security (Chapter 11 [on-line]), designers use *keys*, which are names chosen randomly from a very large and sparsely populated address space. The underlying idea is that if the only way to ask for something is by name, and you don't know and can't guess its name, you can't ask for it, so it is protected.

Name discovery, which was introduced in the preceding chapter, will reappear when we discuss information protection and security. When one user either tries to identify or grant permission to another named user, it is essential to know the authentic name of that other user. If someone can trick you into using the wrong name, you may grant permission to a user who shouldn't have it. That requirement in turn means that one needs to be able to trace the name discovery procedure back to some terminating

direct communication step, verify that the direct communication took place in a credible fashion (such as examining a driver's license), and also evaluate the amount of trust to place in each of the other steps in the recursive name discovery protocol. Chapter 11 [on-line] describes this concern as the *name-to-key binding problem*.

Discovery of user names is one example in which authenticity is clearly of concern, but a similar authenticity concern can apply to any name binding, especially in systems shared by many users or attached to a network. If anyone can tinker with a binding, a user of that binding may make a mistake, such as sending something confidential to a hostile party. Chapter 11 [on-line] addresses in depth techniques of achieving authenticity. The User Internet Architecture research project uses such techniques to provide a secure, global naming system for mobile devices based on physical rendezvous and the trust found in social networks. For more detail, see Suggestions for Further Reading 3.2.5.

There is also a relation between uniqueness of names and security: If someone can trick you into using the same supposedly unique name for two different things, you may make a mistake that compromises security. The Host Identity Protocol addresses this problem by creating a name space of Internet hosts that is protected by cryptographic techniques similar to those described in Chapter 11 [on-line]. For more detail, see Internet Engineering Task Force *Request for Comments RFC 4423*.

This look ahead completes our introduction of concepts related to the design of naming systems. The next two sections of this chapter provide a case study of the relatively complex naming scheme used for pages of the World Wide Web, and a collection of war stories that illustrate what can go wrong when naming concepts fail to receive sufficient design consideration.

## 3.2 CASE STUDY: THE UNIFORM RESOURCE LOCATOR (URL)

The World Wide Web [see Suggestions for Further Reading 3.2.3] is a naming network with no unique root, potentially many different names for the same object, and complex context references. Its name-mapping algorithm is a conglomeration of several different component name-mapping algorithms. Let's fit it into the naming model.*

### 3.2.1 Surfing as a Referential Experience; Name Discovery

The Web has two layers of naming: an upper layer that is user-friendly, and a lower layer, which is also based on character strings, but is nevertheless substantially more mechanical.

*This case study informally introduces three message-related concepts that succeeding chapters will define more carefully: *client* (an entity that originates a request message); *server* (an entity that responds to a client's request); and *protocol* (an agreement on what messages to send and how to interpret their contents.) Chapter 4 expands on the client/service model, and Chapter 7 [on-line] expands the discussion of protocols.

At the upper layer, a Web page looks like any other page of illustrated text, except that one may notice what seem to be an unusually large number of underlined words, for example, Alice's page. These underlined pieces of text, as well as certain icons and regions within graphics, are labels for *hyperlinks* to other Web pages. If you click on a hyperlink, the browser will retrieve and display that other Web page. That is the user's view. The browser's view of a hyperlink is that it is a string in the current Web page written in HyperText Markup Language (HTML). Here is an example of a text hyperlink:

```
<a href="http://web.pedantic.edu/Alice/www/home.html">Alice's page</a>
```

Nestled inside this hyperlink, between the quotation marks, is a *Uniform Resource Locator* or, in Webspeak, a *URL*, which in the example is the name of another Web page at the lower naming layer. We can think of a hyperlink as binding a name (the underlined label) to a value (the URL) that is itself a name in URL name space. Since a context is a set of bindings of names to values, any page that contains hyperlinks can be thought of as a context, albeit not of the simple table-lookup variety. Instead, the name-mapping algorithm is one carried on in the mind of the user, matching ideas and concepts to the various hyperlink labels, icons, and graphics. The user does not usually traverse this naming network by typing path names, but rather by clicking on selected objects. In this naming network, a URL plays the role of a context reference for the links in the page fetched by the URL.

In order to retrieve a page in the World Wide Web, you need its URL. Many URLs can be found in hyperlinks on other Web pages, which helps if you happen to know the URL of one of those Web pages, but somewhere there must be a starting place. Most Web browsers come with one or more built-in Web pages that contain the URL of the browser maker plus a few other useful starting points in the Web. This is one way to get started on name discovery. Another form of name discovery is to see a URL mentioned in a newspaper advertisement.

### 3.2.2 Interpretation of the URL

In the example hyperlink above, we have an absolute URL, which means that the URL carries its own complete, explicit context reference:

```
http://web.pedantic.edu/Alice/www/home.html
```

The name-mapping algorithm for a URL works in several steps, as follows.

1. The browser extracts the part before the colon (here, `http`), considers it to be the name of a network protocol to use, and resolves that name to a protocol handler using a table-lookup context stored in the browser. The name of that context is built in to the browser. The interpretation of the rest of the URL depends on the protocol handler. The remaining steps describe the interpretation for the case of the hypertext transfer protocol (http) handler.

2. The browser takes the part between the `//` and the following `/` (in our example, that would be `web.pedantic.edu`) and asks the Internet Domain Name System (DNS) to resolve it. The value that DNS returns is an Internet address.

Section 4.4 is a case study of DNS that describes in detail how this resolution works.

3. The browser opens a connection to the server at that Internet address, using the protocol found in step 1, and as one of the first steps of that protocol it sends the remaining part of the URL, `/Alice/www/home.html`, to the server.
4. The server looks for a file in its file system that has that path name.
5. If the name resolution of step 4 is successful, the server sends the file with that path name to the client. The client transforms the file into a page suitable for display.

(Some Web servers perform additional name resolution steps. The discussion in Section 3.3.4 describes an example.)

The page sent by the server might contain a hyperlink of its own such as the following:

```
<a href="contacts.html">How to contact Alice.</a>
```

In this case the URL (again, the part between the quotation marks) does not carry its own context. This abbreviated URL is called a *relative* or *partial* URL. The browser has been asked to interpret this name, and in order to proceed it must supply a default context. The URL specification says to derive a context from the URL of the page in which the browser found this hyperlink, assuming somewhat plausibly that this hypertext link should be interpreted in the same context as the page in which it was found. Thus it takes the original URL and replaces its last component (`home.html`) with the partial URL, obtaining

```
http://web.pedantic.edu/Alice/www/contacts.html
```

It then performs the standard name-mapping algorithm on this newly fabricated absolute URL, and it should expect to find the desired page in Alice's `www` directory.

A page can override this default context by providing something called a *base element* (e.g., `<base href="some absolute URL">`). The absolute URL in the base element is a context reference to use in resolving any partial URL found on the page that contains the base element.

### 3.2.3 URL Case Sensitivity

Multiple naming schemes are involved in the Web naming algorithm, as is clear by noticing that some parts of a URL are case sensitive and other parts are not. The result can be quite puzzling. The host name part of a Uniform Resource Locator (URL) is interpreted by the Internet Domain Name System, which is case-insensitive. The rest of the URL is a different matter. The protocol name part is interpreted by the client browser, and case sensitivity depends on its implementation. (Check to see if a URL starting with `"HTTP://"` works with your favorite Web browser.) The Macintosh implementation of Firefox treats the protocol name in a case-preserving fashion, but the now-obsolete Macintosh implementation of Internet Explorer is case-coercing.

The more interesting case-sensitivity questions come after the host name. The Web specifies that the server should interpret this part of the URL using a scheme that depends on the protocol. In the case of the HTTP protocol, the URL specification is insistent that this string is *not* a UNIX file name, but it is silent on case sensitivity. In practice, most systems interpret this string as a path name in their file system, so case-sensitivity depends on the file system of the server. Thus if the server is running a standard UNIX system, the path name is case-sensitive, while if the server is a standard Macintosh, the path name is case-preserving. There are examples that mix things up even further; Section 3.3.4 describes one such example.

### 3.2.4 Wrong Context References for a Partial URL

The practice of interpreting URL path names as path names of the server's file system can result in unexpected surprises. As described earlier, the Web browser supplies a default context reference for relative names (that is, partial URLs) found in Web pages. The default context reference it supplies is simply the URL that the browser used to retrieve the page that contained the relative name, truncated back to the last slash character. This context reference is the name of a directory at the server that should be used to resolve the (first component of) the relative name.

Some servers provide a URL name space by simply using the local (for example, UNIX) file system name space. When the local file system name space allows synonyms (symbolic links and the Network File System mounts described in Section 4.5 are two examples) for directory names, the mapping of local file system name space to the URL name space is not unique. Thus, several different URLs can have different path names for the same object. For example, suppose that there is a UNIX file system with a symbolic link named `/alice/home.html` that is actually an indirect reference to the file named `/alice/www/home.html`. In that case, the URLs

*1* `<http://web.pedantic.edu/alice/home.html>`

and

*2* `<http://web.pedantic.edu/alice/www/home.html>`

refer to the same file. Trouble can arise when the object that has multiple URLs is a directory whose name is used as a context reference. Continuing the example, suppose that file `home.html` contains the hyperlink `<a href = "contacts.html">`. Both `home.html` and `contacts.html` are stored in the directory `/alice/www`. Suppose further that the browser obtained `home.html` by using the URL *1* above.

Now, the user clicks on the hyperlink containing the partial URL `contacts.html`, asking the browser to resolve it. Following the usual procedure, the browser materializes a default context reference by truncating the original URL to obtain:

`http://web.pedantic.edu/alice/`

and then uses this name as a context by concatenating the partial URL:

```
http://web.pedantic.edu/alice/contacts.html
```

This URL will probably produce a not-found response because the file we are looking for actually has the path name `/alice/www/contacts.html`. Or worse, this request could return a different file that happens to be named `contacts.html` in the directory `/alice`. The confusion may be compounded if the different file with the same name turns out to be an out-of-date copy of the current `contacts.html`. On the other hand, if the user originally used URL *2,* the browser would retrieve the file named `/alice/www/contacts.html`, as the Web page designer expected.

A similar problem can arise when interpreting the relative name "..". This name is, conventionally, the name for the parent directory of the current directory. The UNIX system provides a semantic interpretation: look up the name ".." in the current directory, where by convention it evaluates (in inode name space) to the parent directory. In contrast, the Web specifies that ".." is a syntactic signal that means "modify the default context reference by discarding the least significant component of the path name." Despite these drastically different interpretations of "..", the result is usually the same because the parent of an object is usually the thing named by the next-earlier component of that object's path name. The exception (and the problem) arises when the Web's syntactic modification rule is applied to a path name with a component that is an indirect name for a directory. If the path name in the URL does not traverse the directory's parent, syntactic interpretation of ".." creates a default context reference different from the one that would be supplied by semantic interpretation.

Suppose, in our example, that the file `home.html` contains the hyperlink `<a href="../phone.html">`. If the user who reached home.html via URL *1* clicks on this hyperlink, the browser will truncate that URL and concatenate it with the partial URL, to obtain

```
http://web.pedantic.edu/alice/../phone.html
```

and then use the syntactic interpretation of `"..."` to produce the URL

```
http://web.pedantic.edu/phone.html
```

another non-existent file. Again, if the user had started with URL *2*, the result of syntactic interpretation of `"..."` would be to request the file

```
http://web.pedantic.edu/alice/phone.html
```

as originally intended.

This problem could be fixed in at least three different ways:

1. Arrange things so that the default context reference always works.
   a. Always install a UNIX link to the referenced page in the directory that held the referring page. (Or never use UNIX links at all.)
   b. Never use ".." in hyperlinks.

2. Do a better of job of choosing a default context reference.
   a. The client sends the original URL plus the Web link to the server and lets the server figure out what context to use.
3. Provide an explicit context reference.
   a. The server places an absolute URL in a location field of the protocol header.
   b. The client uses that URL as the context reference.

One might suggest that the implementer of the server (or the writer of the pages containing the relative links) failed to heed the following warning in the Web URL specification* for path names: "The similarity to unix and other disk operating system filename conventions should be taken as purely coincidental, and should not be taken to indicate that URIs should be interpreted as file names."

This warning is technically correct, but the suggestion is misleading. Unfortunately, the problem is built in to the Web naming specifications. Those specifications require that relative names be interpreted syntactically, yet they do not require that every object have a unique URL. Unambiguous syntactic interpretation of relative names requires that the context reference be a unique path name. Since the browser derives the context reference from the path name of the object that contained the relative name, and that object's path name does not have to be unique, it follows that syntactic interpretation of relative names will intrinsically be ambiguous. When servers try to map URL path names to UNIX path names, which are not unique, they are better characterized as exposing, rather than causing, the problem.

That analysis suggests that one way to conquer the problem is to change the way in which the browser acquires the context reference. If the browser could somehow obtain a canonical path name for the context reference, the same canonical path name that the UNIX system uses to reach the directory from the root, the problem would vanish.

### 3.2.5 Overloading of Names in URLs

Occasionally, one will encounter a URL that looks something like

```
http://www.amazon.com/exec/obidos/ASIN/0670672262/o/qid=
  921285151/sr=2-2/002-7663546-7016232
```

or perhaps

```
http://www.google.com/search?hl=en&q=books+about+systems&btnG=
  Google+Search&aq=f&oq=
```

Here we have two splendid examples of overloading of names. The first example is of a shopping service. Because the server cannot depend on the client to maintain any state about this shopping session other than the URL of the Web page currently being displayed, the server has encoded the state of the shopping session, in the form of an identifier of a state-maintaining file at the server, in the path name part of the URL.

*Tim Berners-Lee, *Universal Resource Identifiers: Recommendations*.

The second example is of a search service; the browser has encoded the user's search query into the path name part of the URL it has submitted. The tip-off here is the question mark in the middle of the name, which is a syntactic trick to alert the server that the string up to the question mark is the name of a program to be launched, while the string after the question mark is an argument to be given to that program. To see what processing www.google.com does to respond to such a query, see Suggestions for Further Reading 3.2.4.

There is another form of overloading in many URLs: they concatenate the name of a computer site with a path name of a file, neither of which is a particularly stable identifier. Consider the following name for an earthquake information service:

```
http://HPserver14.pedantic.edu/disk05/science/geophysics/
  quakes.html
```

This name is at risk of change if the HP computer is replaced by a Sun server, if the file server is moved to `disk04`, if the geophysics department is renamed "geology" or moves out of the school of science, or if the responsibility for the earthquake server moves to the Institute for Scholarly Studies. A URL such as this example frequently turns out to be unresolvable, even though the page it originally pointed to is still out there somewhere, perhaps having moved to a different site or simply to a different directory at the original site.

One way to avoid trapping the name of a site in the URLs that point to it is to choose a service name and arrange for DNS to bind that service name as an indirect name for the site. Then, if it becomes necessary to move the Web site to a different computer, a change to the binding of the service name is all that is needed for the old URLs to continue working. Similarly, one can avoid trapping an overloaded path name in a URL by judicious use of indirect file names. Thus the name

```
http://quake.org/library/quakes.html
```

could refer to the same Web page, yet it can remain stable through a wide variety of changes.

Considerable intellectual energy has been devoted to inventing a replacement for the URL that has less overloading and is thus more robust in the face of changes of server site and file system structure. Several systems have been proposed: *Permanent URL* (PURL), *Universal Resource Name* (URN), *Digital Object Identifier* (DOI)®, and *handle*. To date, none of these proposals has yet achieved wide enough adoption to replace the URL.

## 3.3 WAR STORIES: PATHOLOGIES IN THE USE OF NAMES

Although designing a naming scheme seems to be a straightforward exercise, it is surprisingly difficult to meet all of the necessary requirements simultaneously. The following are several examples of strange and sometimes surprising results that have been noticed in deployed naming schemes.

### 3.3.1 A Name Collision Eliminates Smiling Faces

A west coast university provides a "visual class list" Web interface that instructors can use to obtain the names and photos of all the students enrolled in a particular section of a class. At the beginning of the fall 2004 teaching term, instructors noticed that their classes had several photographs of the same individual. One might believe a section includes a set of triplets, but not triskaidekatuplets.

*What went wrong:* When there is no picture available for a student, the system inserts an image of a smiley face with the words "No picture available". The system designer stored the image in a file named "smiley.jpg". That fall a new freshman whose last name was Smiley registered the user name "smiley". As one might expect, the freshman's photograph was named "smiley.jpg", and it became the "No picture available" image.

### 3.3.2 Fragile Names from Overloading, and a Market Solution

Internet mailbox names such as `Alice@Awesome.net` can be viewed as two-component addresses. The component before the @-sign identifies a particular mailbox, and the component after the @-sign is an Internet domain name that identifies the Internet service provider (ISP) that provides that mailbox. When two ISPs (say, `Awesome.net` and `Awful.net`) merge, the customers of one of them (and sometimes both) typically receive a letter telling them that their mailbox address, which contained some representation of the name of their former ISP, will have to change. The new ISP may automatically forward mail addressed to the old address, or it may require that the user notify all of his or her correspondents of the new mailbox address. The reason for the change is that the second component of the old mailbox name was overloaded with a trademark. The new provider does not want to continue using that old trademark, and the old provider may not want to see the trademark used by the new provider.

Alice may also find, to her disappointment, that not only does the domain name of her mailbox change from `Awesome.net` to `Awful.net`, but that in `Awful.net`'s mailbox name space, another customer has already captured the personal mailbox name `Alice`, so she may even have to choose a new personal mailbox name, such as `Alice24`.

As the Internet grows, some ISPs have prospered and others have not, so there have been many mergers and buyouts. The resulting fragility of e-mail service provider names has created a market for indirect domain names. The customers in this market are users who require a stable e-mail address, such as people who run private businesses or who have a large number of correspondents. For an annual fee, an indirect name provider will register a new domain name, such as `Alice.com`, and configure a DNS name server so that the mailbox name `Alice@Alice.com` becomes a synonym for `Alice@Awesome.Net`. Then, upon being notified of the ISP merger, Alice simply asks the indirect name provider to rebind the mailbox name `Alice@Alice.com` to `Alice24@Awful.net`, and her correspondents don't have to know that anything happened.

### 3.3.3 More Fragile Names from Overloading, with Market Disruption

The United States Post Office assigns postal delivery codes, called Zip codes, hierarchically, so that it can take advantage of the hierarchy in routing mail. Zip codes have five digits. The first digit identifies one of 10 national areas; New England is area 0 and California, Washington, and Oregon comprise area 9. The next two digits identify a section. The South Station Postal Annex in Boston, Massachusetts, is the headquarters of section 021. All Zip codes beginning with those three digits have their mail sorted at that sectional center. Zip codes beginning with 024 identify the Waltham, Massachusetts, section. The last two digits of the Zip code identify a specific post office (known as a station), such as Waban, Massachusetts, 02468. Zip codes can also have four appended digits (called Zip + 4) that are used to sort mail into delivery order for each mail carrier. Although they are numerical, adjacent zip codes are not necessarily assigned to adjacent stations or adjacent sections, so they are really names rather than physical addresses. Despite not being interpretable as physical addresses, these names are overloaded with routing information.

Although routing is hierarchical, apparently the 10 national areas have no routing significance; everything is done by section. It is reported that if you walk into the South Station Postal Annex in Boston, you will find that outgoing mail is being sorted into 999 bins, one for each sectional center, nationwide. In addition, for mail addressed with Zip codes beginning with 021 (that is, within the South Station section) there are 99 bins, one for each station within the section. The mail in the outgoing bins goes into bags, with each bag containing mail for one section. Then all the bags for Southern California sections, for example, go into the same truck to the airport, where they go onto a plane to Los Angeles. As they come off the plane in Los Angeles, they are loaded onto different trucks that go to the various Southern California sections. The mail in the 99 bins for section 021 also goes into bags, with each bag destined for a different post office within the 021 section.

Mail that originates at a post office and is destined for the same post office still goes to the sectional center for sorting because individual post offices don't have the automatic sorting machines that can put things into delivery order. There used to be many exceptions to the rule that all mail goes to a sectional center, but the number of exceptions has been gradually reduced over the years.

When the volume of mail handled by the South Station Postal Annex began to exceed its capacity in the late 1990s, the Post Office decided to transfer part of that section's work to the newer Waltham, Massachusetts, section. Since the first three digits of the Zip code are overloaded with routing information, to accomplish this change it announced that about half of the Zip codes that began with 021 would, on July 1, 1998, change to 024. The result, as one might expect, was rather chaotic. The Post Office tried to work with large mailers to have them automatically update their address records, but loose ends soon appeared.

For example, American Express, a credit card company, installed a Zip code translator in its mail label printing system, so that its billing statements would go directly to the Waltham section, but it did not change its internal customer address

records because its computer system flags all address changes as "moves", which affect verification procedures as well as credit ratings. So everything that American Express mailed was addressed properly, but their internal records retained the old Zip codes.

Now comes the problem: some Internet vendors will not accept a credit card unless the shipping address is identical to the credit card address. Customers began to encounter situations in which the Internet vendor rejected the Zip code 02168 as being an invalid delivery address, and American Express rejected the Zip code 02468 because it did not match its customer record. When this situation arose, it was not possible to complete a purchase without human intervention.

Despite the vendor check that identifies 02168 as invalid, mail addressed with that Zip code continued to be correctly delivered to addresses in Waban for several years. It just took an extra day to be delivered because it went first to the South Station Postal Annex, which simply forwarded it to the Waltham sectional center. The renaming was done not because the post office was running out of Zip codes, but rather because the sorting capacity of one of its sectional centers was exceeded.

### 3.3.4 Case-Sensitivity in User-Friendly Names

Even though, as described on page 128, the UNIX system propagated case-sensitive file names to many other file systems, not all widely used naming schemes are case-sensitive. The Internet generally is case-preserving. For example, in the Internet Domain Name System described in Section 4.4, one can open a network connection to `cse.pedantic.edu` or to `CSE.Pedantic.edu`; both refer to the same destination. The Internet mail system is also specified to be case-preserving, so you can send mail to `alice@pedantic.edu`, `Alice@pedantic.edu`, and `aLiCe@pedantic.edu`, and all three messages should go to the same mailbox.

In contrast, the Kerberos authentication system (described in Sidebar 11.6 [on-line]) is case-sensitive, so the names "alice" and "Alice" can identify different users. The rationale for this decision is muddy. Requiring that the case accurately match makes it harder for an intruder to guess a user's name, so one can argue that this decision enhances security. But allowing "alice" and "Alice" to identify different users can lead to serious mistakes in setting up permissions, so one can also argue that this decision weakens security. This decision comes to a head, for example, in the implementation of a mail delivery service with Kerberos authentication. It is not possible to correctly do a direct mapping of Kerberos user names to mailbox names because the necessary coercion might merge the identities of two distinct users.

A mixed example is a service-naming service developed at M.I.T. and called Hesiod, which uses the Internet Domain Name System (DNS) as a subsystem. One of the kinds of services Hesiod can name is a remote file system. DNS (and thus Hesiod) is case-insensitive, while file system names in UNIX systems are case-sensitive. This difference leads to another example of a user interface glitch. If a user asks to attach a remote file

system, specifying its Hesiod name, Hesiod will locate the file system using whatever case the user typed, but the UNIX `mount` command mounts the file system using the name coerced to lower case. Thus if the user says, for example, to mount the remote file system named `CSE`, Hesiod will locate that remote file system, but the UNIX system will mount it using the name `cse`. To use this directory in a file name, the user must then type its name in lower case, which may come as a surprise.

Hesiod is used as a subsystem in larger systems, so the mixing of case-sensitive and case-insensitive names can become worse. For example, the current official M.I.T. Web server responds to the URL

```
http://web.mit.edu/Alice/www/home.html
```

by first trying a simple path name resolution of the string `/Alice/www/home.html`. If it gets a `NOT-FOUND` result from the resolution of that path name, it extracts the first component of the path name (`Alice`) and presents it to the Hesiod service naming system, with a request to interpret it as a remote file system name. Since Hesiod is case-insensitive, it doesn't matter whether the presented name is `Alice`, `alice`, or `aLiCe`. Whatever the case of the name presented, Hesiod coerces it to a standard case, and then it returns the standard file system path name of the corresponding remote file system directory, which for this example might be

```
/afs/athena/user/alice
```

The Web server then replaces the original first component (`Alice`) with this path name and attempts to resolve the path name:

```
/afs/athena/user/alice/www/home.html
```

Thus for the current M.I.T. Web server, the first component name after the host name in a URL is case-insensitive, while the rest of the name is case-sensitive.

### 3.3.5 Running Out of Telephone Numbers

*"Nynex is Proposing New '646' Area Code for Manhattan Lines"*

— headline, *Wall Street Journal,* March 3, 1997

The North American telephone numbering plan name space is nicely hierarchical, which would seem to make it easy to add phone numbers. Although this appears to be an example of an unlimited name space, it is not. It is hierarchical, but the hierarchy is rigid—there is a fixed number of levels, and each level has a fixed size.

Much of Europe does it the other way. In some countries it seems that every phone number has a different number of digits. A variable-length numbering plan has a downside. The telephone numbers are longest in the places that grew the most and thus have the most telephone calls. In addition, because the central exchange can't find the end of a variable-length telephone number by counting digits, some other scheme is necessary, such as noticing that the user has stopped dialing for a while.

A European-style solution to the shortage of phone numbers in Manhattan would be to simply announce that from now on, all numbers in Manhattan will be 11 digits long. But since the entire American telephone system assumes that telephone numbers are exactly 10 digits long, the American solution is to introduce a new area code.

A new area code can be introduced in one of two ways: *splitting* and *overlay*. Traditionally, the phone companies have used only splitting, but overlay is beginning to receive wider attention.

Splitting (sometimes called partition) is done by drawing a geographical line across the middle of the old area code—say 84th street in Manhattan—and declaring that everyone north of that line is now in code 646 and everyone south of that line will remain in code 212. When splitting is used, no one "changes" their seven-digit number, but many people must learn a new number when calling someone else. For example,

- Callers from Los Angeles who used to dial (212)–xxx–xxxx must now dial (646)–xxx–xxxx if they are calling to a phone north of 84th street, but they must use the old area code for phones located south of 84th street.
- Calling from one side of 84th street to the other now requires adding an area code, where previously a seven-digit number was all one had to dial.

In the alternative scheme, overlay, area code 212 would continue to cover all of Manhattan, but when there weren't any phone numbers left in that area code, the telephone companies would simply start assigning new numbers with area code 646. Overlay places a burden on the switching system, and it wouldn't have worked with the step-by-step switches developed in the 1920s, in which the telephone number described the route to that telephone. When the Bell System started to design the crossbar switches introduced in the 1940s, it realized that this inflexibility was a killer problem, so it introduced a number-to-route lookup—a name-resolving system called a *translator*—as part of switch design. With modern computer-based switches, translation is easy. So there is now nothing (but old software) to prevent two phones served by the same switch from having numbers with different area codes.

Overlay sounds like a great idea because it means that callers from Los Angeles continue to dial the same numbers they have always dialed. However, as in most engineering trade-offs, someone loses. Everyone in Manhattan now has to dial a 10-digit number to reach other places in Manhattan. One no longer can tell what the area code is by the geographic location of the phone. One also can't pinpoint the location of the target by its area code because the area code has lost its status as geographic metadata. This could be a concern if people become confused as to whether or not they were making a toll call.

Another possibility would be to use as a default context the area code of the originating phone. If calling from a 212 phone, one wouldn't have to dial an area code to call another 212 number. The prevailing opinion—which may be wrong—is that people can't handle the resulting confusion. Two phones on the

same desk, or two adjacent pay phones, may have different area codes, and thus to call someone in the next office one might have to dial different numbers from the two phones.

Here is one way of coping: BankBoston (long since merged into larger banks) once arranged that the telephone number 788–5000 ring its customer service center from every area code in the state of Massachusetts. The nationwide toll-free number (800) 788–5000 also rang there. Although that arrangement did not completely eliminate name translation, it reduced it significantly and made the remaining name translation simple enough that people could actually remember how to do it.

Requiring that all numbers be dialed with all 10 digits encourages a more coherent model: the number you dial to reach a particular target phone does not depend on the number from which you are calling. The trade-off is that every North American number dialed would require 10 digits, even when calling the phone next door. The North American telephone system has been gradually moving in this direction for a long time. In many areas, it was once possible to call people in the same exchange simply by dialing just the last four digits of their number. Then it took five digits. Then seven. The jump to 10 would thus be another step in the sequence.

The newspaper also reports that at the rate telephone numbers are being used up in Manhattan, another area code will be needed within a few years. That observation would seem to affect the decision. Splitting is disruptive every time, but overlay is disruptive only the first time it is done. If there is going to be another area code needed that soon, it might be better to use overlay at the earliest opportunity, since adding still more area codes with overlay will cause no disruption at all.

Overlay is already widely used. Manhattan cell phones and beepers have long used area code 917, and little confusion resulted. Also, in response to an outcry over "yet another number change", in 1997 the Commonwealth of Massachusetts began requiring that future changes to its telephone numbering plan be done with overlay.

## EXERCISES

**3.1** Alyssa asks you for some help in understanding how metadata is handled in the UNIX file system, as described in Section 2.5.

**3.1a** Where does the UNIX system store system metadata about a file?

**3.1b** Where does it store user metadata about the file?

**3.1c** Where does it store system metadata about a file system?

*2008-0-1*

**3.2** Bob and Alice are using a UNIX file system as described in Section 2.5. The file system has two disks, mounted as `/disk1` and `/disk2`. A system administrator

creates a "home" directory containing symbolic links to the home directories of Bob and Alice via the commands:

```
mkdir /home
ln -s /disk1/alice /home/alice
ln -s /disk2/bob /home/bob
```

Subsequently, Bob types the following to his shell:

```
cd /home/alice
cd ../bob
```

and receives an error.

Which of the following best explains the problem?

**A.** The UNIX file system forbids the use of ".." in a cd command when the current working directory contains a symbolic link.

**B.** Since Alice's home directory now has two parents, the system complains that ".." is ambiguous in that directory.

**C.** In Alice's home directory, ".." is a link to `/disk1`, while the directory "bob" is in `/disk2`.

**D.** Symbolic links to directories on other disks are not supported in the UNIX file system; their call-by-name semantics allows their creation but causes an error when they are used.

*2007-1-7*

**3.3** We can label the path names in the previous question as *semantic* path names. If Bob types `"cd ..."` while in working directory `d`, the command changes the working directory to the directory in which `d` was created. To make the behavior of `"..."` more intuitive, Alice proposes that `"..."` should behave in path names *syntactically*. That is, the parent of a directory `d`, `d/..` is the same directory that would obtain if we instead referred to that directory by removing the last path name component of `d`. For example, if Bob's current working directory is `/a/b/c` and Bob types `"cd ..."`, the result is exactly as if Bob had executed `"cd /a/b"`

**3.3a** If the UNIX file system were to implement syntactic path names, in which directory would Bob end up after typing the following two commands?

```
cd /home/alice
cd ../bob
```

**3.3b** Under what circumstances do semantic path names and syntactic path names provide the same behavior?

**A.** When the name space of the file system forms an undirected graph.

**B.** When the name space of the file system forms a tree rooted at "/".

**C.** When there are no synonyms for directories.

**D.** When symbolic links, like hard links, can be used as synonyms only for files.

**3.3c** Bob proposes the following implementation of syntactic names. He will first rewrite a path name syntactically to eliminate the `"..."`, and then resolve the rewritten path name forward from the root. Compared to the implementation of semantic path names as described in Section 2.5, what is a disadvantage of this syntactic implementation?

**A.** The syntactic implementation may require many more disk accesses than for semantic path names.

**B.** This cost of the syntactic implementation scales linearly with the number of path name components.

**C.** The syntactic implementation doesn't work correctly in the presence of hard links.

**D.** The syntactic implementation doesn't resolve `"."` correctly in the current working directory.

*2007-0-1*

**3.4** The inode of a file plays an important role in the UNIX file system. Which of these statements is true of the inode data structure, as described in Section 2.5?

**A.** The inode of a file contains a reference count.

**B.** The reference count of the inode of a directory should not be larger than 1.

**C.** The inode of a directory contains the inodes of the files in the directory.

**D.** The inode of a symbolic link contains the inode number of the target of the link.

**E.** The inode of a directory contains the inode numbers of the files in the directory.

**F.** The inode number is a disk address.

**G.** A file's inode is stored in the first 64 bytes of the file.

*2005-1-4, 2006-1-1, and 2008-1-3*

**3.5** Section 3.3.1 describes a name collision problem. What could the designer of that system have done differently to eliminate (or reduce to a negligible probability) the possibility of this problem arising?

*2008-0-2*

**Additional exercises relating to Chapter 3 can be found in the problem sets beginning on page 425.**

CHAPTER

# Enforcing Modularity with Clients and Services

# 4

## CHAPTER CONTENTS


**Overview** ..... **148**
**4.1 Client/Service Organization** ..... **149**
4.1.1 From Soft Modularity to Enforced Modularity ..... 149
4.1.2 Client/Service Organization ..... 155
4.1.3 Multiple Clients and Services ..... 163
4.1.4 Trusted Intermediaries ..... 163
4.1.5 A Simple Example Service ..... 165
**4.2 Communication Between Client and Service** ..... **167**
4.2.1 Remote Procedure Call (RPC) ..... 167
4.2.2 RPCs are not Identical to Procedure Calls ..... 169
4.2.3 Communicating Through an Intermediary ..... 172
**4.3 Summary and The Road Ahead** ..... **173**
**4.4 Case Study: The Internet Domain Name System (DNS)** ..... **175**
4.4.1 Name Resolution in DNS ..... 176
4.4.2 Hierarchical Name Management ..... 180
4.4.3 Other Features of DNS ..... 181
4.4.4 Name Discovery in DNS ..... 183
4.4.5 Trustworthiness of DNS Responses ..... 184
**4.5 Case Study: The Network File System (NFS)** ..... **184**
4.5.1 Naming Remote Files and Directories ..... 185
4.5.2 The NFS Remote Procedure Calls ..... 187
4.5.3 Extending the UNIX File System to Support NFS ..... 190
4.5.4 Coherence ..... 192
4.5.5 NFS Version 3 and Beyond ..... 194
**Exercises** ..... **195**



**Principles of Computer System Design: An Introduction**


Copyright © 2009 by Jerome H. Saltzer and M. Frans Kaashoek. All rights of reproduction in any form reserved.


DOI: 10.1016/B978-0-12-374957-4.00012-8

## OVERVIEW

The previous chapters established that dividing a system into modules is good and showed how to connect modules using names. If all of the modules were correctly implemented, the job would be finished. In practice, however, programmers make errors, and without extra thought, errors in implementation may too easily propagate from one module to another. To avoid that problem, we need to strengthen the modularity. This chapter introduces a stronger form of modularity, called *enforced modularity*, that helps limit the propagation of errors from one module to another. In this chapter we focus on software modules. In Chapter 8 [on-line] we develop techniques to handle hardware modules.

One way to limit interactions between software modules is to organize systems as clients and services. In the client/service organization, modules interact only by sending messages. This organization has three main benefits:

- Messages are the only way for a programmer to request that a module provide a service. Limiting interactions to messages makes it more difficult for programmers to violate the modularity conventions.
- Messages are the only way for errors to propagate between modules. If clients and services fail independently and if the client and the service check messages, they may be able to limit the propagation of errors.
- Messages are the only way for an attacker to penetrate a module. If clients and services carefully check the messages before they act on them, they can block attacks.

Because of these three benefits, system designers use the client/service organization as a starting point for building modular, fault tolerant, and secure systems.

Designers use the client/service model to separate larger software modules, rather than, say, individual procedures. For example, a database system might be organized as clients that send messages with queries to a service that implements a complete database management system. As another example, an e-mail application might be organized into readers—the clients—that collect e-mail from a service that stores mailboxes.

One effective way to implement the client/service model is to run each client and service module in its own computer and set up a communication path over a wire between the computers. If each module has its own computer, then if one computer (module) fails, the other computer (module) can continue to operate. Since the only communication path is that wire, that is also the only path by which errors can propagate.

Section 4.1 of this chapter shows how the client/service model can enforce modularity between modules. Section 4.2 presents two styles of sending and receiving messages: remote procedure call and publish/subscribe. Section 4.3 summarizes the major issues identified in this chapter but not addressed, and presents a road map for addressing them. Finally, there are detailed case studies of two widely used client/service applications, the Internet Domain Name System and the Network File System.

## 4.1 CLIENT/SERVICE ORGANIZATION

A standard way to create modularity in a large program is to divide it up into named procedures that call one another. Although the resulting structure can be called modular, implementation errors can propagate from caller to callee and vice versa, and not just through their specified interfaces. For example, if a programmer makes a mistake and introduces an infinite loop in a called procedure and the procedure never returns, then the callee will never receive control again. Or since the caller and callee are in the same address space and use the same stack, either one can accidentally store something in a space allocated to the other. For this reason, we identify this kind of modularity as *soft*. Soft modularity limits interactions of correctly implemented modules to their specified interfaces, but implementation errors can cause interactions that go outside the specified interfaces.

To enforce modularity, we need hard boundaries between modules so that errors cannot easily propagate from one module to another. Just as buildings have firewalls to contain fires within one section of the building and keep them from propagating to other sections, so we need an organization that limits the interaction between modules to their defined interfaces.

This section introduces the client/service organization as one approach to structuring systems that limit the interfaces through which errors can propagate to the specified messages. This organization has two benefits: first, errors can propagate only with messages. Second, clients can check for certain errors by just considering the messages. Although this approach doesn't limit the propagation of all errors, it provides a ***sweeping simplification*** in terms of reasoning about the interactions between modules.

### 4.1.1 From Soft Modularity to Enforced Modularity

As a more concrete example of how modules interact, suppose we are writing a simple program that measures how long a function runs. We might want to split it into two modules: (1) one system module that provides an interface to obtain the time in units specified by the caller and (2) one application module that measures the running time of a function by asking for time from the clock device, running the function, and requesting the time from the clock device after the function completes. The purpose of this split is to separate the measurement program from the details of the clock device:

```
procedure MEASURE (func)
    start ← GET_TIME (SECONDS)
    func () // invoke the function
    end ← GET_TIME (SECONDS)
    return end – start
```

```
procedure GET_TIME (units)
    time ← CLOCK
    time ← CONVERT_TO_UNITS (time, units)
    return time
```

The procedure MEASURE takes a function *func* as argument and measures its running time. The procedure GET_TIME returns the time measured in the units specified by the caller. We may desire this clear separation in modules because, for example, we don't want every function that needs the time to know the physical address of the clock (CLOCK in line *2* of GETTIME) in all application programs, such as MEASURE, that use the clock. On one computer, the clock is at physical address $17E5_{hex}$, but on the next computer it is at $24FFF2_{hex}$. Or some clocks return microseconds, and others return sixtieths of a second. By putting the clock's specific properties into GET_TIME, the callers of GET_TIME do not have to be changed when a program is moved to another computer; only GET_TIME must be changed.

This boundary between GET_TIME and its caller, is soft, however. Although procedure call is a primary tool for modularity, errors can still leak too easily from one module to another. It is obvious that if GET_TIME returns a wrong answer, the caller has a problem. It is less obvious that programming errors in GET_TIME can cause trouble for the caller even if GET_TIME returns a correct answer. This section explains why procedure call allows propagation of a wide variety of errors and will introduce an alternative that resembles procedure call but that more strongly limits propagation of errors.

To see why procedure calls allow propagation of many kinds of errors, one must look at the detail of how procedure calls work and at the processor instructions that implement procedure calls. There are many ways to compile the procedures and the call from MEASURE to GET_TIME into processor instructions. For concreteness we pick one procedure call convention. Others differ in the details but exhibit the same issues that we want to explore.

We implement the call to GET_TIME with a stack, so that GET_TIME could call other procedures (although in this example it does not do so). In general, a called procedure may call another procedure or even call itself recursively. To allow for calls to other procedures, the implementation must adhere to the *stack discipline*: each invocation of a procedure must leave the stack as it found it.

To adhere to this discipline, there must be a convention for who saves what registers, who puts the arguments on the stack, who removes them, and who allocates space on the stack for temporary variables. The particular convention used by a system is called the *procedure calling convention*. We use the convention shown in Figure 4.1. Each procedure call results in a new stack frame, which has space for saved registers, the arguments for the callee, the address where the callee should return, and local variables of the callee.

**FIGURE 4.1**

Procedure call convention.

Given this calling convention, the processor instructions for these two modules are shown in Figure 4.2. In this example, the instructions of the

caller (MEASURE) start at address 100, the instructions of the callee (GET_TIME) start at address 200. The stack grows up, from a low address to a high address. The return value of a procedure is passed through register R0. For simplicity, assume that instructions, memory locations, and addresses are all 4 bytes wide. For our example, MEASURE invokes GETTIME as follows:

1. The caller saves content of temporary registers (R1 and R2) at addresses 100 through 112.
2. The caller stores the arguments on the stack (address 116 through 124) so that the callee can find them. (GET_TIME takes one argument: *unit*.)
3. The caller stores a return address on the stack (address 128 through 136) so that the callee can know where the caller should resume execution. (The return address is 148.)

**Machine code for** MEASURE:

```
100: STORE R1, SP          // save content of R1
104: ADD 4, SP             // adjust stack
108: STORE R2, SP          // save content of R2
112: ADD 4, SP             // adjust stack
116: MOV SECONDS, R1       // move argument to GET_TIME in R1
120: STORE R1, SP          // store argument in R1 on stack
124: ADD 4, SP             // adjust stack
128: MOV 148, R1           // place return address in R1
132: STORE R1, SP          // store return address in R1 on stack
136: ADD 4, SP             // adjust stack
140: STORE 200, R1         // load address of GET_TIME into R1
144: JMP R1                // jump to it
148: SUB 8, SP             // adjust top of stack
152: MOV SP, R2            // restore R2's content
156: SUB 4, SP             // adjust stack
160: MOV SP, R1            // restore R1's content
164: SUB 4, SP             // adjust stack
168: MOV R0, start         // store result in local stack variable start
172: .....                 // invoke func and GET_TIME again
```

**Machine code for** GET_TIME:

```
200: MOV SP, R1            // move stack pointer into R1
204: SUB 8, R1             // subtract 8 from SP in R1
208: LOAD R1, R2           // load argument from stack into R2
212: ....                  // instructions for body of GET_TIME
220: MOV time, R0          // move return value in R0
224: MOV SP, R1            // move stack pointer in R1
228: SUB 4, R1             // subtract 4 from SP in R1
232: LOAD R1, PC           // load return address from stack into PC
```

**FIGURE 4.2**

The procedure MEASURE (located at address 100) calls GET_TIME (located at address 200).

4. The caller transfers control to the callee by jumping to the address of its first instruction (address 140 and 144). (The callee, GET_TIME, is located at address 200.) The stack for our example looks now as in the following figure.

5. The callee loads its argument from the stack into R2 (address 200 through 208).
6. The callee computes with the arguments, perhaps calling other functions (address 212).
7. The callee loads the return value of GET_TIME into R0, the register the implementation reserves for returning values (address 220).
8. The callee loads the return address from the stack into PC (address 224 through 232), which causes the caller to resume control at address 148.
9. The caller adjusts the stack (address 148).
10. The caller restores content of R1 and R2 (addresses 152 through 164).

We use the low-level instructions of the processor for the specific example in Figure 4.2 because it exposes the fine print of the contract between the caller and the callee, and shows how errors can propagate. In the MEASURE example, the contract specifies that the callee returns the current time in some agreed-upon representation to the caller. If we look under the covers, however, we see that this functional specification is not the full contract and that the contract doesn't have a good way of limiting the propagation of errors. To uncover the fine print of the contract between modules, we need to inspect how the stack from Figure 4.2 is used to transfer control from one module to another. The contract between caller and callee contains several subtle potential problems:

- By contract, the caller and callee modify only shared arguments and their own variables in the stack. The callee leaves the stack pointer and the stack the way the caller has set it up. If there is a problem in the callee that corrupts the caller's area of the stack, then the caller might later compute incorrect results or fail.

- By contract, the callee returns where the caller told it to. If by mistake the callee returns somewhere else, then the caller probably performs an incorrect computation or loses control completely and fails.

- By contract, the callee stores return values in register `R0`. If by mistake the callee stores the return value somewhere else, then the caller will read whatever value is in register `R0` and probably perform an incorrect computation.
- By contract, the caller saves the values in the temporary registers (`R1`, `R2`, etc.) on the stack before the call to the callee and restores them when it receives control back. If the caller doesn't follow the contract, the callee may have changed the content of the temporary registers when the caller receives control back, and the caller probably performs an incorrect computation.
- Disasters in the callee can have side effects in the caller. For example, if the callee divides by zero and, as a result, terminates, the caller may terminate too. This effect is known colloquially as *fate sharing*.
- If the caller and callee share global variables, then by contract, the caller and callee modify only those global variables that are shared between them. Again, if the caller or callee modifies some other global variable, they (or other modules) might compute incorrectly or fail altogether.

Thus, the procedure call contract provides us with what might be labeled *soft modularity*. If a programmer makes an error or there is an error in the implementation of the procedure call convention, these errors can easily propagate from the callee to the caller. Soft modularity is usually attained through specifications, but nothing forces the interactions among modules to their defined interfaces. If the callee doesn't adhere (intentionally or unintentionally) to the contract, the caller has a serious problem. We have modularity that is not enforced.

There are also other possibilities for propagation of errors. The procedures share the same address space, and, if a defective procedure incorrectly smashes a global variable, even a procedure that did not call the defective one may be affected. Any procedure that doesn't adhere, either intentionally or unintentionally, to the contract may cause trouble for other modules.

Using a constrained and type-safe implementation language such as Java can beef up soft modularity to a certain extent (see Sidebar 4.1) but is insufficient for complete systems. For one, it is uncommon that all modules in a system are implemented in type-safe language. Often some modules of a system are for performance reasons written in a programming language that doesn't enforce modularity, such C, C++, or processor instructions. But even if the whole system is developed in a type-safe language like Java, we have a need for stronger modularity. If any of the Java modules raises an error (because the interpreter raises a type violation, the module allocated more memory than available, the module couldn't open a file, etc.) or has a programming error (e.g., an infinite loop), we would like to ensure that other modules don't immediately fail too. Even if a called procedure doesn't return, we would like to ensure that the caller has a controlled problem.

What we desire in systems is *enforced modularity*: modularity that is enforced by some external mechanism. This external mechanism limits the interaction among modules to the ones we desire. Such a limit on interactions reduces the number of

**Sidebar 4.1 Enforcing Modularity with a High-Level Languages** A high-level language is helpful in enforcing modularity because its compiler and runtime system perform all stack and register manipulation, presumably accurately and in accordance with the procedure calling convention. Furthermore, if the programming language enforces a restriction that programs write only to memory locations that correspond to variables managed by the language and in accordance with their type, then programs cannot overwrite arbitrary memory locations and, for example, corrupt the stack. That is, a program cannot use the value of a variable of type integer as an address of a memory location and then store to that memory location. Such languages are called *strongly typed* and, if a program cannot avoid the type system in any way, *type safe*. Modern examples of strongly typed languages include Java and C#.

But even with strongly typed languages, modularity through procedure calls doesn't limit the interactions between modules to their defined interfaces. For example, if the callee has a programming error and allocates all of the available memory space, then the caller may be unable to proceed. Also, strongly typed languages allow the programmer to escape the type system of the language to obtain direct access to memory or to processor registers and to exercise system features that the language does not support (e.g., reading and writing memory locations that correspond to the control registers and state of a device). But this access opens a path for the programmer to make mistakes that violate the procedure call contract.

Another concern is that in many computer systems different modules are written in different programming languages, perhaps because an existing, older module is being reused, even though its implementation language does not provide the type-safety features, or because a lower-level language fragment is essential for achieving maximum performance. Even when the caller and callee are written in two different, strongly typed languages, unexpected interactions can occur at their interface because their conventions do not match.

Another source of errors, which in practice seem to occur much less often, is an implementation error in the interpreter of the application (though with increasing complexity of compilers, runtime support systems, and processor designs, this source may yet become significant). The compiler may have a programming error, the runtime support system may have set up the stack incorrectly, the processor or operating system may save and restore registers incorrectly on an interrupt, a memory error causes a `LOAD` instruction to return an incorrect value, and so on. Although these sources are less likely to occur than programming errors, it is good to contain the resulting errors so that they don't propagate to other modules.

For all these reasons, designers use the client/service organization. Combining the client/service organization with writing a system in a strongly typed language offers additional opportunities for enforcing modularity; see, for example, the design of the Singularity operating system [Suggestions for Further Reading 5.2.3].

opportunities for propagation of errors. It also allows verification that a user uses a module correctly, and it helps prevent an attacker from penetrating the security of a module.

### 4.1.2 Client/Service Organization

One good way to enforce modularity is to limit the interactions among modules to explicit messages. It is convenient to impose some structure on this organization by identifying participants in a communication as clients or services.

Figure 4.3 shows a common interaction between client and service. The *client* is the module that initiates a request: it builds a message containing all the data necessary for the service to carry out its job and sends it to a service. The *service* is the module that responds: it extracts the arguments from the request message, executes the requested operations, builds a response message, sends the response message back to the client, and waits for the next request. The client extracts the results from the response message. For convenience, the message from the client to the service is called the *request*, and the message is called the *response* or *reply*.

Figure 4.3 shows one common way in which a client and a service interact: a request is always followed by a response. Since a client and a service can interact using many other sequences of messages, designers often represent the interactions using *message timing diagrams* (see Sidebar 4.2). Figure 4.3 is an instance of a simple timing diagram.

Conceptually, the client/service model runs client and services on separate computers, connected by a wire. This implementation also allows client and service to be separated geographically (which can be good because it reduces the risk that both fail owing to a common fault such as a power outage) and restricts all interactions to well-defined messages sent across a wire.

The disadvantage of this implementation is that it requires one computer per module, which may be costly in equipment. It may also have a performance cost because it may take a substantial amount of time to send a message from one computer to another, in particular if the computers are far away geographically. In some cases these disadvantages are unimportant; for cases in which it does matter, Chapter 5 will explain how to implement the client/service model within a single computer using

**FIGURE 4.3**
Communication between client and service.

**Sidebar 4.2 Representation: Timing Diagrams** A *timing diagram* is a convenient representation of the interaction between modules. When the system is organized in a client/service style, this presentation is particularly convenient, because the interactions between modules are limited to messages. In a timing diagram, the lifetime of a module is represented by a vertical line, with time increasing down the vertical axis. The following example illustrates the use of a timing diagram for a sewage pumping system. The label at the top of a timeline names the module (pump controller, sensor service, and pump service). The physical separation between modules is represented horizontally. Since it takes time for a message to get from one point to another, a message going from the pump controller to the pump service is represented by an arrow that slopes downward to the right.

The modules perform actions, and send and receive messages. The labels next to the time indicate actions taken by the module at a certain time. Modules can take actions at the same time, for example, if they are running on different processors.

The arrows indicate messages. The start of the arrow indicates the time the message is sent by the sending module, and the point of an arrow indicates the time the message is received at the destination module. The content of a message is described by the label associated with the arrow. In some examples, messages can be reordered (arrows cross) or lost (arrows terminate midflight before reaching a module).

The simple timing diagram shown in this sidebar describes the interaction between a pump controller and two services: a sensor service and a pump service. There is a request containing the message "measure tank level" from the client to the sensor service, and a response reports the level read by the sensor. There is a third message, "start pump", which the client sends to the pump service when the level is too high. The second message has no response. The diagram shows three actions: reading the sensor, deciding whether the pump must be started, and starting the pump. Figure 7.7 [on-line] shows a timing diagram with a lost message, and Figure 7.9 [on-line] shows one with a delayed message.

an operating system. For the rest of this chapter we will assume that the client and the service each have their own computer.

To achieve high availability or handle big workloads, a designer may choose to implement a service using multiple computers. For instance, a file service might use several computers to achieve a high degree of fault tolerance; if one computer fails, another one can take over its role. An instance of a service running on a single computer is called a *server*.

To make the client/service model more concrete, let's rearrange our MEASURE program into a simple client/service organization (see Figure 4.4). To get a time from the service, the client procedure builds a request message that names the service and specifies the requested operation and arguments (lines *2* and *6*). The requested operation and arguments must be converted to a representation that is suitable for transmission. For example, the client computer may be a big-endian computer (see Sidebar 4.3), while the service computer may be a little-endian computer. Thus, the client must convert arguments into a canonical representation so that the service can interpret the arguments.

This conversion is called *marshaling*. We use the notation {*a*, *b*} to denote a marshaled message that contains the fields *a* and *b*. Marshaling typically involves converting an object into an array of bytes with enough annotation so that the

Client program

```
1   procedure MEASURE (func)
2       SEND_MESSAGE (NameForTimeService, {"Get time", CONVERT2EXTERNAL(SECONDS)})
3       response ← RECEIVE_MESSAGE (NameForClient)
4       start ← CONVERT2INTERNAL (response)
5       func ()      // invoke the function
6       SEND_MESSAGE (NameForTimeService, {"Get time", CONVERT2EXTERNAL(SECONDS)})
7       response ← RECEIVE_MESSAGE (NameForClient)
8       end ← CONVERT2INTERNAL (response)
9       return end – start
```

Service program

```
10  procedure TIME_SERVICE ()
11      do forever
12          request ← RECEIVE_MESSAGE (NameForTimeService)
13          opcode ← GET_OPCODE (request)
14          unit ← CONVERT2INTERNAL(GET_ARGUMENT (request))
15          if opcode = "Get time" and (unit = SECONDS or unit = MINUTES) then
16              time ← CONVERT_TO_UNITS (CLOCK, unit)
17              response ← {"OK", CONVERT2EXTERNAL (time)}
18          else
19              response ←{"Bad request"}
20          SEND_MESSAGE (NameForClient, response)
```

**FIGURE 4.4**

Example client/service application: time service.

**Sidebar 4.3 Representation: Big-Endian or Little-Endian?** Two common conventions exist for numbering bits within a byte, bytes within a word, words within a page, and the like. One convention is called *big-endian*, and the other *little-endian*.* In big-endian the most significant bit, byte, or word is numbered 0, and the significance of bits *decreases* as the address of the bit increases:

| Words | 0 | | | | 1 | | | |
|---|---|---|---|---|---|---|---|---|
| Bytes | 0 | 1 | … | 7 | 0 | 1 | … | 7 |
| Bits | $2^0\ 2^1\ 2^2 \ldots$ | | | $\ldots 2^{63}$ | $2^0\ 2^1\ 2^2 \ldots$ | | | $\ldots 2^{63}$ |

In big-endian the hex number $ABCD_{hex}$ would be stored in memory, so that if you read from memory in increasing memory address order, you would see A-B-C-D. The string "john" would be stored in memory as john.

In little-endian, the other convention, the least significant bit, byte, or word is numbered 0, and the significance of bits *increases* as the address of the bit increases:

| Words | n | | | | n − 1 | | | |
|---|---|---|---|---|---|---|---|---|
| Bytes | 7 | … | 1 | 0 | 7 | … | 1 | 0 |
| Bits | $2^{63} \ldots$ | | | $\ldots 2^2\ 2^1\ 2^0$ | $2^{63} \ldots$ | | | $\ldots 2^2\ 2^1\ 2^0$ |

In little-endian, the hex number $ABCD_{hex}$ would be stored in memory, so that if you read from memory in increasing memory address order, you see D-C-B-A. The string "john" would still be stored in memory as john. Thus, code that extracts bytes from character strings transports between architectures, but code that extracts bytes from integers does not transport.

Some processors, such as the Intel x86 family, use the little-endian convention, but others, such as the IBM PowerPC family, use the big-endian convention. As Danny Cohen pointed out in a frequently cited article "*On holy wars and a plea for peace*" [Suggestions for Further Reading 7.2.4], it doesn't matter which convention a designer uses as long as it is the *same* one when communicating between two processors. The processors must agree on the convention for numbering the bits sent over the wire (that is, send the most significant bit first or send the least significant bit first). Thus, if the communication standard is big-endian (as it is in the Internet protocols), then a client running on a little-endian processor must marshal data in big-endian order. This book uses the big-endian convention.

This book also follows the convention that bit numbers start with zero. This choice is independent of the big-endian convention; we could have chosen to use 1 instead, as some processors do.

*The labels "big-endian" and "little-endian" were coined by Jonathan Swift in Chapter 4 of *Gulliver's Travels*, to identify two quarreling factions that differed over which end of an egg it was best to open.

*unmarshal* procedure can convert it back into a language object. In this example, we show the marshal and unmarshal operations explicitly (e.g., the procedure calls starting with CONVERT), but in many future examples these operations will be implicit to avoid clutter.

After constructing the request, the client sends it (*2* and *6*), waits for a response (line *3* and *7*), and unmarshals the time (*4* and *8*).

The service procedure waits for a request (line *12*) and unmarshals the request (lines *13* and *14*). Then, it checks the request (line *15*), processes it (lines *16* through *19*), and sends back a marshaled response (line *20*).

The *client/service organization* not only separates functions (abstraction), it also enforces that separation (enforced modularity). Compared to modularity using procedure calls, the client/service organization has the following advantages:

- The client and service don't rely on shared state other than the messages. Therefore, errors can propagate from the client to the service, and vice versa, in only one way. If the services (as in line *15*) and the clients check the validity of the request and response messages, then they can control the ways in which errors propagate. Since the client and service don't rely on global, shared data structures such as a stack, a failure in the client cannot directly corrupt data in the service, and vice versa.

- The transaction between a client and a service is an arm's-length transaction. Many errors cannot propagate from one to the other. For instance, the client does not have to trust the service to return to the appropriate return address, as it does using procedure calls. As another example, arguments and results are marshaled and unmarshaled, allowing the client and service to check them.

- The client can protect itself even against a service that fails to return because the client can put an upper limit on the time it waits for a response. As a result, if the service gets into an infinite loop, or fails and forgets about the request, the client can detect that something has gone wrong and undertake some recovery procedure, such as trying a different service. On the other hand, setting timers can create new problems because it can be difficult to predict how long a wait is reasonable. The problem of setting timers for service requests is discussed in detail in Section 7.5.2 [on-line]. In our example, the client isn't defensive against service errors; providing these defenses will make the program slightly more complex but can help eliminate fate sharing.

- Client/Service organization encourages explicit, well-defined interfaces. Because the client and service can interact only through messages, the messages that a service is willing to receive provide a well-defined interface for the service. If those messages are well specified and their specification is public, a programmer can implement a new client or service without having to understand the internals of another client or the service. Clear specification allows clients and service to be implemented by different programmers, and can encourage competition for the best implementation.

Separating state and passing well-defined messages reduce the number of potential interactions, which helps contain errors. If the programmer who developed the service introduces an error and the service has a disaster, the client has only a *controlled* problem. The client's only concern is that the service didn't deliver its part of the contract; apart from this wrong or missing value, the client has no concern for its own integrity. The client is less vulnerable from faults in the service, or, in slightly different words, fate sharing can be reduced. Clients can be mostly independent of service failures, and vice versa.

The client/service organization is an example of a ***sweeping simplification*** because the model eliminates all forms of interaction other than messages. By separating the client and the service from each other using message passing, we have created a firewall between them. As with firewalls in buildings, if there is a fire in the service, it will be contained in the service, and, assuming the client can check for flames in the response, it will not propagate to the client. If the client and service are well implemented, then the only way to go from the client to the service and back is through well-defined messages.

Of course, the client/service organization is not a panacea. If a service returns an incorrect result, then the client has a problem. This client can check for certain problems (e.g., syntactic ones) but not all semantic errors. The client/service organization reduces fate sharing but doesn't eliminate it. The degree to which the client/service organization reduces fate sharing is also dependent on the interface between the client and service. As an extreme example, if the client/service interface has a message that allows a client to write any value to any address in the service's address space, then it is easy for errors to propagate from the client to the service. It is the job of the system designer to define a good interface between client and service so that errors cannot propagate easily. In this chapter and later chapters, we will see examples of good message interfaces.

For ease of understanding, most of the examples in this chapter exhibit modules consisting of a single procedure. In the real world, designers usually apply the client/service organization between software modules of a larger granularity. The tendency toward larger granularity arises because the procedures within an application typically need to be tightly coupled for some practical reason, such as they all operate on the same shared data structure. Placing every procedure in a separate client or service would make it difficult to manipulate the shared data. The designer thus faces a trade-off between ease of accessing the data that a module needs and ease of error propagation within a module. A designer makes this trade-off by deciding which data and procedures to group into a coherent unit with the data that they manipulate. That coherent unit then becomes a separate service, and errors are contained within the unit. The client and service units are often complete application programs or similarly large subsystems.

Another factor in whether or not to apply the client/service organization to two modules is the plan for recovery when the service module fails. For example, in a simulator program that uses a function to compute the square root of its argument, it makes little sense to put that function into a separate service because it doesn't reduce fate sharing. If the square-root function fails, the simulator program cannot proceed. Furthermore, a good recovery plan is for the programmer to reimplement the function

correctly, as opposed to running two square-root servers, and failing over to the second one when the first one fails. In this example, the square-root function might as well be part of the simulator program because the client/service organization doesn't reduce fate sharing for the simulator program and thus there is no reason use it.

A nice example of a widely used system that is organized in a client/service style, with the client and service typically running on separate computers, is the World Wide Web. The Web browser is a client, and a Web site is a service. The browser and the site communicate through well-defined messages and are typically geographically separated. As long as the client and service check the validity of messages, a failure of a service results in a controlled problem for the browser, and vice versa. The World Wide Web provides enforced modularity.

In Figures 4.3 and 4.4, the service always responds with a reply, but that is not a requirement. Figure 4.5 shows the pseudocode for a pump controller for the sewage pumping system in Sidebar 4.2. In this example, there is no need for the pump service to send a reply acknowledging that the pump was turned off. What the client cares about is a confirmation from an independent sensor service that the level in the tank is going down. Waiting for a reply from the pump service, even for a short time, would just delay sounding the alarm if the pump failed.

Client program: pump controller

```
procedure PUMP_CONTROLLER ()
    do forever
        SEND_MESSAGE (NameForSensor, "measure tank level")
        level ← RECEIVE_MESSAGE (NameForClient)
        if level > UpperPumpLimit then SEND_MESSAGE (NameForPump, "turn on pump")
        if level < LowerPumpLimit then SEND_MESSAGE (NameForPump, "turn pump off")
        if level > OverflowLimit then SOUND_ALARM ()
```

Pump service

```
procedure PUMP_SERVICE ()
    do forever
        request ← RECEIVE_MESSAGE (NameForPump)
        if request = "Turn on pump" then SET_PUMP (on)
        else if request = "Turn off pump" then SET_PUMP (off)
```

Sensor service

```
procedure SENSOR_SERVICE ()
    do forever
        request ← RECEIVE_MESSAGE (NameForSensor)
        if request = "Measure tank level" then
            response ← READ_SENSOR ()
            SEND_MESSAGE (NameForClient, response)
```

**FIGURE 4.5**

Example client/service application: controller for a sewage pump.

**Sidebar 4.4 The X Window System** The X Window System [Suggestions for Further Reading 4.2.2] is the window system of choice on practically every engineering workstation and many personal computers. It provides a good example of using the client/service organization to achieve modularity. One of the main contributions of the X Window System is that it remedied a defect that had crept into the UNIX system when displays replaced typewriters: the display and keyboard were the only hardware-dependent parts of the UNIX application programming interface. The X Window System allowed display-oriented UNIX applications to be completely independent of the underlying hardware.

The X Window System achieved this property by separating the service program that manipulates the display device from the client programs that use the display. The service module provides an interface to manage windows, fonts, mouse cursors, and images. Clients can request services for these resources through high-level operations; for example, clients perform graphics operations in terms of lines, rectangles, curves, and the like. The advantage of this split is that the client programs are device independent. The addition of a new display type may require a new service implementation, but no application changes are required.

Another advantage of a client/service organization is that an application running on one machine can use the display on some other machine. This organization allows, for example, a computing-intensive program to run on a high-performance supercomputer, while displaying the results on a user's personal computer.

It is important that the service be robust to client failures because otherwise a buggy client could cause the entire display to freeze. The X Window system achieves this property by having client and service communicate through carefully designed remote procedure calls, a mechanism described in Section 4.2. The remote procedure calls have the property that the service never has to trust the clients to provide correct data and that the service can process other client requests if it has to wait for a client.

The service allows clients to send multiple requests back to back without waiting for individual responses because the rate at which data can be displayed on a local display is often higher than the network data rate between a client and service. If the client had to wait for a response on each request, then the user-perceived performance would be unacceptable. For example, at 80 characters per request (one line of text on a typical display) and a 5-millisecond round-trip time between client and service, only 16,000 characters per second can be drawn, while typical hardware devices are capable of displaying an order of magnitude faster.

Other systems avoid response messages for performance reasons. For example, the popular X Window System (see Sidebar 4.4) sends a series of requests that ask the service to draw something on a screen and that individually have no need for a response.

### 4.1.3 Multiple Clients and Services

In the examples so far, we have seen one client and one service, but the client/service model is much more flexible:

- One service can work for multiple clients. A printer service might work for many clients so that the cost of maintaining the printer can be shared. A file service might store files for many clients so that the information in the files can be shared.
- One client can use several services, as in the sewage pump controller (see Figure 4.5), which uses both a pump service and a sensor service.
- A single module can take on the roles of both client and service. A printer service might temporarily store documents on a file service until the printer is ready to print. In this case, the print service functions as a service for printing requests, but it is also a client of the file service.

### 4.1.4 Trusted Intermediaries

A single service that has multiple clients brings up another technique for enforcing modularity: the *trusted intermediary*, a service that functions as the trusted third party among multiple, perhaps mutually suspicious, clients. The trusted intermediary can control shared resources in a careful manner. For example, a file service might store files for multiple clients, some of which are mutually suspicious; the clients, however, trust the service to keep their affairs distinct. The file service could ensure that a client cannot have access to files not owned by that client, or it could, based on instructions from the clients, allow certain clients to share files.

The trusted intermediary enforces modularity among multiple clients and ensures that a fault in one client has limited (or perhaps no) effect on another client. If the trusted intermediary provides sharing of resources among multiple clients, then it has to be carefully designed and implemented to ensure that the failures of one client don't affect another client. For example, an incorrect update made by one client to its private files shouldn't affect the private files of another client.

A file service is only one example of a trusted intermediary. Many services in client/service applications are trusted intermediaries. E-mail services store mailboxes for many users so that individual users don't have to worry about losing their e-mail. As another example, instant message services provide private buddy lists. It is usually the clients that need some form of controlled sharing, and trusted intermediaries can provide that.

There are also situations in which intermediaries that do not have to be trusted are useful. For example, Section 4.2.3 describes how an untrusted intermediary can be used to buffer and deliver messages to multiple recipients. This use allows communication patterns other than request/response.

Another common use of trusted intermediaries is to simplify clients by having the trusted intermediary provide most functions. The buzzword in trade magazines for

**Sidebar 4.5 Peer-to-peer: Computing without Trusted Intermediaries** Peer-to-peer is a decentralized design that lacks trusted intermediaries. It is one of the oldest designs and has been used by, for example, the Internet e-mail system, the Internet news bulletin service, Internet service providers to route Internet packets, and IBM's Systems Network Architecture. Recently, it has received much attention in the popular press because file-sharing applications have rediscovered some of its advantages.

In a peer-to-peer application, every computer participating in the application is a peer and is equal in function (but perhaps not in capacity) to any other computer. That is, no peer is more important than any other peer; if one peer fails, then this failure may degrade the performance of the application, but it won't fail the application. The client/service organization doesn't have this property: if the service fails, the application fails, even if all client computers are operational.

UsenetNews is a good example of an older peer-to-peer application. UsenetNews, an on-line news bulletin, is one of the first peer-to-peer applications and has been operational since the 1980s. Users post to a newsgroup, from which other users read articles and respond. Nodes in UsenetNews propagate newsgroups to peers and serve articles to clients. An administrator of a node decides with which nodes the administrator's node peers. Because most nodes interconnect with several other nodes, the system is fault tolerant, and the failure of one node leads at most to a performance degradation rather than to a complete failure. Because the nodes are spread across the world in different jurisdictions, it is difficult for any one central authority to censor content (but an administrator of a node can decide not to carry a group). Because of these properties, designers have proposed organizing other applications in a peer-to-peer style. For example, LOCKSS [Suggestions for Further Reading 10.2.3] has built a robust digital library in that style.

Recently, music-sharing applications and improvements in technology have brought peer-to-peer designs into the spotlight. Today, client computers are as powerful as yesterday's computers for services and are connected with high data-rate links to the Internet. In music-sharing applications the clients are peers, and they serve and store music for one another. This organization aggregates the disk space and network links of all clients to provide a tremendous amount of storage and network capacity, allowing many songs to be stored and served. As often happens in the history of computer systems, the first version of this application was developed not by a computer scientist but by an 18-year-old, Shawn Fanning, who developed Napster. It (and its successors) has changed the characteristics of network traffic on the Internet and has raised legal questions as well.

In Napster, clients serve and store songs, but a trusted intermediary stores the location of a song. Because Napster was used for illegal music sharing, the Recording Industry Association of America (RIAA) sued the operators of the intermediary and was able to shut it down. In more recent peer-to-peer designs, developers adopted the design of censor-resistant applications and avoided the use of a trusted intermediary to locate

(*Sidebar continues*)

songs. In these successors to Napster, the peers locate music by querying other peers; if any individual node is shut down, it will not render the service unavailable. The RIAA must now sue individual users.

Accurately and quickly finding information in a large network of peers without a trusted intermediary is a difficult problem. Without an intermediary there is no central, well-known computer to track the locations of songs. A distributed algorithm is necessary to find a song. A simple algorithm is to send a query for a song to all neighbor peers; if they don't have a copy, the peers forward the query to their neighbors, and so on. This algorithm works, but it is inefficient because it sends a query to every node in the network. To avoid flooding the network of peers on each query, one can stop forwarding the query after it has been forwarded a number of times. Bounding a search in this way may cause some queries to return no answer, even though the song is somewhere in the network.

This problem has sparked interest in the research community, leading to the invention of better algorithms for decentralized search services and resulting in a range of new peer-to-peer applications. Some of these topics are covered in problem sets; see, for example, problem sets *20* [on-line] and *23* [on-line].

this use is "thin-client computing". In this use, only the trusted intermediary must run on a powerful computer (or a collection of computers connected by a high-speed network) because the clients don't run complex functions. Because in most applications there are a few trusted intermediaries (compared to the number of clients), they can be managed by a professional staff and located in a secure machine room. Trusted intermediaries of this type may be expensive to design, build, and run because they may need many resources to support the clients. If one isn't careful, the trusted intermediary can become a choke point during a flash crowd when many clients ask for the service at the same time. At the cost of additional complexity, this problem can be avoided by carefully dividing the work between clients and the trusted intermediary and replicating services using the techniques described in Chapters 8 [on-line] through 10 [on-line].

Designs that have trusted intermediaries also have some general downsides. The trusted intermediary may be vulnerable to failures or attacks that knock out the service. Users must trust the intermediary, but what if it is compromised or is subjected to censorship? Fortunately, there are alternative architectures; see Sidebar 4.5.

### 4.1.5 A Simple Example Service

Figure 4.6 shows the file system example of Figure 2.18 organized in a client/service style, along with the messages between the clients and services. The editor is a client of the file service, which in turn is a client of the block-storage service. The figure shows the message interaction among these three modules using a message timing diagram.

In the depicted example, the client constructs an OPEN message, specifying the name of the file to be opened. The service checks permissions and, if the user is allowed access, sends a response back indicating success (OK) and the value of the file pointer (0) (see Section 2.3.2 for an explanation of a file pointer). The client writes text to the file, which results in a WRITE request that specifies the text and the number of bytes to be written. The service writes the file by allocating blocks on the block service, copies the specified bytes into them, and returns a message stating the number of bytes written (16). After receiving a response from the block service, it constructs a response for the client, indicating success and informing the client of the new value of the file pointer. When the client is done editing, the client sends a CLOSE message, telling the service that the client is finished with this file.

This message sequence is too simple for use in practice because it doesn't deal with failures (e.g., what happens if the service fails while processing a write request), concurrency (e.g., what happens if multiple clients update a shared file), or security (e.g., how to ensure that a malicious person cannot write the business plan). A file service that is almost this simple is the Woodstock File System (WFS), designed by researchers at the Xerox Palo Alto Research Center [Suggestions for Further Reading 4.2.1]. Section 4.5 is a case study of a widely used successor, the Network File System (NFS), which is organized as a client/service application, and summarizes how NFS handles failures and concurrency. Handling concurrency, failures, and security in general are topics we explore in a systematic way in later chapters.

The file service is a trusted intermediary because it protects the content of files. It must check whether the messages came from a legitimate client (and not from an attacker), it decides whether the client has permission to perform the requested operation, and, if so, it performs the operation. Thus, as long as the file service does its job correctly, clients can share files (and thus also the block-storage service) in a protected manner.

**FIGURE 4.6**

File service using message timing diagram.

## 4.2 COMMUNICATION BETWEEN CLIENT AND SERVICE

This section describes two extensions to sending and receiving messages. First, it introduces *remote procedure call (RPC),* a stylized form of client/service interaction in which each request is followed by a response. The goal of RPC systems is to make a remote procedure call look like an ordinary procedure call. Because a service fails independently from a client, however, a remote procedure call can generally not offer identical semantics to procedure calls. As explained in the next subsection, some RPC systems provide various alternative semantics and the programmer must be aware of the details.

Second, in some applications it is desirable to be able to send messages to a recipient that is not on-line and to receive messages from a sender that is not on-line. For example, electronic mail allows users to send e-mail without requiring the recipient to be on-line. Using an intermediary for communication, we can implement these applications.

### 4.2.1 Remote Procedure Call (RPC)

In many of the examples in the previous section, the client and service interact in a stylized fashion: the client sends a request, and the service replies with a response after processing the client's request. This style is so common that it has received its own name: *remote procedure call*, or RPC for short.

RPCs come in many varieties, adding features to the basic request/response style of interaction. Some RPC systems, for example, simplify the programming of clients and services by hiding many the details of constructing and formatting messages. In the time service example above, the programmer must call SEND_MESSAGE and RECEIVE_MESSAGE, and convert results into numbers, and so on. Similarly, in the file service example, the client and service have to construct messages and convert numbers into bit strings and the like. Programming these conversions is tedious and error prone.

*Stubs* remove this burden from the programmer (see Figure 4.7). A stub is a procedure that hides the marshaling and communication details from the caller and callee. An RPC system can use stubs as follows. The client module invokes a remote procedure, say GET_TIME, in the same way that it would call any other procedure. However, GET_TIME is actually just the name of a stub procedure that runs inside the client module (see Figure 4.8). The stub marshals the arguments of a call into a message, sends the message, and waits for a response. On arrival of the response, the client stub unmarshals the response and returns to the caller.

Similarly, a service stub waits for a message, unmarshals the arguments, and calls the procedure that the client requests (GET_TIME in the example). After the procedure returns, the service stub marshals the results of the procedure call into a message and sends it in a response to the client stub.

Writing stubs that convert more complex objects into an appropriate on-wire representation becomes quite tedious. Some high-level programming languages

**FIGURE 4.7**
Implementation of a remote procedure call using stubs. The stubs hide all remote communication from the caller and callee.

Client program

```
procedure MEASURE (func)
    start ← GET_TIME (SECONDS)
    func ()        // invoke the function
    end ← GET_TIME (SECONDS)
    return end – start

procedure GET_TIME (unit)        // the client stub for GET_TIME
    SEND_MESSAGE (NameForTimeService, {"Get time", unit})
    response ← RECEIVE_MESSAGE (NameForClient)
    return CONVERT2INTERNAL (response)
```

Service program

```
procedure TIME_SERVICE ()        // the service stub for GET_TIME
    do forever
        request ← RECEIVE_MESSAGE (NameForTimeService)
        opcode ← GET_OPCODE (request)
        unit ← GET_ARGUMENT (request)
        if opcode = "Get time" and (unit = SECONDS or unit = MINUTES) then
            response ← {"ok", GET_TIME (unit)}
        else
            response ← {"Bad request"}
        SEND_MESSAGE (NameForClient, response)
```

**FIGURE 4.8**
GET_TIME client and service using stubs.

such as Java can generate these stubs automatically from an interface specification [Suggestions for Further Reading 4.1.3], simplifying client/service programming even further. Figure 4.9 shows the client for such an RPC system. The RPC system would generate a procedure similar to the GET_TIME stub in Figure 4.8. The client program of Figure 4.9 looks almost identical to the one using a local procedure call on page 149,

The client program

```
1    procedure MEASURE (func)
2        try
3            start ← GET_TIME (SECONDS)
4        catch (signal servicefailed)
5            return servicefailed
6        func ()      // invoke the function
7        try
8            end ← GET_TIME (SECONDS)
9        catch (signal servicefailed)
10           return servicefailed
11       return end – start
```

**FIGURE 4.9**
GET_TIME client using a system that generates RPC stubs automatically.

except that it handles an additional error because remote procedure calls are not identical to procedure calls (as discussed below). The procedure that the service calls on line 7 is just the original procedure GET_TIME on page 149.

Whether a system uses RPC with automatic stub generation is up to the implementers. For example, some implementations of Sun's Network File System (see Section 4.5) use automatic stub generation, but others do not.

### 4.2.2 RPCs are not Identical to Procedure Calls

It is tempting to think that by using stubs one can make a remote procedure call behave exactly the same as an ordinary procedure call, so that a programmer doesn't have to think about whether the procedure runs locally or remotely. In fact, this goal was a primary one when RPC was originally proposed—hence the name remote "procedure call". However, RPCs are different from ordinary procedure calls in three important ways: First, RPCs can reduce fate sharing between caller and callee by exposing the failures of the callee to the caller so that the caller can recover. Second, RPCs introduce new failures that don't appear in procedure calls. These two differences change the semantics of remote procedure calls as compared with ordinary procedure calls, and the changes usually require the programmer to make adjustments to the surrounding code. Third, remote procedure calls take more time than procedure calls; the number of instructions to invoke a procedure (see Figure 4.2) is much less than the cost of invoking a stub, marshaling arguments, sending a request over a network, invoking a service stub, unmarshaling arguments, marshaling the response, receiving the response over the network, and unmarshaling the response.

To illustrate the first difference, consider writing a procedure call to the library program SQRT, which computes the square root of its argument $x$. A careful programmer would plan for the case that SQRT ($x$) will fail when $x$ is negative by providing an explicit exception handler for that case. However, the programmer using ordinary procedure calls almost certainly doesn't go to the trouble of planning for certain possible failures because they have negligible probability. For

example, the programmer probably would not think of setting an interval timer when invoking SQRT (*x*), even though SQRT internally has a successive-approximation loop that, if programmed wrong, might not terminate.

But now consider calling SQRT with an RPC. An interval timer suddenly becomes essential because the network between client and service can lose a message, or the other computer can crash independently. To avoid fate sharing, the RPC programmer must adjust the code to prepare for and handle this failure. When the client receives a "service failure" signal, the client may be able to recover by, for example, trying a different service or choosing an alternative algorithm that doesn't use a remote service.

The second difference between ordinary procedure calls and RPCs is that RPCs introduce a new failure mode, the "no response" failure. When there is no response from a service, the client cannot tell which of two things went wrong: (1) some failure occurred before the service had a chance to perform the requested action, or (2) the service performed the action and then a failure occurred, causing just the response to be lost.

Most RPC designs handle the no-response case by choosing one of three implementation strategies:

- *At-least-once* RPC. If the client stub doesn't receive a response within some specific time, the stub resends the request as many times as necessary until it receives a response from the service. This implementation may cause the service to execute a request more than once. For applications that call SQRT, executing the request more than once is harmless because with the same argument SQRT should always produce the same answer. In programming language terms, the SQRT service has no side effects. Such side-effect-free operations are also *idempotent*: repeating the same request or sequence of requests several times has the same effect as doing it just once. An at-least-once implementation does not provide the guarantee implied by its name. For example, if the service was located in a building that has been blown away by a hurricane, retrying doesn't help. To handle such cases, an at-least-once RPC implementation will give up after some number of retries. When that happens, the request may have been executed more than once or not at all.

- *At-most-once* RPC. If the client stub doesn't receive a response within some specific time, then the client stub returns an error to the caller, indicating that the service may or may not have processed the request. At-most-once semantics may be more appropriate for requests that do have side effects. For example, in a banking application, using at-least-once semantics for a request to transfer $100 from one account to another could result in multiple $100 transfers. Using at-most-once semantics assures that either zero or one transfers take place, a somewhat more controlled outcome. Implementing at-most-once RPC is harder than it sounds because the underlying network may duplicate the request message without the client stub's knowledge. Chapter 7 [on-line] describes an at-most-once implementation, and Birrell and Nelson's paper gives

a nice, complete description of an RPC system that implements at-most-once [Suggestions for Further Reading 4.1.1].

- *Exactly-once* RPC. These semantics are the ideal, but because the client and service are independent it is in principle impossible to guarantee. As in the case of at-least-once, if the service is in a building that was blown away by a hurricane, the best the client stub can do is return error status. On the other hand, by adding the complexity of extra message exchanges and careful record-keeping, one can approach exactly-once semantics closely enough to satisfy some applications. The general idea is that, if the RPC requesting transfer of $100 from account A to B produces a "no response" failure, the client stub sends a separate RPC request to the service to ask about the status of the request that got no response. This solution requires that both the client and the service stubs keep careful records of each remote procedure call request and response. These records must be fault tolerant because the computer running the service might fail and lose its state between the original RPC and the inquiry to check on the RPC's status. Chapters 8 [on-line] through 10 [on-line] introduce the necessary techniques.

The programmer must be aware that RPC semantics differ from those of ordinary procedure calls, and because different RPC systems handle the no-response case in different ways, it is important to understand just which semantics any particular RPC system tries to provide. Even if the name of the implementation implies a guarantee (e.g., at-least-once), we have seen that there are cases in which the implementation cannot deliver it. One cannot simply take a collection of legacy programs and arbitrarily separate the modules with RPC. Some thought and reprogramming is inevitably required. Problem set *2* explores the effects of different RPC semantics in the context of a simple client/service application.

The third difference is that calling a local procedure takes typically much less time than calling a remote procedure call. For example, invoking a remote `SQRT` is likely to be more expensive than the computation for `SQRT` itself because the overhead of a remote procedure call is much higher than the overhead of following the procedure calling conventions. To hide the cost of a remote procedure call, a client stub may deploy various performance-enhancing techniques (see Chapter 6), such as caching results and pipelining requests (as is done in the X Window System of Sidebar 4.4). These techniques increase complexity and can introduce new problems (e.g., how to ensure that the cache at the client stays consistent with the one at the service). The performance difference between procedure calls and remote procedure calls requires the designer to consider carefully what procedure calls should be remote ones and which ones should be ordinary, local procedure calls.

A final difference between procedure calls and RPCs is that some programming language features don't combine well with RPC. For example, a procedure that communicates with another procedure through global variables cannot typically be executed remotely because separate computers usually have separate address spaces. Similarly, other language constructs that use explicit addresses won't work. Arguments

consisting of data structures that contain pointers, for example, are a problem because pointers to objects in the client computer are local addresses that have different bindings when resolved in the service computer. It is possible to design systems that use global references for objects that are passed by reference to remote procedure calls but require significant additional machinery and introduce new problems. For example, a new plan is needed for determining whether an object can be deleted locally because a remote computer might still have a reference to the object. Solutions exist, however; see, for example, the article on Network Objects [Suggestions for Further Reading 4.1.2].

Since RPCs don't provide the same semantics as procedure calls, the word "procedure" in "remote procedure call" can be misleading. Over the years the concept of RPC has evolved from its original interpretation as an exact simulation of an ordinary procedure call to instead mean any client/service interaction in which the request is followed by a response. This text uses this modern interpretation.

### 4.2.3 Communicating through an Intermediary

Sending a message from a sender to a receiver requires that both parties be available at the same time. In many applications this requirement is too strict. For example, in electronic mail we desire that a user be able to send an e-mail to a recipient even if the recipient is not on-line at the time. The sender sends the message and the recipient receives the message some time later, perhaps when the sender is not on-line. We can implement such applications using an intermediary. In the case of communication, this intermediary doesn't have to be trusted because communication applications often consider the intermediary to be part of an untrusted network and have a separate plan for securing messages (as we will see in Chapter 11 [on-line]).

The primary purpose of the e-mail intermediary is to implement *buffered* communication. Buffered communication provides the SEND/RECEIVE abstraction but avoids the requirement that the sender and receiver be present simultaneously. It allows the delivery of a message to be shifted in time. The intermediary can hold messages until the recipient comes on-line. The intermediary might buffer messages in volatile memory or in non-volatile memory, such as a file system. The latter design allows the intermediary to buffer messages across power failures.

Once we have an intermediary, three interesting design opportunities arise. First, the sender and receiver may make different choices of whether to *push* or *pull* messages. Push is when the initiator of a data movement sends the data. Pull is when the initiator of a data movement asks the other end to send it the data. These definitions are independent of whether or not the system uses an intermediary, but in systems with intermediaries it is not uncommon to find both in a single system. For example, the sender in the Internet's e-mail system, Simple Mail Transfer Protocol (SMTP), pushes the mail to the service that holds the recipient's mailbox. On the other hand, the receiving client pulls messages to fetch mail from a mailbox: the user hits the

"Get new mail" button, which causes the mail client to contact the mailbox service and ask it for any new mail.

Second, the existence of an intermediary opens an opportunity to apply the design principle ***decouple modules with indirection*** by having the intermediary, rather than the originator, determine to whom a message is delivered. For example, an Internet user can send a message to `president@whitehouse.gov`. The intermediary that forwards the message will deliver it to whoever happens to be the President. As another example, users should be able to send an e-mail to a mailing list or to post a message to a bulletin board without knowing exactly who is on the mailing list or subscribed to the bulletin board.

Third, when indirection through an intermediary is available, the designer has a choice of when and where to duplicate messages. In the mailing list example, the intermediary sends a copy of the e-mail to each member of the list. In the bulletin board example, an intermediary may group messages and send them as a group to other intermediaries. When a user fetches the bulletin article from its local intermediary, the local intermediary makes a final copy for delivery to the user.

*Publish/subscribe* is a general style of communication that takes advantage of the three design opportunities of communication through an intermediary. In this communication model, the sender is called the publisher and notifies an event service that it has produced a new message on a certain topic. Recipients subscribe to the event service and express their interest in certain topics. If multiple recipients are interested in the same topic, all of them receive a copy of the message. Popular usages of publish/subscribe are electronic mailing lists and instant messaging services that provide chat rooms. A user might join a chat room on a certain topic. When another user publishes a message in the room, all the members of that room receive it. Another publish/subscribe application is Usenet News, a bulletin board service (described in Sidebar 4.5 on peer-to-peer computing).

## 4.3 SUMMARY AND THE ROAD AHEAD

The client/service model enforces modularity and is the basic approach to organizing complex computer systems. The rest of the book works out major issues in building computer systems that this chapter has identified but has not addressed:

- *Enforcing modularity within a computer* (Chapter 5). Restricting the implementation of client/service systems to one computer per module can be too expensive. Chapter 5 shows how an operating system can use a technique called virtualization to create many virtual computers out of one physical computer. The operating system can enforce modularity between each client and each service by giving each client and each service a separate virtual computer.

- *Performance* (Chapter 6). Computer systems have implicit or explicit performance goals. If services are not carefully designed, it is possible that the slowest

service in the system becomes a performance bottleneck, which causes the complete system to operate at the performance of the slowest service. Identifying performance bottlenecks and avoiding them is a challenge that a designer faces in most computer systems.

- *Networking* (Chapter 7 [on-line]). The client/service model must have a way to send the request message from the client to the service, and the response message back. Implementing SEND_MESSAGE and RECEIVE_MESSAGE is a challenging problem, since networks may lose, reorder, or duplicate messages while routing them between the client and the service. Furthermore, networks exhibit a wide range of performance properties, making straightforward solutions inadequate.

- *Fault tolerance* (Chapter 8 [on-line]). We may need for a service to continue to operate even if some of the hardware and software modules fail. For example, we may want to construct a fault tolerant date-and-time service that runs on several computers so that if one of the computers fails, another computer can still deliver a response to requests for the date and time. In systems that harness a large number of computers to deliver a single service, it is unavoidable that at any instant of time some of the computers will have failed. For example, Google, which indexes the Web, reportedly uses more than 100,000 computers to deliver the service. (A description of the systems Google has designed can be found in Suggestions for Further Reading 3.2.4 and 10.1.10.) With so many computers, some of them are certain to be unavailable. Techniques for fault tolerance allow designers to implement reliable services out of unreliable components. These techniques involve detecting failures, containing them, and recovering from them.

- *Atomicity* (Chapter 9 [on-line]). The file service described in this chapter (Figure 4.6 in Section 4.1.6) must work correctly in the face of concurrent access and failures, and use OPEN and CLOSE calls to mark related READ and WRITE operations. Chapter 9 [on-line] introduces a single framework called atomicity that addresses both issues. This framework allows the operations between an OPEN and CLOSE call to be executed as an atomic, indivisible action. As we saw in Section 4.2.2, exactly-once RPC is ideal for implementing a banking application. Chapter 9 [on-line] introduces the necessary tools for exactly-once RPC and building such applications.

- *Consistency* (Chapter 10 [on-line]). This chapter uses messages to implement various protocols to ensure consistency of data stores on different computers.

- *Security* (Chapter 11 [on-line]). The client/service model protects against accidental errors propagating from one module to another module. Some services may need to protect against malicious attacks. This requirement arises, for example, when a file service is storing sensitive data and needs to ensure that malicious users cannot read the sensitive data. Such protection requires that the service reliably identify users so that it can make an authorization decision. The design of systems in the face of malicious users is a topic known as *security*.

The subsystems that address these topics are interesting systems in their own right and are case studies of managing complexity. Typically, these subsystems are internally structured as client/service systems, applying the concept of this chapter recursively. The next two sections provide two case studies of real-world client/service systems and also illustrate the need for the topics addressed in the subsequent chapters.

## 4.4 CASE STUDY: THE INTERNET DOMAIN NAME SYSTEM (DNS)

The Internet Domain Name System (DNS) provides an excellent case study of both a client/service application and a successful implementation of a naming scheme, in this case for naming of Internet computers and services. Although designed for that specific application, DNS is actually a general-purpose name management and name resolution system that hierarchically distributes the management of names among different naming authorities and also hierarchically distributes the job of resolving names to different name servers. Its design allows it to respond rapidly to requests for name resolution and to scale up to extremely large numbers of stored records and numbers of requests. It is also quite resilient, in the sense that it provides continued, accurate responses in the face of many kinds of network and server failures.

The primary use for DNS is to associate user-friendly character-string names, called *domain names*, with machine-oriented binary identifiers for network attachment points, called *Internet addresses*. Domain names are hierarchically structured, the term *domain* being used in a general way in DNS: it is simply a set of one or more names that have the same hierarchical ancestor. This convention means that hierarchical regions can be domains, but it also means that the personal computer on your desk is a domain with just one member. In consequence, although the phrase "domain name" suggests the name of a hierarchical region, every name resolved by DNS is called a domain name, whether it is the name of a hierarchical region or the name of a single attachment point. Because domains typically correspond to administrative organizations, they also are the unit of delegation of name assignment, using exactly the hierarchical naming scheme described in Section 3.1.4.

For our purposes, the basic interface to DNS is quite simple:

*value* ← DNS_RESOLVE (*domain_name*)

This interface omits the context argument from the standard name-resolving interface of the naming model of Section 2.2.1 because there is just a single, universal, default context for resolving all Internet domain names, and the reference to that one context is built into DNS_RESOLVE as a configuration parameter.

In the usual DNS implementation, binding is not accomplished by invoking BIND and UNBIND procedures as suggested by our naming model, but rather by using a text editor or database generator to create and manage tables of bindings. These tables are then loaded into DNS servers by some behind-the-scenes method as often as their managers deem necessary. One consequence of this design is that changes to DNS

bindings don't often occur within seconds of the time you request them; instead, they typically take hours.

Domain names are path names, with components separated by periods (called *dots,* particularly when reading domain names aloud) and with the least significant component coming first. Three typical domain names are

```
ginger.cse.pedantic.edu    ginger.scholarly.edu    ginger.com
```

DNS allows both relative and absolute path names. Absolute path names are supposed to be distinguished by the presence of a trailing dot. In human interfaces the trailing dot rarely appears; instead, DNS_RESOLVE applies a simple form of multiple lookup. When presented with a relative path name, DNS_RESOLVE first tries appending a default context, supplied by a locally set configuration parameter. If the resulting extended name fails to resolve, DNS_RESOLVE tries again, this time appending just a trailing dot to the originally presented name. Thus, for example, if one presents DNS_RESOLVE with the apparently relative path name "ginger.com", and the default context is "pedantic.edu.", DNS_RESOLVE will first try to resolve the absolute path name "ginger.com.pedantic.edu.". If that attempt leads to a NOT-FOUND result, it will then try to resolve the absolute path name "ginger.com."

### 4.4.1 Name Resolution in DNS

DNS name resolution might have been designed in at least three ways:

1. *The telephone book model:* Give each network user a copy of a file that contains a list of every domain name and its associated Internet address. This scheme has a severe problem: to cover the entire Internet, the size of the file would be proportional to the number of network users, and updating it would require delivering a new copy to every user. Because the frequency of update tends to be proportional to the number of domain names listed in the file, the volume of network traffic required to keep it up to date would grow with the cube of the number of domain names. This scheme was used for nearly 20 years in the Internet, was found wanting, and was replaced with DNS in the late 1980s.

2. *The central directory service model:* Place the file on a single well-connected server somewhere in the network and provide a protocol to ask it to resolve names. This scheme would make update easy, but with growth in the number of users its designer would have to adopt increasingly complex strategies to keep it from becoming both a performance bottleneck and a potential source of massive failure. There is yet another problem: whoever controls the central server is by default in charge of all name assignment. This design does not cater well to delegation of responsibility in assignment of domain names.

3. *The distributed directory service model.* The idea is to have many servers, each of which is responsible for resolving some subset of domain names, and a protocol for finding a server that can resolve any particular name. As we shall see in the following descriptions, this model can provide delegation and respond

to increases in scale while maintaining reliability and performance. For those reasons, DNS uses this model.

With the distributed directory service model, the operation of every name server is the same: a server maintains a set of name records, each of which binds a domain name to an Internet address. When a client sends a request for a name resolution, the name server looks through the collection of domain names for which it is responsible, and if it finds a name record, it returns that record as its response. If it does not find the requested name, it looks through a separate set of referral records. Each referral record binds a hierarchical region of the DNS name space to some other name server that can help resolve names in that region of the naming hierarchy. Starting with the most significant component of the requested domain name, the server searches through referral records for the one that matches the most components, and it returns that referral record. If nothing matches, DNS cannot resolve the original name, so it returns a "no such domain" response.

The referral architecture of DNS, though conceptually simple, has a number of elaborations that enhance its performance, scalability, and robustness. We begin with an example of its operation in a simple case, and we later add some of the enhancements. The dashed lines in Figure 4.10 illustrate the operation of DNS when the client computer named `ginger.cse.pedantic.edu`, in the lower left corner, tries to resolve the domain name `ginger.Scholarly.edu`. The first step, shown as request #1, is that `DNS_RESOLVE` sends that domain name to a *root name server*, whose Internet address it somehow knows. Section 4.4.4 explains how `DNS_RESOLVE` discovers that address.

The root name server matches the name in the request with the subset of domain names it knows about, starting with the most significant component of the requested domain name (in this example, `edu`). In this example, the root name server discovers that it has a referral record for the domain `edu`, so it responds with a referral, saying, in this example, "There is a name server for a domain named `edu`. The name record for that name server binds the name `names.edu.` to Internet address 192.14.71.191." This response illustrates that name servers, like any other servers, have both domain names and Internet addresses. Usually, the domain name of a name server gives some clue about what domain names it serves, but there is no necessary correspondence. Responding with a complete name record provides more information than the client really needs (the client usually doesn't care about the name of the name server), but it allows all responses from a name server to be uniform. Because the name server's domain name isn't significant and to reduce clutter in Figure 4.10, that figure omits it in the illustrated response.

When the client's `DNS_RESOLVE` receives this response, it immediately resends the same name resolution request, but this time it directs the request (request 2 in the figure) to the name server located at the Internet address mentioned in response number 1. That name server matches the requested path name with the set of domain names it knows about, again starting with the most significant component. In this case, it finds a match for the name `Scholarly.edu.` in a referral record. It thus sends back a response saying, "There is a name server for a domain named `Scholarly.edu`. The

**FIGURE 4.10**

Structure and operation of the Internet Domain Name System. In this figure, each circle represents a name server, and each rectangle is a name client. The type NS in a table or in a response means that this is a referral to another name server, while the type AP in a table or a response means that this is an Internet address. The dashed lines show the paths of the three requests made by the name client in the lower left corner to resolve the name `ginger.Scholarly.edu`, starting with the root name server. The dotted lines show resolution of a request of the name client in the lower right corner to resolve `anise.pedantic.edu` starting with a local name server.

name record for that name server binds the name `ns.iss.edu.` to Internet address 128.32.136.9." The illustration again omits the domain name of the name server.

This sequence repeats for each component of the original path name, until `DNS_RESOLVE` finally reaches a name server that has the name record for `ginger.Scholarly.edu`. That name server sends back a response saying, "The name record for `ginger.Scholarly.edu.` binds that name to Internet address 169.229.2.16." This being the answer to the original query, `DNS_RESOLVE` returns this result to its caller, which can go on to initiate an exchange of messages with its intended target.

The server that holds either a name record or a referral record for a domain name is known as the *authoritative name server* for that domain name. In our example, the name server `ns3.cse.pedantic.edu.` is authoritative for the `ginger.cse.pedantic.edu.` domain, as well as all other domain names that end with `cse.pedantic.edu.`, and `ns.iss.edu.` is authoritative for the `Scholarly.edu.` domain. Since a name server does not hold the name record for its own name, a name server cannot be the authoritative name server for its own name. Instead, for example, the root name server is authoritative for the domain name `edu.`, while the `names.edu.` name server is authoritative for all domain names that end in `edu`.

That is the basic model of DNS operation. Here are some elaborations in its operation, each of which helps make the system fast-responding, robust, and capable of growing to a large scale.

1. It is not actually necessary to send the initial request to the root name server. `DNS_RESOLVE` can send the request to *any* convenient name server whose Internet address it knows. The name server doesn't care where the request came from; it simply compares the requested domain name with the list of domain names for which it is responsible in order to see if it holds a record that can help. If it does, it answers the request. If it doesn't, it answers by returning a referral to a root name server. The ability to send any request to a local name server means that the common case in which the client, the name server, and the target domain name are all three in the same domain (e.g., `pedantic.edu`) can be handled swiftly with a single request/response interaction. (The dotted lines in the lower right corner of Figure 4.10 show an example, in which `thyme.pedantic.edu.` asks the name server for the `pedantic.edu` domain for the address of `anise.pedantic.edu.`) This feature also simplifies name discovery because all a client needs to know is the Internet address of any nearby name server. The first request to that nearby server for a distant name (in the current example, `ginger.scholarly.edu`) will return a referral to the Internet address of a root name server.

2. Some domain name servers offer what is (perhaps misleadingly) called *recursive* name service. If the name server does not hold a record for the requested name, rather than sending a referral response, the name server takes on the responsibility for resolving the name itself. It forwards the initial request to a root name server, then continues to follow the chain of responses to resolve the complete path name, and finally returns the desired name record to its client. By

itself, this feature seems merely to simplify life for the client, but in conjunction with the next feature it provides a major performance enhancement.

3. Every name server is expected to maintain, in addition to its authoritative records, a cache of all name records it has heard about from other name servers. A server that provides recursive name service thus collects records that can greatly speed up future name resolution requests. If, for example, the name server for `cse.pedantic.edu` offers recursive service and it is asked to resolve the name `flower.cs.scholarly.edu`, in the course of doing so (assuming that it does not in turn request recursive service), its cache might acquire the following records:

```
edu                 refer to names.edu at 198.41.0.4
Scholarly.edu       refer to ns.iss.edu at 128.32.25.19
cs.Scholarly.edu    refer to cs.Scholarly.edu at 128.32.247.24
flower.cs.Scholarly.edu       Internet address is 128.32.247.29
```

Now, when this name server receives, for example, the request to resolve the name `psych.Scholarly.edu`, it will discover the record for the domain `Scholarly.edu` in the cache and it will be able to quickly resolve the name by forwarding the initial request directly to the corresponding name server.

A cache holds a duplicate copy, which may go out of date if someone changes the authoritative name record. On the basis that changes of existing name bindings are relatively infrequent in the Domain Name System and that it is hard to keep track of all the caches to which a domain name record may have propagated, the DNS design does not call for explicit invalidation of changed entries. Instead, it uses expiration. That is, the naming authority for a DNS record marks each record that it sends out with an expiration period, which may range from seconds to months. A DNS cache manager is expected to discard entries that have passed their expiration period. The DNS cache manager provides a memory model that is called *eventual consistency,* a topic taken up in Chapter 10 [on-line].

### 4.4.2 Hierarchical Name Management

Domain names form a hierarchy, and the arrangement of name servers described above matches that hierarchy, thereby distributing the job of name resolution. The same hierarchy also distributes the job of managing the handing out of names, by distributing the responsibility of operating name servers. Distributing responsibility is one of the main virtues of the distributed directory service model.

The way this works is actually quite simple: whoever operates a name server can be a *naming authority,* which means that he or she may add authoritative records to that name server. Thus, at some point early in the evolution of the Internet, some Pedantic University network administrator deployed a name server for the domain `pedantic.edu` and convinced the administrator of the `edu` domain to install a binding for the domain name `pedantic.edu.` associated with the name and Internet

address of the `pedantic.edu` name server. Now, if Pedantic University wants to add a record, for example, for an Internet address that it wishes to name `archimedes.pedantic.edu`, its administrator can do so without asking permission of anyone else. A request to resolve the name `archimedes.pedantic.edu` can arrive at any domain name server in the Internet; that request will eventually arrive at the name server for the `pedantic.edu` domain, where it can be answered correctly. Similarly, a network administrator at the Institute for Scholarly Studies can install a name record for an Internet address named `archimedes.Scholarly.edu` on its own authority. Although both institutions have chosen the name `archimedes` for one of their computers, because the path names of the domains are distinct there was no need for their administrators to coordinate their name assignments. Put another way, their naming authorities can act independently.

Continuing this method of decentralization, any organization that manages a name server can create lower-level naming domains. For example, the Computer Science and Engineering Department of Pedantic University may have so many computers that it is convenient for the department to manage the names of those computers itself. All that is necessary is for the department to deploy a name server for a lower-level domain (named, for example, `cse.pedantic.edu`) and convince the administrator of the `pedantic.edu` domain to install a referral record for that name in its name server.

### 4.4.3 Other Features of DNS

To ensure high availability of name service, the DNS specification calls on every organization that runs a name service to arrange that there be at least two identical replica servers. This specification is important, especially at higher levels of the domain naming hierarchy, because most Internet activity uses domain names and inability to resolve a name component blocks reachability to all sites below that name component. Many organizations have three or four replicas of their name servers, and as of 2008 there were about 80 replicas of the root name server. Ideally, replicas should be attached to the network at places that are widely separated, so that there is some protection against local network and electric power outages. Again, the importance of separated attachment increases at higher levels of the naming hierarchy. Thus, the 80 replicas of the root name server are scattered around the world, but the three or four replicas of a typical organization's name server are more likely to be located within the campus of that organization. This arrangement ensures that, even if the campus is disconnected from the outside world, communication by name within the organization can still work. On the other hand, during such a disconnection, correspondents outside the organization cannot even verify that a name exists, for example, to validate an e-mail address. Therefore, a better arrangement might be to attach at least one of the organization's multiple replica name servers to another part of the Internet.

For the same reason that name servers need to be replicated, many network services also need to be replicated, so DNS allows the same name to be bound to several Internet addresses. In consequence, the *value* returned by `DNS_RESOLVE` can be a list of (presumably) equivalent Internet addresses. The client can choose which

Internet address to contact, based on order in the list, previous response times, a guess as to the distance to the attachment point, or any other criterion it might have available.

The design of DNS allows name service to be quite robust. In principle, the job of a DNS server is extremely simple: accept a request packet, search a table, and send a response packet. Its interface specification does not require it to maintain any connection state, or any other durable, changeable state; its only public interface is idempotent. The consequence is that a small, inexpensive personal computer can provide name service for a large organization, which encourages dedicating a computer to this service. A dedicated computer, in turn, tends to be more robust than one that supplies several diverse and unrelated network services. In addition a server with small, read-only tables can be designed so that when something such as a power failure happens, it can return to service quickly, perhaps even automatically. (Chapters 8 [on-line] and 9 [on-line] discuss how to design such a system.)

DNS also allows synonyms, in the form of indirect names. Synonyms are used conventionally to solve two distinct problems. For an example of the first problem, suppose that the Pedantic University Computer Science and Engineering Department has a computer whose Internet address is named `minehaha.cse.pedantic.edu`. This is a somewhat older and slower machine, but it is known to be very reliable. The department runs a World Wide Web server on this computer, but as its load increases the department knows that it will someday be necessary to move the Web server to a faster machine named `mississippi.cse.pedantic.edu`. Without synonyms, when the server moves, it would be necessary to inform everyone that there is a new name for the department's World Wide Web service. With synonyms, the laboratory can bind the indirect name `www.cse.pedantic.edu` to `minehaha.cse.pedantic.edu` and publicize the indirect name as the name of its Web site. When the time comes for `mississippi.cse.pedantic.edu` to take over the service, it can do so by simply having the manager of the `cse.pedantic.edu` domain change the binding of the indirect name. All those customers who have been using the name `www.cse.pedantic.edu` to get to the Web site will find that name continues to work correctly; they don't care that a different computer is now handling the job. As a general rule, the names of services can be expected to outlive their bindings to particular Internet addresses, and synonyms cater to this difference in lifetimes.

The second problem that synonyms can handle is to allow a single computer to appear to be in two widely different naming domains. For example, suppose that a geophysics group at the Institute of Scholarly Studies has developed a service to predict volcano eruptions but that organization doesn't actually have a computer suitable for running that service. It could arrange with a commercial vendor to run the service on a machine named, perhaps, `service-bureau.com` and then ask the manager of the Institute's name server to bind the indirect name `volcano.iss.edu` to `service-bureau.com`. The Institute could then advertise its service under the indirect name. If the commercial vendor raises its prices, it would be possible to move the service to a different vendor by simply rebinding the indirect name.

Because resolving a synonym requires an extra round-trip through DNS, and the basic name-to-Internet-address binding of DNS already provides a level of indirection, some network specialists recommend just manipulating name-to-Internet-address bindings to get the effect of synonyms.

### 4.4.4 Name Discovery in DNS

Name discovery comes up in at least three places in the Domain Name System: a client must discover the name of a nearby name server, a user must discover the domain name of a desired service, and the resolving system must discover an extension for unqualified domain names.

First, in order for DNS_RESOLVE to send a request to a name server, it needs to know the Internet address of that name server. DNS_RESOLVE finds this address in a configuration table. The real name-discovery question is how this address gets into the configuration table. In principle, this address would be the address of a root server, but as we have seen it can be the address of any existing name server. The most widely used approach is that when a computer first connects to a network it performs a name discovery broadcast to which the Internet service provider (ISP) responds by assigning the attacher an Internet address and also telling the attacher the Internet address of one or more name servers operated by or for the ISP. Another way to terminate name discovery is by direct communication with a local network manager, to obtain the address of a suitable name server, followed by configuring the answer into DNS_RESOLVE.

The second form of name discovery involves domain names themselves. If you wish to use the volcano prediction service at the Institute for Scholarly Studies, you need to know its name. Some chain of events that began with direct communication must occur. Typically, people learn of domain names via other network services, such as by e-mail, querying a search engine, reading postings in newsgroups or while surfing the Web, so the original direct communication may be long forgotten. But using each of those services requires knowing a domain name, so there must have been a direct communication at some earlier time. The purchaser of a personal computer is likely to find that it comes with a Web browser that has been preconfigured with domain names of the manufacturer's suggested World Wide Web query and directory services (as well as domain names of the manufacturer's support sites and other advertisers). Similarly, a new customer of an Internet service provider typically may, upon registering for service, be told the domain name of that ISP's Web site, which can then be used to discover names for many other services.

The third instance of name discovery concerns the extension that is used for unqualified domain names. Recall that the Domain Name System uses absolute path names, so if DNS_RESOLVE is presented with an unqualified name such as `library` it must somehow extend it, for example, to `library.pedantic.edu`. The default context used for extension is usually a configuration parameter of DNS_RESOLVE. The value of this parameter is typically chosen by the human user when initially setting up a computer, with an eye to minimizing typing for the most frequently used domain names.

### 4.4.5 Trustworthiness of DNS responses

A shortcoming of DNS is that, although it purports to provide authoritative name resolutions in its responses, it does not use protocols that allow authentication of those responses. As a result, it is possible (and, unfortunately, relatively easy) for an intruder to masquerade as a DNS server and send out mischievous or malevolent responses to name resolution requests.

Currently, the primary way of dealing with this problem is for the user of DNS to treat all of its responses as potentially unreliable hints and independently verify (using the terminology of Chapters 7 [on-line] and 11 [on-line] we would say "perform end-to-end authentication of") the identity of any system with which that user communicates. An alternative would be for DNS servers to use authentication protocols in communication with their clients. However, even if a DNS response is assuredly authentic, it still might not be accurate (for example, a DNS cache may hold out-of-date information, or a DNS administrator may have configured an incorrect name-to-address binding), so a careful user would still want to independently authenticate the identity of its correspondents.

Chapter 11 [on-line] describes protocols that can be used for authentication; there is an ongoing debate among network experts as to whether or how DNS should be upgraded to use such protocols.

The reader interested in learning more about DNS should explore the documents in the readings for DNS [Suggestions for Further Reading 4.3].

## 4.5 CASE STUDY: THE NETWORK FILE SYSTEM (NFS)

The Network File System (NFS), designed by Sun Microsystems, Inc. in the 1980s, is a client/service application that provides shared file storage for clients across a network. An NFS client grafts a remote file system onto the client's local file system name space and makes it behave like a local UNIX file system (see Section 2.5). Multiple clients can mount the same remote file system so that users can share files.

The need for NFS arose because of technology improvements. Before the 1980s, computers were so expensive that each one had to be shared among multiple users and each computer had a single file system. But a benefit of the economic pressure was that it allowed for easy collaboration because users could share files easily. In the early 1980s, it became economically feasible to build workstations, which allowed each engineer to have a private computer. But users still desired to have a shared file system for ease of collaboration. NFS provides exactly that: it allows a user at any workstation to use files stored on a shared server, a powerful workstation with local disks but often without a graphical display.

NFS also simplifies the management of a collection of workstations. Without NFS, a system administrator must manage each workstation and, for example, arrange for backups of each workstation's local disk. NFS allows for centralized management; for example, a system administrator needs to back up only the server's disks to archive the file system. In the 1980s, the setup also had a cost benefit: NFS allowed organizations

to buy workstations without disks, saving the cost of a disk interface on every workstation and the cost of unused disk space on each workstation.

The designers of NFS had four major goals. NFS should work with existing applications, which means NFS should provide the same semantics as a local UNIX file system. NFS should be deployable easily, which means its implementation should be able to retrofit into existing UNIX systems. The client should be implementable in other operating systems such as Microsoft's DOS, so that a user on a personal computer can have access to the files on an NFS server; this goal implies that the client/service messages cannot be too UNIX system-specific. Finally, NFS should be efficient enough to be tolerable to users, but it doesn't have to provide as high performance as local file systems. NFS only partially achieves these goals because achieving them all is difficult. The designers made a trade-off: simplify the design and lose some of the UNIX semantics.

This section describes version 2 of NFS. Version 1 was never deployed outside of Sun Microsystems, while version 2 has been in use since 1987. The case study concludes with a brief summary of the changes in versions 3 (1990s) and 4 (early 2000s), which address weaknesses in version 2. Problem set *3* explores an NFS-inspired design to reinforce the ideas in NFS.

### 4.5.1 Naming Remote Files and Directories

To programs, NFS appears as a UNIX file system providing the file interface presented in Section 2.5. User programs can name remote files in the same way as local files. When a user program invokes, say, OPEN ("/users/alice/.profile", READONLY ), it cannot tell from the path name whether "users" or "alice" are local or remote directories.

To make naming remote files transparent to users and their programs, the NFS client must mount a remote file system on the local name space. NFS performs this operation by using a separate program, called the *mounter*. This program serves a similar function as the MOUNT call (described in Section 2.5.10); it grafts the remote file system—named by *host*:*path*, where *host* is a DNS name and *path* a path name—onto the local file name space. The mounter sends a remote procedure call to the file server *host* and asks for a *file handle*, a 32-byte name for the object identified by *path*. On receiving the reply, the NFS client marks the mount point in the local file system as a remote file system. It also remembers the file handle for *path* and the network address for the server.

To the NFS client a file handle is a 32-byte opaque name that identifies an object on a remote NFS server. An NFS client obtains file handles from the server when the client mounts a remote file system, or it looks up a file in a directory on the NFS server. In all subsequent remote procedure calls to the NFS server for that file, the NFS client includes the file handle. In many ways the file handle is similar to an inode number; it is not visible to applications, but it used as a name internal to NFS to name files.

To the NFS server a file handle is a structured name—containing a *file system identifier,* an *inode number*, and *a generation number*—which the server can use to locate the file. The file system identifier allows the server to identify the file system responsible for the file. The inode number (see page 58) allows the identified file system to locate the file on the disk.

One might wonder why the NFS designers didn't choose to put path names in file handles. To see why, consider the following scenario with two user programs running on different clients:

| | Program 1 on client 1 | Program 2 on client 2 |
|---|---|---|
| *1* | CHDIR ("dir1") | |
| *2* | *fd* ← OPEN ("f", READONLY) | |
| *3* | | RENAME ("dir1", "dir2") |
| *4* | | RENAME ("dir3", "dir1") |
| *5* | READ (*fd*, *buf*, *n*) | |

Time ↓

RENAME (*source*, *destination*) changes the name of *source* to *destination*. The first rename operation (on line *3*) in program 2 renames "dir1" to "dir2", and the second one (on line *4*) renames "dir3" to "dir1". This scenario raises the following question: when program 1 invokes READ (line *5*) after the two rename operations have completed, does program 1 read data from "dir1/f", or "dir2/f"?

If the two programs were running on the same computer and sharing a local UNIX file system, program 1 would read "dir2/f", according to the UNIX specification. The goal is that NFS should provide the same behavior. If the NFS server were to put path names inside handles, then the READ call would result in a remote procedure call for the file "dir1/f". By putting the inode number in the handle the specification is met.

The file handle includes a generation number to handle scenarios such as the following almost correctly:

| | Program 1 on client 1 | Program 2 on client 2 |
|---|---|---|
| *1* | | *fd* ← OPEN ("f", READONLY) |
| *2* | UNLINK ("f") | |
| *3* | *fd* ← OPEN ("f", CREATE) | |
| *4* | | READ (*fd*, *buf*, *n*) |

Time ↓

A program on a client 1 deletes a file "f" (line *2*) and creates a new file with the same name (line *3*), while another program on a client 2 already has opened the original file (on line *1*). If the two programs were running on the same computer and sharing a local UNIX file system, program 2 would read the old file on line *4*.

If the server should happen to reuse the inode of the old file for the new file, remote procedure calls from client 2 will get the new file, the one created by client 1, instead of the old file. The generation number allows NFS to avoid this incorrect behavior. When the server reuses an inode, it increases the generation number by one. In the example, client 1 and client 2 would receive different file handles, and client 2 will use the old handle. Increasing the generation number makes it always safe for the NFS server to recycle inodes immediately.

For this scenario, NFS does not provide identical semantics to a local UNIX file system because that would require that the server know which files are in use. With NFS, when client 2 uses the file handle, it will receive an error message: "stale file handle".

This case is one example of the NFS designers trading some UNIX semantics to obtain a simpler implementation.

File handles are usable across server failures, so that even if the server computer fails and restarts between a client program opening a file and then reading from the file, the server can identify the file using the information in the file handle. Making file handles (which include a file system identifier and a generation number) usable across server failures requires small changes to the server's on-disk file system: the NFS designers modified the super block to record the file system identifier and modified inodes to record the generation number for the inode. With this information recorded, after a reboot the NFS server will be able to process NFS requests that the server handed out before it failed.

### 4.5.2 The NFS Remote Procedure Calls

Table 4.1 shows the remote procedure calls used by NFS. The remote procedure calls are best explained by example. Suppose we have the following fragment of a user program:

```
fd ← OPEN ("f", READONLY)
READ (fd, buf, n)
CLOSE (fd)
```

**Table 4.1** NFS Remote Procedure Calls

| Remote Procedure Call | Returns |
|---|---|
| NULL () | Do nothing. |
| LOOKUP (*dirfh*, *name*) | fh and file attributes |
| CREATE (*dirfh*, *name*, *attr*) | fh and file attributes |
| REMOVE (*dirfh*, *name*) | status |
| GETATTR (*fh*) | file attributes |
| SETATTR (*fh*, *attr*) | file attributes |
| READ (*fh*, *offset*, *count*) | file attributes and data |
| WRITE (*fh*, *offset*, *count*, *data*) | file attributes |
| RENAME (*dirfh*, *name*, *tofh*, *toname*) | status |
| LINK (*dirfh*, *name*, *tofh*, *toname*) | status |
| SYMLINK (*dirfh*, *name*, *string*) | status |
| READLINK (*fh*) | string |
| MKDIR (*dirfh*, *name*, *attr*) | fh and file attributes |
| RMDIR (*dirfh*, *name*) | status |
| READDIR (*dirfh*, *offset*, *count*) | directory entries |
| STATFS (*fh*) | file system information |

Figure 4.11 shows the corresponding timing diagram where "f" is a remote file. The NFS client implements each file system operation using one or more remote procedure calls.

In response to the program's call to OPEN, the NFS client sends the following remote procedure call to the server:

LOOKUP (*dirfh*, "f")

From before the program runs, the client has a file handle for the current working directory's (*dirfh*). It obtained this handle as a result of a previous lookup or as a result of mounting the remote file system.

On receiving the LOOKUP request, the NFS server extracts the file system identifier and inode number from *dirfh*, and asks the identified file system to look up the inode number in *dirfh*. The identified file system uses the inode number in *dirfh* to locate the directory's inode. Now the NFS server searches the directory identified by the inode number for "f". If present, the server creates a handle for "f". The handle contains the file system identifier of the local file system, the inode number for "f", and the generation number stored in the inode of "f". The NFS server sends this file handle to the client.

On receiving the response, the client allocates the first unused entry in the program's file descriptor table, stores a reference to f's file handle in that entry, and returns the index for the entry (*fd*) to the user program.

Next, the program calls READ (*fd, buf, n*). The client sends the following remote procedure call to the NFS server:

READ (*fh*, 0, *n*)

**FIGURE 4.11**

Example interaction between an NFS client and service. Since the NFS service is stateless, the client does not need to inform the service when the application calls CLOSE. Instead, it just deallocates *fd* and returns.

As with the directory file handle, the NFS server looks up the inode for *fh*. Then, the server reads the data and sends the data in a reply message to the client.

When the program calls CLOSE to tell the local file system that it is done with the file descriptor *fd*, NFS doesn't issue a CLOSE remote procedure call; the protocol doesn't have a CLOSE remote procedure call. Because the application didn't modify the file, the NFS client doesn't have to issue any remote procedure calls. As we shall see in Section 4.5.4, if a program modifies a file, the NFS client will issue remote procedure calls on a CLOSE system call to provide coherence for the file.

The NFS remote procedure calls are designed so that the server can be *stateless*, that is, the server doesn't need to maintain any other state than the on-disk files. NFS achieves this property by making each remote procedure call contain all the information necessary to carry out that request. The server does not maintain any state about past remote procedure calls to process a new request. For example, the client, not the server, must keep track of the file cursor (see Section 2.3.2), and the client includes it as an argument in the READ remote procedure call. As another example, the file handle contains all information to find the inode on the server, as explained above.

This stateless property simplifies recovery from server failures: a client can just repeat a request until it receives a reply. In fact, the client cannot tell the difference between a server that failed and recovered, and a server that is slow. Because a client repeats a request until it receives a response, it can happen that the server executes a request twice. That is, NFS implements at-least-once semantics for remote procedure calls.

Since many requests are idempotent (e.g., LOOKUP, READ, etc.), that is not a problem, but for some requests it results in surprising behavior. Consider a user program that calls UNLINK on an existing file that is stored on a remote file system. The NFS client would send a REMOVE remote procedure call and the server would execute it, but it could happen that the network lost the reply. In that case, the client would resend the REMOVE request, the server would execute the request again, and the user program would receive an error saying that the file didn't exist!

Later implementations of NFS minimize this surprising behavior by avoiding executing remote procedure calls more than once when there are no server failures. In these implementations, each remote procedure call is tagged with a transaction number and the server maintains some "soft" state (it is lost if the server fails), namely, a reply cache. The reply cache is indexed by transaction identifier and records the response for the transaction identifier. When the server receives a request, it looks up the transaction identifier (ID) in the reply cache. If the ID is in the cache, the server returns the reply from the cache, without reexecuting the request. If the ID is not in the cache, the server processes the request.

If the server doesn't fail, a retry of a REMOVE request will receive the same response as the first attempt. If, however, the server fails and restarts between the first attempt and a retry, the request is executed twice. The designers opted to maintain the reply cache as soft state because storing it in non-volatile storage is expensive. Doing so would require that the reply cache be stored, for example, on a disk and would require

a disk write for each remote procedure call to record the response. As explained in Section 6.1.8, disk writes are often a performance bottleneck and much more expensive than a remote procedure call.

Although the stateless property of NFS simplifies recovery, it makes it impossible to implement the UNIX file interface correctly because the UNIX specification requires maintaining state. Consider again the case where one program deletes a file that another program has open. The UNIX specification is that the file exists until the second program closes the file.

If the programs run on different clients, NFS cannot adhere to this specification because it would require that the server keep state. It would have to maintain a reference count per file, which would be incremented on an OPEN system call and decremented on a CLOSE system call, and persist across server failures. In addition, if a client would not respond to messages, the server would have to wait until the client becomes reachable again to decrement the reference count. Instead, NFS just does the easy but slightly wrong thing: remote procedure calls return an error "stale file handle" if a program on another client deletes a file that the first client has open.

NFS does not implement the UNIX specification faithfully because that simplifies the design of NFS. NFS preserves most of the UNIX semantics, and only in rarely encountered situations may users see different behavior. In practice, these situations are not a serious problem, and in return NFS gets by with simple recovery.

### 4.5.3 Extending the UNIX File System to Support NFS

To implement NFS as an extension of the UNIX file system while minimizing the number of changes required to the UNIX file system, the NFS designers split the file system program by introducing an interface that provides *vnodes*, virtual nodes (see Figure 4.12). A vnode is a structure in volatile memory that abstracts whether a file or directory is implemented by a local file system or a remote file system. This design allows many functions in the file system call layer to be implemented in terms of vnodes, without having to worry about whether a file or directory is local or remote. The interface has an additional advantage: a computer can easily support several, different local file systems.

When a file system call must perform an operation on a file (e.g., reading data), it invokes the corresponding procedure through the vnode interface. The vnode interface has procedures for looking up a file name in the contents of a directory vnode, reading from a vnode, writing to a vnode, closing a vnode, and so on. The local file system and NFS support their own implementation of these procedures.

By using the vnode interface, most of the code for file descriptor tables, current directory, name lookup, and the like, can be moved from the local file system module into the file system call layer with minimal effort. For example, with a few changes, the procedure PATHNAME_TO_INODE from Section 2.5 can be modified to be PATHNAME_TO_VNODE and be provided by the file system call layer.

**FIGURE 4.12**

NFS implementation for the UNIX system

To illustrate the vnode design, we consider a user program that invokes OPEN for a file (see Figure 4.12). To open the file, the file system call layer invokes PATHNAME_TO_VNODE, passing the vnode for the current working directory and the path name for the file as arguments. PATHNAME_TO_VNODE will parse the path name, invoking LOOKUP in the vnode layer for each component in the path name. If the directory is a local directory, the vnode-layer LOOKUP invokes the LOOKUP procedure implemented by the local file system to obtain a vnode for the path name component. If the directory is a remote directory, LOOKUP invokes the LOOKUP procedure implemented by the NFS client.

The NFS client invokes the LOOKUP remote procedure call on the NFS server, passing as arguments the file handle of the directory and the path name's component. On receiving the lookup request, the NFS server extracts the file system identifier and inode number from the file handle for the directory to look up the directory's vnode and then invokes LOOKUP in the vnode layer, passing the path name's component as an argument. If the directory is implemented by the server's local file system, the vnode layer invokes the procedure LOOKUP implemented by the server's local file system, passing the path name's component as an argument. The local file system looks up the name and, if present, creates a vnode and returns the vnode to the NFS server. The NFS server sends a reply containing the vnode's file handle and some metadata for the vnode to the NFS client.

On receiving the reply, the NFS client creates a vnode, which contains the file handle, on the client computer and returns it to the file system call layer on the client machine. When the file system call layer has resolved the complete path name, it returns a file descriptor for the file to the user program.

To achieve usable performance, a typical NFS client maintains various caches. A client stores the vnode for every open file so that the client knows the file handles for open files. A client also caches recently used vnodes, their attributes, recently used blocks of those cached vnodes, and the mapping from path name to vnode. Caching reduces the latency of file system operations on remote files because for cached files a client can avoid the cost of remote procedure calls. In addition, because clients make fewer remote procedure calls, a single server can support more clients. If multiple clients cache the same file, however, NFS must ensure read/write coherence in some way.

### 4.5.4 Coherence

When programs share a local file in a UNIX system, the program calling READ observes the data from the most recent WRITE, even if this WRITE was performed by another program. This property is called read/write coherence (see Section 2.1.1.1). If the programs are running on different clients, caching complicates implementing these semantics correctly.

To illustrate the problem, consider a user program on one computer that writes a block of a file. The file system call layer on that computer might perform the update to the block in the cache, delaying the write to the server, just like the local UNIX file system delays a write to disk. If a program on another computer then reads the file from the server, it may not observe the change made on the first computer because that change may not have been propagated to the server yet. Because this behavior would be incorrect, NFS implements a form of read/write coherence.

NFS could guarantee read/write coherence for every operation, or just for certain operations. One option is to provide read/write coherence for only OPEN and CLOSE. That is, if an application OPENs a file, WRITEs, and CLOSEs the file on one client, and if later an application on a second client opens the same file, then the second application will observe the results of the writes by the first application. This option is called *close-to-open consistency.* Another option is to provide read/write coherence for every read and write. That is, if two applications on different clients have the same file open concurrently, then a READ of one observes the results of WRITEs of the other.

Many NFS implementations provide close-to-open consistency because it allows for higher data rates for reading or writing a big file; a client can send several reads or write requests without having to wait for a response after each request. Figure 4.13 illustrates close-to-open semantics in more detail. If, as in case 1, a program on one client calls WRITE and then CLOSE, and then another client calls OPEN and READ, the NFS implementation will ensure that the READ will include the results of the WRITEs by the first client. But, as in case 2, if two clients have the same file open, one client writes a block of the file, and then the other client invokes READ, READ may return the data either from before or after the last WRITE; the NFS implementation makes no guarantees in that case.

**FIGURE 4.13**

Two cases illustrating close-to-open consistency

NFS implementations provide close-to-open semantics as follows. The client stores with each data block in its cache the modification of the block's vnode at the time the client reads the block from the server. When a user program opens a file, the client sends a GETATTR request to fetch the last modification time of the file. The client reads a cached data block only if the block's modification time is the same as its vnode's modification time. If the modification times are not the same, the client removes the data block from its cache and fetches it from the server.

The client implements WRITE by modifying its local cached version, without incurring the overhead of remote procedure calls. Then, in the CLOSE call of Figure 4.11, the client, rather than simply returning, would first send any cached writes to the server and wait for an acknowledgment. This implementation is simple and provides decent performance. The client can perform READs and WRITEs at local memory speeds. By delaying sending the modified blocks until CLOSE, the client absorbs modifications that are overwritten (e.g., the application writes the same block multiple times) and aggregates WRITEs to the same block (e.g., WRITEs that modify different parts of the block).

By providing close-to-open semantics, most user programs written for a local UNIX file system will work correctly when their files are stored on NFS. For example, if a user edits a program on a personal workstation but prefers to compile on a faster compute machine, then NFS with close-to-open consistency works well, requiring no modifications to the editor and the compiler. After the editor has written out the modified file and the user starts the compiler on the compute machine, the compiler will observe the edits.

On the other hand, certain programs will not work correctly using NFS implementations that provide close-to-open consistency. For example, a multiclient database program that reads and writes records stored in a file over NFS will not work correctly because, as the second case in Figure 4.13 illustrates, close-to-open semantics

doesn't specify the semantics when clients execute operations concurrently—for example, if client 2 opens the database file before client 1 closes it and client 3 opens the database file after client 1 closes it. If client 2 and 3 then read data from the file, client 2 may not see the data written by client 1, while client 3 will see the data written by client 1.

Furthermore, because NFS caches blocks (instead of whole files), the file may have blocks from different versions of the file intermixed. When a client fetches a file, it fetches only the inode and perhaps prefetches a few blocks. Subsequent READ RPCs may fetch blocks from a newer version of the file because another client may have written those blocks after this client opened the file.

To provide the correct semantics in this case requires more sophisticated machinery, which NFS implementations don't provide, because databases often have their own special-purpose solutions anyway, as we discuss in Chapters 9 [on-line] and 10 [on-line]. If the database program doesn't provide a special-purpose solution, then tough luck—one cannot run it over NFS.

### 4.5.5 NFS Version 3 and Beyond

NFS version 2 is being replaced by NFS version 3. Version 3 addresses a number of limitations in version 2, but the extensions do not significantly change the preceding description. For example, version 3 supports 64-bit numbers for recording file sizes and adds an asynchronous write (i.e., the server may acknowledge an asynchronous WRITE request as soon as it receives the request, before it has written the data to disk).

NFS version 4, which took a number of lessons from the Andrew File System [Suggestions for Further Reading 4.2.3], is a bigger change than version 3; in version 4 the server maintains some state. Version 4 also protects against intruders who can snoop and modify network traffic using techniques discussed in Chapter 11 [on-line]. Furthermore, it provides a more efficient scheme for providing close-to-open consistency, and it works well across the Internet, where the client and server may be connected using low-speed links.

The following references provide more details on NFS:

1. Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. "Design and implementation of the Sun network file system", *Proceedings of the 1985 Summer Usenix Technical Conference*, June 1985, El Cerrito, CA, pages 119–130.

2. Chet Juszezak, "Improving the performance and correctness of an NFS server", *Proceedings of the 1989 Winter Usenix Technical Conference*, January 1989, Berkeley, CA, pages 53–63.

3. Brian Pawlowski, Chet Juszezak, Peter Staubach, Carl Smith, Diana Lebel, and David Hitz, "NFS Version 3 design and implementation", *Proceedings of the 1990 Summer Usenix Technical Conference*, June 1994, Boston, MA.

4. Brian Pawlowski, Spencer Shepler, Carl Beame, Brent Callaghan, Michael Eisler, David Noveck, David Robinson, and Robert Turlow, "The NFS Version 4 protocol", *Proceedings of Second International SANE Conference*, May 2000, Maastricht, The Netherlands.

## EXERCISES

**4.1** When modularity between a client and a service is enforced, there is no way for errors in the implementation of the service to propagate to its clients. True or False? Explain.

*1995-1-1d*

**4.2** Chapter 1 discussed four general methods for coping with complexity: modularity, abstraction, hierarchy, and layering.

**4.2a** Which of those four methods does client/service use as its primary organizing scheme?

**4.2b** Which does remote procedure call use? Explain.

*1996-1-1b,d*

**4.3** To client software, a notable difference between remote procedure call and ordinary local procedure call is:

**A.** None. That's the whole point of RPC!
**B.** There may be multiple returns from one RPC call.
**C.** There may be multiple calls for one RPC return.
**D.** Recursion doesn't work in RPC.
**E.** The runtime system may report a new type of error as a result of an RPC.
**F.** Arguments to RPCs must be scalars.

*1998-2-4*

**4.4** Which of the following statements is true of the X Window System (see Sidebar 4.4)?

**A.** The X server is a trusted intermediary and attempts to enforce modularity between X clients in their use of the display resource.
**B.** An X client always waits for a response to a request before sending the next request to the X server.
**C.** When a program running on another computer displays its window on your local workstation, that *remote* computer is considered an X server.

*2005-1-6*

**4.5** While browsing the Web, you click on a link that identifies an Internet host named `www.cslab.scholarly.edu`. Your browser asks your Domain Name System (DNS)

name server, *M*, to find an Internet address for this domain name. Under what conditions is each of the following statements true of the name resolution process?

**A.** To answer your query, *M* must contact one of the root name servers.

**B.** If *M* answered a query for `www.cslab.scholarly.edu` in the past, then it can answer your query without asking any other name server.

**C.** M must contact one of the name servers for `cslab.scholarly.edu` to resolve the domain name.

**D.** If *M* has the current Internet address of a working name server for `scholarly.edu` cached, then that name server will be able to directly provide an answer.

**E.** If *M* has the current Internet address of a working name server for `cslab.scholarly.edu` cached, then that name server will be able to directly provide an answer.

**4.6** For the same situation as in Exercise 4.5, which of the following is always true of the name resolution process, assuming that all name servers are configured correctly and no messages are lost?

**A.** If M had answered a query for the IP address corresponding to `www.cslab.scholarly.edu` at some time in the past, then it can respond to the current query without contacting any other name server.

**B.** If M has a valid IP address of a functioning name server for `cslab.scholarly.edu` in its cache, then M will get a response from that name server without any other name servers being contacted.

*2000-2-5 and 2005-2-4*

**4.7** The Network File System (NFS) described in Section 4.5 allows a client machine to run operations on files that are stored at a remote server. For the version of NFS described there, decide if each of these assertions is true or false:

**A.** When the server responds to a client's `WRITE` call, all modifications required by that `WRITE` will have made it to the server's disk.

**B.** An NFS client might send multiple requests for the same operation to the NFS server.

**C.** When an NFS server crashes, after the operating system restarts and recovers the disk contents, the server must also run its own recovery procedure to make its state consistent with that of its clients.

*2005-1-2*

**4.8** Assume that an NFS (described in Section 4.5) server contains a file `/a/b` and that an NFS client mounts the NFS server's root directory in the location `/x`, so that the client can now name the file as `/x/a/b`. Further assume that this is the only client and that the client executes the following two commands:

```
chdir /x/a
rm b
```

The `REMOVE` message from the client to the server gets through, and the server removes the file. Unfortunately, the response from the server to the client is lost

and the client resends the message to remove the (now non-existent) file. The server receives the resent message. What happens next depends on the server implementation. Which of the following are correct statements?

**A.** If the server maintains an in-memory reply cache in which it records all operations it previously executed, and there are no server failures, the server will return "OK".

**B.** If the server maintains an in-memory reply cache but the server has failed, restarted, and its reply cache is empty, both of the following responses are possible: the server may return "file not found" or "OK".

**C.** If the server is stateless, it will return "file not found".

**D.** Because `REMOVE` is an idempotent operation, any server implementation will return "OK".

*2006-2-2*

**Additional exercises relating to Chapter 4 can be found in the problem sets beginning on page 425.**

CHAPTER

# Enforcing Modularity with Virtualization

5

## CHAPTER CONTENTS


**Overview** ..... **200**
**5.1 Client/Server Organization within a Computer Using Virtualization** ..... **201**
5.1.1 Abstractions for Virtualizing Computers ..... 203
5.1.2 Emulation and Virtual Machines ..... 208
5.1.3 Roadmap: Step-by-Step Virtualization ..... 208
**5.2 Virtual Links Using SEND, RECEIVE, and a Bounded Buffer** ..... **210**
5.2.1 An Interface for SEND and RECEIVE with Bounded Buffers ..... 210
5.2.2 Sequence Coordination with a Bounded Buffer ..... 211
5.2.3 Race Conditions ..... 214
5.2.4 Locks and Before-or-After Actions ..... 218
5.2.5 Deadlock ..... 221
5.2.6 Implementing ACQUIRE and RELEASE ..... 222
5.2.7 Implementing a Before-or-After Action Using the One-Writer Principle ..... 225
5.2.8 Coordination between Synchronous Islands with Asynchronous Connections ..... 228
**5.3 Enforcing Modularity in Memory** ..... **230**
5.3.1 Enforcing Modularity with Domains ..... 230
5.3.2 Controlled Sharing Using Several Domains ..... 231
5.3.3 More Enforced Modularity with Kernel and User Mode ..... 234
5.3.4 Gates and Changing Modes ..... 235
5.3.5 Enforcing Modularity for Bounded Buffers ..... 237
5.3.6 The Kernel ..... 238
**5.4 Virtualizing Memory** ..... **242**
5.4.1 Virtualizing Addresses ..... 243
5.4.2 Translating Addresses Using a Page Map ..... 245
5.4.3 Virtual Address Spaces ..... 248
5.4.4 Hardware versus Software and the Translation Look-Aside Buffer ..... 252
5.4.5 Segments (Advanced Topic) ..... 253



**Principles of Computer System Design: An Introduction**


Copyright © 2009 by Jerome H. Saltzer and M. Frans Kaashoek. All rights reserved.


DOI: 10.1016/B978-0-12-374957-4.00013-X

**5.5 Virtualizing Processors Using Threads ..... 255**
5.5.1 Sharing a Processor Among Multiple Threads ..... 255
5.5.2 Implementing YIELD ..... 260
5.5.3 Creating and Terminating Threads ..... 264
5.5.4 Enforcing Modularity with Threads: Preemptive Scheduling ..... 269
5.5.5 Enforcing Modularity with Threads and Address Spaces ..... 271
5.5.6 Layering Threads ..... 271
**5.6 Thread Primitives for Sequence Coordination ..... 273**
5.6.1 The Lost Notification Problem ..... 273
5.6.2 Avoiding the Lost Notification Problem with Eventcounts and Sequencers ..... 275
5.6.3 Implementing AWAIT, ADVANCE, TICKET, and READ (Advanced Topic) ..... 280
5.6.4 Polling, Interrupts, and Sequence Coordination ..... 282
**5.7 Case Study: Evolution of Enforced Modularity in the Intel x86 ..... 284**
5.7.1 The Early Designs: No Support for Enforced Modularity ..... 285
5.7.2 Enforcing Modularity Using Segmentation ..... 286
5.7.3 Page-Based Virtual Address Spaces ..... 287
5.7.4 Summary: More Evolution ..... 288
**5.8 Application: Enforcing Modularity Using Virtual Machines ..... 290**
5.8.1 Virtual Machine Uses ..... 290
5.8.2 Implementing Virtual Machines ..... 291
5.8.3 Virtualizing Example ..... 293
**Exercises ..... 294**

## OVERVIEW

The goal of the client/service organization is to limit the interactions between clients and services to messages. To ensure that there are no opportunities for hidden interactions, the previous chapter assumed that each client module and service module runs on a separate computer. Under that assumption, the network between the computers enforces modularity. This implementation reduces the opportunity for programming errors to propagate from one module to another, but it is also good for achieving security (because the service module can be penetrated only by sending messages) and fault tolerance (service modules can be separated geographically, which reduces the risk that a catastrophic failure such as an earthquake or a massive power failure affects all servers that implement the service).

The main disadvantage of using one computer per module is that it requires as many computers as modules. Since the modularity of a system and its applications shouldn't be dictated by the number of computers available, this requirement is undesirable. If the designer decides to split a system or application into *n* modules and would like to enforce modularity between them, the choice of *n* should not be constrained by the number of computers that happen to be in stock and easily obtained. Instead, the designer needs a way to run several modules on the same computer without resorting to soft modularity.

This chapter introduces *virtualization* as the primary approach to achieve this goal and presents three new abstractions (SEND and RECEIVE with *bounded buffers*, *virtual memory*, and *threads*) that correspond to virtualized versions of the three main abstractions (communication links, memory, and processors). The three new abstractions allow a designer to implement as many virtual computers as needed for running the desired *n* modules.

## 5.1 CLIENT/SERVER ORGANIZATION WITHIN A COMPUTER USING VIRTUALIZATION

To enforce modularity between modules running on the same computer, we create several *virtual* computers using one *physical* computer and execute each module (usually an application or a subsystem) in its own virtual computer.

This idea can be realized using a technique called *virtualization.* A program that virtualizes a physical object simulates the interface of the physical object, but it creates many virtual objects by *multiplexing* one physical instance, or it may provide one large virtual object by *aggregating* many physical instances, or implement a virtual object from a different kind of physical object using *emulation*. For the user of the simulated object, it provides the same behavior as a physical instance, but it isn't the physical instance, which is why it is called virtual. A primary goal of virtualization is to preserve an existing interface. That way, modules designed to use a physical instance of an object don't have to be modified to use a virtual instance. Figure 5.1 gives some examples of the three virtualization methods, which we will discuss in turn.

Hosting several Web sites on a single physical server is an example of virtualization involving multiplexing. If the aggregate peak load of the Web sites is less than what a

| Virtualization Method | Physical Resource | Virtual Resource |
|---|---|---|
| multiplexing | server | Web site |
| multiplexing | processor | thread |
| multiplexing | real memory | virtual memory |
| multiplexing and emulation | real memory and disk | virtual memory with paging |
| multiplexing | wire or communication channel | virtual circuit |
| aggregation | communication channel | channel bonding |
| aggregation | disk | RAID |
| emulation | disk | RAM disk |
| emulation | Macintosh | virtual PC |

**FIGURE 5.1**
Examples of virtualization.

single server computer can support, providers often prefer to use a single server to host several Web sites because it is less expensive than buying one server for each Web site.

The next three examples relate to threads and virtual memory, which we will overview in Section 5.1.1. Some of these usages don't rely on a single method of virtualization but combine several or use different methods to obtain different properties. For example, virtual memory with paging (described in Section 6.2) uses both multiplexing and aggregation.

A virtual circuit virtualizes a wire or communication channel using multiplexing. For example, it allows several phone conversations to take place over a single wire with a technique called time division multiplexing, as we will discuss in Chapter 7 [on-line]. Channel bonding aggregates several communication channels to provide a combined high data rate.

RAID (see Section 2.1.1.4) is an example of virtualization involving aggregation. In RAID, a number of disks are aggregated together in a clever way that provides an identical interface to the one of a single disk, but together the disks provide improved performance (by reading and writing disks concurrently) and durability (by writing information on more than one disk). A system administrator can replace a single disk with a RAID and take advantage of the RAID improvements without having to change the file system.

A RAM disk is an example of virtualization involving emulation. A RAM disk provides the same interface as a physical disk but stores blocks in memory instead of on a disk platter. RAM disks can therefore read and write blocks much faster than a physical disk but, because RAM is volatile, it provides little durability. Administrators can configure a file system to use a RAM disk instead of a physical disk without needing to modify the file system itself. For example, a system administrator may configure the file system to use RAM disk to store temporary files, which allows the file system to read and write temporary files fast. And since temporary files don't have to be stored durably, nothing is lost by storing them on a RAM disk.

A virtual PC is an example of virtualization using emulation. It allows the construction of a virtual personal computer out of a physical personal computer, perhaps of a different type (e.g., using a Macintosh to emulate a virtual PC). Virtual PCs can be useful to run several operating systems on a single computer, or simplify the testing and the development of a new operating system. Section 5.1.2 discusses this virtualization technique in more detail.

Designers are often tempted to tinker slightly with an interface rather than virtualizing it exactly, to improve it or to add a useful feature. Such tinkering can easily cross a line in which the original goal of not having to modify other modules that use the interface is lost. For example, the X Window System described in Sidebar 4.4 implements objects that could be thought of as virtual displays, but because the size of those objects can be changed on the fly and the program that draws on them should be prepared to redraw them on command, it is more appropriate to call them "windows".

Similarly, a file system (see Section 2.3.2) creates objects that store bits and thus has some similarity to a virtualized hard disk, but because files have names, are of adjustable length, allow controlled sharing, and can be organized into hierarchies, they are more appropriately thought of as a different memory abstraction.

The preceding examples suggest how we could implement the client/service organization within a single computer. Consider a computer on which we would like to run five modules: a text editor, an e-mail reader, a keyboard manager, the window service, and the file service. When a user works with the text editor, keyboard input should go to the editor. When the user moves the mouse from the editor window to the mail reader window, the next keyboard input should go to the mail reader. When the text editor saves a file, the file service must execute to store the file. If there are more modules than computers, some solution is needed for sharing a single computer.

The idea is to present each module with its own virtual computer. The power of this idea is that programmers can think of each module independently. From the programmer's perspective, every program module has a virtual computer to itself, which executes independently of the virtual computers of other modules. This idea enforces modularity because a virtual computer can contain a module's errors and no module can halt the progress of other modules.

The virtual computer design does not enforce modularity as well as running modules on physically separate computers because, for example, a power failure will knock out all virtual computers on the same physical computer. Also, once an attacker has broken into one virtual computer, the attacker may discover a way to exploit a flaw in the implementation of virtualization to break into other virtual computers. The primary modularity goal of employing virtual computers is to ensure that module failures due to accidental programming errors don't propagate from one virtual computer to another. Virtual computers can contribute to security goals but are better viewed as only one of several lines of defense.

### 5.1.1 Abstractions for Virtualizing Computers

The main challenge in implementing virtual computers is finding the right abstractions to build them. This chapter introduces three abstractions that correspond to virtualized versions of the main abstractions: SEND and RECEIVE with bounded buffers (virtualizes communication links), virtual memory (virtualizes memory), and threads (virtualizes processors).

These three abstractions are typically implemented by a program that is called the operating system (which was briefly discussed in Sidebar 2.4 but will be discussed in detail in this chapter). Using an operating system that provides the three abstractions, we can implement the client/service organization within a single computer (see Figure 5.2). For example, with this design the text editor running on one virtual computer can send a message over the virtual communication link to the file service, running on a different virtual computer, and ask it, for example, to save a file. In the figure each virtual computer has one virtual processor (implemented by a thread) and its own virtual memory with a virtual address space ranging from 0 to $2^n$. To build an intuition for these abstractions and learn how they can be used to implement a virtual computer, we give a brief overview of them.

**FIGURE 5.2**

An operating system providing the editor and file service module each their own virtual computer. Each virtual computer has a thread that virtualizes the processor. Each virtual computer has a virtual memory that provides each module with the illusion that it has its own memory. To allow communication between virtual computers, the operating system provides SEND, RECEIVE, and a bounded buffer of messages.

#### *5.1.1.1 Threads*

The first step in virtualizing a computer is to virtualize the processor. To provide the editor module (shown in Figure 5.3) with a virtual processor, we create a *thread of execution*, or *thread* for short. A thread is an abstraction that encapsulates the execution state of an active computation. It encapsulates the state of a conceptual interpreter that executes the computation (see Section 2.1.2). The state of a thread consists of the variables internal to the interpreter (e.g., processor registers), which include

1. A reference to the next program step (e.g., a program counter)
2. References to the environment (e.g., a stack, a heap, and other current objects)

The thread abstraction encapsulates enough of the interpreter's state 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 and provides a convenient way of multiplexing a physical processor. Threads are the most widely used implementation strategy to virtualize physical processors. In fact, this implementation strategy is so common that in the context of virtualizing physical processors the words "thread" and "virtual processor" have become synonyms in practice.

The next few paragraphs give a high-level overview of how threads can be used to virtualize physical processors. A user might type the name of the module that the user wants to run, or a user might select the name from a pull-down menu. The command line interpreter or the window system can then start the program as follows:

1. Load the program's text and data stored in the file system into memory.
2. Allocate a thread and start it at a specified address. Allocating a thread involves allocating a stack to allow the thread to make procedure calls, setting the SP register to the top of the stack, and setting the PC register to the starting address.

```
input ← OPEN (keyboard)          // open the keyboard device
file ← OPEN (argument)           // open the file that was passed an argument to the editor
do forever
    n ← READLINE (input, buf)    // read characters from the keyboard into buf
    APPLY (file, buf, n)         // apply them to the file being edited
```

**FIGURE 5.3**

Sketch of the program for the editor module.

A module may have one or several threads. A module with only *one* thread (and thus one processor) is common because then the programmer can think of it as executing a program serially: it starts at the beginning, computes (perhaps producing some output by performing a remote procedure call to a service), and then terminates. This simple structure follows the ***principle of least astonishment*** for programmers. Humans are better at understanding serial programs than at understanding programs that have several, concurrent threads, which can have surprising behavior.

Modules may have more than one thread by creating several threads. A module, for example, may create a thread per device that the module manages so that the module can operate the devices concurrently. Or a module may create several threads to overlap the latency of an expensive operation (e.g., waiting for a disk) by running the expensive operation in another thread. A module may allocate several threads to exploit several physical processors that run each thread concurrently. A service module may create several threads to process requests from different clients concurrently.

The thread abstraction is implemented by a *thread manager*. The thread manager's job is to multiplex the possibly many threads on the limited number of physical processors of the computer, and in such a way that a programming error in one thread cannot interfere with the execution of another thread. Since the thread encapsulates enough of the state so that one can stop a thread at any point in time, and later resume it, the thread manager can stop a thread and allocate the released physical processor to another thread by resuming that thread. Later the thread manager can resume the suspended thread again by reallocating a physical processor to that thread. In this way, the thread manager can multiplex many threads across a number of physical processors. The thread manager can ensure that no thread hogs a physical processor by forcing each thread to periodically give up its physical processor on a clock interrupt.

With the introduction of threads, it is helpful to refine the description of the interrupt mechanism described in Chapter 2. External events (e.g., a clock interrupt or a magnetic disk signals the completion of an I/O) interrupt a physical processor, but the event may have nothing to do with the thread running on the physical processor. On an interrupt, the processor invokes the interrupt handler and after returning from the handler continues running the thread that was running on the physical processor before the interrupt. If one processor should not be interrupted because it is already busy processing an interrupt, the next interrupt may interrupt another processor in the computer, allowing interrupts to be processed concurrently.

Some interrupts do pertain to the currently running thread. We shall refer to this class of interrupts as *exceptions.* The exception handler runs in the context of the interrupted thread; it can read and modify the interrupted thread's state. Exceptions often happen when a thread performs some operation that the hardware cannot complete (e.g., divide by zero). Many programming languages also have a notion of an exception; for example, a square root program may signal an exception if its caller hands it a negative argument. We shall see that because exception handlers run in the context of the interrupted thread, but interrupt handlers run in the context of the operating system, there are different restrictions on what the two kinds of handlers can safely do.

#### 5.1.1.2 Virtual Memory

As described so far, all threads and handlers share the same physical memory. Each processor running a thread sends READ and WRITE requests across a bus along with an address identifying the memory location to be read or written. Sharing memory has benefits, but uncontrolled sharing makes it too easy to make a mistake. If several threads have their programs and data stored in the same physical memory, then the threads of each module have access to every other module's data. In fact, a simple programming error (e.g., the program computes the wrong address) can result in a STORE instruction overwriting another module's data or a JMP instruction executing procedures of another module. Thus, without a memory enforcement mechanism we have, at best, soft modularity. In addition, the physical memory and address space may be too small to fit the applications, requiring the applications to manage the memory carefully.

To enforce modularity, we must ensure that the threads of one module cannot overwrite the data of another module by accident. To do so, we give each module its own *virtual memory*, as Figure 5.2 illustrates. Virtual memory can provide each module with its own *virtual address space*, which has its own *virtual addresses*. That is, the arguments to JMP, LOAD, and STORE instructions are all virtual addresses, which a new hardware gadget (called a *virtual memory manager*) translates to physical addresses. If each module has its own virtual address space, then a module can name only its own physical memory and cannot store to the memory of another module. If a thread of a module by accident calculates an incorrect virtual address and stores to that virtual address, it will affect only that module.

With threads and virtual memory, we can create a virtual computer for each module. Each module has one or more threads that execute the code of the module. The threads of one module share a single virtual address memory that threads of other modules by default cannot touch.

#### 5.1.1.3 Bounded Buffer

To allow client and service modules on virtual computers to communicate, we introduce SEND and RECEIVE with a *bounded buffer* of messages. A thread can invoke SEND, which attempts to insert the supplied message into a bounded buffer of messages. If the bounded buffer is full, the sending thread waits until there is space in the bounded buffer. A thread invokes RECEIVE to retrieve a message from the buffer; if there are no

messages, the calling thread waits. Using SEND, RECEIVE, and bounded buffers, we can implement remote procedure calls and enforce strong modularity between modules on different virtual computers running on the same physical computer.

#### 5.1.1.4 Operating System Interface

To make the abstractions concrete, this chapter develops a minimal operating system that provides the abstractions (see Table 5.1 for its interface). This minimal design exhibits many of the mechanisms that are found in existing operating systems, but to keep the explanation simple it doesn't describe any existing system. Existing systems have evolved over many years, incorporating new ideas as they came along. As a result, few existing systems provide an example of a clean, simple design. In addition, a complete operating system includes many services (such as a file system, a window system, etc.) that are not included in the minimal operating system described in this chapter.

**Table 5.1** The Interface Developed in this Chapter

| Abstraction | Procedure |
|---|---|
| Memory | CREATE_ADDRESS_SPACE |
| | DELETE_ADDRESS_SPACE |
| | ALLOCATE_BLOCK |
| | FREE_BLOCK |
| | MAP |
| | UNMAP |
| Interpreter | ALLOCATE_THREAD |
| | EXIT_THREAD |
| | DESTROY_THREAD |
| | YIELD |
| | AWAIT |
| | ADVANCE |
| | TICKET |
| | ACQUIRE |
| | RELEASE |
| Communication | ALLOCATE_BOUNDED_BUFFER |
| | DEALLOCATE_BOUNDED_BUFFER |
| | SEND |
| | RECEIVE |

### 5.1.2 Emulation and Virtual Machines

The previous section described briefly three high-level abstractions to virtualize processors, memory, and links to enforce modularity. An alternative approach is to provide an interface that is identical to some physical hardware. In this approach, one can enforce modularity by providing each application with its own instance of the physical hardware.

This approach can be implemented using a technique called *emulation.* Emulation simulates some physical hardware so faithfully that the emulated hardware can run any software the physical hardware can. For example, Apple Inc. has used emulation successfully to move customers to new hardware designs. Apple used a program named Classic to emulate Motorola Inc.'s 68030 processor on the PowerPC processor and more recently used a program named Rosetta to emulate the PowerPC processor on Intel Inc.'s x86 processor. As another example, some processors include a microcode interpreter inside the processor to simulate instructions of other processors or instructions from older versions of the same processor. It is also standard practice for a vendor developing a new processor to start by writing an emulator for it and running the emulator on some already existing processor. This approach allows software development to begin before the chip for the new processor is manufactured, and when the chip does become available, the emulator acts as a kind of specification against which to debug the chip.

Emulation in software is typically slow because interpreting the instructions of the emulated machine in software has substantial overhead. Looking at the structure of an interpreter in Figure 2.5, it is easy to see that decoding the simulated instruction, performing its operation, and updating the state of the simulated processor can take tens of instructions on the processor that performs the emulation. As a result, emulation in software can cost a factor 10 in performance and a designer must work hard to do better.

A specialized approach to fast emulation is using *virtual machines*. In this approach, a physical processor is used as much as possible to implement many virtual instances of itself. That is, virtual machines emulate many instances of a machine *M* using a physical machine *M*. This approach loses the portability of general emulation but provides better performance. The part of the operating system that provides virtual machines is often called a *virtual machine monitor*. Section 5.8 of this chapter discusses virtual machines and virtual machine monitors in more detail. Internally, a virtual machine monitor typically uses bounded buffers, virtual memory, and threads, the main topics of this chapter.

### 5.1.3 Roadmap: Step-by-Step Virtualization

This chapter gradually develops the tools needed to provide a virtual computer. We start out assuming that there are more physical processors than threads and that the operating system can allocate each thread its own physical processor. We will even assume that each interrupt handler has its own physical processor, so when

an interrupt occurs, the handler runs on that dedicated processor. Figure 5.4 shows a modified version of Figure 2.2, in which each thread has its own processor. Consider again the example that we would like to run the following five modules on a single computer: a text editor, an e-mail reader, a keyboard manager, the window service, and the file service. Processor 1, for example, might run the text editor thread. Processor 2 might run the e-mail reader thread. The window manager might have one thread per window, each running on a separate processor. Similarly, the file service might have several threads, each running on a separate processor. The LOAD and STORE instructions of threads refer to addresses that name memory locations or registers of the various controllers. That is, threads share the memories and controllers.

Given this setup, Section 5.2 shows how the client/server organization can be implemented in a computer with many processors and a single address space by allowing the threads of different modules to communicate through a bounded buffer. This implementation takes advantage of the fact that processors within a computer can interact with one another through a shared memory. That ability will prove useful to implement virtual communication links, but such unconstrained interaction through shared memory would drastically compromise modularity. For this reason, Section 5.3 will adjust this assumption to show how to provide and enforce walls between the memory regions used by different modules to restrict and control sharing of memory.

Sections 5.4, 5.5, and 5.6 of this chapter remove restrictions on the design presented in Sections 5.2 and 5.3. In Section 5.4 we will remove the restriction that processors must share one single, large address space, and provide each module with its own virtual memory, while still allowing controlled sharing. In Section 5.5 we remove the restriction that each thread must have its own physical processor while still ensuring that no thread can halt the progress of other threads involuntarily. Finally,

**FIGURE 5.4**

A computer with several hardware modules connected by a shared bus. Each thread of the software modules has its own processor allocated to it.

in Section 5.6 we remove the restriction that a thread must use a physical processor continuously to test if another thread has sent a message.

The operating system, thread manager, virtual memory manager, and SEND and RECEIVE with bounded buffers presented in this chapter are less complex than the designs found in contemporary computer systems. One reason is that most contemporary designs have evolved over time with changing technologies, while also allowing users to continue to run old programs. As an example of this evolution, Section 5.7 briefly describes the history of the Intel x86 processor, a widely used general-purpose processor design that has, over the years, provided increasing support for enforced modularity.

## 5.2 VIRTUAL LINKS USING SEND, RECEIVE, AND A BOUNDED BUFFER

Operating systems designers have developed many abstractions for virtual communication links. One popular abstraction is pipes [Suggestions for Further Reading 2.2.1 and 2.2.2], which allow two programs to communicate using procedures from the file system call interface. Because SEND and RECEIVE with a bounded buffer mirror a communication link directly, we describe them in more detail in this chapter. The implementation of SEND and RECEIVE with a bounded buffer also mirrors implementations of sockets, an interface for virtual links provided in operating systems such as UNIX and Microsoft Windows.

The main challenge in implementing SEND and RECEIVE with bounded buffers is that several threads, perhaps running in parallel on separate physical processors, may add and remove messages from the same bounded buffer concurrently. To ensure correctness, the implementation must coordinate these updates. This section will present bounded buffers in detail and introduce some techniques to coordinate concurrent actions.

### 5.2.1 An Interface for SEND and RECEIVE with Bounded Buffers

An operating system might provide the following interface for SEND and RECEIVE with bounded buffers:

- *buffer* ← ALLOCATE_BOUNDED_BUFFER (*n*): allocate a bounded buffer that can hold *n* messages.
- DEALLOCATE_BOUNDED_BUFFER (*buffer*): free the bounded buffer *buffer*.
- SEND (*buffer*, *message*): if there is room in the bounded buffer *buffer*, insert message in the buffer. If not, stop the calling thread and wait until there is room.
- *message* ← RECEIVE (*buffer*): if there is a message in the bounded buffer *buffer*, return the message to the calling thread. If there is no message in the bounded buffer, stop the calling thread and wait until another thread sends a message to buffer *buffer*.

SEND and RECEIVE with bounded buffers allow sending and receiving messages as described in Chapter 4. By building stubs that use these primitives, we can implement remote procedure calls between threads on the same physical computer in the same

way as remote procedure calls between physical computers. That is, from the client's point of view in Figure 4.8, there is no difference between sending a message to a local virtual computer or to a remote physical computer. In both cases, if the client or service module fails because of a programming error, then the other module needs to provide a recovery strategy, but it doesn't necessarily fail.

### 5.2.2 Sequence Coordination with a Bounded Buffer

The implementation with bounded buffers requires coordination between sending and receiving threads because a thread may have to wait until buffer space is available or until a message arrives. Two quite different approaches to thread coordination have developed over the years by researchers in different fields. One approach, usually taken by operating system designers, assumes that the programmer is an all-knowing genius who makes no mistakes. The other approach, usually taken by database designers, assumes that the programmer is a mere mortal, so it provides strong automatic support for coordination correctness, but at some cost in flexibility.

The next couple of subsections exhibit the genius approach to coordination, not because it is the best way to tackle coordination problems, but rather to give some intuition about why it requires a coordination genius, and thus should be subcontracted to such a specialist whenever possible. In addition, to implement the database approach the designer of the automatic coordination support approach must use the genius approach. Chapter 9 [on-line] uses the concepts introduced in this chapter to implement the database approach for mere mortals.

The scenario is that we have two threads (a sending thread and a receiving thread) that share a buffer into which the sender puts messages and the receiver removes those messages. For clarity we will assume that the sending and receiving thread each have their own processor allocated to them; that is, for the rest of this section we can equate thread with processor, and thus threads can proceed concurrently at independent rates. As mentioned earlier, Section 5.5 will explore what happens when we eliminate that assumption.

The buffer is bounded, which means that it has a fixed size. To ensure that the buffer doesn't overflow, the sending thread should hold off putting messages into the buffer when the number of messages there reaches some predefined limit. When that happens, the sender must wait until the receiver has consumed some messages.

The problem of sharing a bounded buffer between two threads is an instance of the *producer and consumer problem*. For correct operation, the consumer and the producer must coordinate their activities. In our example, the constraint is that the producer must first add a message to the shared buffer before the consumer can remove it and that the producer must wait for the consumer to catch up when the buffer fills up. This kind of coordination is an example of *sequence coordination*: a coordination constraint among threads stating that, for correctness, an event in one thread must precede an event in another thread.

Figure 5.5 shows an implementation of SEND and RECEIVE using a bounded buffer. This implementation requires making some subtle assumptions, but before diving

```
1   shared structure buffer                    // A shared bounded buffer
2       message instance message[N]            // With a maximum of N messages
3       integer in initially 0                 // Counts number of messages put in the buffer
4       integer out initially 0                // Counts number of messages taken out of the buffer

5   procedure SEND (buffer reference p, message instance msg)
6       while p.in − p.out = N do nothing       // If buffer is full, wait for room
7       p.message [p.in modulo N] ← msg         // Put message in the buffer
8       p.in ← p.in + 1                         // Increment in

9   procedure RECEIVE (buffer reference p)
10      while p.in = p.out do nothing           // If buffer is empty, wait for message
11      msg ← p.message [p.out modulo N]        // Copy item out of buffer
12      p.out ← p.out + 1                       // Increment out
13      return msg                              // Return message to receiver
```

**FIGURE 5.5**

An implementation of a SEND and RECEIVE using bounded buffers.

into these assumptions let's first consider how the program works. The two threads implement the sequence coordination constraint using N (the size of the shared bounded buffer) and the variables *in* (the number of items produced) and *out* (the number of items consumed). If the buffer contains items (i.e., *in* > *out* on line *10*), then the receiver can proceed to consume the items; otherwise, it loops until the sender has put some items in the buffer. Loops in which a thread is waiting for an event without giving up its processor are called *spin loops*.

To ensure that the sender waits when the buffer is full, the sender puts new items in the buffer only if *in* − *out* < N (line *6*); otherwise, it spins until the receiver made room in the buffer by consuming some items. This design ensures that the buffer does not overflow.

The correctness of this implementation relies on several assumptions:

1. The implementation assumes that there is one sending thread and one receiving thread and that only one thread updates each shared variable. In the program only the receiver thread updates *out,* and only the sender thread updates *in*. If several threads update the same shared variable (e.g., multiple sending threads update *in* or the receiving thread and the sending thread update a variable), then the updates to the shared variable must be coordinated, which this implementation doesn't do.

   This assumption exemplifies the principle that coordination is simplest when each shared variable has just one writer:

**One-writer principle**

*If each variable has only one writer, coordination becomes easier.*

That is, if you can, arrange your program so that two threads don't update the same shared variable. Following this principle also improves modularity because information flows in only one direction: from the single writer to the reader. In our implementation, *out* contains information that flows from the receiver thread to the sender, and *in* contains information that flows from the sender thread to the receiver. This restriction of information flow simplifies correctness arguments and, as we will see in Chapter 11 [on-line], can also enhance security.

A similar observation holds for the way the bounded buffer *buffer* is implemented. Because *messages* is a fixed-size array, the entries are written only by the sender thread. If the buffer had been implemented as a linked list, we might have a situation in which the sender and the receiver need to update a shared variable at the same time (e.g., the pointer to the head of the linked list) and then these updates would have to be coordinated.

2. The spin loops on lines *6* and *10* require the previously mentioned assumption that the sender and the receiver threads each run on a dedicated processor. When we remove that assumption in Section 5.5 we will have to do something about these spin loops.

3. This implementation assumes that the variables *in* and *out* are integers whose representation must be large enough that they will never overflow for the life of the buffer. Integers of width 64 or 96 bits would probably suffice for most applications. (An alternative way to remove this assumption is to make the implementation of the bounded buffer more complicated: perform all additions involving *in* and *out* modulo *N,* and reserve one additional slot in the buffer to distinguish a full buffer from an empty one.)

4. The implementation assumes that the shared memory provides read/write coherence (see Section 2.1.1.1) for *in* and *out*. That is, a LOAD of the variable *in* or *out* by one thread must be guaranteed to obtain the result of the most recent store to that variable by the other thread.

5. The implementation assumes before-or-after atomicity for the variables *in* and *out*. If these two variables fit in a 16- or 32-bit memory cell that can be read and written with a single LOAD or STORE, this assumption is likely to be true. But a 64- or 96-bit integer would probably require multiple memory cells. If they do, reading and writing *in* and *out* would require multiple LOADs or STOREs, and additional measures will be necessary to make these multistep sequences atomic.

6. The implementation assumes that the result of executing a statement becomes visible to other threads in program order. If an optimizing compiler or processor reorders statements to achieve better performance, this program could work incorrectly. For example, if the compiler generates code that reads *p.in* once, holds it in a temporary register for use in lines *6* through *8*, and updates the memory copy of *p.in* immediately, then the receiver may read the contents of the *in*th entry of the shared buffer before the sender has copied its message into that entry.

The rest of this section explains what problems occur when assumptions *1* (the one-writer principle) and *5* (before-or-after atomicity of multistep LOAD and STORE sequences) don't hold and introduces techniques to ensure them. In Section 5.5 we will find out how to remove assumption *2* (more processors than threads). Throughout, we assume that assumptions *3*, *4*, and *6* always hold.

### 5.2.3 Race Conditions

To illustrate the importance of the six assumptions in guaranteeing the correctness of the program in Figure 5.5, let's remove two of those assumptions, one at a time, to see just what goes wrong. What we will find is that to deal with the removed assumptions we need additional mechanisms, mechanisms that Section 5.2.4 will introduce. This illustration reinforces the observation that concurrent programming needs the attention of specialists: all it takes is one subtle change to make a correct program wrong.

To remove the first assumption, let's allow several senders and receivers. This change will violate the one-writer principle, so we should not be surprised to find that it introduces errors. Multiple senders and receivers are common in practice. For example, consider a printer that is shared among many clients. The service managing the printer may receive requests from several clients. Each request adds a document to the shared buffer of to-be-printed documents. In this case, we have several senders (the threads adding jobs to the buffer) and one receiver (the printer).

As we will see, the errors that will manifest themselves are difficult to track down because they don't always show up. They appear only with a particular ordering of the instructions of the threads involved. Thus, concurrent programs are not only difficult to get right, but also difficult to debug when they are wrong.

The solution in Figure 5.5 doesn't work when several senders execute the code concurrently. To see why, let's assume N is 20 and that all entries in the buffer are empty (e.g., *out* is 0 and *in* is 0), and each thread is running on its own processor:

If two sending threads run concurrently—one on processor A and one on processor B—the threads issue instructions independently of each other, at their own

pace. The processors may have different speeds and take interrupts at different times, or instructions may hit in the cache on one processor and miss on another, so there is no way to predict the relative timing of the LOAD and STORE instructions that the threads issue.

This scenario is an instance of asynchronous interpreters (described in Section 2.1.2). Thus, we should make no assumptions about the sequence in which the memory operations of the two threads execute. When analyzing the concurrent execution of two threads, both executing instructions *6* through *8* in Figure 5.5, we can assume they execute in some *serial* sequence (because the bus arbiter will order any memory operations that arrive at the bus at the same time). However, because the relative speeds of the threads are unpredictable, we can make no assumptions about the order in the sequence.

We represent the execution of instruction *6* by thread A as "*A6*". Using this representation, we see that one possible sequence might be as follows: *A6, A7, A8, B6, B7, B8*. In this case, the program works as expected. Suppose we just started, so variables *in* and *out* are both zero. Thread *A* performs all of its three instructions before thread *B* performs any of its three instructions. With this order, thread *A* inserts an item in entry 0 and increments *in* from 0 to 1. Thread *B* adds an item in entry 1 and increments *in* from 1 to 2.

Another possible, but undesirable, sequence is *A6, B6, B7, A7, A8, B8*, which corresponds to the following timing diagram:

With this order, thread *A*, at *A6*, discovers that entry 0 of the buffer is free. Then, at *B6*, *B* also discovers that buffer entry 0 is free. At *B7*, *B* stores an item in entry 0 of *buffer*. Then, *A* proceeds: at *A7* it also stores an item in entry 0, overwriting *B*'s item. Then, both increment *in* (*A8* and *B8*), setting *in* first to 1 and then to 2. Thus, at the end of this order of instructions, one print job is lost (thread *B*'s job), and (because both threads incremented *in*) the receiver will find that entry 1 in the buffer was never filled in.

This type of error is called a *race condition* because it depends on the exact timing of two threads. Whether or not an error happens cannot be controlled. It is nasty, since some sequences deliver a correct result and some sequences deliver an incorrect result.

Worse, small timing changes between invocations might result in different behavior. If we notice that *B*'s print job was lost and we run it again to see what went wrong, we might get a correct result on the retry because the relative timing of

the instructions has changed slightly. In particular, if we add instructions (e.g., for debugging) on the retry, it is almost guaranteed that the timing is changed (because the threads execute additional instructions) and we will observe a different behavior. Bugs that disappear when the debugger starts to close in on them are colloquially called "Heisenbugs" in a tongue-in-cheek pun on the Heisenberg uncertainty principle of quantum mechanics. Heisenbugs are difficult to reproduce, which makes debugging difficult.

Race conditions are the primary pitfall in writing concurrent programs and the main reason why developing concurrent programs should be left to specialists, despite the existence of tools to help identifying races (e.g., see Savage et al. [Suggestions for Further Reading 5.5.6]). Concurrent programming is subtle. In fact, with several senders the program of Figure 5.5 has a second race condition. Consider the statement *8* that the senders execute:

$in \leftarrow in + 1$

In reality, a thread executes this statement in three separate steps, which can be expressed as follows:

```
1  LOAD in, R0      // Load the value of in into a register
2  ADD R0, 1        // Increment
3  STORE R0, in     // Store result back to in
```

Consider two sending threads running simultaneously, threads *A* and *B*, respectively. The instructions of the threads might execute in the sequence *A1*, *A2*, *A3*, *B1*, *B2*, *B3*, which corresponds to the following timing diagram:

In this case *in* is incremented by two, as the programmer intended.

But now consider the execution sequence *A1*, *B1*, *A2*, *A3*, *B2*, *B3*, which corresponds to the following timing diagram:

When the two threads finish, this ordering of memory references has increased *in* by only 1. At *A1*, thread *A* loads the R0 register of its thread with the value of *in*, which is 0. At *B1*, thread *B* does exactly the same thing, loading its thread's register R0 with the value 0. Then, at *A2*, thread A computes the new value in R0 and at *A3* updates *in* with the value 1. Next, at *B2* and *B3*, thread *B* does the same thing: it computes the new value in R0 and updates *in* with the value 1. Thus, *in* ends up containing 1 instead of the intended 2. Any time two threads update a shared variable concurrently (i.e., the one-writer principle is violated), a race condition is possible.

We caused this second race condition by allowing multiple senders. But the manipulation of the variables *in* and *out* also has a potential race even if there is only one sender and one receiver, and we remove assumption *5* (the before-or-after atomicity requirement). Let's assume that we want to make *in* and *out* of type **long integer** so that there is little risk of overflowing those two variables. In that case, *in* and *out* each span two memory cells instead of one, and updates to *in* and *out* are no longer atomic operations. That change creates yet another race.

If *in* is a long integer, then updating *in* would require two instructions:

```
1  STORE R0, in+1    // Update the least-significant word of in
2  STORE R1, in      // Update the most-significant word of in
```

To read *in* would also require two instructions:

```
3  LOAD in+1, R0     // Load the least-significant word of in into a register
4  LOAD in, R1       // Load the most-significant word of in into a register
```

If the sender executes instructions *1* and *2* at about the same time that the receiver executes instructions *3* and *4*, a race condition could manifest itself. Let's assume that two threads call SEND and RECEIVE $2^{32}-1$ times, and interleave their calls perfectly. At this point there are no messages in the buffer and $in = out = 00000000FFFFFFFF_{hex}$ using big-endian notation.

Let's consider the scenario in which thread *A* has just added a message to the buffer, has read *in* into R0 and R1 (at instructions 3 and *4*), has computed the new value for *in* in the registers R0 and R1, and has executed instruction *1* to update *in* to memory. But before A executes instruction *2*, thread *B* adds a message:

In this case, the program works incorrectly because *A* has stored a message in entry 15 of the buffer ($00000000FFFFFFFF_{hex}$ **modulo** 20 = 15), *B* stores a message in entry 0, and *A* completes the update of *in*, which sets *in* to $0000000100000000_{hex}$. *B*'s message in entry 0 will be lost because entry 0 will be overwritten by the next caller to SEND.

Race conditions are not uncommon in complex systems. Two notorious ones occurred in CTSS and in the Therac-25 machine. In CTSS, an early operating system, all running instances of a text editor used the same name for temporary files. At some point, two administrators were concurrently editing the file with the message of the day and a file containing passwords. The content of the two files ended up being exchanged (see Section 11.11.2 [on-line] for the details): when users logged into CTSS, it displayed the pass phrases of all other users as the message of the day.

The Therac-25 is a machine that delivers medical irradiation to human patients [Suggestions for Further Reading 1.9.5]. A race condition between a thread and the operator allowed an incorrect radiation intensity to be set: as a result, some patients died. The repairman could not reproduce the problem, since he typed more slowly than the more experienced operator of the machine.

Problem sets *4*, *5*, and *6* ask the reader to find race conditions in a few small, concurrent code fragments.

### 5.2.4 Locks and Before-or-After Actions

From the examples in the preceding section we can see that the program in Figure 5.5 was carefully written so that it didn't violate assumptions *1* and *5*. If we make slight modifications to the program or use the program in slightly different ways than it was intended to be used, we violate the assumptions and the program exhibits race conditions. We would like a technique by which a developer can systematically avoid race conditions. This section introduces a mechanism called a *lock*, with which a designer can make a multistep operation behave like a single-step operation. By using locks carefully, we can modify the program in Figure 5.5 so that it enforces assumptions *1* and *5*, and thus avoids the race conditions systematically.

A lock is a shared variable that acts as a flag to coordinate usage of other shared variables. To work with locks we introduce two new primitives: ACQUIRE and RELEASE, both of which take the name of a lock as an argument. A thread may ACQUIRE a lock, hold it for a while, and then RELEASE it. While a thread is holding a lock, other threads that attempt to acquire that same lock will wait until the first thread releases the lock. By surrounding multistep operations involving shared variables with ACQUIRE and RELEASE, the designer can make the multistep operation on shared variables behave like a single-step operation and avoid undesirable interleavings of multistep operations.

Figure 5.6 shows the code of Figure 5.5 with the addition of ACQUIRE and RELEASE invocations. The modified program uses only one lock (*buffer_lock*) because there is a single data structure that must protected. The lock guarantees that the program works correctly when there are several senders and receivers. It also guarantees correctness when *in* and *out* are long integers. That is, the two assumptions under which the program of Figure 5.5 is correct are now guaranteed by the program itself.

The ACQUIRE and RELEASE invocations make the reads and writes of the shared variables *p.in* and *p.out* behave like a single-step operation. The lock set by ACQUIRE and RELEASE ensures the test, and manipulation of the buffer is executed as one indivisible action; thus, no undesirable interleavings and races can happen. If two threads attempt

to execute the multistep operation between ACQUIRE and RELEASE concurrently, one thread acquires the lock and finishes the complete multistep operation before the other thread starts on the operation. The ACQUIRE and RELEASE primitives have the effect of dynamically implementing the one-writer principle on those variables: they ensure there is only a single writer at any instant, but the identity of the writer can change.

It is important to keep in mind that when a thread acquires a lock, the shared variables that the lock is supposed to protect are not mechanically protected from access by other threads. Any thread can still read or write those variables without acquiring the lock. The lock variable merely acts as a flag, and for correct coordination all threads must honor an additional convention: they must not perform operations on shared variables unless they hold the lock. If any thread fails to honor that convention, there may be undesirable interleavings and races.

To ensure correctness in the presence of concurrent threads, a designer must identify *all* potential races and carefully insert invocations of ACQUIRE and RELEASE to prevent them. If the locking statements don't ensure that multistep operations on shared variables appear as single-step operations, then the program may have a race condition. For example, if in the SEND procedure of Figure 5.6 the programmer places the ACQUIRE and RELEASE statements around just the statements on lines *11* through

```
1   shared structure buffer                    // A shared bounded buffer
2       message instance message[N]            // with a maximum of N messages
3       long integer in initially 0            // Counts number of messages put in the buffer
4       long integer out initially 0           // Counts number of messages taken out of the buffer
5       lock instance buffer_lock initially UNLOCKED // Lock to order sender and receiver

6   procedure SEND (buffer reference p, message instance msg)
7       ACQUIRE (p.buffer_lock)
8       while p.in − p.out = N do              // Wait until there is room in the buffer
9           RELEASE (p.buffer_lock)            // While waiting release lock so that RECEIVE
10          ACQUIRE (p.buffer_lock)            // can remove a message
11      p.message[p.in modulo N] ← msg         // Put message in the buffer
12      p.in ← p.in + 1                        // Increment in
13      RELEASE (p.buffer_lock)

14  procedure RECEIVE (buffer reference p)
15      ACQUIRE (p.buffer_lock)
16      while p.in = p.out do                  // Wait until there is a message to receive
17          RELEASE (p.buffer_lock)            // While waiting release lock so that SEND
18          ACQUIRE (p.buffer_lock)            // can add a message
19      msg ← p.message[p.out modulo N]        // Copy item out of buffer
20      p.out ← p.out + 1                      // Increment out
21      RELEASE (p.buffer_lock)
22      return msg
```

**FIGURE 5.6**

An implementation of SEND and RECEIVE that adds locks so that there can be multiple senders and receivers. The RELEASE and ACQUIRE on lines 9 and 10 are explained in Section 5.25.

*12*, then several race conditions may happen. If the lock doesn't protect the test of whether there's space in the buffer (line *8*), a buffer with only one space free could be appended to by multiple concurrent invocations to SEND. Also, before-or-after atomicity for *in* and *out* (assumption *5*) could be violated during the comparisons of *p.in* with *p.out*, so the race described in Section 5.2.3 could still occur. Programming with locks requires great attention to detail. Chapter 9 [on-line] will explore schemes that allow the designer to systematically ensure correctness for multistep operations involving shared variables.

A lock can be used to implement before-or-after atomicity. During the time that a thread holds a lock that protects one or more shared variables, it can perform a multistep operation on these shared variables. Because other threads that honor the lock protocol will not concurrently read or write any of the shared variables, from their point of view the multiple steps of the first thread appear to happen indivisibly: before the lock is acquired, none of the steps have occurred; after the lock is released all of them are complete. Any operation by a concurrent thread must happen either completely before or completely after the before-or-after atomic action.

The need for before-or-after atomicity has been realized in different contexts, and as a result that concept and before-or-after atomic actions are known by various names. The database literature uses the terms *isolation* and *isolated actions*; the operating system literature uses the terms *mutual exclusion* and *critical sections*; and the computer architecture literature uses the terms *atomicity* and *atomic actions*. Because Chapter 9 [on-line] introduces a second kind of atomicity, this text uses the qualified term "before-or-after atomicity" for precision as well as for its self-defining and mnemonic features.

In general, in the computer science community, a tremendous amount of work has been done on approaches to finding race conditions in programs and on approaches to avoid them in the first place. This text introduces the fundamental ideas in concurrent programming, but the interested reader is encouraged to explore the literature to learn more.

The usual implementation of ACQUIRE and RELEASE guarantees that only a single thread can acquire a given lock at any one time. This requirement is called the *single-acquire protocol*. If the programmer knows more details about how the protected shared variables will be used, a more relaxed protocol may be able to allow more concurrency. For example, Section 9.5.4 describes a multiple-reader, single-writer locking protocol.

In larger programs with many shared data structures, a programmer often uses several locks. For example, if each of the several data structures is used by different operations, then we might introduce a separate lock for each shared data structure. That way, the operations that use different shared data structures can proceed concurrently. If the program used just one lock to protect all of the data structures, then all operations would be serialized by the lock. On the other hand, using several locks raises the complexity of understanding a program by another notch, as we will see next.

Problem sets *4* and *5* explore several possible locations for ACQUIRE and RELEASE statements in an attempt to remove a race condition while still allowing for concurrent execution of some operations. Birrell's tutorial [Suggestions for Further Reading

5.3.1] provides a nice introduction on how to write concurrent programs with threads and locks.

### 5.2.5 Deadlock

A programmer must use locks with care, because it is easy to create other undesirable situations that are as bad as race conditions. For example, using locks, a programmer can create a *deadlock*, which is an undesirable interaction among a group of threads in which each thread is waiting for some other thread in the group to make progress.

Consider two threads, A and B, that both must acquire two locks, *L1* and *L2*, before they can proceed with their task:

| Thread A | Thread B |
|---|---|
| ACQUIRE(*L1*) | ACQUIRE(*L2*) |
| ACQUIRE(*L2*) | ACQUIRE(*L1*) |

This code fragment has a race condition that results in deadlock, as shown in the following timing diagram:

Thread A cannot make forward progress because thread B has acquired *L2*, and thread B cannot make forward progress because thread A has acquired *L1*. The threads are in a deadly embrace.

If we had modified the code so that both threads acquire the locks in the same order (*L1* and then *L2*, or vice versa), then no deadlock could have occurred. Again, small changes in the order of statements can result in good or bad behavior.

A convenient way to represent deadlocks is using a *wait-for* graph. The nodes in a wait-for graph are threads and resources such as locks. When a thread acquires a lock, it inserts a directed edge from the lock node to the thread node. When a thread must wait for a resource, it inserts another directed edge from the thread node to the resource node. As an example, the race condition with threads A, B, and locks *L1* and *L2* results in the following wait-for graph:

When thread A acquires lock *L1*, it inserts arrow 1. When thread B acquires lock *L2*, it inserts arrow 2. When thread A must wait for lock *L2*, it inserts arrow 3. When thread B attempts to acquire lock *L1* but must wait, it inserts arrow 4. When a thread must wait, we check if the wait-for graph contains a cycle. A cycle indicates deadlock: everyone is waiting for someone else to release a resource. In general, if, and only if, a wait-for graph contains a cycle, then threads are deadlocked.

When there are several locks, a good programming strategy to avoid deadlock is to enumerate all lock usages and ensure that all threads of the program acquire the locks in the same order. This rule will ensure there can be no cycles in the wait-for graph and thus no deadlocks. In our example, if thread B above did ACQUIRE(*L1*) before ACQUIRE (*L2*), the same order that thread A used, then there wouldn't have been a problem. In our example program, it is easy for the programmer to modify the program to ensure that locks are acquired in the same order because the ACQUIRE statements are shown next to each other and there are only two locks. In a real program, however, the four ACQUIRE statements may be buried deep inside two separate modules that threads A and B happen to call indirectly in different orders, and ensuring that all locks are acquired in a static global order requires careful thinking and design.

A deadlock doesn't always have to involve multiple locks. For example, if the sender forgets to release and acquire the lock on lines *9* and *10* of Figure 5.6, then a deadlock is also possible. If the buffer is full, the receiver will not get a chance to remove a message from the buffer because it cannot acquire the lock, which is being held by the sender. In this case, the sender is waiting on the receiver to change the value of *p.out* (in a wait-for graph, the resource is buffer space represented by the value of *p.out*), and the receiver is waiting on the sender to release the lock. Simple programming errors can lead to deadlocks.

A problem related to deadlock is *livelock*—an interaction among a group of threads in which each thread is repeatedly performing some operations but is never able to complete the whole sequence of operations. An example of livelock is given in Sidebar 5.2, which presents an algorithm to implement ACQUIRE and RELEASE.

### 5.2.6 Implementing ACQUIRE and RELEASE

A correct implementation of ACQUIRE and RELEASE must enforce the single-acquire protocol. Several threads may attempt to acquire the lock at the same time, but only one should succeed. This requirement makes the implementation of locks challenging. In essence, we must make sure that ACQUIRE itself is a before-or-after action.

To see what goes wrong if ACQUIRE is not a before-or-after action, consider the too-simple implementation of ACQUIRE as shown in Figure 5.7. This implementation is broken because it has a race condition. If two threads labeled A and B call

FAULTY_ACQUIRE at the same time, the threads may execute the statements in the order A5, B5, A6, B6, which corresponds to the following timing diagram:

The result of this sequence of events is that both threads acquire the lock, which violates the single-acquire protocol.

The faulty ACQUIRE has a multistep operation on a shared variable (the lock), and we must ensure in some way that ACQUIRE itself is a before-or-after action. Once ACQUIRE is a before-or-after action, we can use it to turn arbitrary multistep operations on shared variables into before-or-after actions. This reduction is an example of a technique called *bootstrapping*, which resembles an inductive proof. Bootstrapping means that we look for a systematic way to reduce a general problem (e.g., making multistep operations on shared variables before-or-after actions) to some much-narrower particular version of the same problem (e.g., making an operation on a single shared lock a before-or-after action). We then solve the narrow problem using some specialized method that might work for only that case because it takes advantage of the specific situation. The general solution then consists of two parts: a method for solving the special case and a method for reducing the general problem to the special case. In the case of ACQUIRE, the solution for the specific problem is either building a special hardware instruction that is itself a before-or-after action or programming very carefully.

```
structure lock
    integer state

procedure FAULTY_ACQUIRE (lock reference L)
    while L.state = LOCKED do nothing      // spin until L is UNLOCKED
    L.state ← LOCKED                       // the while test failed, got the lock

procedure RELEASE (lock reference L)
    L.state ← UNLOCKED
```

**FIGURE 5.7**

Incorrect implementation of ACQUIRE. LOCKED and UNLOCKED are constants that have different values; for example, LOCKED is 1 and UNLOCKED is 0.

We first look at a solution involving a special instruction, *Read and Set Memory* (RSM). RSM performs the statements in the block **do atomic** as a before-or-after action:

```
1    procedure RSM (reference mem)     // RSM memory location mem
2        do atomic
3            r ← mem                   // Load value stored at mem into r
4            mem ← LOCKED              // Store LOCKED into memory location mem
5        return r
```

Most modern computers implement some version of the RSM procedure in hardware, as an extension to the memory abstraction. RSM is then often called *test-and-set*; see Sidebar 5.1. For the RSM instruction to be a before-or-after action, the bus arbiter that controls the bus connecting the processors to the memory must guarantee that the LOAD (line *3*) and STORE (line *4*) instruction execute as before-or-after actions—for example, by allowing the processor to read a value from a memory location and to write a new value into that same location in a single bus cycle. We have thus pushed the problem of providing a before-or-after action down to the bus arbiter, a piece of hardware whose precise function is turning bus operations into before-or-after

**Sidebar 5.1 RSM, Test-and-Set, and Avoiding Locks** RSM is often called "test-and-set" or "test-and-set-locked" for accidental reasons. An early version of the instruction tested the lock and performed the store only if the test showed that the lock was not set. The instruction also set a bit that the software could test to find out whether or not the lock had been set. Using this instruction, one can implement the body of ACQUIRE as follows:

```
while TEST_AND_SET (L) = LOCKED do nothing
```

This version appears to be shorter than the one shown in Figure 5.8, but the hardware performs a test that is redundant. Thus, later hardware designers removed the test from test-and-set, but the name stuck.

In addition to RSM, there are many other instructions, including "test-and-test-and-set" (which allows for a more efficient implementation of a spin lock) and COMPARE_AND_SWAP (*v1, m, v2*) (which compares, in a before-or-after action the content of a memory location *m* to the value *v1* and, if they are the same, stores *v2* in *m*). The "compare-and-swap" instruction can be used, for example, to implement a linked list in which threads can insert elements concurrently without having to use locks, avoiding the risk of spinning until other threads have completed their insert [see Suggestions for Further Reading 5.5.8 and 5.5.9]. Such implementations are called *non-blocking*.

The Linux kernel uses yet another form of coordination that avoids locks. It is called read-copy update and is tailored to data structures that are mostly read and infrequently updated [see Suggestions for Further Reading 5.5.7].

```
1    procedure ACQUIRE (lock reference L)
2        R1 ← RSM (L.state)                  // read and set lock L
3        while R1 = LOCKED do                // was it already locked?
4            R1 ← RSM(L.state)               // yes, do it again, till we see it wasn't
5
6    procedure RELEASE (lock reference L)
7        L.state ← UNLOCKED
```

**FIGURE 5.8**

ACQUIRE and RELEASE using RSM.

actions: the arbiter guarantees that if two requests arrive at the same time, one of those requests is executed completely before the other begins.

Using the RSM instruction, we can implement any other before-or-after action. It is the one essential before-or-after action from which we can bootstrap any other set of before-or-after actions. Using RSM, we can implement ACQUIRE and RELEASE as shown in Figure 5.8. This implementation follows the single-acquire protocol: if *L* is LOCKED, then one thread has the lock *L*; if *L* contains UNLOCKED, then no thread has acquired the lock *L*.

To see that the implementation is correct, let's assume that *L* is UNLOCKED. If some thread calls ACQUIRE (*L*), then after RSM, *L* is LOCKED and R1 contains UNLOCKED, so that thread has acquired the lock. The next thread that calls ACQUIRE (*L*) sees LOCKED in R1 after the RSM instruction, signaling that some other thread holds the lock. The thread that tried to acquire will spin until R1 contains UNLOCKED. When releasing a lock, no test is needed, so an ordinary STORE instruction can do the job without creating a race condition.

This implementation assumes that the shared memory provides read/write coherence. For example, if a manager thread sets *L* to UNLOCKED on line *7*, then we assume that the thread observes that store and falls out of the spinning loop on line *3* in ACQUIRE. Some memories provide more relaxed semantics than read/write coherence; in that case, additional mechanisms are needed to make this program work correctly.

With this implementation, even a single thread can deadlock itself by calling ACQUIRE twice on the same lock. With the first call to ACQUIRE, the thread obtains the lock. With the second call to ACQUIRE the thread deadlocks, since some thread (itself) already holds the lock. By storing the thread identifier of the lock's owner in *L* (instead of true or false), ACQUIRE could check for this problem and return an error.

Problem set *6* explores concurrency issues using a SET-AND-GET remote procedure call, which executes as a before-or-after action.

### 5.2.7 Implementing a Before-or-After Action Using the One-Writer Principle

The RSM instruction can also be implemented without extending the memory abstraction. In fact, one can implement RSM as a procedure in software using ordinary load and store instructions, but such implementations are complex. The key

**Sidebar 5.2 Constructing a Before-or-After Action Without Special Instructions**

In 1959, E. Dijkstra, a well-known Dutch programmer and researcher, posed to his colleagues the problem of providing a before-or-after action with ordinary read and write instructions as an amusing puzzle. Th. J. Dekker provided a solution for two threads, and Dijkstra generalized the idea into a solution for an arbitrary number of threads [Suggestions for Further Reading 5.5.2]. Subsequently, numerous researchers have looked for provable, efficient solutions. We present a simple implementation of RSM based on L. Lamport's solution. Lamport's solution, like other solutions, relies on the existence of a bus arbiter that guarantees that any single LOAD or STORE is a before-or-after action with respect to every other LOAD and STORE. Given this assumption, RSM can be implemented as follows:

```
shared boolean flag[N]                          // one boolean per thread

1   procedure RSM (lock reference L)            // set lock L and return old value
2       do forever                              // me is my index in flag
3           flag[me] ← TRUE                     // warn other threads
4           if ANYONE_ELSE_INTERESTED (me) then // is another thread warning us?
5            flag[me] ← FALSE                   // yes, reset my warning, try again
6           else
7            R ← L.state                        // set R to value of lock
8            L.state ← LOCKED                   // and set the lock
9            flag[me] ← FALSE
10           return R
11
12  procedure ANYONE_ELSE_INTERESTED (me)       // is another thread updating L?
13      for i from 0 to N–1 do
14          if i ≠ me and flag[i] = TRUE then return TRUE
15      return FALSE
```

To guarantee that RSM is indeed a before-or-after action, we need to assume that each entry of the shared array is in its own memory cell, that the memory provides read/write coherence for memory cells, and that the instructions execute in program order, as we did for the sender and receiver in Figure 5.5.

Under these assumptions, RSM ensures that the shared variable *L* is never written by two threads at the same time. Each thread has a unique number, *me*. Before *me* is allowed to write *L*, it must express its interest in writing *L* by setting *me*'s entry in the boolean array *flag* (line *3*) and check that no other thread is interested in writing *L* (line *4*). If no other thread has expressed interest, then *me* acquires *L* (line *8*).

If two threads A and B call RSM at the same time, either A or B may acquire *L*, or both may retry, depending on how the shared memory system orders the accesses of A and B to the *flag*[i] array. There are three cases:

1. A sets *flag*[A], calls ANYONE_ELSE_INTERESTED, and reads flags at least as far as *flag*[B] before B sets *flag*[B]. In this case, A sees no other flags set and proceeds to acquire *L*;

(*Sidebar continues*)

B discovers A's flag and tries again. On its next try, B encounters no flags, but by the time B writes LOCKED to *L*, *L* is already set to LOCKED, so B's write will have no effect.

2. B sets *flag*[B], calls ANYONE_ELSE_INTERESTED, and reads flags at least as far as *flag*[A] before A sets *flag*[A]. In this case, B sees no other flags set and proceeds to acquire *L*; A discovers B's flag and tries again. On its next try, A encounters no flags, but by the time A writes *locked* to *L*, *L* is already set to *locked*, so A's write will have no effect.

3. A sets *flag*[A] and B sets *flag*[B] before either of them gets far enough through ANYONE_ELSE_INTERESTED to reach the other's flag location. In this case, both A and B reset their own *flag*[*i*] entries and try again. On the retry, all three cases are again possible.

The implementation of RSM has a livelock problem because the two threads A and B might end up in the final case (neither of them gets to update *L*), every time they retry. RSM could reduce the chance of livelock by inserting a random delay before retrying, a technique called *random backoff*. Chapter 7 [on-line] will refine the random backoff idea to make it applicable to a wider range of problems.

This implementation of RSM is not the most efficient one; it is linear in the number of threads because ANYONE_ELSE_INTERESTED reads all but one element of the array *flag*. More efficient versions of RSM exist, but even the best implementation [Suggestions for Further Reading 5.5.3] requires two loads and five stores (if there is no contention for *L*), which can be proven to be optimal under the given assumptions.

problem that our implementation without RSM of ACQUIRE has is that several threads are attempting to *modify* the same shared variable (*L* in our example). For two threads to read *L* concurrently is fine (the bus arbiter ensures that LOADs are before-or-after actions, and both threads will read the same value), but reading *and* modifying *L* is a multistep operation that must be performed as a before-or-after action. If not, this multistep operation can lead to a race condition in which the outcome may be a violation of the single-acquire protocol. This observation suggests an approach to implementing RSM based on the one-writer principle: ensure that only *one* thread modifies *L*.

Sidebar 5.2 describes a software solution that follows that approach. This software solution is complex compared to the hardware implementation of RSM. To ensure that only one thread writes *L*, the software solution requires an array with one entry per thread. Such an array must be allocated for each lock. Moreover, the number of memory accesses to acquire a lock is linear in the number of threads. Also, if threads are created dynamically, the software solution requires a more complex data structure than an array. Between the need for efficiency and the requirement for an array of unpredictable size, designers generally implement RSM as a hardware instruction that invokes a special bus cycle.

If one follows the *one-writer principle* carefully, one can write programs without locks (for example, as in Figure 5.5). This approach without locks can improve a program's performance because the expense of locks is avoided, but eliminating locks makes it more difficult to reason about the correctness.

The designers of the computer system for the space shuttle used many threads sharing many variables, and they deployed a systematic design approach to encourage a correct implementation. Designed in the late 1970s and early 1980s, the computers of the space shuttle were not efficient enough to follow the principled way of protecting all shared variables using locks. Understanding the risks of sharing variables among concurrent threads, however, the designers followed a rule that the program declaration for each unprotected shared variable must be accompanied by a comment, known as an *alibi*, explaining why no race conditions can occur even though that variable is unprotected. At each new release of the software, a team of engineers inspects all alibis and checks whether they still hold. Although this method has a high verification overhead, it helps discover many race conditions that otherwise might go undetected until too late. The use of alibis is an example of design for iteration.

### 5.2.8 Coordination between Synchronous Islands with Asynchronous Connections

As has been seen in this chapter, all implementations of before-or-after actions rely on bootstrapping from a properly functioning hardware arbiter. This reliance should catch the attention of hardware designers, who are aware that under certain conditions, it can be problematic (indeed, theoretically impossible) to implement a perfect arbiter. This section explains why and how hardware designers deal with this problem in practice. System designers need to be aware of how arbiters can fail, so that they know what questions to ask the designer of the hardware on which they rely.

The problem arises at the interface between asynchronous and synchronous components, when an arbiter that provides input to a synchronous subsystem is asked to choose between two asynchronous but closely spaced input signals. An asynchronous-input arbiter can enter a metastable state, with an output value somewhere between its two correct values or possibly oscillating between them at a high rate.* After applying asynchronous signals to an arbiter, one must therefore wait for the arbiter's output to settle. Although the probability that the output of the arbiter has not settled falls exponentially fast, for any given delay time some chance always remains that the arbiter has not settled yet, and a sample of its output may find it still changing. By waiting longer, one can reduce the probability of it not having settled to as small a figure as necessary for any particular application, but it is impossible to drive it to zero within a fixed time. Thus if the component that acquires the output of the arbiter is synchronous, when its clock ticks there is a chance that the component's

*Our colleague Andreas Reuter points out that the possibility that an arbiter may enter a metastable state has been of concern since antiquity: "How long halt ye between two opinions"?—*1 Kings* 18:21.

input (that is, the arbiter's output) is not ready. When that happens, the component may behave unpredictably, launching a chain of failure. Although the arbiter itself will certainly come to a decision at some point, not doing so before the clock ticks is known as *arbiter failure*.

Arbiter failure can be avoided in several ways:

- Synchronize the clocks of the two components. If the two processors, the arbiter, and the memory all operate with a common clock (more precisely, all of their interfaces are synchronous), arbiter design becomes straightforward. This technique is used, for example, to arbitrate access within some chips that have several processors.
- Design arbiters with multiple stages. Multiple stages do not eliminate the possibility of arbiter failure, but each additional stage multiplicatively reduces the probability of failure. The strategy is to provide enough stages that the probability of failure is so low that it can be neglected. With current technology, two or three stages are usually sufficient, and this technique is used in most interfaces between asynchronous and synchronous devices.
- Stop the clock of the synchronous component (thus effectively making it asynchronous) and wait for the arbiter's output to settle before restarting. In modern high-performance systems, clock distribution requires continuous ticks to provide signals for correcting phase errors, so one does not often encounter this technique in practice.
- Make all components asynchronous. The component that takes the output of the arbiter then simply waits until the arbiter reports that it has settled. A flurry of interest in asynchronous circuit design arose in the 1970s, but synchronous circuits proved to be easier to design and so won out. However, as clock speeds increase to the point that it is difficult to distribute clock even across a single chip, interest is reawakening.

Communication across a network link is nearly always asynchronous, communication between devices in the same box (for example, between a disk drive and a processor) is usually asynchronous, and as mentioned in the last bullet above, as advancing technology reduces gate delays, it is becoming challenging to maintain a common, fast-enough clock all the way across even a single chip. Thus, within-chip intercommunication is becoming more network-like, with synchronous islands connected by asynchronous links (see, for example, Suggestions for Further Reading 1.6.3).

As pointed out, arbiter failure is an issue only at the boundary between synchronous and asynchronous components. Over the years, that boundary has moved with changing technology. The authors are not aware of any current implementations of RSM () or their equivalents that cross a synchronous/asynchronous boundary (in other words, current multiprocessor practice is to use the method of the first bullet above). Thus, before-or-after atomicity based on RSM () is not at risk of arbiter failure. But that was not true in the past, and it may not be true again at

some point in the future. The system designer thus needs to be aware of where arbiters are being used and verify that they are specified appropriately for the application.

## 5.3 ENFORCING MODULARITY IN MEMORY

The implementation of bounded buffers took advantage of the fact that all threads share the same physical memory (see Figure 5.4 on page 209), but sharing memory does not enforce modularity well. A program may calculate a shared address incorrectly and write to a memory location that logically belongs to another module. To enforce modularity, we must ensure that the threads of one module cannot overwrite the data of another module. This section introduces domains and a memory manager to enforce memory boundaries, assuming that the address space is very large (i.e., so large that we can consider it unlimited). In Section 5.4, we will remove that assumption.

### 5.3.1 Enforcing Modularity with Domains

To contain the memory references of a thread, we restrict the thread's references to a *domain*, a contiguous range of memory addresses. When a programmer calls ALLOCATE_THREAD, the programmer specifies a domain in which the thread is to run. The thread manager records a thread's domain.

To enforce the rule that a thread should refer only to memory within its domain, we add a domain register to each processor and introduce a special interpreter, a *memory manager*, that is typically implemented in hardware and placed between a processor and the bus (see Figure 5.9). A processor's domain register contains the

**FIGURE 5.9**

An editor thread running with its domain.

lowest (*low*) and highest address (*high*) that the currently running thread is allowed to use. ALLOCATE_THREAD loads the processor's domain register with the thread's domain.

The memory manager checks for each memory reference that the address is equal or higher than *low* and smaller than *high*. If it is, the memory manager issues the corresponding bus request. If not, it interrupts the processor, signaling a *memory reference exception*. The exception handler can then decide what to do. One option is to deliver an error message and destroy the thread.

This design ensures that a thread can make references only to addresses that are in its domain. Threads cannot overwrite or jump to memory locations of other threads.

This domain design achieves the main goal, but it lacks a number of desirable features:

1. A thread may need more than one domain. By using many domains, threads can control what memory they share and what memory they keep private. For example, a thread might allocate a domain for a bounded buffer and share that domain with another thread, but allocate private domains for the text of its program and private data structures.

2. A thread should be unable to change its own domain. That is, we must ensure that the thread cannot change the content of its processor's domain register directly or indirectly. If a thread can change the content of its processor's domain register, then the thread can make references to addresses that it shouldn't.

The rest of Section 5.3 adds these features.

### 5.3.2 Controlled Sharing Using Several Domains

To allow for sharing, we extend the design to allow each thread to have several domains and give each processor several domain registers, for the moment, as many as a thread needs. Now a designer can partition the memory of the programs shown in Figure 5.9 and control sharing. For example, a designer may split a client into four separate domains (see Figure 5.10): one domain containing the program text for the client thread, one domain containing the data for the client, one domain containing the stack of the client thread, and one domain containing the bounded message buffer. The designer may split a service in the same way. This setup allows both threads to use the shared bounded buffer domain, but restricts the other references of the threads to their private domains.

To manage this hardware design, we introduce a software component to the memory manager, which provides the following interface:

- *base_address* ← ALLOCATE_DOMAIN (*size*): Allocate a new domain of *size* bytes and return the base address of the domain.

- MAP_DOMAIN (*base_address*): Add the domain starting at address *base_address* to the calling thread's domains.

The memory manager can implement this interface by keeping a list of memory regions that are not in use, allocate *size* bytes of memory on an ALLOCATE_DOMAIN request, and maintain a *domain table* with allocated domains. An entry in the domain table records the *base_address* and *size*.

MAP_DOMAIN loads the domain's bounds from the domain table into a domain register of the thread's processor. If two or more threads map a domain, then that domain is shared among those threads.

We can improve the control mechanism by extending each domain register to record access permissions. In a typical design, a domain register might include three bits, which separately control permission to READ, WRITE, or EXECUTE (i.e., retrieve and use as an instruction) any of the bytes in the associated domain.

With these permissions, the designer can give the threads in Figure 5.10 EXECUTE and READ permissions to their text domains, and READ and WRITE permissions to their stack domains and the shared bounded buffer domain. This setup prevents a thread from taking instructions from its stack, which can help catch programming mistakes and also help avoid buffer overrun attacks (see Sidebar 11.4 [on-line]). Giving the program text only READ and EXECUTE permissions helps avoid the mistake of accidentally writing data into the text of the program.

The permissions also allow more controlled sharing: one thread can have access to a shared domain with only READ permission, whereas another thread can have READ and WRITE permissions.

To provide permissions, we modify the MAP_DOMAIN call as follows:

- MAP_DOMAIN (*base_address, permission*): loads the domain's bounds from the domain table into one of the calling thread's domain registers with permission *permission*.

To check permissions, the memory manager must know which permissions are needed for each memory reference. A LOAD instruction requires READ permission for its address, and thus the memory manager must check that the address is in a domain with READ access. A STORE instruction requires WRITE permission for its address, and

**FIGURE 5.10**

A client and service, each with three private domains and one shared domain.

thus the memory manager must check that the address is in a domain with WRITE access. To execute an instruction at the address in the PC requires EXECUTE permission. The domain holding instructions may also require READ permission because the program may have stored constants in the program text.

The pseudocode in Figure 5.11 details the check performed by the memory manager. Although we describe the function of the memory manager using pseudocode, in practice the memory manager is a hardware device that implements its function in digital circuitry. In addition, the memory manager is typically integrated with the processor so that the address checks run at the speed of the processor. As Section 5.3 develops, we will add more functions to the memory manager, some of which may be implemented in software as part of the operating system. Later, in Section 5.4.4, we discuss the trade-offs involved in implementing parts of the memory manager in software.

As shown in the figure, on a memory reference, the memory manager checks all the processor's domain registers. For each domain register, the memory manager calls CHECK_DOMAIN, which takes three arguments: the address the processor requested, a bit mask with the permissions needed by the current instruction, and the domain register. If *address* falls between *low* and *high* of the domain and if the permissions needed are a subset of permissions authorized for the domain, then CHECK_DOMAIN returns TRUE and the memory manager will issue the desired bus request. If *address* falls between *low* and *high* of the domain but the permissions needed aren't sufficient, then CHECK_DOMAIN interrupts the processor, indicating a memory reference exception as before. Now, however, it is useful to demultiplex the memory reference exception in two different categories: ***illegal memory reference exception*** and ***permission error exception***. If *address* doesn't fall in any domain, the exception handler indicates an illegal memory reference exception. If the address falls in a domain, but the threads didn't have sufficient permissions, the exception handler indicates a permission error exception.

```
procedure LOOKUP_AND_CHECK (integer address, perm_needed)
    for each domain do                  // for each domain register of this processor
        if CHECK_DOMAIN (address, perm_needed, domain) return domain
    signal memory_exception

procedure CHECK_DOMAIN (integer address, perm_needed, domain) returns boolean
    if domain.low ≤ address and address < domain.high then
        if PERMITTED (perm_needed, domain.permission) then return TRUE
        else signal permission_exception
    return FALSE

procedure PERMITTED (perm_needed, perm_authorized) returns boolean
    if perm_needed ⊂ perm_authorized then return TRUE // is perm_needed a subset?
    else return FALSE           // permission violation
```

**FIGURE 5.11**

The memory manager's pseudocode for looking up an address and checking permissions.

The demultiplexing of memory reference exceptions can be implemented either in hardware or software. If implemented in hardware, the memory manager signals an illegal memory reference exception or a permission error exception to the processor. If implemented in software, the memory manager signals a memory reference exception, and the exception handler for memory reference exceptions demultiplexes it by calling either the illegal memory reference exception handler or the permission error exception handler. As we will see in Chapter 6 (see Section 6.2.2), we will want to refine the categories of memory exceptions further. In processors in the field, some of the demultiplexing is implemented in hardware with further demultiplexing implemented in software in the exception handler.

In practice, only a subset of the possible combinations of permissions are useful. The ones used are: READ permission, READ and WRITE permissions, READ and EXECUTE permissions, and READ, WRITE, and EXECUTE permissions. The READ, WRITE, and EXECUTE combination of permissions might be used for a domain that contains a program that generates instructions and then jumps to the generated instructions, so-called self-modifying programs. Supporting self-modifying programs is risky, however, because this also allows an adversary to write a new procedure as data into a domain (e.g., using a buffer overrun attack) and then execute that procedure. In practice, self-modifying programs have proven to be more trouble than they are worth, except to system crackers. A domain with only EXECUTE permission or with just WRITE and EXECUTE permissions isn't useful in practice.

If memory-mapped I/O (described in Section 2.3.1) is used, then domain registers can also control which devices a thread can use. For example, a keyboard manager thread may have access to a domain that corresponds to the registers of the keyboard controller. If none of the other threads has access to this domain, then only the keyboard manager thread has access to the keyboard device. Thus, the same technique that controls separation of memory ranges can also control access to devices.

The memory manager can implement security policies because it controls which threads have access to which parts of memory. It can deny or grant a thread's request for allocating a new domain or for sharing an existing domain. In the same way, it can control which threads have access to which devices. How to implement such security policies is the topic of Chapter 11 [on-line].

### 5.3.3 More Enforced Modularity with Kernel and User Mode

Domain registers restrict the addresses to which a thread can make reference, but we must also ensure that a thread cannot change its domains. That is, we need a mechanism to control changes to the *low*, *high*, and *permission* fields of a domain register. To complete the enforcement of domains, we modify the processor to prevent threads from overwriting the content of domain registers as follows:

- Add one bit to the processor indicating whether the processor is in *kernel mode* or *user mode*, and modify the processor as follows. Instructions can change the value of the processor's domain registers only in kernel mode, and instructions

**FIGURE 5.12**

Threads with a kernel domain containing the memory manager and its domain table.

that attempt to change the domain register in user mode generate an *illegal instruction exception*. Similarly, instructions can change the mode bit only in kernel mode.

- Extend the set of permissions for a domain to include KERNEL-ONLY, and modify the processor to make it illegal for threads in user mode to reference addresses in a domain with KERNEL-ONLY permission. A thread in user mode that attempts to read or write memory with KERNEL-ONLY permission causes a permission error exception.
- Switch to kernel mode on an interrupt and on an exception so that the handler can process the interrupt (or exception) and invoke privileged instructions.

We can use the mechanisms as illustrated in Figure 5.12. Compared to Figure 5.10, each thread has two additional domains, which are marked K for KERNEL-ONLY. A thread must be in kernel mode to be able to make references to this domain. This domain contains the program text and data for the memory manager. These mechanisms ensure that a thread running in user mode cannot change its processor's domain registers; only when a thread executes in kernel mode can it change the processor domain registers. Furthermore, because the memory manager and its table are in kernel domains, a thread in user mode cannot change its domain information. We see that the kernel/user mode bit helps in enforcing modularity by restricting what threads in user mode can do.

### 5.3.4 Gates and Changing Modes

Because threads running in user mode cannot invoke procedures of the memory manager directly, a thread must have a way of changing from user mode to kernel mode and entering the memory manager in a controlled manner. If a thread could enter at an arbitrary address, it might create problems; for example, if a thread could enter a

domain with kernel permission at the instruction that sets the user-mode bit to KERNEL, it might be able to gain control of the processor with kernel privileges. To avoid this problem, a thread may enter a kernel domain only at certain addresses, called *gates*.

We implement gates by adding one more special instruction to the processor, the *supervisor call instruction* (SVC), which specifies in a register the name for the intended gate. Upon executing the SVC instruction, the processor performs two operations as one action:

1. Change the processor mode from user to kernel.
2. Set the PC to an address predefined by the hardware, the entry point of the gate manager.

The gate manager now has control in kernel mode; it can call the appropriate procedures to serve the thread's request. Typically, gate names are numbers, and the gate manager has a table that records for each gate number the corresponding procedure. For example, the table might map gate 0 to ALLOCATE_DOMAIN, gate 1 to MAP_DOMAIN, gate 2 to ALLOCATE_THREAD, and so on.

Implementing SVC has a slight complication: the steps to enter the kernel must happen as a before-or-after action: they must all be executed *without* interruption. If the processor can be interrupted in the middle of these steps, a thread might end up in kernel mode but with the program counter still pointing to an address in one of its user-level domains. Now the thread is executing instructions from an application module in kernel mode. To avoid this potential problem, processors complete all steps of an SVC instructions before executing another instruction.

When the thread wants to return to user mode, it executes the following instructions:

1. Change mode from kernel to user.
2. Load the program counter from the top of the stack into the processor's PC.

Processors don't have to perform these steps as a before-or-after action. After step 1, it is fine for a processor to return to kernel mode, for example, to process an outstanding interrupt.

The return sequence assumes that a thread has pushed the return address on its stack before invoking SVC. If the thread hasn't done so, then the worst that can happen when the thread returns to user mode is that it resumes at some arbitrary address, which might cause the thread to fail (as with many other programming errors), but it cannot create a problem for a domain with KERNEL-ONLY permission because the thread cannot refer to that domain in user mode.

The difference between entering and leaving kernel mode is that on leaving, the value loaded in the program counter isn't a predefined value. Instead, the kernel sets it to the saved address.

Gates can also be used to handle interrupts and exceptions. If the processor encounters an interrupt, the processor enters a special gate for interrupts and the gate manager dispatches the interrupt to the appropriate handler, based on the source of the interrupt (clock interrupt, permission error, illegal memory reference, divide by zero, etc.). Some

processors have a different gate for errors caused by exceptions (e.g., a permission error); others have one gate for interrupts (e.g., clock interrupt) and exceptions.

Problem set *9* explores in a minimal operating system the interactions between hardware and software for setting modes and handling interrupts.

### 5.3.5 Enforcing Modularity for Bounded Buffers

The implementation of SEND and RECEIVE in Figure 5.6 assumes that the sending and receiving threads share the bounded buffer, using, for example, a shared domain, as shown in Figure 5.12. This setup enforces a boundary between all domains of the threads, except for the domain containing the shared buffer. A thread can modify the shared buffer accidentally because both threads have write permissions to the shared domain. Thus, an error in one thread could indirectly affect the other thread; we would like to avoid that and enforce modularity for the bounded buffer.

We can protect the shared bounded buffer, too, by putting the buffer in a shared kernel domain (see Figure 5.13). Now the threads cannot directly write the shared buffer in user mode. The threads must transition to kernel mode to copy messages into the shared buffer. In this design, SEND and RECEIVE are supervisor calls. When a thread invokes SEND, it changes to kernel mode and copies the message from the sending thread's domain into the shared buffer. When the receiving thread invokes RECEIVE, it changes to kernel mode and copies a message from the shared buffer into the receiving's domain. As long as the program that is running in kernel mode is written carefully, this design provides stronger enforced modularity because threads in user mode have no direct access to a bounded buffer's messages.

This stronger enforced modularity comes at a performance cost for performing supervisor calls for SEND and RECEIVE. This cost can be significant because transitions between user mode and kernel mode can be expensive. The reason is that a processor typically maintains state in its pipeline and a cache as a speedup mechanism. This state

**FIGURE 5.13**

Threads with a kernel domain containing the shared buffer, the memory manager, and the domain table.

may have to be flushed or invalidated on a user-kernel mode transition because otherwise the processor may execute instructions that are still in the pipeline incorrectly.

Researchers have come up with techniques to reduce the performance cost by optimizing the kernel code paths for the SEND and RECEIVE supervisor calls, by having a combined call that sends and receives, by cleverly setting up domains to avoid the cost of copying large messages, by passing small arguments through processor registers, by choosing a suitable layout of data structures that reduces the cost of user-kernel mode transitions, and so on [Suggestions for Further Reading 6.2.1, 6.2.2, and 6.2.3]. Problem set 7 illustrates a lightweight remote procedure call implementation.

### 5.3.6 The Kernel

The collection of modules running in kernel mode is usually called the kernel program, or *kernel* for short. A question that arises is how the kernel and the first domain come into existence. Sidebar 5.3 details how a processor starts in kernel mode with domain checking disabled, how the kernel can then bootstrap the first domain, and how the kernel can create user-level domains.

The kernel is a trusted intermediary because it is the only program that can execute privileged instructions (such as storing to a processor's domain registers) and the application modules rely on the kernel to operate correctly. Because the kernel must be a trusted intermediary for the memory manager hardware, many designers also make the kernel the trusted intermediary for all other shared devices, such as the clock, display, and disk. Modules that manage these devices must then be part of the kernel program. In this design, the window manager module, network manager module, and file manager module run in kernel mode. This kernel design, in which most of the operating system runs in kernel mode, is called a *monolithic kernel* (see Figure 5.14).

In kernel mode, errors such as dividing by zero are fatal and halt the computer because these errors are typically caused by programming mistakes in the kernel program, from which there is no easy way to recover. Since kernel errors are fatal, we must program and structure the kernel carefully.

**FIGURE 5.14**

Monolithic organization: the kernel implements the operating system.

**Sidebar 5.3 Bootstrapping an Operating System** When the user switches on the power for the computer, the processor starts with all registers set to zero; thus, the user-mode bit is off. The first instruction the processor executes is the instruction at address 0 (the value of the pc register). Thus after a reset, the processor fetches its first instruction from address 0.

Address 0 typically corresponds to a read-only memory (ROM). This memory contains some initial code, the boot code, a rudimentary kernel program, which loads the full kernel program from a magnetic disk. The computer manufacturer burns into the read-only memory the boot program, after which the boot program cannot be changed. The boot program includes a rudimentary file system, which finds the kernel program (probably written by a software manufacturer) at a pre-agreed location on disk. The boot code reads the kernel into physical memory and jumps to the first instruction of the kernel.

Bootstrapping the kernel through a small boot program provides modularity. The hardware and software manufacturers can develop their products independently, and users can change kernels, for example, to upgrade to a newer version or to use a different kernel vendor, without having to modify their hardware.

Sometimes there are multiple layers of booting to handle additional constraints. For example, the first boot loader may be able to load only a single block, which can be too small to hold the rudimentary kernel program. In such cases, the boot code may load first an even simpler kernel program, which then loads the rudimentary kernel program, which then loads the kernel program.

Once it is running, the kernel allocates a thread for itself. This thread allocation involves allocating a domain for use as a stack so that it can invoke procedure calls, allowing the rest of the kernel to be written in a high-level language. It may also allocate a few other domains, for example, one for the domain table.

Once the kernel has initialized, it typically creates one or more threads to run non-kernel services. It allocates to each service one or more domains (e.g., one for program text, one for a stack, and one for data). The kernel typically preloads some of the domains with the program text and data of the non-kernel services. A common solution to locating the program text and data is to assume that the first non-kernel program, like the kernel program, is at a predefined address on the magnetic disk or part of the data of the kernel program.

Once thread is running in user mode, it can reenter the kernel using a gate for a kernel procedure. Using the kernel procedures, the user-level thread can create more threads, allocate domains for these threads, and, when done, exit.

Errors by threads in user mode (e.g., dividing by zero or using an address that is not in the thread's domains or violates permissions) cause an exception, which changes the processor to kernel mode. The exception handler can then clean up the thread.

We would like to keep the kernel small because the number of bugs in a program is at least proportional to the size of a program—and some even argue to the square of the size of program. In a monolithic kernel, if the programmer of the file manager module has made an error, the file manager module may overwrite kernel data structures unrelated to the file system, thus causing unrelated parts of the kernel to fail.

The *microkernel* architecture structures the operating system itself in a client/service style (see Figure 5.15). By applying the idea of enforced modularity to the operating system itself, we can avoid some of the major problems of a monolithic organization. In the microkernel architecture, system modules run in user mode in their own domain, as opposed to being part of a monolithic kernel. The microkernel itself implements a minimal set of abstractions, primarily domains to contain modules, threads to run programs, and virtual communication links to allow modules to send messages to one another. The kernel described in this chapter with its interface shown in Table 5.1 is an example of a microkernel.

In the microkernel organization, for example, the window service module runs in its own domain with access to the display, the file service module runs in its own domain with access to a disk extent, and the database service runs in its own domain with its own disk extent. Clients of the services communicate with them by invoking remote procedure calls, whose stubs in turn invoke the SEND and RECEIVE supervisor calls. An early, clean design for a microkernel is presented by Hansen [Suggestions for Further Reading 5.1.1].

A benefit of the microkernel organization is that errors are contained within a module, simplifying debugging. A programming error in the file service module affects only the file service module; no other module ever has its internal data structures unintentionally modified because of an error by the programmer of the file service module. If the file service fails, a programmer of the file service can focus on debugging the file service and rule out the other services immediately. In contrast with the monolithic kernel approach, it is difficult to attribute an error in the kernel to a particular module because the modules aren't isolated from each other and an error in one module may be caused by a flaw in another module.

**FIGURE 5.15**

Microkernel organization: the operating system organized using the client/service model.

In addition, if the file service fails, the database service may be able to continue operating. Of course, if the file service module fails, its clients cannot operate, but they may be able to invoke a recovery procedure that repairs the damage and restarts the file service. In the monolithic kernel approach if the file service fails, the kernel usually fails too, and the entire operating system must reboot.

Few widely used operating systems implement the microkernel approach in its purest form. In fact, most widely used operating systems today have a mostly monolithic kernel. Many critical services run inside the kernel, and only a few run outside the kernel. For example, in the GNU/Linux operating system the file and the network service run in kernel mode, but the X Window System runs in user mode.

Monolithic operating systems dominate the field for several reasons. First, if a service (e.g., a file service) is critical to the functioning of the operating system, it doesn't matter much if it fails in user mode or in kernel mode; in either case, the system is unusable.

Second, some services are shared among many modules, and it can be easier to implement these services as part of the kernel program, which is already shared among all modules. For example, a cache of recently accessed file data is more effective when shared among all modules. Furthermore, this cache may need to coordinate its memory use with the memory manager, which is typically part of the kernel.

Third, the performance of some services is critical, and the overhead of SEND and RECEIVE supervisor calls may be too large to split subsystems into smaller modules and separate each module.

Fourth, monolithic systems can enjoy the ease of debugging microkernel systems if the monolithic kernel comes with good kernel debugging tools.

Fifth, it may be difficult to reorganize existing kernel programs. In particular, there is little incentive to change a kernel program that already works. If the system works and most of the errors have been eradicated, the debugging advantage of microkernels begins to evaporate, and the cost of SEND and RECEIVE supervisor calls begins to dominate.

In general, if one has the choice between a working system and a better designed, but new system, one doesn't want to switch over to the new system unless it is much better. One reason is the overhead of switching: learning the new design, reengineering the old system to use the new design, rediscovering undocumented assumptions, and discovering unrealized assumptions (large systems often work for reasons that weren't fully understood). Another reason is the uncertainty of the gain of switching. Until there is evidence from the field, the claims about the better design are speculative. In the case of operating systems, there is little experimental evidence that microkernel-based systems are more robust than existing monolithic kernels. A final reason is that there is an opportunity cost: one can spend time reengineering existing software, or one can spend time developing the existing software to address new needs. For these reasons, few systems have switched to a pure microkernel design. Instead many existing systems have stayed with monolithic kernels, perhaps running services that are not as performance critical as user-mode programs. Microkernel designs exist in more specialized areas, and research on microkernels continues to be active.

## 5.4 VIRTUALIZING MEMORY

To address one problem at a time, the previous section assumed that memory and its address space is very large, large enough to hold all domains. In practice, memory and address space are limited. Thus, when a programmer invokes ALLOCATE_DOMAIN, we would like the programmer to specify a reasonable size. To allow a program to grow its domain if the specified size turns out to be too small, we could offer the programmer an additional primitive GROW_DOMAIN.

Growing domains, however, creates memory management problems. For example, assume that program A allocates domain 1 and program 2 allocates domain 2, right after domain 1. Even if there is free memory after domain 2, program A cannot grow domain 1 because it would cross into domain 2. In this case, the only option left for program A is to allocate a new domain of the desired size, copy the contents of domain 1 into the new domain, change the addresses in the program that refer to addresses in domain 1 to instead refer to corresponding addresses in the new domain 2, and deallocate domain 1.

This memory management complicates writing programs and can make programs slow because of the memory copies. To reduce the programming burden of managing memory, most modern computer systems virtualize memory, a step that provides two features:

1. *Virtual addresses*. If programs address memory using virtual addresses and the memory manager translates the virtual addresses to physical addresses on the fly, then the memory manager can grow and move domains in memory behind the program's back.

2. *Virtual address spaces*. A single address space may not be large enough to hold all addresses of all applications at the same time. For example, a single large database program by itself may need all the address space available in the hardware. If we can create virtual address spaces, then we can give each program its own address space. This extension also allows a thread to have its program loaded at an address of its choosing (e.g., address 0).

A memory manager that virtualizes memory is called a *virtual memory manager*. The design we work out in this section replaces the domain manager but incorporates the main features of domains: controlled sharing and permissions. We describe the virtual memory design in two steps. For the first step, Sections 5.4.1 and 5.4.2 introduce virtual addresses and describe an efficient way to translate them. For the second step, Section 5.4.3 introduces virtual address spaces. Section 5.4.4 discusses the trade-offs of software and hardware aspects of implementing a virtual memory manager. Finally, the section concludes with an advanced virtual memory design.

### 5.4.1 Virtualizing Addresses

The virtual memory manager will deal with two types of addresses, so it is convenient to give them names. The threads issue *virtual addresses* when reading and writing to memory (see Figure 5.16). The memory manager translates each virtual address issued by the processor into a *physical address*, a bus address of a location in memory or a register on a controller of a device.

Translating addresses as they are being used provides design flexibility. One can design computers whose physical addresses have a different width than its virtual addresses. The memory manager can translate several virtual addresses to the same physical address, but perhaps with different permissions. The memory manager can allocate virtual addresses to a thread but postpone allocating physical memory until the thread makes a reference to one of the virtual addresses.

Virtualizing addresses exploits the design principle *decouple modules with indirection*. The virtual memory manager provides a layer of indirection between the processor and the memory system by translating the virtual addresses of program instructions into physical addresses, instead of having the program directly issue physical memory addresses. Because it controls the translation from the addresses issued by the program to the addresses understood by the memory system, the virtual memory manager can translate any particular virtual address to different physical memory addresses at different times. Thanks to the translation, the virtual memory manager can rearrange the data in the memory system without having to modify any application program.

**FIGURE 5.16**

A virtual memory manager translating virtual addresses to physical addresses.

From a naming point of view, the virtual memory manager creates a name space of virtual addresses on top of a name space of physical addresses. The virtual memory manager's naming scheme translates virtual addresses into physical addresses.

Virtual memory has many uses. In this chapter, we focus on managing physical memory transparently. Later, in Section 6.2 of Chapter 6, we describe how virtual memory can also be used to transparently simulate a larger memory than the computer actually possesses.

To see how address translation can help memory management, consider a virtual memory manager with a virtual address space that is very large (e.g., $2^{64}$ bytes) but with a physical address space that is smaller. Let's assume that a thread has allocated two domains of size 100 bytes (see Figure 5.17a). The memory manager allocated the domains in physical memory contiguously, but in the virtual address space the domains are far away from each other. (ALLOCATE_DOMAIN returns a virtual address.) When a thread makes a reference to a virtual address, the virtual memory manager translates the address to the appropriate physical address.

Now consider the thread requesting to grow domain 1 from size 8 kilobytes to, say, 16 kilobytes. Without virtual addresses, the memory manager would deny this request because the domain cannot grow in physical memory without running into domain 2. With virtual addresses (see Figure 5.17b), however, the memory manager can grow the domain in the virtual address space, allocate the requested amount of physical memory, copy the content of domain 1 into the newly allocated physical memory, and update its mapping for domain 1. With virtual addresses, the application doesn't have to be aware that the memory manager moved the contents of its domain in order to grow it.

Even ignoring the cost of copying the content of a domain, introducing virtual addresses comes at a cost in complexity and performance. The memory manager must manage virtual addresses in addition to a physical address space. It must allocate

**FIGURE 5.17**

(a) A thread has allocated a domain 1 and 2; they are far apart in virtual memory but next to each other in physical memory. (b) In response to the thread's request to grow domain 1, the virtual memory manager transparently moved domain 1 in physical memory and adjusted the translation from virtual to physical addresses.

and deallocate them (if the virtual address space isn't large), it must set up translations between virtual and physical addresses, and so on. The translation happens on-the-fly, which may slow down memory references. The rest of this section investigates these issues and presents a plan that doesn't even require copying the complete content of a domain when growing the domain.

### 5.4.2 Translating Addresses Using a Page Map

A naïve way of translating virtual addresses into physical addresses is to maintain a map that for each virtual address records its corresponding physical address. Of course, the amount of memory required to maintain this map would be large. If each physical address is a word (8 bytes) and the address space has $2^{64}$ virtual addresses, then we might need $2^{72}$ bytes of physical memory just to store the mapping.

A more efficient way of translation is using a *page map*. The page map is an array of page map entries. Each entry translates a fixed-sized range of contiguous bytes of virtual addresses, called a *page*, to a range of physical addresses, called a *block*, which holds the page. For now, the memory manager maintains a single page map, so that all threads share the single virtual address space, as before.

With this organization, we can think of the memory that threads see as a set of contiguous pages. A virtual address then is a name overloaded with structure consisting of two parts: a page number and a byte offset within that page (see Figure 5.18). The page number uniquely identifies an entry in the page map, and thus a page, and the byte offset identifies a byte within that page. (If the processor provides word addressing instead of byte addressing, *offset* would specify the word within a page.) The size of a page, in bytes, is equal to the maximum number of different values that can be stored in the byte offset field of the virtual address. If the offset field is 12 bits wide, then a page contains 4,096 ($2^{12}$) bytes.

With this arrangement, the virtual memory manager records in the page map, for each page, the block of physical memory that contains that page. We can think of a *block* as the container of a page. Physical memory is then a contiguous set of blocks, holding pages, but the pages don't have to be contiguous in physical memory; that is, block 0 may hold page 100, block 1 may hold page 2, and so forth. The mapping between pages and the blocks that hold them can be arbitrary.

**FIGURE 5.18**

A virtual memory manager that translates virtual addresses by translating page numbers to block numbers.

The page map simplifies memory management because the memory manager can allocate a block anywhere in physical memory and insert the appropriate mapping into the page map, without having to copy domains in physical memory to coalesce free space.

A physical address can also be viewed as having two parts: a block number that uniquely identifies a block of memory and an offset that identifies a byte within that block. Translating a virtual address to a physical address is now a two-step process:

1. The virtual memory manager translates the page number of the virtual address to a block number that holds that page by means of some mapping from page numbers to block numbers.
2. The virtual memory manager computes the physical address by concatenating the block number with the byte offset from the original virtual address.

Several different representations are possible for the page map, each with its own set of trade-offs for translating addresses. The simplest implementation of a page map is an array implementation, often called a *page table*. It is suitable when most pages have an associated block. Figure 5.19 demonstrates the use of a page map implemented as a linear page table. The virtual memory manager resolves virtual addresses into physical addresses by taking the page number from the virtual address and using it as an index into the page table to find the corresponding block number. Then, the manager computes the physical address by concatenating the byte offset with the

**FIGURE 5.19**

An implementation of a virtual memory manager using a page table.

block number found in the page-table entry. Finally, it sends this physical address to the physical memory.

If the page size is $2^{12}$ bytes and virtual addresses are 64 bits wide, then a linear page table is large ($2^{52} \times 52$ bits). Therefore, in practice, designers use a more efficient representation of a page map, such as a two-level one or an inverted one (i.e., indexed by physical address instead of virtual), but these designs are beyond the scope of this text.

To be able to perform the translation, the virtual memory manager must have a way of finding and storing the page table. In the usual implementation, the page table is stored in the same physical memory that holds the pages, and the *physical* address of the base of the page map is stored in a reserved processor register, typically named the *page-map address register*. To ensure that user-level threads cannot change translation directly and bypass enforced modularity, processor designs allow threads to write the page-map address register only in kernel mode and allow only the kernel to modify the page table directly.

Figure 5.20 shows an example of how a kernel could use the page map. The kernel has allocated a page map in physical memory at address 0. The page map provides modules with a contiguous universal address space, without forcing the kernel to allocate blocks of memory for a domain contiguously. In this example, block 100 contains page 12 and block 500 contains page 13. When a thread asks for a new domain or to grow an existing domain, the kernel can allocate any unused block and insert it in the page map. The level of indirection provided by the page map allows the kernel to do this transparently—the running threads are unaware.

**FIGURE 5.20**

A virtual memory manager using a page table. The page table is located at physical address 0. It maps pages (e.g., 12) to blocks (e.g., 100).

## 5.4.3 Virtual Address Spaces

The design so far has assumed that all threads share a single virtual address space that is large enough that it can hold all active modules and their data. Many processors have a virtual address space that is too small to do that. For example, many processors use virtual addresses that are 32 bits wide and thus have only $2^{32}$ addresses, which represent 4 gigabytes of address space. This might be barely large enough to hold the most frequently used part of a large database, leaving little room for other modules. We can eliminate this assumption by virtualizing the physical address space.

### *5.4.3.1 Primitives for Virtual Address Spaces*

A *virtual address space* provides each application with the illusion that it has a complete address space to itself. The virtual memory manager can implement a virtual address space by giving each virtual address space its own page map. A memory manager supporting multiple virtual address spaces may have the following interface:

- *id* ← CREATE_ADDRESS_SPACE (): create a new address space. This address space is empty, meaning that none of its virtual pages are mapped to real memory. CREATE_ADDRESS_SPACE returns an identifier for that address space.
- *block* ← ALLOCATE_BLOCK (): ask the memory manager for a block of memory. The manager attempts to allocate a block that is not in use. If there are no free blocks, the request fails. ALLOCATE_BLOCK returns the physical address of the block.
- MAP (*id, block, page_number, permission*): put a block into *id*'s address space. MAP maps the physical address *block* to virtual page *page_number* with permissions *permission*. The memory manager allocates an entry in the page map for address space *id*, mapping the virtual page *page_number* to block *block*, and setting the page's permissions to *permission*.
- UNMAP (*id*, *page_number*): remove the entry for *page_number* from the page map so that threads have no access to that page and its associated block. An instruction that refers to a page that has been deleted is an illegal instruction.
- FREE_BLOCK (*block*): add the block *block* to the list of free memory blocks.
- DELETE_ADDRESS_SPACE (*id*): destroy an address space. The memory manager frees the page map and its blocks of address space *id*.

Using this interface, a thread may allocate its own address space or share its address space with other threads. When a programmer calls ALLOCATE_THREAD, the programmer specifies the address space in which the thread is to run. In many operating systems, the word "*process*" is used for the combination of a single virtual address space shared by one or more threads, but not consistently (see Sidebar 5.4).

The virtual address space is a thread's domain, and the page map defines how it resides in physical memory. Thus the kernel doesn't have to maintain a separate domain table with domain registers. If a physical block doesn't appear in an address space's page map, then the thread cannot make a reference to that physical block.

If a physical block appears in an address space's page map, then the thread can make a reference to that physical block. If a physical block appears in two page maps, then threads in both address spaces can make references to that physical block, which allows sharing of memory.

The memory manager can support domain permissions by placing the permission bits in the page-map entries. For example, one address space may have a block mapped with READ and WRITE permissions, while another address space has only READ permissions for that block. This design allows us to remove the domain registers, while keeping the concept of domains.

Figure 5.21 illustrates the use of several address spaces. It depicts two threads, each with its own address space but sharing block 800. Threads A and B have block 800 mapped at page 12. (In principle, the threads could map block 800 at different virtual

**Sidebar 5.4 Process, Thread, and Address Space** The operating systems community uses the word "process" often, but over the years it has come up with enough variants on the concept that when you read or hear the word you need to guess its meaning from its context. In the UNIX system (see Section 2.5), a process may mean one thread in a private address space (as in the early version of that system), or a group of threads in a private address space (as in later versions), or a thread (or group of threads) in an address space that is partly or completely shared (as in later versions of UNIX that also allow processes to share memory). That range of meanings is so broad as to render the term less than useful, which is why this text uses process only in the context of the early version of the UNIX system and otherwise uses only the terms *thread* and *address space*, which are the two core concepts.

**FIGURE 5.21**

Each thread has its own page-map address register. Thread A runs with the page map stored at address 300, while B runs with the page map stored at address 500. The page map contains the translation from page (p) to block (b), and the permissions required (P).

addresses, but that complicates naming of the shared data in the block.) Thread A maps block 800 with READ permission, while thread B maps block 800 with READ and WRITE permissions. In addition to a shared block, each thread has two private blocks. Each thread has a block mapped with READ and EXECUTE permissions for, for example, its program text and a block mapped with READ and WRITE permissions for, for example, its stack.

To support virtual address spaces, the page-map address register of a processor holds the physical address of the page map of the running thread on the processor and translation works then as follows:

```
1  procedure TRANSLATE (integer virtual, perm_needed) returns physical_address
2      page ← virtual[0:41]                          // Extract page number
3      offset ← virtual[42:63]                       // Extract offset
4      page_table ← PMAR                             // Use the current page table
5      perm_page ← page_table[page].permissions      // Lookup permissions for page
6      if PERMITTED (perm_needed, perm_page) then
7          block ← page_table[page].address          // Index into page map
8          physical ← block + offset                 // Concatenate block and offset
9          return physical                           // Return physical address
10     else return error
```

Although usually implemented in hardware, in pseudocode form, we can view the linear page table as an array that is indexed by a page number and that stores the corresponding block number. Line *2* extracts the page number, *page*, by extracting the leftmost 42 bits. (As explained in Sidebar 4.3, this book uses the big-endian convention for numbering bits and begins numbering with zero.) Then, it extracts the *offset*, the 12 rightmost bits of the virtual address (line *3*). Line *4* reads the address from the active page map out of PMAR. Line *5* looks up the permissions for the page. If the permissions necessary for using *virtual* are a subset of the permissions for the page (line *6*), then TRANSLATE looks up the corresponding block number by using *page* as an index into *page_table* and computes the physical address by concatenating *block* with *offset* (lines *7* and *8*). Now the virtual memory manager issues the bus request for the translated physical address or interrupts the processor with an illegal memory reference exception.

#### *5.4.3.2 The Kernel and Address Spaces*

There are two options for setting up page maps for the kernel program. The first option is to have each address space include a mapping of the kernel into its address space. For example, the top half of the address space might contain the kernel, in which case the bottom half contains the user program. With this setup, switching from the user program to the kernel (and vice versa) doesn't require changing the processor's page-map address register; only the user-mode bit must be changed. To protect the kernel, the kernel sets the permissions for kernel pages to KERNEL-ONLY; in user mode, performing a STORE to kernel pages is an illegal instruction. An additional advantage of this design is that in kernel mode, it is easy for the kernel to read data structures of the user program because the user program and kernel share the same address space. A disadvantage of this design is that it reduces the available address

space for user programs, which could be a problem in a legacy architecture that has small address spaces.

The second option is for the memory manager to give the kernel its own separate address space, which is inaccessible to user-level threads. To implement this option, we must extend the SVC instruction to switch the page-map address register to the kernel's page map when entering kernel mode. Similarly, when returning from kernel mode to user mode, the kernel must change the page-map address register to the page map of the thread that entered the gate.

The second option separates the kernel program and user programs completely, but the memory manager, which is part of the kernel, must be able to create new address spaces for user programs, and so on. The simple solution is to include the page tables of all user address spaces in the kernel address space. By modifying the page table for a user address space, the memory manager can modify that address space. Since a page table is smaller than the address space it defines, the second option wastes less address space than the first option.

If the kernel program and user programs have their own address spaces, the kernel cannot refer to data structures in user programs using kernel virtual addresses, since those virtual addresses refer to locations in the kernel address space. User programs must pass arguments to supervisor calls by value or the kernel must use a more involved method for copying data from a user address space to a kernel address space (and vice versa). For example, the kernel can compute the physical address for a user virtual address using the page table for that user address space, map the computed physical address into the kernel address space at an unused address, and then use that address.

#### *5.4.3.3 Discussion*

In the design with many virtual address spaces, virtual addresses are relative to an address space. This property has the advantage that programs don't have to be compiled to be position independent (see Sidebar 5.5). Every program can be stored at virtual address 0 and can use absolute addresses for making references to memory in its address space. In practice, this advantage is unimportant because it is not difficult for compiler designers to generate position-independent instructions.

A disadvantage of the design with many address spaces is that sharing can be confusing and less flexible. It can be confusing because a block to be shared can be mapped by threads into two different address spaces at different virtual addresses.

**Sidebar 5.5 Position-Independent Programs** Position-independent programs can be loaded at any memory address. To provide this feature, a compiler translating programs into processor instructions must generate relative, but not absolute, addresses. For example, when compiling a **for** loop, the compiler should use a jump instruction with an offset relative to the current PC to return to the top of a **for** loop rather than a jump instruction with an absolute address.

It can be less flexible because either threads share a complete address space or a designer must accept a restriction on sharing. Threads in different address spaces can share objects only at the granularity of a block: if two threads in different address spaces share an object, that object must be mapped at a page boundary, and holding the object requires allocating an integral number of pages and blocks. If the shared object is smaller than a page, then part of the address space and the block will be wasted. Section 5.4.5 describes an advanced design that doesn't have this restriction, but it is rarely used, since the waste isn't a big problem in practice.

### 5.4.4 Hardware versus Software and the Translation Look-Aside Buffer

An ongoing debate between hardware and software designers concerns what parts of the virtual memory manager should be implemented in hardware as part of the processor and what parts in software as part of the operating system, as well as what the interface between the hardware module and the software module should be.

There is no "right" answer because the designers must make a trade-off between performance and flexibility. Because address translation is in the critical path of processor instructions that use addresses, the memory manager is often implemented as a digital circuit that is part of the main processor so that it can run at the speed of the processor. A complete hardware implementation, however, reduces the opportunities for the operating system to exploit the translation between virtual and physical addresses. This trade-off must be made with care when implementing the memory manager and its page table.

The page table is usually stored in the same memory as the data, reachable over the bus. This design requires that the processor make an additional bus reference to memory each time it interprets an address: the processor must first translate the virtual address into a physical address, which requires reading an entry in the page map.

To avoid these additional bus references for translating virtual to physical addresses, the processor typically maintains a cache of entries of the page map in a smaller fast memory within the processor itself. The hope is that when the processor executes the next instruction, it will discover a previously cached entry that can translate the address, without making a bus reference to the larger memory. Only when the cache memory doesn't contain the appropriate entry must the processor retrieve an entry from memory. In practice, this design works well because most programs exhibit locality of reference. Thus, caching translation entries pays off, as we will see when we study caches in Chapter 6. Caching page table entries in the processor introduces new complexities: if a processor changes a page table entry, the cached versions must be updated too, or invalidated.

A final design issue is how to implement the cache memory of translations efficiently. A common approach is to use an associative memory instead of an indexed

memory. By making the cache memory associative, any entry can store any translation. Furthermore, because the cache is much smaller than physical memory, an associative memory is feasible. In this design, the cache memory of translations is usually referred to as the *translation look-aside buffer (TLB)*.

In the hardware design of Figure 5.19, the format of the page table is determined by the hardware. RISC processors typically don't fix the format of the page table in hardware but leave the choice of data structure to software. In these RISC designs, only the TLB is implemented in hardware. When a translation is not in the TLB, the processor generates a *TLB miss exception*. The handler for this interrupt looks up the mapping in a data structure implemented in software, inserts the translation in the TLB, and returns from the interrupt. With this design, the memory manager has complete freedom in choosing the data structure for the page map. If a module uses only a few pages, a designer may be able to save memory by storing the page map as a linked list or tree of pages. If, as is common, the union of virtual addresses is much larger than the physical memory, a designer may be able to save memory by inverting the page map and storing one entry per physical memory block; the contents of the entry identify the number of the page currently in the block.

In almost all designs of virtual addresses, the operating system manages the content of the page map in software. The hardware design may dictate the format of the table, but the kernel determines the values stored in the table entries and thus the mapping from virtual addresses to physical addresses. By allowing software to control the mapping, designers open up many uses of virtual addresses. One use is to manage physical memory efficiently, avoiding problems due to fragmentation. Another use is to extend physical memory by allowing pages to be stored on other devices, such as magnetic disks, as explained in Section 6.2.

### 5.4.5 Segments (Advanced Topic)

An address space per program (as in Figure 5.21) limits the way objects can be shared between threads. An alternative way is to use *segments*, which provide each object with a virtual address space starting at 0 and ending at the size of the object. In the segment approach, a large database program may have a segment for each table in a database (assuming the table isn't larger than a segment's address space). This allows threads to share memory at the granularity of objects instead of blocks, and in a flexible manner. A thread can share one object (segment) with one thread and another object (segment) with another thread.

To support segments, the processor must be modified because the addresses that programs use are really two numbers. The first identifies the segment number, and the second identifies the address within that segment. Unlike the model that has one virtual address space per program, where the programmer is unaware that the virtual address is implemented as a page number and an offset, in the segment model, the compiler and programmer must be aware that an address contains two parts. The programmer must specify which segment to use for an instruction, the compiler must

put the generated code in the right segment, and so on. Problem set *11* explores segments with a simple operating system and a processor with minimal support for segments.

In the address space per program, a thread can do arithmetic on addresses because the program's address space is linear. In the segment model, a thread cannot do arithmetic on addresses in different segments because adding to a segment number yields a meaningless result; there is no notion of contiguity for segment numbers.

If two threads share an object, they typically use the same segment number for the object; otherwise naming shared objects becomes cumbersome too.

Segments can be implemented by reintroducing slightly modified domain registers. Each segment has its own domain register, but we add a *page_table* field to the domain register. This *page_table* field contains the physical address of the page table that should be used to translate virtual addresses of the segment. When domain registers are used in this way, the literature calls them *segment descriptors*. Using this implementation, the memory manager translates an address as follows: the memory manager uses the segment number to look up the segment descriptor and uses the *page_table* in the segment descriptor to translate the virtual address within the segment to a physical address.

Giving each object of an application its own segment potentially requires a large number of segment descriptors per processor. We can solve this problem by putting the segment descriptors in memory in a *segment descriptor table* and giving each processor a single register that points to the segment descriptor table.

An advantage of the segment model is that the designer doesn't have to predict the maximum size of objects that grow dynamically during computation. For example, as the stack of a running computation grows, the virtual memory manager can allocate more pages on demand and increase the length of the stack segment. In the address space per program model, the thread's stack may grow into another data structure in the virtual address space. Then either the virtual memory manager must raise an error, or the complete stack must be moved to a place in the address space that has a large enough range of unused contiguous addresses.

The programming model that goes with a segment per object can be a good match for new programs written in an object-oriented style: the methods of an object class can be in a segment with READ and EXECUTE permissions, the data objects of an instance of that class in a segment with READ and WRITE permissions, and so on. Porting an old program to the segment model can be easy if one stores the complete program, code, and data in a single segment, but this method loses much of the advantage of the segment model because the entire segment must have READ, WRITE, and EXECUTE permission. Restructuring an old program to take advantage of multiple segments can be challenging because addresses are not linear; the programmer must modify the old program to specify which segment to use. For example, upgrading a kernel program to take advantage of segments in its internal construction is disruptive. A number of processors and kernels tried but failed (see Section 5.7).

Although virtual memory systems supporting segments have advantages and have been influential (see, for example, the Multics virtual memory design [Suggestions

for Further Reading 5.4.1]), most virtual memory systems today follow the address space per program approach instead of the segment approach. A few processors, such as the Intel x86 (see Section 5.7), have support for segments, but today's virtual memory systems don't exploit them. Virtual memory managers for the address space per program model tend to be less complex because sharing is not usually a primary requirement. Designers view an address space per program primarily as a method for achieving enforced modularity, rather than an approach to sharing. Although one can share pages between programs, that isn't the primary goal, but it is possible to do it in a limited way. Furthermore, porting an old application to the one address space per program model requires little effort at the outset: just allocate a complete address space for the application. If any sharing is necessary, it can be done later. In practice, sharing patterns tend to be simple, so no sophisticated support is necessary. Finally, today, address spaces are usually large enough that a program doesn't need an address space per object.

## 5.5 VIRTUALIZING PROCESSORS USING THREADS

In order to focus on one new idea at a time, the previous sections assumed that a separate processor was available to run each thread. Because there are usually not enough processors to go around, this section extends the thread manager to remove the assumption that each thread has its own processor. This extended thread manager shares a limited number of processors among a larger number of threads.

Sharing of processors introduces a new concern: if a thread hogs a processor, either accidentally or intentionally, it can slow down or even halt the progress of other threads, thereby compromising modularity. Because we have proposed that a primary requirement be to enforce modularity, one of the design challenges of the thread manager is to eliminate this concern.

This section starts with the design of a simple thread manager that does not avoid the hogging of a processor, and then moves to a design that does. It makes the design concrete by providing a pseudocode implementation of a thread manager. This implementation captures the essence of a thread manager. In practice, thread managers differ in many details and sometimes are much more complex than our example implementation.

### 5.5.1 Sharing a Processor Among Multiple Threads

Recall from Section 5.1.1 that a thread is an abstraction that encapsulates the state of a running module. A thread encapsulates enough state of the interpreter (e.g., a processor) that executes the thread that a thread manager can stop the thread and resume the thread later. A thread manager animates a thread by giving it a processor. This section explains how this ability to stop a thread and later resume it can be used to multiplex many threads over a limited number of physical processors.

To make the thread abstraction concrete, the thread manager might support this simple version of a THREAD_ALLOCATE procedure:

- *thread_id* ← ALLOCATE_THREAD (*starting_procedure, address_space_id*): allocate a new thread in address space *address_space_id*. The new thread is to begin with a call to the procedure specified in the argument *starting_procedure*. ALLOCATE_THREAD returns an identifier that names the just-created thread. If the thread manager cannot allocate a new thread (e.g., it doesn't have enough free memory to allocate a new stack), ALLOCATE_THREAD returns an error.

The thread manager implements ALLOCATE_THREAD as follows: it allocates a range of memory in *address_space_id* to be used as the stack for procedure calls, selects a processor, and sets the processor's PC to the address *starting_procedure* in *address_space_id* and the processor's SP to the bottom of the allocated stack.

Using ALLOCATE_THREAD, an application can create more threads than there are processors. Consider the applications running on the computer in Figure 5.4. These applications can have more threads than there are processors; for example, the file service might launch a new thread for each new client request. Starting a new module will also create additional threads. So the problem is to share a limited number of processors among potentially many threads.

We can solve this problem by observing that most threads spend much of their time waiting for events to happen. Most modules that run on the computer in Figure 5.4 call READ for input from the keyboard, the file system, or the network, and will wait by spinning until there is input. Instead of spinning, the thread could, while it is waiting, let another thread use its processor. If the consumer thread finds that it cannot proceed (because the buffer is empty), then it could release its processor, giving the keyboard manager (or any other thread) a chance to run.

This observation is the basic idea for virtualizing the processor: when a thread is waiting for an event, its processor can switch from that thread to another one by saving the state of the waiting thread and loading the state of a different thread. Since in most system designs many threads spend much of their life waiting for a condition to become true, this idea is general. For example, most of the other modules (the window manager, the mail reader, etc.) are consumers. They spend much of their existence waiting for input to arrive.

Over the years people have developed various labels for processor virtualization schemes, such as "time-sharing", "processor multiplexing", "multiprogramming", or "multitasking". For example, the word "time-sharing" was introduced in the 1950s to describe virtualization of a computer system so that it could be shared among several interactive human users. All these schemes boil down to the same idea: virtualizing the processor, which this section describes in detail.

To make the discussion more concrete, consider the implementation of SEND and RECEIVE with a bounded buffer in Figure 5.6. This spin-loop solution is appropriate if there is a processor for each thread, but it is inappropriate if there are fewer processors than threads. If there is just one processor and if the receiver started before the sender, then we have a major problem. The receiver thread executes its

spinning loop, and the sender never gets a chance to run and add an item to the buffer.

A solution with thread switching is shown in Figure 5.22. Comparing this code with the code in Figure 5.6, we find that the only change is the addition of two calls to a procedure named YIELD (lines *10* and *20*). YIELD is an entry to the thread manager. When a thread invokes YIELD, the thread manager gives the calling thread's processor to some other thread. At some time in the future, the thread manager returns a processor to this thread by returning from the call to YIELD. In the case of the receiver, when the processor returns at line *21*, the receiving thread reacquires the lock again and tests *out* = *in*. If *out* is now less than *in*, there is at least one new item in the buffer, so the thread extracts an item from the buffer. If not, the thread releases the lock and calls YIELD again to allow another thread to run. A thread therefore alternates between two states, which we name RUNNING (executing on a processor) and RUNNABLE (ready to run but waiting for a processor to become available).

```
1   shared structure buffer                // A shared bounded buffer
2       message instance message[N]        // with a maximum of N messages
3       long integer in initially 0        // Counts number of messages put in the buffer
4       long integer out initially 0       // Counts number of messages taken out of the buffer
5       lock instance buffer_lock initially UNLOCKED // Lock to coordinate sender and receiver

6   procedure SEND (buffer reference p, message instance msg)
7       ACQUIRE (p.buffer_lock)
8       while p.in − p.out = N do                // Wait until there room in the buffer
9           RELEASE (p.buffer_lock)              // Release lock so that receiver can remove
10          YIELD ()                             // Let another thread use the processor
11          ACQUIRE (p.buffer_lock)              // Processor came back, maybe there is room
12                                               // Wait loop end, go back to test
13      p.message[p.in modulo N] ← msg           // Put message in the buffer
14      p.in ← p.in + 1                          // Increment in
15      RELEASE (p.buffer_lock)

16  procedure RECEIVE (buffer reference p)
17      ACQUIRE (p.buffer_lock)
18      while p.in = p.out do                    // Wait until there is a message to receive
19          RELEASE (p.buffer_lock)              // Release lock so that sender can add
20          YIELD ()                             // Let another thread use the processor
21          ACQUIRE (p.buffer_lock)              // YIELD returned, maybe there is a message
22                                               // Wait loop end, go back to test
23      msg ← p.message[p.out modulo N]          // Copy item out of buffer
24      p.out ← p.out + 1                        // Increment out
25      RELEASE (p.buffer_lock)
26      return msg
```

**FIGURE 5.22**

An implementation of a virtual communication link for a system with more threads than processors.

**FIGURE 5.23**

Multiplexing *m* processors among *n* threads ($n > m$).

The job of YIELD is to switch a processor from one thread to another. In its essence, YIELD is a simple three-step operation:

1. *Save* this thread's state so that it can resume later.
2. *Schedule* another thread to run on this processor.
3. *Dispatch* this processor to that thread.

The concrete problem that YIELD solves is multiplexing many threads over a potentially smaller number of processors (see Figure 5.23). Each processor typically has an identifier (ID), a stack pointer (SP), a program counter (PC), and a page-map address register (PMAR), pointing to the page map that defines the thread's address space. Processors may have additional state, such as floating point registers. Each thread has virtual versions of ID, SP, PC, and PMAR, and any additional state. YIELD must multiplex perhaps many threads in the thread layer over a limited number of processors in the processor layer.

YIELD implements the multiplexing as follows. When a thread running in the thread layer calls YIELD, YIELD enters the processor layer. The processor saves the state of the thread that is currently running. When that processor later exits from the processor layer, it runs a new thread, usually one that is different from the one it was running when it entered. This new thread may run in a different address space from the one used by the thread that called YIELD, or it may run in the same address space, depending on how the two threads were originally allocated.

Because the implementation of YIELD is specific to a processor and must load and save state that is often stored in processor registers (i.e., SP, PC, PMAR), YIELD is written in the instruction repertoire of the processor and can be thought of as a software extension of the processor. Programs using YIELD may be written in any programming language, but the implementation of YIELD itself is usually written in low-level processor-specific instructions. YIELD is typically a kernel procedure reached by a supervisor call.

Using this layering picture, we can also explain how interrupts and exceptions are multiplexed with threads. Interrupts invoke an interrupt handler, which always runs in the processor layer, even if the interrupt occurs while in the thread layer. On an interrupt, the interrupted processor runs the corresponding interrupt handler

(e.g., when a clock interrupt occurs, it runs the clock handler) and then continues with the thread that the processor was running before the interrupt.

Exceptions happen in the thread layer. That is, the exception handler runs in the context of the interrupted thread; it has access to the interrupted thread's state and can invoke procedures on behalf of the interrupted thread.

As discussed in Sidebar 5.6, the literature is inconsistent both about the labels and about the distinction between the concepts of interrupts and exceptions. For purposes of this text, we define interrupts as events that may have no relation to the currently running thread, whereas exceptions are events that specifically pertain to the currently running thread. While both exceptions and interrupts are discovered by the processor, interrupts are handled by the processor layer, and exceptions are usually referred to a handler in the thread layer.

This difference places restrictions on what code can run in an interrupt handler: in general, an interrupt handler shouldn't invoke procedures (e.g., YIELD) of the thread layer that assume that the thread is running on the current processor because the interrupt may have nothing to do with the currently running thread. An interrupt handler can invoke an exception handler in the thread layer if the handler determines that this interrupt pertains to the thread running on the interrupted processor. The exception handler can then adjust the thread's environment. We will see an example of this case in Section 5.5.4 when the thread manager uses a clock interrupt to force the currently running thread to invoke YIELD.

Although the essence of multiplexing is simple, the code implementing YIELD is often among the most mysterious in an operating system. To dispel the mysteries, Section 5.5.2 develops a simple implementation of a thread manager that supports YIELD. Section 5.5.3 describes how this implementation can be extended to support creation and termination of threads. Section 5.5.4 explains how an operating system can enforce modularity among threads using interrupts and Section 5.5.5 adds enforcement of separate address spaces. Section 5.5.6 explains how systems use multiplexing recursively to implement several layers of processor virtualization.

**Sidebar 5.6 Interrupts, Exceptions, Faults, Traps, and Signals** The systems and architecture literature uses the words "interrupt" and "exception" inconsistently, and some authors use different words, such as "fault", "trap", "signal", and "sequence break". Some designers call a particular event an interrupt, while another designer calls the same event an exception or a signal. An operating system designer may label the handler for a hardware interrupt as an exception, trap, or fault handler. Terminology questions also arise because an interrupt handler in the operating system may invoke a thread's exception handler, which raises the question of whether the original event is an interrupt or an exception. The layered model helps answer this question: at the processor layer the event is an interrupt, and at the thread layer it is an exception.

### 5.5.2 Implementing YIELD

To keep the implementation of YIELD as simple as possible, let's temporarily restrict its implementation to a fixed number of threads, say, seven, and assume there are fewer than seven processors. (If there were seven or more processors and only seven threads, then processor virtualization would be unnecessary.) We further start by assuming that all seven threads run in the same address space, so we don't have to worry about saving and restoring a thread's PMAR. Finally, we will assume that the threads are already running. (Section 5.5.3 will remove the last assumption, explaining how threads are created and how the thread manager starts.)

With these assumptions we can implement YIELD as shown in Figure 5.24. The implementation of yield relies on four procedures: GET_THREAD_ID, ENTER_PROCESSOR_LAYER, EXIT_PROCESSOR_LAYER, and SCHEDULER. Each procedure has only a few lines of code, but they are subtle; we investigate them in detail.

As shown in the figure, the code for the procedures maintains two shared arrays: an array with one entry per processor, known as the *processor_table*, and an array with one entry per thread, known as the *thread_table*. The *processor_table* array records information for each processor. In this simple implementation, the information is just the identity of the thread that the processor is currently running. In later versions, we will need to keep track of more information. To be able to index into this table, a processor needs to know what its identity is, which is usually stored in a special register CPUID. That is, the procedure GET_THREAD_ID returns the identity of the thread running on processor CPUID (line *7*). The procedure GET_THREAD_ID virtualizes the register CPUID to create a virtual ID register for each thread, which records a thread's identity.

Entry *i* of *thread_table* holds the stack pointer for thread *i* (whenever thread *i* is not actually running on a processor) and records whether thread *i* is RUNNING (i.e., a processor is running thread *i*) or RUNNABLE (i.e., thread *i* is waiting to receive a processor). In a system with $n$ processors, $n$ threads can be in the RUNNING state at the same time.

With these data structures, the processor layer works as follows. Suppose that two processors, A and B, are busy running seven threads and that thread 0, which is running on processor A, calls YIELD. YIELD acquires *thread_table_lock* at line *9* so that the processor layer can implement switching threads as a before-or-after action. (The lock is needed because there is more than one processor, so different threads might try to invoke YIELD at the same time.) YIELD then calls ENTER_PROCESSOR_LAYER to release its processor.

The statement on line *14* records that the calling thread will no longer be running on the processor but that it is runnable. That is, if there are no other threads waiting to run, the processor layer can schedule thread 0 again.

Line *15* saves thread 0's stack pointer (held in processor A's SP register) into thread 0's entry in *thread_table*. The stack pointer is the only thread state that must be saved because the processor layer always suspends a thread in ENTER_PROCESSOR_LAYER; it is unnecessary to save and restore the program counter. We are assuming that all threads run in the same address space so PMAR doesn't have to be saved and restored either. Other processors or calling conventions (or if a thread may be resumed at a different address than in ENTER_PROCESSOR_LAYER) might require that ENTER_PROCESSOR_LAYER save

```
1   shared structure processor_table[7]
2       integer thread_id              // identity of thread running on a processor
3   shared structure thread_table[7]
4       integer topstack               // value of this thread's stack pointer
5       integer state                  // RUNNABLE OR RUNNING
6   shared lock instance thread_table_lock

7   procedure GET_THREAD_ID() return processor_table[CPUID].thread_id

8   procedure YIELD ()
9       ACQUIRE (thread_table_lock)
10      ENTER_PROCESSOR_LAYER (GET_THREAD_ID())
11      RELEASE (thread_table_lock)
12      return

13  procedure ENTER_PROCESSOR_LAYER (this_thread)
14      thread_table[this_thread].state ← RUNNABLE     // switch state to RUNNABLE
15      thread_table[this_thread].topstack ← SP        // store yielding's thread SP
16      SCHEDULER()
17      return

18  procedure SCHEDULER()
19      j = GET_THREAD_ID()
20      do                                             // schedule a RUNNABLE thread
21          j ← (j + 1) modulo 7
22      while thread_table[j].state ≠ RUNNABLE         // skip running threads
23      thread_table[j].state ← RUNNING                // set state to RUNNING
24      processor_table[CPUID].thread_id ← j           // record that processor runs thread j
25      EXIT_PROCESSOR_LAYER (j)                       // dispatch this processor to thread j
26      return

27  // EXIT_PROCESSOR_LAYER returns from the new thread's invocation of
28  // ENTER_PROCESSOR_LAYER and returns control to the new thread's invocation of YIELD.
29  procedure EXIT_PROCESSOR_LAYER (new)
30      SP ← thread_table[new].topstack                // dispatch: load SP of new thread
31      return
```

**FIGURE 5.24**

An implementation of YIELD. EXIT_PROCESSOR_LAYER will return to YIELD because EXIT_PROCESSOR_LAYER uses the SP that was saved in ENTER_PROCESS_LAYER. To make it easier to follow, the procedures have explicit **return** statements.

additional thread state. In that case, the *thread_table* entries must have additional fields, and ENTER_PROCESSOR_LAYER would save the additional state in the additional fields of the *thread_table* entries.

The part of the processor layer that chooses the next thread is called the *scheduler*. In our simple implementation, statements on lines *20* through *22* constitute the core of the scheduler. Processor A cycles through the thread table, skips threads that are already running on another processor, stops searching when it finds a runnable

thread (let's say thread 6), and sets thread 6's state to RUNNING (line *23*) so that another processor doesn't again select thread 6. This implementation schedules threads in a *round-robin* fashion, but many other policies are possible; we discuss some others in Chapter 6 (Section 6.3).

This implementation of the processor layer assumes that the number of threads is more than (or at least equal to) the number of processors. Under this assumption, processor A will select and run a thread different from the one that called YIELD, unless the number of threads is the same as the number of processors, in which case processor A will cycle back to the thread that called YIELD because all the other threads are running on other processors. If there are fewer threads than processors, this implementation leaves processor A cycling forever through *thread_table* without giving up *thread_table_lock*, preventing any other thread from calling YIELD. We will fix this problem in Section 5.5.3, where we introduce a version of YIELD that supports the dynamic creation and termination of threads.

After selecting thread 6 to run, the processor records that thread 6 is running on this processor (line *24*), so that on the next call to ENTER_PROCESSOR_LAYER the processor knows which thread it is running. The processor leaves the processor layer by calling EXIT_PROCESSOR_LAYER, which dispatches processor A to thread 6. This part of the thread manager is often called the *dispatcher*.

The procedure EXIT_PROCESSOR_LAYER loads the saved stack pointer of thread 6 into processor A's SP register (line *30*). (In implementations that have additional thread state that must be restored, these lines would need to be expanded.) Now processor A is running thread 6.

Because line *30* replaces SP with the value that thread 6 saved on line *15* when it last ran, the flow of control when the processor reaches the return on line *31* requires some thought. That return pops a return address off the stack. The return address is the address that thread 6 pushed on its stack when it called ENTER_PROCESSOR_LAYER at line *10*. Thus, the line *31* return actually goes to the caller of ENTER_PROCESSOR_LAYER, namely, YIELD, at line *11*. Line *12* pops the next return address off the stack, returning control to the program in thread 6 that originally called YIELD. The overall effect is that thread 0 called YIELD, but control returns to the instruction after the call to YIELD in thread 6.

This flow of control has the curious effect of abandoning two stack frames, the ones allocated on the calls to SCHEDULER and EXIT_PROCESSOR_LAYER. The original save of SP in thread 6 at line *15* actually accomplished two goals: (1) save the value of SP for future use when control returns to thread 6, and (2) mark a place that the processor layer thread can use as a stack when executing SCHEDULER and EXIT_PROCESSOR_LAYER. The reloading of SP at line *30* similarly accomplishes two goals: (1) restore the thread 6 stack and (2) abandon the processor layer stack, which is no longer needed. A more elaborate thread manager design, as we will see in Section 5.5.3, switches to a separate processor layer stack rather than borrowing space atop an existing thread layer stack.

To understand why this implementation of YIELD works, consider two threads: one running the SEND procedure of Figure 5.22 and one running the RECEIVE procedure. Furthermore, assume that the sender thread is thread 0, that the receiver thread is thread 6, and that the data and instructions of the procedures are located at address

1001 and up in memory. Finally, assume the following saved thread state for thread 0 and the following current state for processor A:

*thread_table*:

| | *Saved* SP | *State* |
|---|---|---|
| 0 | 100 | RUNNABLE |
| | ... | |
| 6 | | |

| | |
|---|---|
| Processor A's *thread_id*: | 6 |
| Processor A's PC: | 1009 |
| Processor A's SP: | 204 |

At some time in the past, thread 0 called YIELD and ENTER_PROCESSOR_LAYER stored the value of thread 0's stack pointer (100) into the thread table and went on to run some other thread. Processor A is currently running thread 6: A's entry in the *processor_table* array contains 6, A's SP register points to the top of the stack of thread 6, and A's PC register contains address 1009, which holds the first instruction of YIELD (see line *9*).

YIELD invokes the procedure ENTER_PROCESSOR_LAYER, following the procedure call convention of 4.1.1, which pushes some values on thread 6's stack—in particular, the return address (1011)—and change A's SP to 220 (204 + 16).* ENTER_PROCESSOR_LAYER knows that the current thread has index 6 by reading the processor's entry in the *processor_table* array. Line *15* saves thread 6's current top of stack (220) by storing processor A's SP into thread 6's entry into *thread_table*.

The statements at lines *19* through *22* choose which thread to run next, using a simple round-robin algorithm, and select thread 0. The scheduler invokes EXIT_PROCESSOR_LAYER to dispatch processor A to thread 0.

Line *30* loads the saved SP of thread 0 so that processor A can find the top of the stack at memory address 100. At the top of thread 0's stack will be the return address; this address will be 1011 (the line after the call to ENTER_PROCESSOR_LAYER into YIELD, line *11*), since thread 0 entered ENTER_PROCESSOR_LAYER from YIELD. Thread 0 releases *thread_table_lock* so that another thread can enter ENTER_PROCESSOR_LAYER and return from YIELD. Thread 0 returns from EXIT_PROCESSOR_LAYER following the procedure call convention, which pops off the return address from the top of the stack. The address that EXIT_PROCESSOR_LAYER uses is 1011 because EXIT_PROCESSOR_LAYER uses the SP saved by ENTER_PROCESSOR_LAYER and thus returns to YIELD at line *11*. YIELD releases the *thread_table_lock* and returns control to the program in thread 0 that originally called YIELD.

At this point, the thread switch has completed, and thread 0, rather than thread 6, is running on processor A; the state is as follows:

*thread_table*:

| | *Saved* SP | *State* |
|---|---|---|
| 0 | 100 | RUNNING |
| | ... | |
| 6 | 220 | RUNNABLE |

| | |
|---|---|
| Processor A's *thread_id*: | 0 |
| Processor A's PC: | 1011 |
| Processor A's SP: | 72 |

*The 16 bytes provide space to save R0, R1, one argument, and a return address.

At some time in the future, the thread manager will resume thread 6, at the instruction at address 1011.

From this example we can see that a thread always releases its processor by calling ENTER_PROCESSOR_LAYER and that the thread always resumes right after the call to ENTER_PROCESSOR_LAYER. This stylized flow of control in which a thread always releases its processor at the same point and resumes at that point is an example of what is sometimes called *co-routine*.

To ensure that the thread switch is atomic, the thread that invokes ENTER_PROCESSOR_LAYER acquires *thread_table_lock* and the thread that resumes using EXIT_PROCESSOR_LAYER releases *thread_table_lock* (line *11*). Because the scheduler is likely to choose a different thread to run from the one that called YIELD, the thread that releases the lock is most likely a different thread from the one that acquired the lock. In essence, the thread that releases the processor passes the lock along to the thread that next receives the processor.

Thread switching relies on a detailed understanding of the processor and the procedure call convention. In most systems, the implementation of thread switching is more complex than the implementation in Figure 5.24 because we made several assumptions that often don't hold in real systems: there is a fixed number of threads, all threads are runnable, and scheduling threads round-robin is an acceptable policy. In the next sections, we will eliminate some of these assumptions.

### 5.5.3 Creating and Terminating Threads

The example YIELD procedure supports only a fixed number of threads. A full-blown thread manager allows threads to be created and terminated on demand. To support a variable number of threads, we would need to modify the implementation of ALLOCATE_THREAD and extend the thread manager with the following procedures:

- EXIT_THREAD (): destroy and clean up the calling thread. When a thread is done with its job, it invokes EXIT_THREAD to release its state.
- DESTROY_THREAD (*id*): destroy the thread identified by *id*. In some cases, one thread may need to terminate another thread. For example, a user may have started a thread that turns out to have a programming error such as an endless loop, and thus the user wants to terminate it. For these cases, we might want to provide a procedure to destroy a thread.

For the most part, the implementation of these procedures is relatively straightforward, but there are a few subtle issues. For example, if threads can terminate, we have to fix the problem that the previous code required at least as many threads as processors. To get at these issues, we detail their implementation. First, we create a separate thread for each processor (which we will call a *processor-layer thread*, or *processor thread* for short), which runs the procedure SCHEDULER (see Figure 5.25). The way to think about this setup is that the SCHEDULER runs in the processor layer, and it virtualizes its processor. A processor thread per processor is necessary because a thread in the thread layer (a *thread-layer thread*) cannot deallocate its own stack since it cannot call a procedure (e.g., DEALLOCATE or YIELD) on a stack that it has released. Instead,

we set it up so that the processor-layer thread cleans up thread-layer threads. When starting the operating system kernel (e.g., after turning the computer on), the kernel creates processor-layer threads as follows:

```
procedure RUN_PROCESSORS ()
    for each processor do
        allocate stack and set up a processor thread
        shutdown ← FALSE
        SCHEDULER ()
        deallocate processor thread stack
        halt processor
```

This procedure allocates a stack and sets up a processor thread for each processor. This thread runs the scheduler procedure until some procedure sets the global variable *shutdown* to TRUE. Then, the computer restarts or halts.

We first revisit YIELD with this setup, and we then see how this generalization supports thread creation and deletion. Using a separate processor thread, we find that switching a processor from one thread-layer thread to another actually requires two thread switches: one from the thread that is releasing its processor to the processor thread and one from the processor thread to the thread that is to receive the processor (see Figure 5.26). In more detail, let's suppose, as before, that thread 0 calls YIELD on processor A and that thread 6 is runnable and has called YIELD earlier. Thread 0 switches to the processor thread by invoking ENTER_PROCESSOR_LAYER (line *12*). The implementation of ENTER_PROCESSOR_LAYER is almost identical to ENTER_PROCESSOR_LAYER of Figure 5.24: it saves the stack pointer in the calling thread's *thread_table* entry, but it loads a new stack pointer from CPUID's *processor_table* entry. When ENTER_PROCESSOR_LAYER returns, it will switch to the processor thread and resume at line *24* (right after EXIT_PROCESSOR_LAYER).

The processor thread will cycle through the thread table until it hits thread 6, which is runnable. The SCHEDULER sets thread 6's state to RUNNING (line *21*), records that thread 6 will run on this processor (line *22*), and invokes EXIT_PROCESSOR_LAYER, to switch the processor to thread 6 (line *23*). EXIT_PROCESSOR_LAYER saves the scheduler's thread state into CPUID's entry in the *processor_table* and loads thread 6's state in the processor. Because line *37* of EXIT_PROCESSOR_LAYER has loaded SP, the **return** statement at line *38* acts like a return from the procedure that saved SP. That procedure was ENTER_PROCESSOR_LAYER at line *33*, so control passes to the caller of ENTER_PROCESSOR_LAYER, namely, YIELD, at line *13*. YIELD releases *thread_table_lock* and returns control to the program of thread 6 that originally called it.

With this setup of thread switching in place, we can return to creating and deallocating threads dynamically. To keep track of which *thread_table* entries are in use, we extend the set of possible states of each entry with the additional state FREE. Now we can implement ALLOCATE_THREAD as follows:

1. Allocate space in memory for a new stack.
2. Place on the new stack an empty frame containing just a return address and initialize that return address with the address of EXIT_THREAD.

```
1   shared structure processor_table[7]  // each processor maintains the following information:
2       integer topstack                 // value of stack pointer
3       byte reference stack             // preallocated stack for this processor
4       integer thread_id                // identity of thread currently running on this processor
5   shared structure thread_table[7]     // each thread maintains the following information:
6       integer topstack                 // value of the stack pointer
7       integer state                    // RUNNABLE, RUNNING, or FREE
8       boolean kill_or_continue         // terminate this thread? initialized to CONTINUE
9       byte reference stack             // stack for this thread

10  procedure YIELD ()
11      ACQUIRE (thread_table_lock)
12      ENTER_PROCESSOR_LAYER (GET_THREAD_ID(), CPUID)     // See caption below!
13      RELEASE (thread_table_lock)
14      return
15
16  procedure SCHEDULER ()
17      while shutdown = FALSE do
18          ACQUIRE (thread_table_lock)
19          for i from 0 until 7 do
20              if thread_table[i].state = RUNNABLE then
21                  thread_table[i].state ← RUNNING
22                  processor_table[CPUID].thread_id ← i
23                  EXIT_PROCESSOR_LAYER (CPUID, i)
24                  if thread_table[i].kill_or_continue = KILL then
25                      thread_table[i].state ← FREE
26                      DEALLOCATE(thread_table[i].stack)
27                      thread_table[i].kill_or_continue = CONTINUE
28          RELEASE (thread_table_lock)
29      return                                         // Go shut down this processor

30  procedure ENTER_PROCESSOR_LAYER (tid, processor)
31      thread_table[tid].state ← RUNNABLE
32      thread_table[tid].topstack ← SP                // save state: store yielding's thread SP
33      SP ← processor_table[processor].topstack       // dispatch: load SP of processor thread
34      return

35  procedure EXIT_PROCESSOR_LAYER (processor, tid)    // transfers control to after line 14
36      processor_table[processor].topstack ← SP       // save state: store processor thread's SP
37      SP ← thread_table[tid].topstack                // dispatch: load SP of thread
38      return
```

**FIGURE 5.25**

YIELD with support for dynamic thread creation and deletion. Control flow is not obvious because some of those procedures reload SP, which changes the place to which they return. To make it easier to follow, the procedures have explicit **return** statements. The procedure called on line *12* actually returns by passing control to line *24*, and the procedure called on line *23* actually returns by passing control to line *13*. Figure 5.26 shows the control flow graphically.

**FIGURE 5.26**

Control flow example when thread 0 yields to thread 6.

3. Place on the stack a second empty frame containing just a return address and initialize this return address with the address of *starting_procedure*.
4. Find an entry in the thread table that is FREE and initialize that entry for the new thread in the thread table by storing the top of the new stack.
5. Set the state of newly created thread to RUNNABLE.

If the thread manager cannot complete these steps (e.g., all entries in the thread table are in use), then THREAD_ALLOCATE returns an error.

To illustrate this implementation, consider the following state for a newly created thread 1:

Thread 1's stack is located at address 292 and its saved stack pointer is 300. With this initial setup, it appears that EXIT_THREAD called the procedure STARTING_PROCEDURE, and thread 1 is about to return to this procedure. Thus, when SCHEDULER selects this thread, its return statement will go to the procedure *starting_procedure*. In detail, when the scheduler selects the new thread (1) as the next thread to execute, it sets its stack pointer to the top of the new stack (300) in EXIT_PROCESSOR_LAYER. When the processor returns from EXIT_PROCESSOR_LAYER, it will set its program counter to the address on top of the stack (*starting_procedure*), and start execution at that location. The procedure *starting_procedure* releases *thread_table_lock* and the new thread is running.

With this initial setup, when a thread finishes the procedure *starting_procedure*, it returns using the standard procedure return convention. Since the THREAD_CREATE procedure has put the address of the EXIT_THREAD procedure on the stack, this return transfers control to the first instruction of the EXIT_THREAD procedure.

The EXIT_THREAD procedure can be implemented as follows:

```
1  procedure EXIT_THREAD()
2      ACQUIRE (thread_table_lock)
3      thread_table[tid].kill_or_continue ← KILL
4      ENTER_PROCESSOR_LAYER (GET_THREAD_ID (), CPUID)
```

EXIT_THREAD sets the *kill_or_continue* variable for thread and calls ENTER_PROCESSOR_LAYER, which switches the processor to the processor thread. The processor thread checks the variable *kill_or_continue* on line *24* to see if a thread is done, and, if so, it

marks the thread entry as reusable (line *25*) and deallocates its stack (line *26*). Since no thread is using that stack, it is safe to deallocate it.

The implementation of DESTROY_THREAD is also a bit tricky because the target thread to be destroyed might be running on one of the processors. Thus, the calling thread cannot just free the target thread's stack; the processor running the target thread must do that. We can achieve that in an indirect way. DESTROY_THREAD just sets the *kill_or_continue* variable of the target thread to KILL and returns. When a thread invokes YIELD and enters the processor layer, the processor thread will check this variable and release the thread's resources. (Section 5.5.4 will show how to ensure that each thread running on a processor will call YIELD at least occasionally.)

The implementation described for allocating and deallocating threads is just one of many ways of handling the creation and destruction of threads. If one opens up the internals of half a dozen different thread packages, one will find half a dozen quite different ways to handle launching and terminating threads. The goal of this section was not to exhibit a complete catalog, but rather, by illustrating one example in detail, to dispel any mystery and expose the main issues that every implementation must address. Problem set *10* explores an implementation of a thread package in a trivial operating system for a single processor and two threads.

### 5.5.4 Enforcing Modularity with Threads: Preemptive Scheduling

The thread manager described so far switches to a new thread only when a thread calls YIELD. This scheduling policy, where a thread continues to run until it gives up its processor, is called *non-preemptive scheduling*. It can be problematic because the length of time a thread holds its processor is entirely under the control of the thread itself. If, for example, a programming error sends one thread into an endless loop, no other thread will ever be able to use that processor again. Non-preemptive scheduling might be acceptable for a single module that has several threads (e.g., a Web server that has several threads to increase performance) but not for several modules.

Some systems partially address this problem by having a gentlemen's agreement called *cooperative scheduling (*which in the literature sometimes is called *cooperative multitasking*): every thread is supposed to call YIELD periodically, for instance, once per 100 milliseconds. This solution is not robust because it relies on modules behaving well and not having any errors. If a programmer forgets to put in a YIELD, or if the program accidentally gets into an endless loop that does not include a YIELD, that processor will no longer participate in the gentlemen's agreement. If, as is common with cooperative multitasking designs, there is only a single processor, the processor may appear to freeze, since the other threads won't have an opportunity to make progress.

To enforce modularity among multiple threads, the operating system thread manager must ensure thread switching by using what is called *preemptive scheduling*. The thread manager must force a thread to give up its processor after, for example, 100 milliseconds. The thread manager can implement preemptive scheduling by setting the interval timer of a clock device. When the timer expires, the clock triggers

an interrupt, switching to kernel mode in the processor layer. The clock interrupt handler can then invoke an exception handler, which runs in the thread layer and forces the currently running thread to yield. Thus, if a thread is in an endless loop, it receives 100 milliseconds to run on its turn, but it cannot stop other threads from getting at least some use of the processor, too.

Supporting preemptive scheduling requires some changes to the thread manager because in the implementation described so far an interrupt handler shouldn't invoke procedures in the thread layer at all, not even when the interrupt pertains to the currently running thread. To see why, consider a processor that invokes an interrupt handler that calls YIELD. If the interrupt happens right after the thread on that processor has acquired *thread_table_lock* in YIELD, then we will create a deadlock. The YIELD call in the handler will try to acquire *thread_table_lock* too, but it already has been acquired by the interrupted thread. That thread cannot continue and release the lock because it has been interrupted by the handler.

The problem is that we have concurrent activity within the processor layer (see Figure 5.23): the thread manager (i.e., YIELD) and the interrupt handler. The concurrent execution within the thread layer is coordinated with locks, but the processor needs its own mechanism. The processor may stop processing instructions of a thread at any time and switch to processing interrupt instructions. We are lacking a mechanism to turn the processor instruction stream and the interrupt instruction stream into separate before-or-after actions.

One solution to prevent the interrupt instruction stream from interfering with the processor instruction stream is to enable/disable interrupts. Disabling interrupts for a region greater than the region in which the *thread_table_lock* is set ensures that both streams are separate before-or-after actions. When a thread is about to acquire the *thread_table_lock*, it also disables interrupts on its processor. Now the processor will not switch to an interrupt handler when an interrupt arrives; interrupts are delayed until they are enabled again. After the thread has released the *thread_table_lock*, it is safe to reenable interrupts. Any pending interrupts will then execute immediately, but it is now safe since no thread on this processor can be inside the thread manager. This solution avoids the deadlock problem. For a more detailed description of the challenges and the solution in the Plan 9 operating system, see Suggestions for Further Reading 5.3.5.

Problem set *9* explores an implementation of a thread package with preemptive scheduling for a trivial operating system tailored to a single processor, which allows for other solutions to coordinating interrupts.

Preemptive scheduling is the mechanism that enforces modularity among threads because it isolates threads from one another's behavior, guaranteeing that no thread can halt the progress of other threads. The programmer can thus write a module as a standard computer program, execute it with its own thread, and not have to worry about any other modules in the system. Even though several programs are sharing the processors, programmers can consider each module independently and can think of each module as having a processor to itself. Furthermore, if a programming error causes a module to enter into an endless loop, another module that interacts

with the user gets a chance to run at some point, thus allowing the user to destroy the ill-behaving thread by calling the THREAD_DESTROY procedure.

### 5.5.5 Enforcing Modularity with Threads and Address Spaces

Preemptive scheduling enforces modularity in the sense that one thread cannot stop the progress of another thread, but if all threads share a single address space, then they can modify each other's memory accidentally. That may be okay for threads that are working together on a common problem, but unrelated threads need to be protected from erroneous or malicious stores of one another. We can provide that protection by making the thread manager aware of the virtual address spaces of Section 5.4.

This awareness can be implemented by having the thread manager, when it switches a processor from one thread to another, also switch the address space. That is, ENTER_PROCESSOR_LAYER saves the contents of the processor's PMAR in the *thread_table* entry of the thread that is releasing the processor, and EXIT_PROCESSOR_LAYER loads the processor's PMAR with the value in the *thread_table* entry of the new thread.

Loading the PMAR adds one significant complication to the thread manager: starting at the instant that the processor loads a new value into the PMAR, the processor will translate virtual addresses using the new page table, so that it will take its next instruction from some location in the new virtual address space. As mentioned in Section 5.4.3.2, one way to deal with this complication is to map both the instructions and the data of the thread manager into the same set of virtual addresses in every virtual address space. Another possibility is to design hardware that can load the PMAR, SP, and PC as a single before-or-after action, thereby returning control to the thread in the new virtual address space at the saved location and with the saved stack pointer.

### 5.5.6 Layering Threads

Figure 5.23 and the program fragments in Figures 5.24 and 5.25 showed how to create several threads in the thread layer from one thread in the processor layer. In particular, Figure 5.25 explained how a processor thread in the processor layer can be used to dynamically create and delete threads in the thread layer. Many systems generalize this implementation to support interrupt handling and multiple layers of thread management, as shown in Figure 5.27.

To support interrupts, we can think of a processor as a hard-wired thread manager with two threads: (1) a processor thread (e.g., the thread that runs SCHEDULER in Figure 5.25) and (2) an interrupt thread that runs interrupt handlers in kernel mode. On an interrupt, a processor's hard-wired thread manager switches from a processor thread to an interrupt thread that runs an interrupt handler in kernel mode, which may invoke a thread-layer exception handler that calls YIELD.

The operating system thread layer uses the processor threads of the processor layer to implement a second layer of threads and gives each application module one

**FIGURE 5.27**

Thread managers applied recursively.

or more preemptively scheduled virtual processors. When the operating system thread manager switches to another thread, it may also have to load the chosen thread's page-map address into the page-map address register to switch to the address space of the chosen thread. The operating system thread manager runs in kernel mode.

Each application module in turn may implement, if it desires, its own, user-mode, third-layer thread manager using one or more virtual processors provided by the operating system layer. For example, some Web servers have an embedded Java interpreter to run Java programs, which may use several Java threads. To support threads at the Java level, the Java interpreter has its own thread manager. Typically, a third-layer thread manager uses non-preemptive scheduling because all threads belong to the same application module and don't have to be protected from each other.

Generalizing, we get the picture in Figure 5.27, where a number of threads at layer $n$ can be used to implement higher-layer threads at layer $n + 1$. Each hardware processor at the lowest layer creates two threads: a processor thread and an interrupt thread. One layer up, the operating system uses the processor threads to provide one or more threads per module: one thread for the editor, one thread for the window manager, one thread for the keyboard manager, and several threads for the file service. One layer further up, the file service thread creates three application-level threads out of two operating system threads: one to wait for the disk and one for each of two client requests. At each layer, a thread manager switches one or more threads of layer $n - 1$ among several layer $n$ threads.

Although the layering idea is simple in the abstract, in practice a number of issues must be carefully thought through—for example, if a thread blocks in a layer different than the layer where it was created and where its scheduler runs. Clark [Suggestions for Further Reading 5.3.3] and Anderson et al. [Suggestions for Further Reading 5.3.2] discuss some of the practical issues.

## 5.6 THREAD PRIMITIVES FOR SEQUENCE COORDINATION

The thread manager described in Section 5.5 allows processors to be shared among many threads. A thread can release its processor so that other threads get a chance to run, as the sender and receiver do using YIELD in Figure 5.22. When the sender or receiver is scheduled again, it retests the shared variables *in* and *out*. This mode of interaction, where a thread continually tests a shared variable, is called *polling*. Polling in software is usually undesirable because every time a thread discovers that the test for a shared variable fails, it has acquired and released its processor needlessly. If a system has many polling threads, then the thread manager spends much time performing unnecessary thread switches instead of running threads that have productive work to perform.

Ideally, a thread manager should schedule a thread only when the thread has useful work to perform. That is, we would prefer a way of waiting that avoids spinning on calls to YIELD. For example, a sender with a bounded buffer should be able to tell the thread manager not to run it until $in - out < N$. (That is, until the buffer has room.) One way to approach this goal is for a thread manager to support primitives for sequence coordination, which is what this section explores.

### 5.6.1 The Lost Notification Problem

To see what we need for the primitives for sequence coordination, consider an obvious, but incorrect, implementation of sender and receiver, as shown in Figure 5.28. This implementation uses a variable shared between the sender and receiver, and two new, but inadequate, primitives—WAIT and NOTIFY—that take as argument the name of the shared variable:

- WAIT (*event_name*) is a before-or-after action that sets this thread's state to WAITING, places *event_name* in the thread table entry for this thread, and yields its processor.
- NOTIFY (*event_name*) is a before-or-after action that looks in the thread table for a thread that is in the state WAITING for *event_name* and changes that thread to the RUNNABLE state.

To support this interface, the thread manager must add the WAITING state to the RUNNING and RUNNABLE state for threads in the thread table. When the scheduler runs (for example, when some thread invokes YIELD), it skips over any thread that is in the WAITING state.

The program in the figure uses these primitives as follows. A thread invokes WAIT to allow the thread manager to release the thread's processor until a call to NOTIFY (lines *15* and *25*). The thread that changes *in* invokes NOTIFY (line *15*) to tell the thread manager to give a processor to a receiver thread waiting on *nonempty* (line *22*), since now there is a message in the buffer (i.e., $out < in$). There is a similar call to NOTIFY by the thread that updates *out* (line *25*), to tell the thread manager to give a processor to a sending thread waiting on *room* (line *12*), since now there is room to add a message to the buffer. This implementation avoids needless thread switches because the waiting receiver thread receives a processor only if NOTIFY has been called.

```
1   shared structure buffer                  // A shared bounded buffer
2       message instance message[N]          // with a maximum of N messages
3       long integer in initially 0          // Counts number of messages put in the buffer
4       long integer out initially 0         // Counts number of messages taken out of the buffer
5       lock instance buffer_lock initially UNLOCKED // Lock to coordinate sender and receiver
6       event instance room                  // Event variable to wait until there is room in buffer
7       event instance notempty              // Event variable to wait until the buffer is not empty

8   procedure SEND (buffer reference p, message instance msg)
9       ACQUIRE (p.buffer_lock)
10      while p.in − p.out = N do                // Wait until there room in the buffer
11          RELEASE (p.buffer_lock)              // Release lock so that receiver can remove
12          WAIT(p.room)                         // Release processor
13          ACQUIRE (p.buffer_lock)
14      p.message[p.in modulo N] ← msg           // Put message in the buffer
15      if p.in = p.out then NOTIFY(p.notempty)  // Signal thread that there is a message
16      p.in ← p.in + 1                          // Increment in
17      RELEASE (p.buffer_lock)

18  procedure RECEIVE (buffer reference p)
19      ACQUIRE (p.buffer_lock)
20      while p.in = p.out do                    // Wait until there is a message to receive
21          RELEASE (p.buffer_lock)              // Release lock so that sender can add
22          WAIT(p.notempty)                     // Release processor
23          ACQUIRE (p.buffer_lock)
24      msg ← p.message[p.out modulo N]          // Copy item out of buffer
25      if p.in − p.out = N then NOTIFY(p.room)  // Signal thread that there is room now
26      p.out ← p.out + 1                        // Increment out
27      RELEASE (p.buffer_lock)
28      return msg
```

**FIGURE 5.28**

An implementation of a virtual communication link for a system with locks, NOTIFY, and WAIT.

Unfortunately, the use of WAIT and NOTIFY introduces a race condition. Let's assume that the buffer is empty (i.e., *in* = *out*) and a receiver and a sender run on separate processors. The following order of statements will result in a lost notification: *A20*, *A21*, *B9* through *B17*, and *A22*:

The receiver finds that *buffer* is empty (*A20*) and releases the lock (*A21*), but before the receiver executes *A22*, the sender executes *B9* through *B17*, which adds an item

to the buffer and notifies the receiver. The notification is lost because the receiver hasn't called WAIT yet. Now the receiver executes WAIT (*A22*) and waits for a notification that will never come. The sender continues adding items to the buffer until the buffer is full and then calls WAIT. Now both the receiver and sender are waiting.

We could modify the program to call NOTIFY on each invocation of SEND, but that won't fix the problem. It will make it more unlikely that the notification will be lost, but it won't eliminate the possibility. The following ordering of statements could happen: the receiver executes *A20* and *A21*, then it is interrupted long enough for the sender to add N items, and then the receiver calls *A22*. With this ordering, the receiver misses all of the repeated notifications.

Swapping statements *21* and *22* will result in a lost notification too. Then the receiver would call WAIT while still holding the lock on *buffer_lock*. But the sender needs to be able to acquire *buffer_lock* in order to notify the receiver, so everything would come to a halt.

The problem is that we have three operations on the shared buffer state that must be turned into a before-or-after action: (1) testing if there is room in the shared buffer, and (2) if not, going to sleep until there is room and (3) releasing the shared lock so that another thread can make room. If these three operations are not a before-or-after action, then the risk of the lost notification problem arises.

The pseudocode that uses WAIT and NOTIFY illustrates a tension between modularity and locks. An observant reader might ask: if the problem is a race condition caused by having concurrent threads running multistep actions (e.g., the sender (1) tests for space and (2) calls WAIT, at the same time that the receiver (1) makes space and (2) calls NOTIFY), why don't we just make those steps into before-or-after actions by putting a lock around them? The problem is that the steps we would like to make into an atomic action are for the example of the sender (1) comparing *in* and *out* and (2) changing the thread table entry from RUNNING to WAITING. But the variables *in* and *out* are owned by the sender and receiver modules, whereas the thread table is owned by the thread manager module. These are not only separate modules, but the thread manager is probably in the kernel. So who should own the lock that creates the before-or-after action? We can't allow correctness of the kernel to depend on a user program properly setting and releasing a kernel lock, nor can we allow the correctness of the kernel to depend on a user lock being correctly implemented. The real problem here is that the lock needed to create the before-or-after action must protect an invariant that is a relation between a piece of application-owned state and a piece of system-owned state.

### 5.6.2 Avoiding the Lost Notification Problem with Eventcounts and Sequencers

Designers have identified various solutions to the problem of creating before-or-after actions to eliminate lost notifications. A general property of all these solutions is that they bring some additional thread state that characterizes the event for which the thread is waiting under protection of the thread table lock (i.e., *thread_table_lock*). By extending the semantics of WAIT and NOTIFY to include examining and modifying the

variable *event_name*, it is possible to avoid lost notifications. We leave that solution as an exercise to the reader and instead offer simpler and more widely used solutions based on primitives other than WAIT and NOTIFY. Problem set *13* introduces a solution in which the additional thread state is held in what is called a ***condition variable***, and Birrell's tutorial does a nice job of explaining how to program with threads and condition variables [Suggestions for Further Reading 5.3.1]. Sidebar 5.7 and problem set *12* describe a solution in which the additional thread state is a variable known as a ***semaphore***. In this section we describe a solution (one that is intended to be particularly easy to reason about) in which the additional thread state is found in variables called ***eventcounts*** and ***sequencers*** [Suggestions for Further Reading 5.5.4]. In all of these solutions, the additional thread state must be shared between the application (e.g., SEND and RECEIVE) and the thread manager, so the semantics of WAIT/NOTIFY, condition variables, semaphores, eventcounts, and other similar solutions all contain non-obvious and sometimes quite subtle aspects. A good discussion of some of these subtle issues is provided by Lampson and Redell [Suggestions for Further Reading 5.5.5].

Eventcounts and sequencers are variables that are shared among the sender, the receiver, and the thread manager. They are manipulated using the following interface:

- AWAIT (*eventcount, value*) is a before-or-after action that compares eventcount to value. If *eventcount* exceeds *value*, AWAIT returns to its caller. If *eventcount* is less than or equal to *value*, AWAIT changes the state of the calling thread to WAITING, places *value* and the name of *eventcount* in this thread's entry in the thread table, and yields its processor.

- ADVANCE (*eventcount*) is a before-or-after action that increments *eventcount* by one and then searches the thread table for threads that are waiting on this eventcount. For each one it finds, if *eventcount* now exceeds the value for which that thread is waiting, ADVANCE changes that thread's state to RUNNABLE.

- TICKET (*sequencer*) is a before-or-after action that returns a non-negative value that increases by one on each call. 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.

- READ (*eventcount* or *sequencer*) is a before-or-after action that returns to the caller the current value of the variable. Having an explicit READ procedure ensures before-or-after atomicity for eventcounts and sequencers whose value may grow to be larger than a memory cell.

To implement this interface, the scheduler skips over any thread that is in the WAITING state.

To understand these primitives, consider first the implementation of a bounded buffer for a single sender and receiver. Using eventcounts, we can rewrite the implementation of the bounded buffer from Figure 5.6 as shown in Figure 5.29. SEND waits until there is space in the buffer. Because AWAIT implements the waiting operation, the code in Figure 5.29 does not need the **while** loop that waits for success in Figure 5.6. Once there is space, the sender adds the message to the buffer and increments *in* using

**Sidebar 5.7 Avoiding the Lost Notification Problem with Semaphores** Semaphores are counters with special semantics for sequence coordination. A semaphore supports two operations:

- DOWN (*semaphore*): if *semaphore* > 0 decrement *semaphore* and return otherwise, wait until another thread increases *semaphore* and then try to decrement again.

- UP (*semaphore*): increment *semaphore*, wake up all threads waiting on *semaphore*, and return.

Semaphores are inspired by the ones that the railroad system uses to coordinate the use of a shared track. If a semaphore is down, trains must stop until the current train on the track leaves the track and raises the semaphore. If a semaphore can take on only the values 0 and 1 (sometimes called a binary semaphore), then UP and DOWN operate similar to a railroad semaphore. Semaphores were introduced in computer systems by the Dutch programmer Edgar Dijkstra (see also Sidebar 5.2), who called the DOWN operation P ("pakken", for grabbing in Dutch) and the UP operation V ("verhogen", for raising in Dutch) [Suggestions for Further Reading 5.5.1].

The implementation of DOWN and UP must be before-or-after actions to avoid the lost notification problem. This property can be realized in the same way as the eventcount operations:

```
structure semaphore
    integer count

procedure UP (semaphore reference sem)
    ACQUIRE (thread_table_lock)
    sem.count ← sem.count + 1
    for i from 0 to 6 do  // wakeup all threads waiting on this semaphore
        if thread_table[i].state = WAITING and thread_table[i].sem = sem then
            thread_table[i].state ← RUNNABLE
    RELEASE (thread_table_lock)

procedure DOWN (semaphore reference sem)
    ACQUIRE (thread_table_lock)
    id ← GET_THREAD_ID()
    thread_table[id].sem ← sem  // record the semaphore ID is waiting on
    while sem.count < 1 do      // give up the processor when sem < 1
        thread_table[id].state ← WAITING
        ENTER_PROCESSOR_LAYER (id, CPUID)
    sem.count ← sem.count − 1
    RELEASE (thread_table_lock)
```

Using semaphores, one can implement SEND and RECEIVE with a bounded buffer without lost notifications (see problem set *12*).

```
shared structure buffer
    message instance message[N]
    eventcount instance in initially 0
    eventcount instance out initially 0

procedure SEND (buffer reference p, message instance msg)
    AWAIT (p.out, p.in − N)                      // Wait until there is space in buffer
    p.message[READ(p.in) modulo N] ← msg         // Copy message into buffer
    ADVANCE (p.in)                               // Increment in and alert receiver

procedure RECEIVE (buffer reference p)
    AWAIT (p.in, p.out)                          // Wait till something in buffer
    msg ← p.message[READ(p.out) modulo N]        // Copy message out of buffer
    ADVANCE (p.out)                              // Increment out and Alert sender
    return msg
```

**FIGURE 5.29**

An implementation of a virtual communication link for a single sender and receiver using eventcounts.

ADVANCE, which may change the receiver's state to RUNNABLE. Because AWAIT and ADVANCE operations are before-or-after actions, the lost notification problem cannot occur.

Again, because AWAIT implements the waiting operation, the receiver implementation is also simple. RECEIVE waits until there is a message in the buffer. If so, the receiver extracts the message from the buffer and increments *out* using ADVANCE, which may change the sender's state to RUNNABLE.

Figure 5.30 shows the implementation for the case of multiple senders with a single receiver. To ensure that several senders don't try to write into the same location within the buffer, we need to coordinate their actions. We can use the TICKET primitive to solve this problem, which requires changes only to SEND. The main difference between Figure 5.30 and Figure 5.29 is that the senders must obtain a ticket to serialize their operations. SEND obtains a ticket from the sequencer *sender* (line *7*). TICKET operates likes the "take a number" machine in a bakery or post office. The returned tickets create an ordering of senders and tell each sender its position in the order. Each sender thread then waits until its turn comes up by invoking AWAIT, passing as arguments the eventcount *sent* and the value of its TICKET (*t*) (line *8*). When *sent* reaches the number on the ticket of a sender thread, that sender thread proceeds to the next step, which is to wait until there is space in the buffer (line *9*), and only then does it add its item to its entry in *buffer*. Because TICKET is a before-or-after action, no two threads will get the same number. Again, because AWAIT and ADVANCE operations are before-or-after actions, the lost notification problem cannot happen.

Again, this solution doesn't use a **while** loop that waits for the success in Figure 5.6. With multiple senders, it is slightly tricky to see why this is correct. AWAIT guarantees that *eventcount* exceeded *value* at some instant after AWAIT was called, but if there are other, concurrent, threads that may increment *value*, by the time AWAIT's caller gets control back, *eventcount* may no longer exceed *value*. The proper view is that a

```
shared structure buffer
    message instance message[N]
    eventcount instance in initially 0
    eventcount instance out initially 0
    sequencer instance sender

procedure SEND (buffer reference p, message instance msg)
    t ← TICKET (p.sender)                          // Allocate a buffer slot
    AWAIT (p.in, t)                                // Wait till previous slots are filled
    AWAIT (p.out, READ(p.in) − N)                  // Wait till there is space in buffer
    p.message[READ(p.in) modulo N] ← msg           // Copy message into buffer
    ADVANCE (p.in)                                 // Increment in and alert receiver
```

**FIGURE 5.30**

An implementation of a virtual communication link for several senders using eventcounts.

return from AWAIT is a hint that the condition AWAIT was waiting for was true and it may still be true, but the program that called AWAIT must check again to be sure.

The issue seems to arise when there are multiple senders. Suppose the buffer is full (say *in* and *out* are 10) and there are two sending threads that are both waiting for a slot to become empty. The first one of those sending threads that runs will absorb the buffer entry and change *in* to 11. The second sending thread will find that *in* is 11 but *out* is also 11, so from its point of view, AWAIT returned with $in = out$. Yet it doesn't recheck the condition. Closer inspection of the code reveals that this case can never arise because the second sender is actually waiting its turn on the ticket returned by the sequencer *sender*, not waiting for $in < out$. There is never a case in which two senders are waiting for the same condition to become true. If the program had used a different way of coordinating the senders, it might have required a retest of the condition when AWAIT returns. This is another example of why programming with concurrent threads requires great care.

**FIGURE 5.31**

Thread state diagram. In any of the three states RUNNABLE, WAITING, or RUNNING, a call to DESTROY_THREAD sets a flag that causes the scheduler to force the state to NOT_ALLOCATED the next time that thread would have entered the RUNNING state.

If the implementation must also work with multiple receivers, then a similar sequencer is needed in RECEIVE to allow the receivers to serialize themselves.

With these additional primitives for sequence coordination, we can describe the life of a thread as a state machine with four states (see Figure 5.31). The thread manager

creates a thread in the RUNNABLE state. The thread manager schedules one of the runnable threads and dispatches a processor to it; that thread changes to the RUNNING state. By calling YIELD, the thread reenters the RUNNABLE state, and the manager can select another thread and dispatch to it. Alternatively, a thread can change from the RUNNING state to the NOT_ALLOCATED state by calling EXIT_THREAD. Or a running thread can enter the WAITING state by calling AWAIT when it cannot proceed until some event occurs. Another thread, by calling ADVANCE, can make the waiting thread enter the RUNNABLE state again.

These primitives provide new opportunities for a programmer to create deadlocks. For example, thread A may call AWAIT on an eventcount that it expects thread B to ADVANCE, but thread B may be AWAITing an eventcount that only thread A is in a position to ADVANCE. Eventcounts and tickets can eliminate lost notifications, but the primitives that manipulate them must still be used with care. The last few questions of problem set *11* explore the problem of lost notifications by comparing a simple Web service implemented using NOTIFY and ADVANCE.

### 5.6.3 Implementing AWAIT, ADVANCE, TICKET, and READ (Advanced Topic)

To implement AWAIT, ADVANCE, TICKET, and READ we extend the thread manager as follows. YIELD doesn't require any modifications to support AWAIT and ADVANCE, but we must extend the *thread_table* to record, for threads in the WAITING state, a reference to the eventcount on which it is waiting:

```
shared structure thread_table[7]
    integer topstack                        // value of the stack pointer
    integer state                           // WAITING, RUNNABLE, TERMINATE, NOT_ALLOCATED
    eventcount reference event              // if waiting, the eventcount we are waiting on
    long integer value                      // if waiting, what value are we waiting for
shared lock instance thread_table_lock      // lock to protect entries of thread_table
```

The field *event* is a reference to an eventcount so that the thread manager and the calling thread can share it. This sharing is the key to resolving the tension mentioned earlier: it allows a calling thread variable to be protected by the thread manager lock.

We implement AWAIT by testing the eventcount, setting the state to WAITING if the test fails, and calling ENTER_PROCESSOR_LAYER to switch to the processor thread:

```
1  structure eventcount
2      long integer count

3  procedure AWAIT (eventcount reference event, value)
4      ACQUIRE (thread_table_lock)
5      id ← GET_THREAD_ID ()
6      thread_table[id].event ← event
7      thread_table[id].value ← value
8      if event.count ≤ value then thread_table[id].state ← WAITING
9      ENTER_PROCESSOR_LAYER (id, CPUID)
10     RELEASE (thread_table_lock)
```

This implementation of AWAIT releases its processor unless eventcount *event* exceeds value in a before-or-after action. As before, the thread data structures are protected by

the lock *thread_table_lock*. In particular, the lock ensures that the line *8* comparison of *event* with *value* and the potential change of state from RUNNING to WAITING are two steps of a before-or-after action that must occur either completely before or completely after any concurrent call to ADVANCE that might change the value of *event* or the state of this thread. The lock thus prevents lost notifications.

ENTER_PROCESSOR_LAYER in AWAIT causes control to switch from this thread to the processor thread, which may give the processor away. The thread that calls ENTER_PROCESSOR_LAYER passes the lock it acquired to the processor thread, which passes it to the next thread to run on this processor. Thus, no other thread can modify the thread state while the thread that invoked AWAIT holds *thread_table_lock*. A return from that call to ENTER_PROCESSOR_LAYER means that some other thread called AWAIT or YIELD and the processor thread has decided it is appropriate to assign a processor to this thread again. The thread will return to line *10*, release *thread_table_lock*, and return to the caller of AWAIT.

The ADVANCE procedure increments the eventcount *event*, finds all threads that are waiting on *count* and whose *value* is less than *count's*, and changes their state to RUNNABLE:

```
1  procedure ADVANCE (eventcount reference event)
2      ACQUIRE (thread_table_lock)
3      event.count ← event.count + 1
4      for i from 0 until 7 do
5          if thread_table[i].state = WAITING and thread_table[i].event = event and
6                                      event.count > thread_table[i].value then
7              thread_table[i].state ← RUNNABLE
8      RELEASE (thread_table_lock)
```

The key in the implementation of ADVANCE is that it uses *thread_table_lock* to make ADVANCE a before-or-after action. In particular, the line *6* comparison of *event.count* with *thread*[*i*].*value* and the line *7* change of *state* to RUNNABLE of the thread that called AWAIT are now two steps of a before-or-after action. No thread calling AWAIT can interfere with a thread that is in ADVANCE. Similarly, no thread calling ADVANCE can interfere with a thread that is in AWAIT. This setup avoids races between AWAIT and ADVANCE, and thus the lost notification problem.

ADVANCE just makes a thread runnable; it doesn't call ENTER_PROCESSOR_LAYER to release its processor. The runnable thread won't run until some other thread (perhaps the caller of ADVANCE) calls YIELD or AWAIT, or until the scheduler preemptively releases a processor.

We implement a sequencer and the TICKET operation as follows:

```
1  structure sequencer
2      long integer ticket

3  procedure TICKET (sequencer reference s)
4      ACQUIRE (thread_table_lock)
5      t ← s.ticket
6      s.ticket ← t + 1
7      RELEASE (thread_table_lock)
8      return t
```

For completeness, the implementation of READ of an eventcount is as follows:

```
procedure READ (eventcount reference event)
    ACQUIRE (thread_table_lock)
    e ← event.count
    RELEASE (thread_table_lock)
    return e
```

To ensure that READ provides before-or-after atomicity, READ is implemented as a before-or-after action using locks. The implementation of READ of a sequencer is similar.

Recall that in Figure 5.8, ACQUIRE itself is implemented with a spin loop, polling the lock continuously instead of releasing the processor. Given that ACQUIRE and RELEASE are used to protect only short sequences of instructions, a spinning implementation is acceptable. Furthermore, inside the thread manager we must use a spinning lock because if ACQUIRE (*thread_table_lock*) were to call AWAIT to wait until the lock is unlocked, then the thread manager would be calling itself, but it isn't designed to be recursive. In particular, it does not have a base case that could stop recursion.

### 5.6.4 Polling, Interrupts, and Sequence Coordination

Some threads must interact with external devices. For example, the keyboard manager must be able to interact with the keyboard controller on the keyboard, which is a separate, special-purpose processor. As we shall see, this interaction is just another example of sequence coordination.

The keyboard controller is a special-purpose processor, which runs a single program that gathers key strokes. In the terminology of this chapter, we can think of the keyboard controller as a single, hard-wired thread running with its own dedicated processor. When the user presses a key, the keyboard controller thread raises a signal line long enough to set a flip-flop, a digital circuit that can store one bit that the keyboard manager can read. The controller thread then lowers the signal line until next time (i.e., until the next keystroke). The flip-flop shared between the controller and the manager allows them to coordinate their activities.

In fact, using the shared flip-flop, the keyboard manager can run a wait-for-input loop similar to the one in the receiver:

```
while FLIP_FLOP = 0 do
    YIELD ()
```

In this case, the keyboard controller sets the flip-flop and the keyboard manager reads the flip-flip and tests it. If the flip-flop is not set, it reads 0, and the manager yields. If it is set, it falls out of the loop. As a side-effect of reading the flip-flop, it is reset to 0, thus providing a kind of coordination lock.

Here we have another example of polling. In polling, a thread keeps checking whether another (perhaps hardware) thread needs attention. In our example, the keyboard manager runs every time the scheduler offers it a chance, to see if

any new keys have been pressed. The keyboard manager thread is continually in a RUNNABLE state, and whenever the scheduler selects it to run, the thread checks the flip-flop.

Polling has several disadvantages, especially if it is done by a program. If it is difficult to predict the time until the event will occur, then there is no good choice for how often a thread should poll. If the polling thread executes infrequently (e.g., because the processors are busy executing other threads), then it might take a long time before a device receives attention. In this case, the computer system might appear to be unresponsive; for example, if a user must wait a long time before the computer processes the user's keyboard input, the user has a bad interactive experience. On the other hand, if the scheduler selects the polling thread frequently (e.g., faster than users can type), the thread wastes processor cycles, since often there will be no input available. Finally, some devices might require that a processor executes their managers by a certain deadline because otherwise the device won't operate correctly. For example, the keyboard controller may have only a single keystroke register available to communicate with the keyboard manager. If the user types a second keystroke before the keyboard manager gets a chance to run and absorb the first one, the first keystroke may be lost.

These disadvantages are similar to the disadvantages of not having explicit primitives for sequence coordination. Without AWAIT and ADVANCE, the thread scheduler doesn't know when the receiver thread must run; therefore, the receiver thread may make unnecessary, repeated calls to YIELD. This situation with the keyboard manager is similar; ideally, when the controller has input that needs to be processed, it should be able to alert the scheduler that the keyboard manager thread should run. We would like to program the keyboard manager and keyboard controller as a sender and a receiver using the primitives for sequence coordination, much as in Figure 5.30, except we could use a solution that works for a single sender and a single receiver. Unfortunately, the controller cannot invoke procedures such as AWAIT and ADVANCE directly; it shares only a single flip-flop with the processors.

The trick is to move the polling loop down into the hardware by using interrupts. The keyboard manager enables interrupts by setting a processor's interrupt control register to ON, indicating to that processor that it must take interrupts from the keyboard controller. Then, to check for an interrupt, the processor polls the shared flip-flop at the beginning of every instruction cycle. When the processor finds that the shared flip-flop has changed, instead of proceeding to the next instruction, the processor executes the interrupt handler. In other words, interrupts are actually implemented as a polling loop inside a processor. A processor may support multiple interrupts by providing multiple shared flip-flops and a map that associates a different interrupt handler with each flip-flop.

A simple interrupt handler for the keyboard device calls ADVANCE, the call that the keyboard controller is unable make directly, and then returns. The interrupted thread continues operation without realizing that anything happened. But the next time any thread calls YIELD or AWAIT, the thread manager can notice that the keyboard manager thread has become runnable. When it runs, the keyboard manager can then copy the

keystrokes from the device, translate them to a character representation, put them in a shared buffer, (e.g., for the receiver), and wait for the next keystroke.

Because the interrupt handler gains control of a processor within one instruction time, it can be used to meet deadlines. For example, the interrupt handler for the keyboard device could copy the user's keystrokes to a buffer owned by the keyboard manager immediately, instead of waiting until the keyboard manager gets a chance to run. This way the keyboard device is immediately ready for the user's next keystroke. To meet such deadlines, interrupt handlers are usually more elaborate than a single call to ADVANCE. It is common to place modest-sized chunks of code in an interrupt handler to move data out of the device buffers (e.g., keystrokes out of the keyboard device) or immediately restart an I/O device that has turned itself off.

Putting code in an interrupt handler must be done with great care. An interrupt handler must be cautious in reading or writing shared variables because it may be invoked between any pair of instructions. Therefore, the handler cannot be sure of the state of the thread currently running on the processor or on other processors.

Since interrupt handlers are not threads managed by the operating system thread manager, the interrupt handlers and the operating system thread manager must be carefully programmed. For example, the thread manager should call ACQUIRE and RELEASE on the *thread_table_lock* with interrupts disabled because otherwise a deadlock might occur, as we saw in Section 5.5.4. As another example, an interrupt handler should never call AWAIT because AWAIT may release its processor to the surprise of the interrupted thread—the interrupted thread may be a thread that has nothing to do with the interrupt but just happened to be running on the processor when the interrupt occurred. On the other hand, an interrupt handler can invoke ADVANCE without causing any problems.

The restrictions on exception handlers that process errors caused by the currently running thread (e.g., a divide-by-zero error) are less severe because the handler runs on behalf of the thread currently running on the processor. So, in that case, the handler can call YIELD or AWAIT.

## 5.7 CASE STUDY: EVOLUTION OF ENFORCED MODULARITY IN THE INTEL X86

The previous sections introduced the main ideas for enforcing modularity within a computer using a simple processor. This section presents a case study of how the popular Intel x86 processor provides support for enforced modularity and how commonly used operating systems use this support. The next section provides a case study of enforcing modularity at the processor level using virtual machines.

The Intel x86 processor architecture is currently the most widely used architecture for microprocessors of personal computers, laptops, and servers. The x86 architecture started without any support for enforced modularity. As the robustness of software on personal computers, laptops, and servers has become

important, the Intel designers have added support for enforcing modularity. The Intel designers didn't get it right on the first try. The evolution of x86 architecture to include enforced modularity provides some good examples of the rapid improvement in technology and challenges of designing complex systems, including market pressure.

### 5.7.1 The Early Designs: No Support for Enforced Modularity

In 1971 Intel produced its first microprocessor, the 4004, intended for calculators and implemented in 2,250 transistors. The 4004 is a 4-bit processor (i.e., the word size is 4 bits and the processor computes with 4-bit wide operands) and can address as much as 4 kilobytes of program memory and 640 bytes of data memory. The 4004 provides a stack that can store only three stack frames, no interrupts, and no support for enforcing modularity. Hardware support for the missing features was well known in 1971, but there is little need for them in a calculator.

The follow-on processor, the 8080 (1974), was Intel's first microprocessor that was used in a personal computer, namely, the Altair, produced by MITS. Unlike the 4004, the 8080 is a general-purpose microprocessor. The 8080 has 5,000 transistors: an 8-bit processor that can address up to 64 kilobytes of memory (16-bit addresses), without support for enforcing modularity. Bill Gates and Paul Allen of Microsoft fame developed a program that could run BASIC applications on the Altair. Since the Altair couldn't run more than a single, simple program at one time, there was still no need for enforcing modularity.

The 8080 was followed by the 8086 in 1978, with 29,000 transistors. The 8086 is a 16-bit processor but with 20-bit bus addresses, allowing access to 1 megabyte of memory. To make a 20-bit address out of a 16-bit address register, the 8086 has four 16-bit wide segment descriptors. The 8086 combines the value in the segment descriptors and the 16-bit address in an operand as follows: (16-bit segment descriptor $\times$ 16) + 16-bit address, producing a 20-bit value. The segment descriptor can be viewed as a memory address to which the 16-bit address in the operand field of the instruction is added.

The primary purpose of these segments is to extend physical memory, as opposed to providing enforced modularity. Using the four segment descriptors, a program can refer to a total of 256 kilobytes of memory at one time. If a program needs to address other memory, the programmer must save one of the segment descriptors and load it with a new value. Thus, writing programs for the 8086 that use more than 256 kilobytes of memory is inconvenient because the programmer must keep track of segment descriptors and where segment data is located.

Although the 8086 has a different instruction repertoire from the 8080, programs for the 8080 could run on the 8086 unmodified using a translator provided by Intel. As we will see, backwards compatibility is a recurring theme in the evolution of the Intel processor architecture and one key to Intel's success.

The 8088 (1979) was put together hastily in response to IBM's request for a processor for its personal computer. The 8088 is identical to the 8086, except that it

has an 8-bit data bus, which made the processor less expensive. Most devices at that time had an 8-bit interface anyway. Microsoft supplied the operating system, named Microsoft Disk Operating System (MS-DOS), for the IBM PC. Microsoft first licensed the operating system from Seattle Computer Products and then acquired it shortly before the release of the PC for $50,000. The IBM PC was a commercial success and started the rise of Intel and Microsoft.

The IBM PC reserved the first 640 kilobytes of the 1-megabyte physical address space for programs and the top 360 kilobytes for input and output. The designers assumed that no programs on a personal computer needed more than 640 kilobytes of memory. To keep the price and complexity down, neither 8088 nor MS-DOS had any support for enforcing modularity.

### 5.7.2 Enforcing Modularity Using Segmentation

Because the IBM PC was inexpensive, it became widely used; more and more new software was developed for it, and the existing software became richer in features. In addition, users wanted to run several programs at the same time; that is, they wanted to easily switch from one program to another without having to exit a program and start it again later. These developments posed three new design goals for Intel and Microsoft: larger address spaces to run more complex programs, running several programs at once, and enforcing modularity between them. Unfortunately, the last goal conflicts with backwards compatibility because existing programs took full advantage of having direct access to physical memory.

Intel's first attempt to achieve some of these goals was the 80286* (1982), a 16-bit processor that can address up to 16 megabytes of memory (24-bit physical addresses) and has 134,000 transistors. The 80286 has two modes, named *real* and *protected*: in real mode old 8086 programs can run; in protected mode new programs can take advantage of enforced modularity through a change in the interpretation of segment descriptors. In protected mode the segment descriptors don't define the base address of a segment (as in real mode); rather, they select a segment descriptor out of a table of segment descriptors. This application of the design principle ***decouple modules with indirection*** allows a protected-mode program to refer to $2^{14}$ segments. Furthermore, the low 2 bits of a segment selector are reserved for permission bits; 2 bits supports four protection levels so that operating systems designers can exploit several protection rings†. In practice, protection rings are of limited usefulness, and operating system designers use only two rings (user and

*In 1982 Intel also introduced the 80186 and 80188, but these 6-mHz processors were used mostly as embedded processors instead of processors for personal computing. One of the major contributions of the 80186 is the reduction in the number of chips required because it included a DMA controller, an interrupt controller, and a timer.

†Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM* 15, 3 (March 1972), pages 157–170.

kernel) to ensure, for example, that user-level programs cannot access kernel-only segments.

Although Intel sold 15 million 80286s, it achieved the three goals only partially. First, 24 bits was small compared to the 32 bits of address space offered by competing processors. Second, although it is easy to go from real to protected mode, there was no easy way (other than exploiting an unrelated feature in the design of the processor) to switch from protected mode back to real. This restriction meant that an operating system could not easily switch between old and new programs. Third, it took years after the introduction of the 80286 to develop an operating system, OS/2, that could take advantage of the segmentation provided by the 80286. OS/2 was jointly created by Microsoft and IBM, for the purpose of taking advantage of all the protected-mode features of the 80286. But when Microsoft grew concerned about the project, it disowned OS/2, gave it to IBM, and focused instead on Windows 2.0. Most buyers didn't wait for IBM and Microsoft to get their operating system acts together and instead simply treated the 80286-based PC as a faster 8086 PC that could use more memory.

Overlapping with the 80286, Intel invested over 100 person-years in the design of a full-featured segment-based processor architecture known as the i432. This processor was a ground-up design to enforce modularity and support object-oriented programming. The segment-based architecture included direct support for capabilities, a protection technique for access control (see Chapter 11 [on-line]). The resulting implementation was so complex that it didn't fit on a single chip and it ran slower than the 80286. It was eventually abandoned, not because it enforced modularity, but because it was overly complex, slow, and lacked backward compatibility with the x86 processor architectures.

### 5.7.3 Page-Based Virtual Address Spaces

Under market pressure from Motorola, which was selling a 32-bit processor with support for page-based virtual memory, Intel scratched the i432 and followed the 80286 with the 80386 in 1985. The 80386 has 270,000 transistors and addresses the main shortcomings of the 80286, while still being backwards compatible with it. The 80386 is a 32-bit processor, which can refer to up to 4 gigabytes of memory (32-bit addresses) and supports 32-bit external data and address busses. Compared with the two real and protected modes of the 80286, the 80386 provides an additional mode, called virtual real mode, which allows several real-mode programs to run at the same time in virtual environments fully protected from one another. The 80386 design also allows a single segment to grow to $2^{32}$ bytes, the maximum size of physical memory. Within a segment, the 80386 designers added support for virtual memory using paging with a separate page table for each segment. Operating system designers can choose to use virtual memory with segments, or pages, or both.

This design allows several old programs to run in virtual real mode, each in its own paged address space. This design also allows old programs to have access to

more memory than on the 80286, without being forced to use multiple segments. Furthermore, because the 80386 segmentation was backwards compatible with the 80286, 80286 programs and the Windows 2.0 successor (Windows 3.0) could use the larger segments without any modification. For these reasons, the 80386 was a big hit immediately, but it took a while until 32-bit operating systems were available. GNU/Linux, a widely-used open-source UNIX-based system came out in 1991, and Microsoft's Windows 3.1 and IBM's OS/2 2.0 in 1992. All of these systems incorporated the enforced modularity ideas, pioneered in the time-sharing systems of the 1960s and 1970s.

### 5.7.4 Summary: More Evolution

After 1985, the Intel processor architecture was extended with new instructions, but the core instruction repertoire remained the same. The main changes occurred under the hood. Intel and other companies figured out how to implement processors that provide the complex x86 instruction repertoire—some instructions are 1 byte, and others can be up to 17 bytes long, which is why the literature calls the x86 a Complex Instruction Set Computer, or CISC, while still running as fast as processor architectures designed from scratch with a RISC instruction repertoire. This effort has paid off in terms of performance but has required a large number of transistors to achieve it.

Figure 5.32 shows the growth of Intel processors in terms of transistors over the period 1970–2008*. The *y*-axis is on a logarithmic scale, and the straight line suggests that the growth has been approximately exponential. The Pentium was originally designated the 80586, but Intel redesignated the 80586 the "Pentium" in order to secure a trademark. This growth is a nice example of *d(technology)/dt* in action (see Sidebar 1.6).

The growth in software is also large. Figure 5.33 shows the growth of the Linux kernel in terms of lines of code during the period 1991–2008†. In this graph, the *y*-axis is on a linear scale. As can be seen, the growth in terms of lines of code has been large, and what is shown is just the kernel. A large contributor to this growth is device drivers for new hardware devices.

The success of the x86 illustrates the importance of a specific instance of the ***unyielding foundations rule***: provide backwards compatibility. If one must change an interface, keep the old interface around or simulate the old version of the interface using the new version of the interface, so that clients keep working without modifications. It is typically much less work to develop a simulation layer that provides backwards compatibility than to reimplement all of the clients from scratch.

For processors, backwards compatibility is particularly important because legacy software is a big factor in the success of a processor architecture. The reason is that

*Source: Intel Web page (http://www.intel.com/pressroom/kits/quickreffam.htm).

†The sum of number of lines in all C files (source and include) in a kernel release.

**FIGURE 5.32**

Growth of the number of transistors in Intel processor chips. The label on each point is the commercial name of the chip. (Log scale on *y*-axis).

**FIGURE 5.33**

Growth of the number of lines of code in the Linux kernel. The label on each point is the Linux release number. (Linear scale on *y*-axis).

legacy software is expensive to modify—the original programmers usually have departed (or forgotten about it) and have not documented it well. Experience shows that even minor modifications risk violating some undocumented assumptions, so it is necessary for someone to understand the old program completely, which takes almost as much effort as writing a completely new one. So customers will nearly always choose the architecture that allows them to continue to run legacy software unchanged. Because the x86 architecture provided backwards compatibility, it was able to survive the competition from RISC processors.

Today we see the legacy software scenario being played out in the change from 32-bit virtual addresses to 64-bit virtual addresses. Intel's Itanium architecture is gradually disappearing beneath the waves because it is not backwards compatible, while competitor Advanced Micro Devices (AMD)'s 64-bit Athlon is backwards compatible with the billion or so x86 processors currently in the field. At the time of writing, Intel is abandoning the Itanium architecture and following AMD.

Backwards compatibility can also backfire. For example, Xerox decided it looked more promising to create a PC-clone rather than to commercialize a workstation that Xerox developed in its research lab, which had a mouse, a window manager, and a WYSIWYG editor [Suggestions for Further Reading 1.3.3]. Steve Jobs saw the prototype and developed an equivalent—the Apple Macintosh. The benefits of the Macintosh were so great compared to PCs that customers were willing to buy it. (The later evolution of the Macintosh is a different, less successful story.)

## 5.8 APPLICATION: ENFORCING MODULARITY USING VIRTUAL MACHINES

This chapter has introduced several high-level abstractions to virtualize processors, memory, and links to enforce modularity. Applications interact with these abstractions through a supervisor-call interface, and interrupt and exception handlers. Another approach uses *virtual machines*. In this approach, a real physical machine is used as much as possible to implement many virtual instances of itself (including its privileged instructions, such as loading and storing to the page-map address register). That is, virtual machines emulate many instances of a machine A using a real machine A. The software that implements the virtual machines is known as a *virtual machine monitor*. This section discusses virtual machines and virtual machine monitors in more detail.

### 5.8.1 Virtual Machine Uses

A virtual machine is useful in a number of situations:

- To run several *guest* operating systems side by side. For example, on one virtual Intel x86 machine, one can run the GNU/Linux operating system, and on another

one can run the Windows/XP operating system. If the virtual machine monitor implements the Intel x86 faithfully (i.e., instructions, state, protection levels, page tables), then one can run GNU/Linux, Windows/XP, and their applications on top of the monitor without modifications.

- To contain errors in a guest operating system. Because the guest runs inside a virtual machine, errors in the guest operating system cannot affect the operating systems software on other virtual machines. This feature is handy for debugging a new operating system or for containing an operating system that is flaky but important for certain applications.
- To simplify development of operating systems. The virtual machine monitor can virtualize the physical hardware to provide a simpler interface, which may simplify the development of an operating system. For example, the virtual machine monitor may turn a multiprocessor computer into a few uniprocessor computers to allow the guest operating system to be written for a uniprocessor, which simplifies coordination.

### 5.8.2 Implementing Virtual Machines

Virtual machine monitors can be implemented in two ways. First, one can run the monitor directly on hardware in kernel mode, with the guest operating systems in user mode. Second, one can run the monitor as an application in user mode on top of a *host* operating system. The latter may be less complex to implement because the monitor can take advantage of the abstractions provided by the host operating systems, but it is only possible if the host operating system forwards all the events that monitor needs to perform its job. For simplicity, we assume the first approach (see Figure 5.34); the issues are the same in either case.

**FIGURE 5.34**

A virtual machine monitor providing two virtual machines, each running a different guest operating with its own applications.

To implement virtual machine, the virtual machine monitor must provide three primary functions:

1. *Virtualizing the computer*. For example, if a guest operating system stores a new value into the page-map address register, then the monitor must make the guest operating system believe that it can do so, even though the guest is running in user mode.
2. *Dispatch events*. For example, the monitor must forward interrupts, exceptions, and supervisor calls invoked by the applications to the appropriate guest operating systems.
3. *Allocate resources*. For example, the monitor must divide physical memory among the guest operating systems.

Virtualizing the computer is easy if all instructions are *virtualizable*. That is, all the instructions that allow a guest to tell the difference between running on the physical and running on a virtual machine must result in an exception to the monitor so that the monitor can emulate the intended behavior. In addition, the exception must leave enough information for the exception handler to emulate the instruction and restart the guest operating system as if it has executed the instruction.

Consider instructions that load the page-map address register. These instructions behave differently in user mode and kernel mode. In user mode, these instructions result in an illegal instruction exception (because they are privileged), and in kernel mode the hardware performs them. If a guest operating system invokes such an instruction, for example, to switch to another application on the guest, the monitor must emulate that instruction faithfully so that the application will run with the right page-map. Thus, a requirement for such an instruction is that it results in an exception so that the monitor receives control, that it leaves enough information around that the monitor can emulate it, and that the monitor can restart the guest as if it executed the instruction. That is, the guest should not be able to tell that the monitor emulated the instruction.

If an instruction behaves differently in kernel mode than in user mode and doesn't result in an exception, then the instruction is called *non-virtualizable*. For example, on the Intel x86 processor enabling interrupts is done by setting the interrupt-enable bit in a register called `EFLAGS`. This instruction behaves differently in user mode and in kernel mode. In user mode, the instruction does not have any effect (i.e., the processor just ignores it), but in kernel mode, the instruction sets the bit in the `EFLAGS` register and allows interrupts. If a guest operating system invokes this instruction in user space, it will do nothing, but the guest operating system assumes that it is running in kernel mode and that the instruction will enable interrupts. This instruction is an example of a non-virtualizable instruction, and handling instructions like these requires a more sophisticated plan, which is beyond the scope of this text. The paper by Adams and Agesen explains it well [Suggestions for Further Reading 5.6.4].

Allocating resources well among the guest operating systems is more challenging than the usual scheduling problem. For example, the monitor must guess which blocks of physical memory are not in use so that it can use those blocks for other guests; the monitor cannot directly inspect the guest's list of free memory blocks. The paper by Waldspurger introduces a nice trick for addressing this problem [Suggestions for Further Reading 5.6.3]. As another example, the monitor must guess when a guest operating system has no work to do; the monitor cannot directly observe that the guest is in its idle loop. The literature on virtual machines contains schemes to address these challenges.

### 5.8.3 Virtualizing Example

To make concrete what the implementation challenges of these functions are, consider a guest operating system that implements its own page tables, mapping virtual addresses to physical addresses. Let's assume that this guest operating system runs on the processor developed in this text. The goal of the virtual machine monitor is to run several guest operating system by virtualizing the example processor used in this book (see Section 2.1.2), extended with the instructions documented in this chapter.

To allow each guest operating system to address all physical memory, but not other guests' physical memory, the virtual machine monitor must guard the guest's physical addresses. One way to do so is to virtualize addresses recursively. That is, the guest and virtual machine translate application virtual addresses to virtual machine addresses; the monitor translates machine virtual addresses to physical addresses. One challenge in designing the monitor is to maintain this mapping from application virtual to virtual machine to physical addresses. The general plan is for the monitor to emulate loads and stores to the page-map address register, and keep its own translation map per virtual machine, which we will refer to as the machine map.

The monitor can deduce which virtual machine memory a guest is using and the mappings from virtual to machine addresses when the guest invokes a store instruction to the page-map address register. Because this instruction is privileged, the processor will generate an illegal-instruction exception and transfer control to the monitor. The argument to the store instruction contains the machine address of a page map. The monitor can read that memory and see which virtual machine memory the guest is planning to use and what the guest's mappings from virtual to machine are (including the permissions).

For each machine page (including the one that holds the guest page map), the monitor can allocate a physical page and record in the machine map the translation from virtual to machine to physical address, together with its permissions. Equipped with this information, the monitor can construct a new page map that maps the guest's virtual addresses to physical addresses and install that new map in the real page-map address register (which will succeed since the monitor is

running in kernel mode). Thus, although there are two layers of page maps (virtual to machine and machine to physical), the translation performed by the physical processor is only one level: it translates application virtual addresses directly to physical addresses, using the new page map set up by the monitor. To support this double translation plan efficiently, Intel and AMD have added additional hardware support.

As the final step, the monitor can resume the guest operating system at the instruction after the store to the page-map addresses register, providing the illusion to the guest that it updated the page-map address register directly. Now the guest and the applications can continue execution.

If the guest changes its page map (e.g., it switches to one of its other applications), the monitor will learn about this event because the store to the page-map address register will result in an exception (because the instruction is privileged) and invoke an exception handler in the monitor. The exception handler emulates this instruction by updating the physical page-map address register as above and resumes the guest.

If the monitor wants to switch to another guest OS, it can just switch the page-map address register to the new guest's page map, like a switch between applications.

If the application addresses a page that is not part of its address space, the hardware will generate a missing-page exception, which will invoke an exception handler in the monitor. Then, the exception handler in the monitor can invoke the exception handler of the appropriate guest. The guest exception handler now believes it received the missing-page exception directly from the processor, and it can take appropriate action.

A reader interested in learning more about this topic might find the readings on virtual machines useful [Suggestions for Further Reading 5.6].

## EXERCISES

**5.1** Chapter 1 discussed four general methods for coping with complexity: modularity, abstraction, hierarchy, and layering.

**5.1a** Which of those four methods does virtual memory use as its primary organizing scheme?

**5.1b** Which does a microkernel use? Explain.

*1996-1-1c,e*

**5.2** Alyssa is trying to organize her notes on virtual memory systems, and it occurred to her that virtual memory systems can usefully be analyzed as naming systems. She went through Chapter 3 and made a list of some technical terms about naming systems; that list is on the right, below. She then listed some mechanisms found in virtual memory systems on the left. But she isn't sure which naming

concept goes with which mechanism. Help Alyssa out by telling her which letters on the right apply to each numbered mechanism on the left.

| | |
|---|---|
| **1.** page map | **a.** Search path |
| **2.** virtual address | **b.** Naming network |
| **3.** physical address | **c.** Context reference |
| **4.** a TLB entry | **d.** Object |
| **5.** the page-map address register | **e.** Name |
| | **f.** Context |
| | **g.** None of the above |

*1994-2-4*

**5.3** The Modest Mini Corporation's best-selling computer allows at most two users to run at a time. Its only addressing architecture feature is a single page map, which creates a simple linear address space for the processor. The time-sharing system for this computer loads the page map with a set of memory block addresses before running a user; to switch to the other user, it reloads the entire page map with a new set of memory block addresses. Normally, the set of memory blocks belonging to one user has no overlap with the set of memory blocks belonging to the other user, except that memory block 19 is always assigned as page 3 in every user's address space, providing a "communication region".

**5.3a** Protection and privacy are obviously a problem with a completely public communication area, but is there any *other* difficulty in using the communication region for any of the following types of data?

**A.** The character string name of the payroll file

**B.** An integer representing the number of names in the current payroll file

**C.** The virtual memory address, within the communication region, of another data item

**D.** The virtual memory address of a program that lies outside the communication region

**E.** A small program that is designed to remain within the communication region and execute there

*1980-2-4a*

**5.3b** Ben Bitdiddle has decided that programming with page 3 always preassigned is a nuisance. He has therefore proposed that a call to the system be added that reassigns the communication region to a different page of the calling user's address space, while not affecting the other users. What effect would this proposal have on your answers to 5.3a?

*1980-2-4b*

**5.4** One advantage of a microkernel over a monolithic kernel is that it reduces the load on the translation look-aside buffer, and thereby increases its hit rate and its consequent effect on performance. True or False? Explain.

*1994-1-3a*

**5.5** Louis writes a multithreaded program, which produces an incorrect answer some of the time, but always completes. He suspects a race condition. Which of the following are strategies that can reduce, and with luck eliminate, race conditions in Louis's program?

**A.** Separate a multithreaded program into multiple single-threaded programs, run each thread in its own address space, and share data between them via a communication link that uses SEND and RECEIVE.

**B.** Apply the one-writer rule.

**C.** Ensure that for each shared variable $v$, it is protected by some lock $l_v$.

**D.** Ensure that all locks are acquired in the same order.

*2006-1-4*

**5.6** Which of the following statements about operating system kernels are true?

**A.** Preemptive scheduling allows the kernel's thread manager to run applications in a way that helps avoid fate sharing.

**B.** The kernel serves as a trusted intermediary between programs running on the same computer.

**C.** In an operating system that provides virtual memory, the kernel must be invoked to resolve every memory reference.

**D.** When a kernel switches a processor from one application to another, the target application sets the page-map address register appropriately after it is running in user space.

*2007-1-4*

**5.7** Two threads, A and B, execute a procedure named GLORP but always at different times (that is, only one of the threads calls the procedure at a given time). GLORP contains the following code:

```
procedure GLORP ()
    ACQUIRE (lock_a)
        ACQUIRE (lock_b)
        ...
        RELEASE (lock_b)
    RELEASE (lock_a)
    ...
    ACQUIRE (lock_b)
        ACQUIRE (lock_a)
        ...
        RELEASE (lock_a)
    RELEASE (lock_b)
```

**5.7a** Assuming that no other code in other procedures ever acquires more than one lock at a time, can there be a deadlock? (If yes, give an example; if not, argue why not.)

*1995-1-3a*

**5.7b** Now, assuming that the two threads can be in the code fragment above at the same time, can the program deadlock? (If yes, give an example; if not, argue why not.)

*1995-1-3b*

**5.8** Consider three threads, concurrently executing the three programs shown here. The variables $x$, $y$, and $z$ are integers with initial value 0.

| Thread 1: | Thread 2: | Thread 3: |
|---|---|---|
| **for** $i$ **from** 1 **to** 100 **do** | **for** $i$ **from** 1 **to** 100 **do** | **for** $i$ **from** 1 **to** 100 **do** |
| ACQUIRE ($A$) | ACQUIRE ($B$) | ACQUIRE ($A$) |
| ACQUIRE ($B$) | ACQUIRE ($C$) | ACQUIRE ($C$) |
| $x \leftarrow x + 1$ | $y \leftarrow z + 1$ | $z \leftarrow x + 1$ |
| RELEASE ($B$) | RELEASE ($C$) | RELEASE ($C$) |
| RELEASE ($A$) | RELEASE ($B$) | RELEASE ($A$) |

**5.8a** Can executing these three threads concurrently produce a deadlock? (If yes, give an example; if not, argue why not.)

*1993-1-5a*

**5.8b** Does your answer change if the order of the release operations in each thread is reversed? (If they can deadlock, give an example; if not, argue why not.)

*1993-1-5b*

**Additional exercises relating to Chapter 5 can be found in the problem sets beginning on page 425.**

CHAPTER

# Performance

# 6

## CHAPTER CONTENTS

**Overview** ..... **300**
**6.1 Designing for Performance** ..... **300**
6.1.1 Performance Metrics ..... 302
6.1.2 A Systems Approach to Designing for Performance ..... 304
6.1.3 Reducing Latency by Exploiting Workload Properties ..... 306
6.1.4 Reducing Latency using Concurrency ..... 307
6.1.5 Improving Throughput: Concurrency ..... 309
6.1.6 Queuing and Overload ..... 311
6.1.7 Fighting Bottlenecks ..... 313
6.1.8 An Example: The I/O Bottleneck ..... 316
**6.2 Multilevel Memories** ..... **321**
6.2.1 Memory Characterization ..... 322
6.2.2 Multilevel Memory Management using Virtual Memory ..... 323
6.2.3 Adding Multilevel Memory Management to a Virtual Memory ..... 327
6.2.4 Analyzing Multilevel Memory Systems ..... 331
6.2.5 Locality of Reference and Working Sets ..... 333
6.2.6 Multilevel Memory Management Policies ..... 335
6.2.7 Comparative Analysis of Different Policies ..... 340
6.2.8 Other Page-Removal Algorithms ..... 344
6.2.9 Other Aspects of Multilevel Memory Management ..... 346
**6.3 Scheduling** ..... **347**
6.3.1 Scheduling Resources ..... 348
6.3.2 Scheduling Metrics ..... 349
6.3.3 Scheduling Policies ..... 352
6.3.4 Case Study: Scheduling the Disk Arm ..... 360
**Exercises** ..... **362**

**Principles of Computer System Design: An Introduction**
Copyright © 2009 by Jerome H. Saltzer and M. Frans Kaashoek. All rights of reproduction in any form reserved.
DOI: 10.1016/B978-0-12-374957-4.00014-1

## OVERVIEW

The specification of a computer system typically includes explicit (or implicit) performance goals. For example, the specification may indicate how many concurrent users the system should be able to support. Typically, the simplest design fails to meet these goals because the design has a *bottleneck*, a stage in the computer system that takes longer to perform its task than any of the other stages. To overcome bottlenecks, the system designer faces the task of creating a design that performs well, yet is simple and modular.

This chapter describes techniques to avoid or hide performance bottlenecks. Section 6.1 presents ways to identify bottlenecks and the general approaches to handle them, including exploiting workload properties, concurrent execution of operations, speculation, and batching. Section 6.2 examines specific versions of the general techniques to attack the common problem of implementing multilevel memory systems efficiently. Section 6.3 presents scheduling algorithms for services to choose which request to process first, if there are several waiting for service.

## 6.1 DESIGNING FOR PERFORMANCE

Performance bottlenecks show up in computer systems for two reasons. First, limits imposed by physics, technology, or economics restrict the rate of improvement in some dimensions of technology, while other dimensions improve rapidly. An obvious class of limits are the physical ones. The speed of light limits how fast signals travel from one end of a chip to the other, how many memory elements can be within a given latency from the processor, and how fast a network message can travel in the Internet. Many other physical limits appear in computer systems, such as power and heat dissipation.

These limits force a designer to make trade-offs. For example, by shrinking a chip, a designer can make the chip faster, but it also reduces the area from which heat can be dissipated. Worse, the power dissipation increases as the designer speeds up the chip. A related trade-off is between the speed of a laptop and its power consumption. A designer wants to minimize a laptop's power consumption so that the battery lasts longer, yet customers want laptops with fast processors and large, bright screens.

Physical limits are only a subset of the limits a designer faces; there are also algorithmic, reliability, and economic limits. More limits mean more trade-offs and a higher risk of bottlenecks.

The second reason bottlenecks surface in computer systems is that several clients may share a device. If a device is busy serving one client, other clients must wait until the device becomes available. This property forces the system designer to answer questions such as which client should receive the device first. Should the device first perform the request that requires little work, perhaps at the cost of delaying the request that requires a lot of work? The designer would like to devise a scheduling plan that doesn't starve some clients in favor of others, provides low turnaround time

for each individual client request, and has little overhead so that it can serve many clients. As we will see, it is impossible to maximize all of these goals simultaneously, and thus a designer must make trade-offs. Trade-offs may favor one class of requests over another and may result in bottlenecks for the unfavored classes of requests.

Designing for performance creates two major challenges in computer systems. First, one must consider the benefits of optimization in the context of technology improvements. Some bottlenecks are intrinsic ones; they require careful thinking to ensure that the system runs faster than the performance of the slowest stage. Some bottlenecks are technology dependent; time may eliminate these, as technology improves. Unfortunately, it is sometimes difficult to decide whether or not a bottleneck is intrinsic. Not uncommonly, a performance optimization for the next product release is irrelevant by the time the product ships because technology improvements have removed the bottleneck completely. This phenomenon is so common in computer design that it has led to formulation of the design hint: ***when in doubt use brute force***. Sidebar 6.1 discusses this hint.

**Sidebar 6.1 Design Hint: When in Doubt use Brute Force** This chapter describes a few design hints that help a designer resolve trade-offs in the face of limits. These design hints are hints because they often guide the designer in the right direction, but sometimes they don't. In this book we cover only a few, but the interested reader should digest *Hints for computer system design* by B. Lampson, which presents many more practical guidelines in the form of hints [Suggestions for Further Reading 1.5.4].

The design hint "when in doubt use brute force" is a direct corollary of the *d(technology)/dt* curve (see Section 1.4). Given computing technology's historical rate of improvement, it is typically wiser to choose simple algorithms that are well understood rather than complex, badly characterized algorithms. By the time the complex algorithm is fully understood, implemented, and debugged, new hardware might be able to execute the simple algorithm fast enough. Thompson and Ritchie used a fixed-size table of processes in the UNIX system and searched the table linearly because a table was simple to implement and the number of processes was small. With Joe Condon, Thompson also built the Belle chess machine that relied mostly on special-purpose hardware to search many positions per second rather than on sophisticated algorithms. Belle won the world computer chess championships several times in the late 1970s and early 1980s and achieved an ELO rating of 2250. (ELO is a numerical rating systems used by the World Chess Federation (FIDI) to rank chess players; a rating of 2250 makes one a strong competitive player.) Later, as technology marched on, programs that performed brute-force searching algorithms on an off-the-shelf PC conquered the world computer chess championships. As of August 2005, the Hydra supercomputer (64 PCs, each with a chess coprocessor) is estimated by its creators to have an ELO rating of 3200, which is better than the best human player.

A second challenge in designing for performance is maintaining the simplicity of the design. For example, if the design uses different devices with approximately the same high-level function but radically different performance, a challenge is to abstract devices such that they can be used through a simple uniform interface. In this chapter, we see how a clever implementation of the READ and WRITE interface for memory can transparently extend the effective size of RAM to the size of a magnetic disk.

## 6.1.1 Performance Metrics

To understand bottlenecks more fully, recall that computer systems are organized in modules to achieve the benefits of modularity and that to process a request, the request may be handed from one module to another. For example, a camera may generate a continuous stream of requests containing video frames and send them to a service that digitizes each frame. The digitizing service in turn may send its output to a file service that stores the frames on a magnetic disk.

By describing this application in a client/service style, we can obtain some insights about important performance metrics. It is immediately clear that in a computer system such as this one, four metrics are of importance: the capacity of the service, its utilization, the time clients must wait for request to complete, and throughput, the rate at which services can handle requests. We will discuss each metric in turn.

### *6.1.1.1 Capacity, Utilization, Overhead, and Useful Work*

Every service has some *capacity*, a consistent measure of a service's size or amount of resources. *Utilization* is the percentage of capacity of a resource that is used for some given workload of requests. A simple measure of processor capacity is cycles. For example, the processor might be utilized 10% for the duration of some workload, which means that 90% of its processor cycles are unused. For a magnetic disk, the capacity is usually measured in sectors. If a disk is utilized 80%, then 80% of its sectors are used to store data.

In a layered system, each layer may have a different view of the capacity and utilization of the underlying resources. For example, a processor may be 95% utilized but delivering only 70% of its cycles to the application because the operating system uses 25%. Each layer considers what the layers below it do to be *overhead* in time and space, and what the layers above it do to be *useful work*. In the processor example, from the application point of view, the 25% of cycles used by the operating system is overhead and the 70% is useful work. In the disk example, if 10% of the disk is used for storing file system data structures, then from the application point of view that 10% used by the file system is overhead and only 90% is useful capacity.

### *6.1.1.2 Latency*

*Latency* is the delay between a change at the input to a system and the corresponding change at its output. From the client/service perspective, the latency of a request is the time from issuing the request until the time the response is received from the service.

**FIGURE 6.1**

A simple service composed of several stages.

This latency has several components: the latency of sending a message to the service, the latency of processing the request, and the latency of sending a response back.

If a task, such as asking a service to perform a request, is a sequence of subtasks, we can think of the complete task as traversing stages of a pipeline, where each stage of the pipeline performs a subtask (see Figure 6.1). In our example, the first stage in the pipeline is sending the request, the second stage is the service digitizing the frame, the third stage is the file service storing the frame, and the final stage is sending a response back to the client.

With this pipeline model in mind, it is easy to see that latency of a pipeline with stages A and B is greater than or equal to the sum of the latencies for each stage in the pipeline:

$$latency_{A+B} \geq latency_A + latency_B$$

It is possibly greater because passing a request from one stage to another might add some latency. For example, if the stages correspond to different services, perhaps running on different computers connected by a network, then the overhead of passing requests from one stage to another may add enough latency that it cannot be ignored.

If the stages are of a single service, that additional latency is typically small (e.g., the overhead of invoking a procedure) and can usually be ignored for first-order analysis of performance. Thus, in this case, to predict the latency of a service that isn't running yet but is expected to perform two functions, A and B, with known latencies, a designer can approximate the joint latency of A and B by adding the latency of A and the latency of B.

### 6.1.1.3 Throughput

*Throughput* is a measure of the rate of useful work done by a service for some given workload of requests. In the camera example, the throughput we might care about is how many frames per second the system can process because it may determine what quality camera we want to buy.

The throughput of a system with pipelined stages is less than or equal to the minimum of the throughput for each stage:

$$throughput_{A+B} \leq minimum(throughput_A,\ throughput_B)$$

Again, if the stages are of a single service, passing the request from one stage to another usually adds little overhead and has little impact on total throughput. Thus, for first-order analysis that overhead can be ignored, and the relation is usually close to equality.

Consider a computer system with two stages: one that is able to process data at a rate of 1,000 kilobytes per second and a second one at a rate of 100 kilobytes per second. If the fast stage generates one byte of output for each byte of input, the overall throughput must be less than or equal to 100 kilobytes per second. If there is negligible overhead in passing requests between the two stages, then the throughput of the system is equal to the throughput of the bottleneck stage, 100 kilobytes per second. In this case, the utilization of stage 1 is 10% and that of stage 2 is 100%.

When a stage processes requests serially, the throughput and the latency of a stage are directly related. The average number of requests a stage handles is inversely proportional to the average time to process a single request:

$$throughput = \frac{1}{latency}$$

If all stages process requests serially, the average throughput of the complete pipeline is inversely proportional to the average time a request spends in the pipeline. In these pipelines, reducing latency improves throughput, and the other way around.

When a stage processes requests concurrently, as we will see later in this chapter, there is *no* direct relationship between latency and throughput. For stages that process requests concurrently, an increase in throughput may *not* lead to a decrease in latency. A useful analogy is pipes through which water flows with a constant velocity. One can have several parallel pipes (or one fatter pipe), which improves throughput but doesn't change latency.

### 6.1.2 A Systems Approach to Designing for Performance

To gauge how much improvement we can hope for in reducing a bottleneck, we must identify and determine the performance of the slowest and the next-slowest bottleneck. To improve the throughput of a system in which all stages have equal throughput requires improving *all* stages. On the other hand, improving the stage that has a throughput that is 10 times lower than any other stage's throughput may result in a factor of 10 improvement in the throughput of the whole system. We might determine these bottlenecks by measurements or by using simple analytical calculations based on the performance characteristics of each bottleneck. In principle, the performance of any issue in a computer system can be explained, but sometimes it may require substantial digging to find the explanation; see, for example, the study by Perl and Sites on Windows NT's performance [Suggestions for Further Reading 6.4.1].

One should approach performance optimization from a systems point of view. This observation may sound trivial, but many person-years of work have disappeared in optimizing individual stages that resulted in small overall performance improvements. The reason that engineers are tempted to fine-tune a single stage is that

optimizations result in some measurable benefits. An individual engineer can design an optimization (e.g., replacing a slow algorithm with a faster algorithm, removing unnecessary expensive operations, reorganizing the code to have a fast path, etc.), implement it, and measure it, and can usually observe some performance improvement in that stage. This improvement stimulates the design of another optimization, which results in new benefits, and so on. Once one gets into this cycle, it is difficult to keep the ***law of diminishing returns*** in mind and realize that further improvements may result in little benefit to the system as a whole.

Since optimizing individual stages typically runs into the law of diminishing returns, an approach that focuses on overall performance is preferred. The iterative approach articulated in Section 1.5.2 achieves this goal because at each iteration the designer must consider whether or not the next iteration is worth performing. If the next iteration identifies a bottleneck that, if removed, shows diminished returns, the designer can stop. If the final performance is good enough, the designer's job is done. If the final performance doesn't meet the target, the designer may have to rethink the whole design or revisit the design specification.

The iterative approach for designing for performance has the following steps:

1. Measure the system to find out whether or not a performance enhancement is needed. If performance is a problem, identify which aspect of performance (throughput or latency) is the problem. For multistage pipelines in which stages process requests concurrently, there is no direct relationship between latency and throughput, so improving latency and improving throughput might require different techniques.

2. Measure again, this time to identify the performance bottleneck. The bottleneck may not be in the place the designer expected and may shift from one design iteration to another.

3. Predict the impact of the proposed performance enhancement with a simple back-of-the-envelope model. (We introduce a few simple models in this chapter.) This prediction includes determining where the next bottleneck will be. A quick way to determine the next bottleneck is to unrealistically assume that the planned performance enhancement will remove the current bottleneck and result in a stage with zero latency and infinite throughput. Under this assumption, determine the next bottleneck and calculate its performance. This calculation will result in one of two conclusions:

   a. Removing the current bottleneck doesn't improve system performance significantly. In this case, stop iterating, and reconsider the whole design or revisit the requirements. Perhaps the designer can adjust the interfaces between stages with the goal of tolerating costly operations. We will discuss several approaches in the next sections.

   b. Removing the current bottleneck is likely to improve the system performance. In this case, focus attention on the bottleneck stage. Consider brute-force methods of relieving the bottleneck stage (e.g., add more memory). Taking

advantage of the $\frac{d(technology)}{dt}$ curve may be less expensive than being clever. If brute-force methods won't relieve the bottleneck, be smart. For example, try to exploit properties of the workload or find better algorithms.

4. Measure the new implementation to verify that the change has the predicted impact. If not, revisit steps 1–3 and determine what went wrong.

5. Iterate. Repeat steps 1–5 until the performance meets the required level.

The rest of this chapter introduces various systems approaches to reducing latency and increasing throughput, as well as simple performance models to predict the resulting performance.

### 6.1.3 Reducing Latency by Exploiting Workload Properties

Reducing latency is difficult because the designer often runs into physical, algorithmic, and economic limits. For example, sending a message from a client on the east coast of the United States to a service on the west coast is dominated by the speed of light. Looking up an item in a hash table cannot go faster than the best algorithm for implementing hash tables. Building a very large memory that has uniform low latency is economically infeasible.

Once a designer has run into such limits, the common approach is to reduce the latency of some requests, perhaps even at the cost of increasing the latency for other requests. A designer may observe that certain requests are more common than other requests, and use that observation to improve the performance of the frequent operations by splitting the staged pipeline into a *fast path* for the frequent requests and a *slow path* for other requests (see Figure 6.2). For example, a service might remember the results of frequently asked requests so that when it receives a repeat of a recently handled request, it can return the remembered result immediately without having to recompute it. In practice, exploiting non-uniformity in applications

**FIGURE 6.2**

A simple service with a slow and fast path.

**Sidebar 6.2 Design Hint: Optimize for the Common Case** A cache (see Section 2.1.1.3) is the most common example of optimizing for the most frequent cases. We saw caches in the case study of the Domain Name System (in Section 4.4). As another example, consider a Web browser. Most Web browsers maintain a cache of recently accessed Web pages. This cache is indexed by the name of the Web page (e.g., http://www.Scholarly.edu) and returns the page for that name. If the user asks to view the same page again, then the cache can return the cached copy of the page immediately (a fast path); only the first access requires a trip to the service (a slow path). In addition to improving the user's interactive experience, the cache helps reduce the load on services and the load on the network. Because caches are so effective, many applications use several of them. For example, in addition to caching Web pages, many Web browsers have a cache to store the results of looking up names, such as "www.Scholarly.edu", so that the next request to "www.Scholarly.edu" doesn't require a DNS lookup.

The design of multilevel memory in Section 6.2 is another example of how well a designer can exploit non-uniformity in a workload. Because applications have locality of reference, one can build large and fast memory systems out of a combination of a small but fast memory and a large but slow memory.

works so well that it has led to the design hint *optimize for the common case* (see Sidebar 6.2).

To evaluate the performance of systems with a fast and slow path, designers typically compute the average latency. If we know the latency of the fast and slow paths, and the frequency with which the system will take the fast path, then the average latency is:

$$AverageLatency = Frequency_{fast} \times Latency_{fast} + Frequency_{slow} \times Latency_{slow} \quad (6.1)$$

Whether introducing a fast path is worth the effort is dependent on the relative difference in latency between the fast and slow path, and the frequency with which the system can use the fast path, which is dependent on the workload. In addition, one might be able to change the design so that the fast path becomes faster at the cost of a slower slow path. If the frequency of taking the fast path is low, then introducing a fast path (and perhaps optimizing it at the cost of the slow path) is likely not worth the complexity. In practice, as we will see in Section 6.2, many workloads don't have a uniform distribution of requests, and introducing a fast path works well.

### 6.1.4 Reducing Latency using Concurrency

Another way to reduce latency that may require some intellectual effort but that can be effective is to parallelize a stage. We take the processing that a stage must do for a single request and divide that processing up into subtasks that can be performed concurrently. Then, whenever several processors are available they can be assigned to run

those subtasks in parallel. The method can be applied either within a multiprocessor system or (if the subtasks aren't too entangled) with completely separate computers.

If the processing parallelizes perfectly (i.e., each subtask can run without any coordination with other subtasks and each subtask requires the same amount of work), then this plan can, in principle, speed up the processing by a factor $n$, where $n$ is the number of subtasks executing in parallel. In practice, the speedup is usually less than $n$ because there is overhead in parallelizing a computation—the subtasks need to communicate with each other, for example, to exchange intermediate results; because the subtasks do not require an equal amount of work; because the computation cannot be executed completely in parallel, so some fraction of the computation must be executed sequentially; or because the subtasks interfere with each other (e.g., they contend for a shared resource such as a lock, a shared memory, or a shared communication network).

Consider the processing that a search engine needs to perform in order to respond to a user search query. An early version of Google's search engine—described in more detail in Suggestions for Further Reading 3.2.4—parallelized this processing as follows. The search engine splits the index of the Web up in $n$ pieces, each piece stored on a separate machine. When a front end receives a user query, it sends a copy of the query to each of the $n$ machines. Each machine runs the query against its part of the index and sends the results back to the front end. The front end accumulates the results from the $n$ machines, chooses a good order in which to display them, generates a Web page, and sends it to the user. This plan can give good speedup if the index is large and each of the $n$ machines must perform a substantial, similar amount of computation. It is unlikely to achieve a full speedup of a factor $n$ because there is parallelization overhead (to send the query to the $n$ machines, receive $n$ partial results, and merge them); because the amount of work is not balanced perfectly across the $n$ machines and the front end must wait until the slowest responds; and because the work done by the front end in farming out the query and merging hasn't been parallelized.

Although parallelizing can improve performance, several challenges must be overcome. First, many applications are difficult to parallelize. Applications such as search have exploitable parallelism, but other computations don't split easily into $n$ mostly independent pieces. Second, developing parallel applications is difficult because the programmer must manage the concurrency and coordinate the activities of the different subtasks. As we saw in Chapter 5, it is easy to get this wrong and introduce race conditions and deadlocks. Systems have been developed to make development of parallel applications easier, but they are often limited to a particular domain. The paper by Dean and Ghemawat [Suggestions for Further Reading 6.4.3] provides an example of how the programming and management effort can be minimized for certain stylized applications running in parallel on hundreds of machines. In general, however, programmers must often struggle with threads and locks, or explicit message passing, to obtain concurrency.

Because of these two challenges in parallelizing applications, designers traditionally have preferred to rely on continuous technology improvements to reduce application latency. However, physical and engineering limitations (primarily the problem of heat dissipation) are now leading processor manufacturers away from making processors

faster and toward placing several (and soon, probably, several hundred or even several thousand, as some are predicting [Suggestions for Further Reading 1.6.4]) processors on a single chip. This development means that improving performance by using concurrency will inevitably increase in importance.

### 6.1.5 Improving Throughput: Concurrency

If the designer cannot reduce the latency of a request because of limits, an alternative approach is to *hide* the latency of a request by overlapping it with other requests. This approach doesn't improve the latency of an individual request, but it can improve system throughput. Because hiding latency is often much easier to achieve than improving latency, it has led to the hint: ***instead of reducing latency, hide it*** (see Sidebar 6.3). This section discusses how one can introduce concurrency in a multistage pipeline to increase throughput.

To overlap requests, we give each stage in the pipeline its own thread of computation so that it can compute concurrently, operating much like an assembly line (see Figure 6.3). If a stage has completed its task and has handed off the request to the next stage, then the stage can start processing the second request while the next stage processes the first request. In this fashion, the pipeline can work on several requests concurrently.

An implementation of this approach has two challenges. First, some stages of the pipeline may operate more slowly than other stage. As a result, one stage might not be able to hand off the request to the next stage because that next stage is still working on a previous request. As a result, a queue of requests may build up, while other stages might be idle. To ensure that a queue between two stages doesn't grow without bound, the stages are often coupled using a bounded buffer. We will discuss queuing in more detail in Section 6.1.6.

The second challenge is that several requests must be available. One natural source of multiple requests is if the system has several clients, each generating a request. A single client can also be a source of multiple requests if the client operates asynchronously. When an asynchronous client issues a request, rather than waiting for the response, it continues computing, perhaps issuing more requests. The main challenge

**FIGURE 6.3**

A simple service composed of several stages, with each stage operating concurrently using threads.

**Sidebar 6.3 Design Hint: Instead of Reducing Latency, Hide it** Latency is often not under the control of the designer but rather is imposed on the designer by physical properties such as the speed of light. Consider sending a message from the east coast of the United States to the west coast at the speed of light. This takes about 20 milliseconds (see Section 7.1 [online]); in the same time, a processor can execute millions of instructions. Worse, each new generation of processors gets faster every year, but the speed of light doesn't improve. As David Clark, a network researcher, put it succinctly: "One cannot bribe God." The speed of light shows up as an intrinsic barrier in many places of computer design, even when the distances are short. For example, dies are so large that for a signal to travel from one end of a chip to another is a bottleneck that limits the clock speed of a chip.

When a designer is faced with such intrinsic limits, the only option is to design systems that hide latency and try to exploit performance dimensions that do follow d(technology)/dt. For example, transmission rates for data networks have improved dramatically, and so if a designer can organize the system such that communication can be overlapped with useful computation and many network requests can be batched into a large request, then the large request can be transferred efficiently. Many Web browsers use this strategy: while a large transfer runs in the background, users can continue browsing Web pages, hiding the latency of the transfer.

in issuing multiple requests asynchronously is that the client must then match the responses with the outstanding requests.

Once the system is organized to have many requests in flight concurrently, a designer may be able to improve throughput further by using *interleaving*. The idea is to make $n$ instances of the bottleneck stage and run those $n$ instances concurrently (see Figure 6.4). Stage 1 feeds the first request to instance 1, the second request to instance 2, and so on. If the throughput of a single instance is $t$, then the throughput using interleaving is $n \times t$, assuming enough requests are available to run all instances

**FIGURE 6.4**

Interleaving requests.

concurrently at full speed and the requests don't interfere with each other. The cost of interleaving is additional copies of the bottleneck stage.

RAID (see Section 2.1.1.4) interleaves several disks to achieve a high aggregate disk throughput. RAID 0 stripes the data across the disks: it stores block 0 on disk 0, block 1 on disk 1, and so on. If requests arrive for blocks on different disks, the RAID controller can serve those requests concurrently, improving throughput. In a similar style one can interleave memory chips to improve throughput. If the current instruction is stored in memory chip 0 and the next one is in memory chip 1, the processor can retrieve them concurrently. The cost of this design is the additional disks and memory chips, but often systems already have several memory chips or disks, in which case the added cost of interleaving can be small in comparison with the performance benefit.

### 6.1.6 Queuing and Overload

If a stage in Figure 6.3 operates at its capacity (e.g., all physical processors are running threads), then a new request must wait until the stage becomes available; a queue of requests builds up waiting for the busy stage, while other stages may run idle. For example, the thread manager of Section 5.5 maintains a table of threads, which records whether a thread is runnable; a runnable thread must wait until a processor is available to run it. The stage that runs with an input queue while other stages are running idle is a bottleneck.

Using queuing theory* we can estimate the time that a request spends waiting in a queue for its turn to be processed (e.g., the time a thread spends in the ready queue). In queuing theory, the time that it takes to process a request (e.g., the time from when a thread starts running on the processor until it yields) is called the *service time*. The simplest queuing theory model assumes that requests (e.g., a thread entering the ready queue) arrive according to a random, memoryless process and have independent, exponentially distributed service times. In that case, a well-known queuing theory result tells us that the average queuing delay, measured in units of the average service time and including the service time of this request, will be $1/(1-\rho)$, where $\rho$ is the service utilization. Thus, as the utilization approaches 1, the queuing delay will grow without bound.

This same phenomenon applies to the delays for threads waiting for a processor and to the delays that customers experience in supermarket checkout lines. Any time the demand for a service comes from many statistically independent sources, there will be fluctuations in the arrival of load and thus in the length of the queue at the bottleneck stage and the time spent waiting for service. The rate of arrival of requests for service is known as the *offered load*. Whenever the offered load is greater than the capacity of a service for some duration, the service is said to be *overloaded* for that time period.

*The textbook by Jain is an excellent source to learn about queuing theory and how to reason about performance in computer systems [Suggestions for Further Reading 1.1.2].

In some constrained cases, where the designer can plan the system so that the capacity just matches the offered load of requests, it is possible to calculate the degree of concurrency necessary to achieve high throughput and the maximum length of the queue needed between stages. For example, suppose we have a processor that performs one instruction per nanosecond using a memory that takes 10 nanoseconds to respond. To avoid having the processor wait for the memory, it must make a memory request 10 instructions in advance of the instruction that needs it. If every instruction makes a request of memory, then by the time the memory responds, the processor will have issued 9 more. To avoid being a bottleneck, the memory therefore must be prepared to serve 10 requests concurrently.

If half of the instructions make a request of memory, then on average there will be five outstanding requests. Thus, a memory that can serve five requests concurrently would have enough capacity to keep up. To calculate the maximum length of the queue needed for this case depends on the application's pattern of memory references. For example, if every second instruction makes a memory request, a fixed-size queue of size five is sufficient to ensure that the queue never overflows. If the processor performs five instructions that make memory references followed by five that don't, then a fixed-size queue of size five will work, but the queue length will vary in length and the throughput will be different. If the requests arrive randomly, the queue can grow, in principle, without limit. If we were to use a memory that can handle 10 requests concurrently for this random pattern of memory references, then the memory would be utilized at 50% of capacity, and the average queue length would be $(1/(1-0.5) = 2$. With this configuration, the processor observes latencies for some memory requests of 20 or more instruction cycles, and it is running much slower than the designer expected. This example illustrates that a designer must understand non-uniform patterns in the references to memory and exploit them to achieve good performance.

In many computer systems, the designer cannot plan the offered load that precisely, and thus stages will experience periods of overload. For example, an application may have several threads that become runnable all at the same time and there may not be enough processors available to run them. In such cases, at least occasional overload is inevitable. The significance of overload depends critically on how long it lasts. If the duration is comparable to the service time, 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 service. Put another way, a queue handles short bursts of too much demand by time-averaging with adjacent periods when there is excess capacity.

If overload persists over long periods of time, the system designer has only two choices:

1. *Increase the capacity of the system*. If the system must meet the offered load, one approach is to design a system that has less overhead so that it can perform more useful work or purchase a better computer system with higher capacity. In computer systems, it is typically less expensive to buy the next generation of

the computer system that has higher capacity because of technology improvements than trying to squeeze the last ounce out of the implementation through complex algorithms.

2. *Shed load*. If purchasing a computer system with higher capacity isn't an option and system performance cannot be improved, the preferred method is to shed load by reducing or limiting the offered load until the load is less than the capacity of the system.

One approach to control the offered load is to use a bounded buffer (see Figure 5.5) between stages. When the bounded buffer ahead of the bottleneck stage is full, then the stage before it must wait until the bounded buffer empties a slot. Because the previous stage is waiting, its bounded buffer may fill up too, which may cause the stage before it to wait, and so on. The bottleneck may be pushed all the way back to the beginning of the pipeline. If this happens, the system cannot accept any more input, and what happens next depends on how the system is used.

If the source of the load needs the results of the output to generate the next request, then the load will be self-managing. This model of use applies to some interactive systems, in which the users cannot type the next command until the previous one finishes. This same idea will be used in Chapter 7 [on-line] in the implementation of self-pacing network protocols.

If the source of the load decides not to make the request at all, then the offered load decreases. If the source, however, simply holds on to the request and resubmits it later, then the offered load doesn't decrease, but some requests are just deferred, perhaps to a time when the system isn't overloaded.

A crude approach to limiting a source is to put a *quota* on how many requests a source may have outstanding. For example, some systems enforce a rule that an application may not create more than some fixed number of active threads at the same time and may not have more than some fixed number of open files. If a source has reached its quota for a given service, the system denies the next request, limiting the offered load on the system.

An alternative to limiting the offered load is reducing it when a stage becomes overloaded. We will see one example of this approach in Section 6.2. If the address spaces of a number of applications cannot fit in memory, the virtual memory manager can swap out a complete address space of one or more applications so that the remaining applications fit in memory. When the offered load decreases to normal levels, the virtual memory manager can swap in some of the applications that were swapped out.

### 6.1.7 Fighting Bottlenecks

If the designer cannot remove a bottleneck with the techniques described above, it may be possible instead to fight the bottleneck using one or more of three different techniques: batching, dallying, and speculation.

#### 6.1.7.1 Batching

*Batching* is performing several requests as a group to avoid the setup overhead of doing them one at a time. Opportunities for batching arise naturally at a bottleneck stage, which may have a queue of requests waiting to be processed. For example, if a stage has several requests to send to the next stage, the stage can combine all of the messages into a single message and send that one message to the next stage. This use of batching divides the overhead of an expensive operation (e.g., sending a message) over the several messages. More generally, batching works well when processing a request has a fixed delay (e.g., transmitting the request) and a variable delay (e.g., performing the operation specified in the request). Without batching, processing $n$ requests takes $n \times (f + v)$, where $f$ is the fixed delay and $v$ is the variable delay. With batching, processing $n$ requests takes $f + n \times v$.

Once a stage performs batching, the potential arises for additional performance wins. Batching may create opportunities for the stage to avoid work. If two or more write requests in a batch are for the same disk block, then the stage can perform just the last one.

Batching may also provide opportunities to improve latency by *reordering* the processing of requests. As we will see in Section 6.3.4, if a disk controller receives a batch of requests, it can schedule them in an order that reduces the movement of the disk arm, reducing the total latency for the batch of requests.

#### 6.1.7.2 Dallying

*Dallying* is delaying a request on the chance that the operation won't be needed, or to create more opportunities for batching. For example, a stage may delay a request that overwrites a disk block in the hope that a second one will come along for the same block. If a second one comes along, the stage can delete the first request and perform just the second one. As applied to writes, this benefit is sometimes called *write absorption*.

Dallying also increases the opportunities for batching. It purposely increases the latency of some requests in the hope that more requests will come along that can be combined with the delayed requests to form a batch. In this case, dallying increases the latency of some requests to improve the average latency of all requests.

A key design question in dallying is to decide how long to wait. There is no generic answer to this question. The costs and benefits of dallying are application and system specific.

#### 6.1.7.3 Speculation

*Speculation* is performing an operation in advance of receiving a request on the chance that it will be requested. The goal is that the results can be delivered with less latency and perhaps with less setup overhead. Speculation can achieve this goal in two different ways. First, speculation can perform operations using otherwise idle resources. In this case, even if the speculation is wrong, performing the additional operations has no downside. Second, speculation can use a busy resource to do an

operation that has a long lead time so that the result of the operation can be available without waiting if it turns out to be needed. In this case, speculation might increase the delay and overhead of other requests without benefit because the prediction that the results may be needed might turn out to be wrong.

Speculating may sound bewildering because how can a computer system predict the input of an operation if it hasn't received the request yet, and how can it predict if the result of the operation will be useful in the future? Fortunately, many applications have request patterns that a system designer can exploit to predict an input. In some cases, the input value is evident; for example, a future instruction may add register 5 to register 9, and these register values may be available now. In some cases, the input values can be predicted accurately; for example, a program that asks to read byte $n$ is likely to want to read bytes $n + 1$, $n + 2$, and so on, too. Similarly, for many applications a system can predict what results will be useful in the future. If a program performs instruction $n$, it will likely soon need the result of instruction $n + 1$; only when the instruction $n$ is a JMP will the prediction be wrong.

Sometimes a system can use speculation even if the system cannot predict accurately what the input to an operation is or whether the result will be useful. For example, if an input has only two values, then the system might create a new thread and have the main thread run with one input value and the second thread with the other input value. Later, when the system knows the value of the input, it terminates the thread that is computing with the wrong value and undoes any changes that thread might have made. This use of speculation becomes challenging when it involves shared state that is updated by different thread, but using techniques presented in Chapter 9 [on-line] it is possible to undo the operations of a thread, even when shared state is involved.

Speculation creates more opportunities for batching and dallying. If the system speculates that a read request for block $n$ will be followed by read requests for blocks $n + 1$ through $n + 8$, then the system can batch those read requests. If a write request might soon be followed by another write request, the system can dally for a while to see if any others come in and, if so, batch all the writes together.

Key design questions associated with speculation are when to speculate and how much. Speculation can increase the load on later stages. If this increase in load results in a load higher than the capacity of a later stage, then requests must wait and latency will increase. Also, any work done that turns out to be not useful is overhead, and performing this unnecessary work may slow down other requests. There is no generic answer to this design question; instead, a designer must evaluate the benefits and cost of speculation in the context of the system.

#### *6.1.7.4 Challenges with Batching, Dallying, and Speculation*

Batching, dallying, and speculation introduce complexity because they introduce concurrency. The designer must coordinate incoming requests with the requests that are batched, dallied, or speculated. Furthermore, if the requested operations share variables, the designer must coordinate the references to these variables. Since coordination is difficult to get right, a designer must use these performance-enhancing

techniques with discipline. There is always the risk that by the time the designer has worked out the concurrency problems and the system has made it through the system tests, technology improvements will have made the extra complexity unnecessary. Problem set *14* explores several performance-enhancing techniques and their challenges with a simple multithreaded service.

### 6.1.8 An Example: The I/O Bottleneck

We illustrate design for performance using batching, dallying, and speculation through a case study involving a magnetic disk such as was described in Sidebar 2.2. The performance problem with disks is that they are made of mechanical components. As a result, reading and writing data to a magnetic disk is slow compared to devices that have no mechanical components, such as RAM chips. The disk is therefore a bottleneck in many applications. This bottleneck is usually referred to as the *I/O bottleneck*.

Recall from Sidebar 2.2 that the performance of reading and writing a disk block is determined by (1) the time to move the head to the appropriate track (the seek latency); (2) plus the time to wait until the requested sector rotates under the disk head (the rotational latency); (3) plus the time to transfer the data from the disk to the computer (the transfer latency).

The I/O bottleneck is getting worse over time. Seek latency and rotational latency are not improving as fast as processor performance. Thus, from the perspective of programs running on ever faster processors, I/O is getting slower over time. This problem is an example of problems due to incommensurate rates of technology improvement. Following the ***incommensurate scaling rule*** of Chapter 1, applications and systems have been redesigned several times over the last few decades to cope with the I/O bottleneck.

To build some intuition for the I/O bottleneck, consider a typical disk of the last decade. The average seek latency (the time to move the head over one-third of the disk) is about 8 milliseconds. The disks spin at 7,200 rotations per minute, which is one rotation every 8.33 milliseconds. On average, the disk has to wait a half rotation for the desired block to be under the disk head; thus, the average rotational latency is 4.17 milliseconds.

Bits read from a disk encounter two potential transfer rate limits, either of which may become the bottleneck. The first limit is mechanical: the rate at which bits spin under the disk heads on their way to a buffer. The second limit is electrical: the rate at which the I/O channel or I/O bus can transfer the contents of the buffer to the computer. A typical modern 400-gigabyte disk has 16,383 cylinders, or about 24 megabytes per cylinder. That disk would probably have 8 two-sided platters and thus 16 read/write heads, so there would be $24/16 = 1.5$ megabytes per track. When rotating at 7,200 revolutions per minute (120 revolutions per second), the bits will go by a head at $120 \times 1.5 = 180$ megabytes per second. The I/O channel speed depends on which standard bus connects the disk to the computer. For the Integrated Device Electronics (IDE) bus, 66 megabytes per second is a common number in practice; for

the Serial ATA 3 bus the limit is 3 gigabytes per second. Thus, the IDE bus would be the bottleneck at 66 megabytes per second; with a Serial ATA 3 bus, the disk mechanics would be the bottleneck at 180 megabytes per second.

Using such a disk and I/O standard, reading a 4-kilobyte block chosen at random takes:

average seek time + average rotation latency + transmission of 4 kilobytes
= 8 + 4.17 + (4 / (180 × 1024)) × 1000 milliseconds
= 8 + 4.17 + 0.02 milliseconds
= 12.19 milliseconds

The throughput for reading randomly chosen blocks one by one is:

= 1000/12.19 × 4 kilobytes per second
= 328 kilobytes per second

The main opportunity to handle the I/O bottleneck is to drive the disk at the transfer rate (180 megabytes per second) instead of the rate of seeks and rotations (327 kilobytes per second). This strategy is an example of hiding latency (moving the disk arm) by exploiting throughput (the high transfer rate between computer and disk).

Consider the following prototypical program, which processes a large input file sequentially and produces an output file sequentially:

```
1    in ← OPEN ("in", READ)              // open "in" for reading
2    out ← OPEN ("out", WRITE)           // open "out" for reading
3
4    while not ENDOFFILE (in) do
5        block ← READ (in, 4096)         // read 4 kilobyte block from in
6        block ← COMPUTE (block)         // compute for 1 millisecond
7        WRITE (out, block, 4096)        // write 4 kilobyte block to out
8    CLOSE (in)
9    CLOSE (out)
```

If we think of this application as a pipeline, then there are the following stages: (1) the file system, which reads data from a disk in response to a READ (line 5); (2) the application, which computes new data using the data read (line 6); and (3) the file system, which writes the new data to the disk (line 7).

If the application is organized naively, without batching, dallying, and speculation, the average time to go around the loop is equal to the latency of the three stages. The latencies of the two file system stages are dominated by the latency of the disk operations, and thus we can approximate the average latency of the loop as follows:

reading 4 kilobytes + 1 millisecond of computation + writing 4 kilobytes
= 12.19 + 1 + 12.19 milliseconds
= 25.38 milliseconds

In practice, the latency might be lower because this calculation assumes that each disk access involves an average seek time, but if the file system has allocated the blocks near each other on the disk, the disk might have to perform only a short seek.

How can we improve the performance of this program? The program reads the file only once, and thus a cache cannot improve the latency of reading a block. The only alternative is to hide the latency of read and write operations. The simplest optimization is to overlap the reading and writing of blocks with the computation on line 6. Let's start with reading.

When the application READs a block, the file system can speculate that the application will read a few blocks following the requested block. This speculation can improve performance for our application if we combine it with two further optimizations. First, we modify the file system to lay out the blocks of a file contiguously. Second, we modify the file system to prefetch an entire track of data on each read. Our prototypical application is perfect for prefetching, since the whole data set is read sequentially.

These optimizations eliminate rotational delay before reading can start. An entire track can be read in:

average seek time + 1 rotational delay
= 8 + 8.33 milliseconds
= 16.33 milliseconds

With 1.5 megabytes (1,536 kilobytes) per track, the file system issues one read request per 384 (1536/4) loop iterations, and we have the following timing diagram:

The average time for 384 iterations is:

= reading 1536 kilobytes + 384 × (1 millisecond of computation + writing 4 kilobytes)
= 16.33 + 384 × (1 + 12.19) milliseconds
= 16.33 + 5065 milliseconds
= 5081 milliseconds

Thus, the average time for a loop iteration is 5081/384 = 13.23 milliseconds, a substantial improvement over 25.38 milliseconds.

We can improve the performance of writing blocks by dallying and batching write requests. We modify WRITE to use a buffer of blocks in RAM (see Figure 6.5). The WRITE call stores the updated block into this buffer and returns immediately, and the application thread can continue. When the buffers fill up, the file system can batch the blocks in the buffer and combine them into a single disk request, which the disk can process in parallel with the processor running the application. Batching allows the

disk controller to execute writes to adjacent sectors with no rotational delay. Because blocks are written contiguously in our example, the file system may take 384 contiguous writes and batch them together to form a complete track write. These optimizations result in the following timing diagram:

This optimization reduces the average time around the loop:

= (16.33 + 384 + 16.33)/384 milliseconds
= 1.09 milliseconds

If we modify the file system to prefetch the next track before the application calls the 385th READ, we can overlap computation and I/O completely. If we modify the file system to read the next track after it has processed, say, half of the last track read, then we obtain the following timing diagram for each block of 384 loop iterations, other than the first one:

**FIGURE 6.5**

Using a buffer to delay writes.

Now the system overlaps computation with I/O completely, the average time around the loop is 1 millisecond, and the application is now bottlenecked by computation, rather than by I/O.

The optimizations take advantage of the facts that the application processes the input file sequentially and that the file system allocates blocks for the output contiguously on disk. However, even for applications that process blocks not in the order in which they are laid out on the disk, these optimizations can be beneficial. The file system, for example, can reorder the disk requests for a batch in the order of their track number, thereby minimizing disk arm movement, and thus improving performance for the whole batch of requests. (To understand what a good algorithm is for disk scheduling, we need to think more broadly about scheduling requests in computer systems, which is the topic of Section 6.3.)

The analysis assumes a simple performance model for the disk; for a more in-depth discussion of the performance of disks, see Suggestions for Further Reading 6.3.1. The analysis also assumes a single disk; using several disks can offer opportunities for improving performance. For example, RAIDs have several disks (see Section 2.1.1.4), which allows the file system to interleave read and write requests instead of serving them one by one, providing additional opportunities for increasing performance. Finally, practical, alternative storage technologies are emerging, which change the trade-offs. For example, designing a high performance storage system with Flash disks provides new opportunities and new challenges (see, for example, Suggestions for Further Reading 6.3.4).

A buffer without write-through can provide substantial performance improvements but can lose on reliability. If the computer system fails before the file system has written out data to the disk, some data is lost. The basic problem is how long to delay before forcing the data to the disk. The longer the file system delays writes, the larger the opportunity for higher performance will be, but the greater the probability that data will be lost if, for example, the power fails and the volatile RAM resets.

There are at least four choices as to when the WRITE request to the disk can be issued:

- Before WRITE returns to the caller (write-through).
- On an explicit *force* request from the user (user-controlled write).
- When the file is closed (another kind of user-controlled write).
- When a certain number of write requests have been accumulated or when some fixed time has passed since the last write request. This option can be a bad idea if one needs to control the order of writes.

A buffer without write-through also introduces some other complexities, mostly related to reliability in the face of system failures. First, if the file system batches several write requests in a single disk request, then the disk may write the blocks in an order different from the order issued by the file system to reduce seek time. Thus, the disk may not reflect a consistent state if the system crashes halfway through the batched write request. Second, the disk controller may also use a buffer without write-through. The file system may think the data has been stored reliably on disk when, in fact, the disk controller is caching it. We shall see systematic ways of controlling the problem caused by caches without write-through in Chapter 9 [on-line]; a nice application of these systematic ways to design a high-performance and robust file system is given by Ganger and Patt [Suggestions for Further Reading 6.3.3]. In general, here we have a good example that increased performance comes at the cost of increased complexity, as illustrated by Figure 1.1.

The prototypical application represents one particular workload for which the techniques described above improve performance well. Improving the performance of the prototypical application is challenging because it doesn't reuse a block. Many applications read and write a block multiple times, and in that case additional techniques are available to improve performance. In particular, in that case it is worthwhile for the file system to maintain a cache of recently read blocks in RAM. If an

application reads a block that is already in the cache, then the file system doesn't have to perform any disk operations.

Introducing a cache leads to additional coordination constraints. The file system may have to coordinate WRITE and READ operations with outstanding disk requests. For example: a READ operation may force the removal of a modified block from the cache to make space for the block to be read. But the file system cannot throw out a modified block until it has been written it to the disk, so the file system must wait until the write request of the modified block has completed before proceeding with the READ operation.

Understanding for what workloads a cache works well, learning how to design a cache (e.g., which block to throw out to make space for a new block), and analyzing a cache's performance benefits are sophisticated topics, which we discuss next. Problem set *16* explores these issues, as well as topics related to scheduling, in the context of a simple high-performance video server.

## 6.2 MULTILEVEL MEMORIES

The previous section described how to address the I/O bottleneck by using two types of digital memory devices: a RAM chip and a magnetic disk, which have different capacities, costs, and speeds. A system designer would like to have a single memory device that is both as large and as fast as the application requires, and that at the same time is affordable. Unfortunately, application requirements often exceed one or another of these three parameters—a memory device that is both fast enough and large enough is usually too expensive—so the designer must make some trade-offs. The usual trade-off is to use more than one memory device, for example, one that is fast but expensive (and thus necessarily too small), and another that is large and cheap (but slower than desired). But fitting an application into such an environment adds the complexity of deciding which parts of the application should use the small, fast memory and which parts the large, slow one. It may also increase maintenance effort if the memory configuration changes.

One might think that improvements in technology may eventually make a brute-force solution economical—someday the designer can just buy a memory that is both large and fast enough. But there are two problems with that thought: one practical and one intrinsic. The practical problem is that historically the increase in memory size has been matched by an equal increase in problem sizes. That is, the data that people want to manipulate has grown along with memory technology.

The intrinsic problem is that memory has a trade-off between latency and size. This trade-off becomes clear when we consider the underlying physics. Even if one has an unlimited budget to throw at the design problem, the speed of light interferes. To see why, imagine a processor that occupies a single point in space, with memory clustered around it in a sphere, using the densest packing that physics allows. With this packing, some of the memory cells will end up located quite near the processor, so the latency (that is, the time required for access to those cells, which requires a

propagation of a signal at the speed of light from the processor to the bit and back) will be short. But because only a few memory cells can fit in the space near the processor, most memory cells will be farther away, and they will necessarily have a larger latency. Put another way, for any specified minimum latency requirement, there will be some memory size for which at least some cells must exceed that latency, based on speed-of-light considerations alone. Moreover, the geometry of spheres (the volume of a shell of radius $r$ grows with the square of $r$) dictates that there must be more high-latency cells than low-latency ones.

In practical engineering terms, available technologies also exhibit analogous packing problems. For example, the latency of a memory array on the same chip as the processor (where it is usually called an L1 cache) is less than the latency of a separate memory chip (which is usually called an L2 cache), which in turn is less than the latency of a much larger memory implemented as a collection of memory chips on a separate card. The result is that the designer is usually forced to deal with a composite memory system in which different component memories have different parameters of latency, capacity, and cost. The challenge then becomes that of achieving overall maximum performance by deciding which data items to store in the fastest memory device, which can be relegated to the slower devices, and deciding if and when to move data items from one memory device to another.

### 6.2.1 Memory Characterization

Different memory devices are characterized not just by dimensions of capacity, latency, and cost, but also by cell size and throughput. In more detail, these dimensions are:

- *Capacity*, measured in bits or bytes. For example, a RAM chip may have a capacity from a few to tens of megabytes, whereas magnetic disks have capacities measured in scores or hundreds of gigabytes.
- *Average random latency*, measured in seconds or processor clock cycles, for a memory cell chosen at random. For example, the average latency of RAM is measured in nanoseconds, which might correspond to hundreds of processor clock cycles. (On closer examination, RAM READ latency is actually more complicated—see Sidebar 6.4.) Magnetic disks have an average latency measured in milliseconds, which corresponds to millions of processor clock cycles. In addition, magnetic disks, because of their mechanical components, usually have a much wider variance in their latency than does RAM.
- *Cost*, measured in some currency per storage unit. The cost of RAM is typically measured in cents per megabyte, while the cost of magnetic disks is measured in dollars per gigabyte.
- *Cell size,* measured as the number of bits or bytes transferred in or out of the device by a single READ or WRITE operation. For example, the cell size of RAM is typically a few bytes, perhaps 4, 8, or 16. The cell size of a magnetic disk is typically 512 bytes or more.

**Sidebar 6.4 RAM Latency** Performance analysis sometimes requires a better model of random access memory latency for READ operations. Most random access memory devices actually have two latency parameters of interest: *cycle time* and *access time*. The distinction arises because the physical memory device may need time to recover from one access before it can handle the next one. For example, some memory READ mechanisms are destructive: to READ a bit out, the memory device literally smashes the bit and examines the resulting debris to determine the value that the bit had. Once it determines that value, the memory device writes the bit back so that its value can again be available for future READs. This write-back operation typically cannot be overlapped with an immediately following READ operation. Thus, the *cycle time* of the memory device is the minimum time that must pass between issuance of one READ request and issuance of the next one. However, the result of the READ may be available for delivery to the processor well before the cycle time is complete. The time from issuance of the READ to delivery of the response to the processor is known as the *access time* of the memory device. The following figure illustrates.

- *Throughput*, measured in bits per second. RAM can typically transfer data at rates measured in gigabytes per second, while magnetic disks transfer at the rate of hundreds of megabytes per second.

The differences between RAM and magnetic disks along these dimensions are orders of magnitude in all cases. RAM is typically about five orders of magnitude faster than magnetic disk and two orders of magnitude more expensive. Many, but not all, of the dimensions have been improving rapidly. For example, the capacity of magnetic disks has doubled, and cost has fallen by a factor of 2 every year for the last two decades, while the average latency has improved by only a factor of 2 in that same 20 years. Latency has not improved much because it involves mechanical operations as opposed to all-electronic ones, as described in Sidebar 2.2. This incommensurate rate of technology improvement makes effective memory management a challenge to implement well.

### 6.2.2 Multilevel Memory Management using Virtual Memory

Because larger latency and larger capacity usually go hand in hand, it is customary and useful to describe the various available memory devices as belonging to different levels, with the fastest, smallest device being at the highest level and slower, larger

**FIGURE 6.6**

A multilevel memory pyramid.

devices being at lower levels. A memory system constructed of devices from more than one level is called a *multilevel memory*. Figure 6.6 shows a popular way of depicting the multiple levels, using a pyramid, in which higher levels are narrower, suggesting that their capacity is smaller. The memories in the top of the hierarchy are fast and expensive, and they are therefore small; the memories at the bottom of the hierarchy are slow and inexpensive; and so they can be much bigger. In a modern computer system, an information item can be in the registers of the processor, the L1 cache memory, the L2 cache memory, main memory, a RAM disk cache, on a magnetic disk, or even on another computer that is accessible through a network.

Two quite different ways can be used to manage a multilevel memory. One way is to leave it to each application programmer to decide in which memory to place data items and when to move them. The second way is automatic management: a subsystem independent of any application program observes the pattern of memory references being made by the program. With that pattern in mind, the automatic memory management subsystem decides where to place data items and when to move them from one memory device to another.

Most modern memory management is automatic because (1) there exist automatic algorithms that have good performance on average and (2) automatic memory management relieves the programmer of the need to conform the program to specifics of the memory system such as the capacities of the various levels. Without automatic memory management, the application program explicitly allocates memory space within each level and moves data items from one memory level to another. Such programs become dependent on the particular hardware configuration for which they were written, which makes them difficult to write, to maintain, or to move to a different computer. If someone adds more memory to one of the levels, the program will probably have to be modified to take advantage of it. If some memory is removed, the program may stop working.

As Chapter 2 described, there are two commonly encountered memory interfaces: an interface to small-cell memory to which threads refer using `READ` and `WRITE` operations, and an interface to large-cell memory to which threads refer using `GET` and

PUT operations. These two interfaces correspond roughly to the levels of a multilevel memory; higher levels typically have small cells and use the READ/WRITE interface, while lower levels typically have large cells and use the GET/PUT interface.

One opportunity in the design of an automatically managed multilevel memory system is to combine it with a virtual memory manager in such a way that the small-cell READ/WRITE interface appears to the application program to apply to the entire memory system. This creates what is sometimes called a *one-level store*, an idea first introduced in the Atlas system.* Put another way, this scheme virtualizes the entire memory system around the small-cell READ/WRITE interface, thus hiding from the application programmer the GET/PUT interface as well as the specifics of latency, capacity, cost, cell size, and throughput of the component memory devices. The programmer instead sees a single memory system that appears to have a large capacity, a uniform cell size, a modest average cost per bit, and a latency and throughput that depend on the memory access patterns of the application.

Just as with virtualization of addresses, virtualization of the READ/WRITE memory interface further exploits the design principle *decouple modules with indirection*. In this case, indirection allows the virtual memory manager to translate any particular virtual address not only to different physical memory addresses at different times but also to addresses in a different memory level. With the support of the virtual memory manager, a *multilevel memory manager* can then rearrange the data among the memory levels without having to modify any application program. By adding one more feature, the *indirection exception*, this rearrangement can become completely automatic. An indirection exception is a memory reference exception that indicates that memory manager cannot translate a particular virtual address. The exception handler examines the virtual address and may bind or rebind that value before resuming the interrupted thread.

With these techniques, the virtual memory manager not only can contain errors and enforce modularity, but it also can help make it appear to the program that there is a single, uniform, large memory. The multilevel memory management feature can be slipped in underneath the application program *transparently*, which means that the application program does not need to be modified.

Virtualization of widely used interfaces creates an opportunity to transparently add features and thus evolve a system. Since by definition many modules use a widely used interface, the transparent addition of features beneath such an interface can have a wide impact, without having to change the clients of the interface. The memory interface is an example of such a widely used interface. In addition to implementing single-level stores, here are several other ways in which systems designers have used a virtual memory manager with indirection exceptions:

- *Memory-mapped files*. When an application opens a file, the virtual memory manager can map files into an application's address space, which allows the application to read and write portions of a file as if they were located in RAM. Memory-mapped files extend the idea of a single-level store to include files.

*T. Kilburn, D.B.J. Edwards, M.J. Lanigan, and F.H. Sumner. One-level storage system. *IRE Transactions on Electronic Computers, EC-11*, 2 (April 1962), pages 223–235.

- *Copy-on-write*. If two threads are working on the same data concurrently, then the data can be stored once in memory by mapping the pages that hold the data with only READ permissions. If one of the threads attempts to write a shared page, the virtual memory hardware will interrupt the processor with a permission exception. The handler can demultiplex this exception as an indirection exception of the type copy-on-write. In response to the indirection exception, the virtual memory manager transparently makes a copy of the page and maps the copy with READ and WRITE permissions in the address space of the threads that wants to write the page. With this technique, only changed pages must be copied.
- *On-demand zero-filled pages*. When an application starts, a large part of its address space must be filled with zeros—for instance, the parts of the address space that aren't preinitialized with instructions or initial data values. Instead of allocating zero-filled pages in RAM or on disk, the virtual memory manager can map those pages without READ and WRITE permissions. When the application refers to one of those pages, the virtual memory hardware will interrupt the processor with a memory reference exception. The exception handler can demultiplex this exception as an indirection exception of the type zero-fill. In response to this zero-fill exception, the virtual memory manager allocates a page dynamically and fills it with zeros. This technique can save storage in RAM or on disk because the parts of the address space that the application doesn't use will not take up space.
- *One zero-filled page*. Some designers implement zero-filled pages with a copy-on-write exception. The virtual memory manager allocates just one page filled with zeros and maps that one page in all page-map entries for pages that should contain all zeros, but granting only READ permission. Then, if a thread writes to this read-only zero-filled page, the exception handler will demultiplex this indirect exception as a copy-on-write exception, and the virtual memory manager will make a copy and update that thread's page table to have WRITE permission for the copy.
- *Virtual shared memory*. Several threads running on different computers can share a single address space. When a thread refers to a page that isn't in its local RAM, the virtual memory manager can fetch the page over the network from a remote computer's RAM. The remote virtual memory manager unmaps the page and sends the content of the page back. The Apollo DOMAIN system (mentioned in Suggestions for Further Reading 3.2.1) used this idea to make a collection of distributed computers look like one computer. Li and Hudak use this idea to run parallel applications on a collection of workstations with shared virtual memory [Suggestions for Further Reading 10.1.8].

The virtual memory design for the Mach operating system [Suggestions for Further Reading 6.1.3] provides an example design that supports many of these features and that is used by some current operating systems.

The remainder of this section focuses on building large virtual memories using automatic multilevel memory management. To do so, a designer must address some

challenging problems, but, once it is designed, application programmers do not have to worry about memory management. Except for embedded devices (e.g., a computer acting as the controller of a microwave oven), nearly all modern computer systems use virtual memory to contain errors, enforce modularity, and manage multiple memory levels.

### 6.2.3 Adding Multilevel Memory Management to a Virtual Memory

Suppose for the moment that we have two memory devices, one that has a READ/WRITE interface, such as a RAM, and the second that has a GET/PUT interface, such as a magnetic disk. If the processor is already equipped with a virtual memory manager such as the one illustrated in Figure 5.20, it is straightforward to add multilevel memory management to create a one-level store.

The basic idea is that at any instant, only some of the pages listed in the page map are actually in RAM (because the RAM has limited capacity) and the rest are on the disk. To support this idea, we add to each entry of the page map a single bit, called the *resident* bit, in the column identified as *r*? in Figure 6.7. If the resident bit of a page is TRUE, that means that the page is in a block of RAM and the physical address in the page map identifies that block. If the resident bit of a page is FALSE, that means that the page is not currently in any block of RAM; it is instead on some block on the disk.

In the example, pages 10 and 12 are in RAM, while page 11 is only on the disk. Thus, references to pages 10 and 12 can proceed as usual, but if the program tries to refer to page 11, for example, with a LOAD instruction, the virtual memory manager must take some action because the processor can't refer to the disk with READ/WRITE operations. The action it takes is to alert the multilevel memory manager that it needs

**FIGURE 6.7**

Integrating a virtual memory manager with a multilevel memory manager. The virtual memory manager is typically implemented in hardware, while the multilevel memory manager is typically implemented in software as part of the operating system.

```
7.1   if resident of page_table[page] = FALSE then      // check if page is resident
7.2       signal MISSING_PAGE (page)                    // no, signal a missing-page exception
7.3   else
7.4       block ← page_table[page].address              // Index into page map
8         physical ← block + offset                     // Concatenate block and offset
9         return physical                               // return physical address
```

**FIGURE 6.8**

Replacement for lines *7–9* of procedure TRANSLATE of Chapter 5, to implement a multilevel memory.

to use the GET/PUT interface of the disk to copy that page from the disk block into some block in the RAM where the processor can directly refer to it. For this purpose, the multilevel memory manager (at least conceptually) maintains a second, parallel map that translates page numbers to disk block addresses. In practice, real implementations may merge the two maps.

The pseudocode of Figure 6.8 (which replaces lines *7–9* of the version of the TRANSLATE procedure of Section 5.4.3.1) illustrates the integration. When a program makes a reference to a virtual memory address, the virtual memory manager invokes TRANSLATE, which (after performing the usual domain and permission checks) looks up the page number in the page map. If the requested address is in a page that is resident in memory, the manager proceeds as it did in Chapter 5, translating the virtual address to a physical address in the RAM. If the page is not resident, the manager signals that the page is missing.

The pseudocode of Figure 6.8 describes the operation of the virtual memory manager as a procedure, but to maintain adequate performance a virtual memory manager is nearly always implemented in hardware because it must translate every virtual address the processor issues. With this page-based design, the virtual memory manager interrupts the processor with an indirect exception that is called a *missing-page exception* or a *page fault*.

The exception handler examines the value in the program counter register to determine which instruction caused the missing-page exception, and it then examines that instruction in memory to see what address that instruction issued. Next, it calls SEND (see Figure 5.30) with a request containing the missing page number to the port for the multilevel memory manager. SEND invokes ADVANCE, which wakes up a thread of the multilevel memory manager. Then, the handler invokes AWAIT on behalf of the application program's thread (i.e., with the stack of the thread that caused the exception). The AWAIT procedure yields the processor.

The multilevel memory manager receives the request and copies pages between RAM blocks and disk blocks as they are needed. For each missing-page exception, the multilevel memory manager first looks up that page in its parallel page map to determine the address of the disk block that holds the page. Next, it locates an unused block in RAM. With these two parameters, it issues a GET for the disk block that holds the

page, writing the result into the unused RAM block. The multilevel memory manager then informs the virtual memory manager about the presence of the page in RAM by writing the block number in the virtual memory manager's page map and changing the resident bit to TRUE, and makes the thread that experienced the missing-page exception runnable by calling ADVANCE.

When that thread next runs, it backs up the program counter found in its return point so that after the return to user mode the application program will reexecute the instruction that encountered the missing page. Since that page is now resident in RAM and the multilevel memory manager has updated the mappings of the virtual memory manager, this time the TRANSLATE function will be able to translate the virtual address to a physical address.

If all blocks in RAM are occupied with pages, the multilevel memory manager must select some page from RAM and remove it to make space for the missing page. The page selected for removal is known colloquially as the *victim*, and the algorithm that the multilevel memory manager uses to select a victim is called the *page-removal policy*. A bad choice (for example, systematically selecting for removal the page that will be needed by the next memory access) could cause the multilevel memory system to run at the rate that pages can be retrieved from the disk, rather than the rate that words can be retrieved from RAM. In practice, a selection algorithm that exploits a property of most programs known as *locality* can allow those programs to run with only occasional missing-page exceptions. The locality property is discussed in Section 6.2.5, and several different page removal policies are discussed in Section 6.2.6.

If the selected page was modified while it was in RAM, the multilevel memory manager must PUT the modified page back to the disk before issuing a GET for the new page. Thus, in the worst case, a missing-page exception results in two accesses to the disk: one to PUT a modified page back to the disk and one to GET the page requested by the missing-page exception handler. In the best case, the page in RAM has not been modified since being read from disk, so it is identical to the disk copy. In this case, the multilevel memory manager can simply adjust the virtual memory page-map entry to show that this page is no longer resident, and the number of disk accesses needed is just the one to GET the missing page. This scheme maintains a copy of *every* virtual memory page on the disk, whether or not that page is also resident in RAM, so the disk must be larger than the RAM and the effective virtual memory capacity is equal to the space allocated for virtual memory on the disk.

A concern about this scheme is that it introduces what sometimes are called *implicit I/Os*. The multilevel memory manager performs I/O operations beyond the ones performed by the application (which are then called *explicit I/Os*). Given that a disk is often an I/O bottleneck (see Section 6.1.8), these implicit I/Os may risk slowing down the application. Problem set *15* explores some of the issues related to implicit I/Os in the context of a page-based and an object-based single-level store.

To mitigate the I/O bottleneck for missing-page exceptions, a designer can exploit concurrency by implementing the multilevel memory manager with multiple threads. When a missing-page exception occurs, the next available multilevel memory manager thread can start to work on that missing page. The thread begins a GET operation

and waits for the GET to complete. Meanwhile, the thread manager can assign the processor to some other thread. When the GET completes, an interrupt notifies the multilevel memory manager thread and it completes processing of the missing-page exception. With this organization, the multilevel memory manager can overlap the handling of a missing-page exception with the computation of other threads, and it can handle multiple missing-page exceptions concurrently.

A quite different, less modular organization is used in many older systems: integrate the multilevel memory manager with the virtual memory manager in the kernel, with the goal of reducing the number of instructions required to handle a missing-page exception, and thus improving performance. Typically, when integrated, the multilevel memory manager runs in the application thread in the kernel, thus reducing the number of threads and avoiding the cost of context switches. Most such systems were designed decades ago when instruction count was a major concern.

Comparing these two organizations, one benefit of the modularity of a separate multilevel memory manager is that several multilevel memory managers can easily coexist. For example, one multilevel memory manager that reads and writes blocks to a magnetic disk to provide applications with the illusion of a large memory may

**Sidebar 6.5 Design Hint: Separate Mechanism from Policy** If a module needs to make a policy decision, it is better to leave the policy decision to the clients of the module so that they can make a decision that meets their goals. If the interface between the mechanism and policy module is well defined, then this split allows the schedule policies to be changed without having to change the implementation of the mechanism. For example, one could replace the page-removal policy without having to change the mechanism for handling missing-page exceptions. Furthermore, when porting the missing-page exception mechanism to another processor, the missing-page handler may have to be rewritten, but the policy module may require no modifications.

Of course, if a change in policy requires changes to the interface between the mechanism and policy modules, then both modules must be replaced. Thus, the success of following the hint is limited by how well the interface between the mechanism and policy module is designed. The potential downsides of separating mechanism and policy are a loss in performance due to control transfers between the mechanism and policy module, and increased complexity if flexibility is unneeded. For example, if one policy is always the right one, then separating policy and mechanism may just be unnecessary complexity.

In the case of multilevel memory management, separating the missing-page mechanism from the page replacement policy is mostly for ease of porting because the least recently used page-replacement policy (discussed in Section 6.2.5) works well in practice for most applications.

coexist with another multilevel memory manager that provides memory-mapped files. These different multilevel memory managers can be implemented as separate modules, as opposed to being integrated together with the virtual memory manager. Separating the multilevel memory manager from the virtual memory manager is an example of the design hint *separate mechanism from policy*, discussed in Sidebar 6.5. The Mach virtual memory system is an example of a modern, modular design [Suggestions for Further Reading 6.1.3].

If the multilevel managers are implemented as separate modules from the virtual memory manager, then the designer has the choice of running the multilevel manager modules in kernel mode or as separate applications in user mode. For the same reasons that many deployed systems are monolithic kernel systems (see Section 5.3.6), designers often choose to run the multilevel manager modules in kernel mode. In a few systems, the multilevel managers run as separate user applications with their own address spaces.

One question that requires some careful thought is what to do if a multilevel memory manager encounters a missing-page exception in its own procedures or data. In principle, there is no problem with recursive missing-page exceptions as long as the recursion bottoms out. To ensure that the recursion does bottom out, it is necessary to make sure that some essential set of pages (for example, the pages containing the instructions and tables of the interrupt handler and the kernel thread manager) is never selected for removal from RAM. The usual method is to add a mark to the page-map entries for those essential pages saying, in effect, "Don't remove this page." Pages so marked are commonly said to be *wired down*.

### 6.2.4 Analyzing Multilevel Memory Systems

Multilevel memories are common engineering practice. From the processor perspective, stored instructions and data traverse some pyramid of memory devices such as the one that was illustrated in Figure 6.6. But when analyzing or constructing a multilevel memory, we do so by analyzing each adjacent pair of levels individually as a two-level memory system, and then stacking the several two-level memory systems. (One reason for doing it this way is that it seems to work. Another is that no one has yet figured out a more satisfactory way to analyze or manage a three- or more-level memory as a single system.)

Devices that function as the fast level in a two-level memory system are called *primary devices*, and devices that function as the slow level are called *secondary devices*. In virtual memory systems, the primary device is usually some form of RAM; the secondary device can be either a slower RAM or a magnetic disk. Web browsers typically use the local disk as a cache that holds pages of remote Web services. In this case, the primary device is a magnetic disk; the remote service is the secondary device, which may itself use magnetic disks for storage. The multilevel memory management algorithms described in the remainder of this section apply to both of these different configurations, and many others.

A cache and a virtual memory are two similar kinds of multilevel memory managers. They are so similar, in fact, that the only difference between them is in the name space they provide for memory cells:

- The user of a *cache* identifies memory cells using the name space of the secondary memory device.
- The user of a *virtual memory* identifies memory cells using the name space of the primary memory device.

Apart from that difference, designers of virtual memories and caches choose policies for multilevel memory management from the same range of possibilities.

The pyramid of Figure 6.6 is typically implemented with the highest level explicitly managed by the application, a cache design at some levels and a virtual memory design at other levels. For example, a multilevel memory system that includes all six levels of the figure might be organized something like the following:

1. At the highest level, the registers of the processor are the primary device, and the rest of the memory system is the secondary device. The application program (as constructed by the compiler code generator) explicitly loads and stores the registers to and from the rest of the memory system.

2. When the processor issues a READ or WRITE to the rest of the memory system, it provides as an argument a name from the main memory name space, but this name goes to a primary memory device located on the same chip as the processor. Since the name is from the lower level main memory name space, this level of memory is being managed as a cache, commonly known as a "level 1 cache" or "L1 cache".

3. If the named cell is not found in the level 1 cache, a multilevel memory manager looks in its secondary memory, an off-chip memory device, but again using the name from the main memory name space. The off-chip memory is thus another example of a cache, this one known as a "level 2 cache" or "L2 cache".

4. The level 2 cache is now the primary device, and if the named memory cell is not found there, the next lower multilevel manager (the one that manages the level 2/main memory pair) looks in its secondary device—the main memory—still using the name from the main memory name space.

5. At the next level, the main memory is the primary device. If an addressed cell is not in main memory, a virtual memory manager invokes the next lower level multilevel memory manager (the one described in Section 6.2.3, that manages movement between main and disk memory) but still using the name from the main memory name space. The multilevel memory manager translates this name to a disk block address.

6. The sequence may continue down another layer; if the disk block is not found on the (primary) local disk, yet another multilevel memory manager may retrieve it from some remote (secondary) system. In some systems, this last memory pair is managed as a cache, and in others as a virtual memory.

It should be apparent that the above example is just one of a vast range of possibilities open to the multilevel memory designer.

### 6.2.5 Locality of Reference and Working Sets

It is not obvious that an automatically managed multilevel memory system should perform well. The basic requirement for acceptable performance is that all information items stored in the memory must not have equal frequency of use. If every item is used with equal frequency, then a multilevel memory cannot have good performance, since the overall memory will operate at approximately the speed of the slowest memory component. To illustrate this effect, consider a two-level memory system. The average latency of a two-level memory is:

$$AverageLatency = R_{bit} \times Latency_{primary} + R_{miss} \times Latency_{secondary} \tag{6.2}$$

The term $R_{bit}$ (known as the *hit ratio*) is the frequency with which items are found in the primary device, and $R_{miss}$ is $(1 - R_{bit})$. This formula is a direct application of Equation 6.1, (in Section 6.1) which gives the average performance of a system with a fast and slow path. Here the fast path is a reference to the primary device, while the slow path is a reference to the secondary device.

If accesses to every cell of the primary and secondary devices were of equal frequency, then the average latency would be proportional to the number of cells of each device:

$$AverageLatency = \frac{S_{primary}}{S_{primary} + S_{secondary}} \times T_{primary} + \frac{S_{secondary}}{S_{primary} + S_{secondary}} \times T_{secondary} \tag{6.3}$$

where $S$ is the capacity of a memory device and $T$ is its average latency. In a multilevel memory, it is typical that $T_{primary} \ll T_{secondary}$ and $S_{secondary} \gg S_{primary}$ (as, for example, with RAM for primary memory and magnetic disk for secondary memory), in which case the first term is much smaller than the second, the coefficient of the second term approaches 1, and $AverageLatency \approx T_{secondary}$. Thus, if accesses to every cell of primary and secondary are equally likely, a multilevel memory doesn't provide any performance benefit.

On the other hand, if the frequency of use of some stored items is significantly higher than the frequency of use of other stored items, even for a short time, automatically managed multilevel memory becomes feasible. For example, if, somehow, 99% of accesses were directed to the faster memory and only 1% to the slower memory, then the average latency would be:

$$AverageLatency = 0.99 \times T_{primary} + 0.01 \times T_{secondary} \tag{6.4}$$

Thus if the primary device is L2 cache with 1 nanosecond latency and the secondary device is main memory with 10 nanoseconds latency, the average latency becomes 0.99 + 0.10 = 1.09 nanoseconds, which makes the composite memory,

with a capacity equal to that of the main memory, nearly as fast as the L2 cache. For a second example, if the primary device is main memory with 10 nanoseconds latency and the secondary device is magnetic disk with average latency of 10 milliseconds, the average latency of the multilevel memory is

$$0.99 \times 10 \text{ nanoseconds} + 0.01 \times 10 \text{ milliseconds} = 100.0099 \text{ microseconds}$$

That latency is substantially larger than the 10 nanosecond primary memory latency, but it is also much smaller than the 10 millisecond secondary memory latency. In essence, a multilevel memory just exploits the design hint *optimize for the common case*.

Most applications are not so well behaved that one can identify a static set of information that is both small enough to fit in the primary device and for which reference is so concentrated that it is the target of 99% of all memory references. However, in many situations most memory references are to a small set of addresses for significant periods of time. As the application progresses, the area of concentration of access shifts, but its size still typically remains small. This concentration of access into a small but shifting locality is what makes an automatically managed multilevel memory system feasible. An application that exhibits such a concentration of accesses is said to have *locality of reference*.

Analyzing the situation, we can think of a running application as generating a stream of virtual addresses, known as the *reference string*. A reference string can exhibit locality of reference in two ways:

- *Temporal locality*: the reference string contains several closely spaced references to the same address.
- *Spatial locality*: the reference string contains several closely spaced references to adjacent addresses.

An automatically managed multilevel memory system can exploit temporal locality by keeping in the primary device those memory cells that appeared in the reference string recently—thus applying speculation. It can exploit spatial locality by moving into the primary device memory cells that are adjacent to those that have recently appeared in the reference string—a combination of speculation and batching (because issuing a `GET` to a secondary device can retrieve a large block of data that can occupy many adjacent memory cells in the primary device).

There are endless ways in which applications exhibit locality of reference:

- Programs are written as a sequence of instructions. Most of the time, the next instruction is stored in the memory cell that is physically adjacent to the previous instruction, thus creating spatial locality. In addition, applications frequently execute a loop, which means there will be repeated references to the same instructions, creating temporal locality. Between loops, conditional tests, and jumps, it is common to see many instruction references directed to a small subset of all the instructions of an application for an extended time. In addition, depending on the conditional structure, large parts of an application program may not be exercised at all.

- Data structures are typically organized so that a reference to one component of the structure makes references to physically nearby components more likely. Arrays are an example; reference to the first element is likely to be followed shortly by reference to the second. Similarly, if an application retrieves one field of a record, it will likely soon retrieve another field of the same record. Each of these examples creates spatial locality.
- Information processing applications typically process files sequentially. For example, a bank audit program may examine accounts one by one in physical storage order (creating spatial locality) and may perform multiple operations on each account (creating temporal locality).

Although most applications naturally exhibit a significant amount of locality of reference, to a certain extent the concept also embodies an element of self-fulfilling prophecy. Application programmers are usually aware that multilevel memory management is widely used, so they try to write programs that exhibit good locality of reference in the expectation of better performance.

If we look at an application that exhibits locality of reference, in a short time the application refers to only a subset of the total collection of memory cells. The set of references of an application in a given interval $\Delta t$ is called its *working set.* In one such interval, the application may execute a procedure or loop that operates on a group of related data items, causing most references to go to the text of the procedure and that group of data items. Then, the application might call another procedure, causing most references to go to the text and related data items of that procedure. The working set of an application thus grows, shrinks, and shifts with time.

If at some instant the current working set of an application is entirely stored in the primary memory device, the application will make no references to the secondary device. On the other hand, if the current working set of an application is larger than the primary device, the application (or at least the multilevel memory manager) will have to make at least some references to the secondary device, and it will therefore run more slowly. An application whose working set is much larger than the primary device is likely to cause repeated movement of data back and forth between the primary and secondary devices, a phenomenon called *thrashing*. A design goal is to avoid, or at least minimize, thrashing.

### 6.2.6 Multilevel Memory Management Policies

Equipped with the concepts of locality of reference and working set, we can now examine the behavior of some common multilevel memory management policies, algorithms that choose which stored objects to place in the primary device, which to place in the secondary device, and when to move a stored object from one device to the other. To make the discussion concrete, we will analyze multilevel memory management policies in the context of a virtual memory system with two levels: RAM (the primary device) and a magnetic disk (the secondary device), in which the stored objects are pages of uniform size. However, it is important to keep in mind that the

same analysis applies to any multilevel memory system, whether organized as a cache or a virtual memory, with uniform or variable-sized objects, and any variety of primary and secondary devices.

Each level of a multilevel memory system can be characterized by four items:

- *The string of references directed to that level.* In a virtual memory system, the reference string seen by the primary device is the sequence of page numbers extracted from virtual addresses of both instructions and data, in the order that the application makes references to them. The reference string seen by the secondary device is the sequence of page numbers that were misses in the primary device. The secondary device reference string is thus a shortened version of the primary device reference string.
- *The bring-in policy for that level.* In a virtual memory system, the usual bring-in policy for the primary device is *on-demand*: whenever a page is used, bring it to the primary device if it is not already there. The only remaining policy decision is whether or not to bring along some adjacent pages. In a two-level memory system there is no need for a bring-in policy for the secondary device.
- *The removal policy for that level.* In the primary device of a virtual memory system, this policy chooses a page to evict (the victim) to make room for a new page. Again, in a two-level memory system there is no need for a removal policy for the secondary device.
- *The capacity of the level.* In a virtual memory system, the capacity of the primary level is the number of primary memory blocks, and the capacity of the secondary level is the number of secondary memory blocks. Since the secondary memory normally contains a copy of every page, the capacity of the multilevel memory system is equal to the capacity of the secondary device.

The goal of a multilevel memory system is to have the primary device serve as many references in its reference string as possible, thereby minimizing the number of references in the secondary device reference string. In the example of the multilevel memory manager, this goal means to minimize the number of missing-page exceptions. One might expect that increasing the capacity of the primary device would guarantee a reduction (or at least not an increase) in the number of missing-page exceptions. Surprisingly, this expectation is not always true. As an example, consider the *first-in, first-out (FIFO) page-removal policy*, in which the page selected for removal is the one that has been in the primary device the longest. (That is, the first page that was brought in will be the first page to be removed. This policy is attractive because it is easy to implement by managing the pages of the primary device as a circular buffer.) If the reference string is 0 1 2 3 0 1 4 0 1 2 3 4, and the primary device starts empty, then a primary device with a capacity of three pages will experience nine missing-page exceptions, while a primary device with a capacity of four pages will experience ten missing-page exceptions, as shown in Tables 6.1 and 6.2:

**Table 6.1** FIFO Page-Removal Policy with a Three-Page Primary Device

| **Time** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Reference string** | **0** | **1** | **2** | **3** | **0** | **1** | **4** | **0** | **1** | **2** | **3** | **4** | |
| Primary device contents | – | 0 | 0 | 0 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 4 | Pages brought in |
| | – | – | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | |
| | – | – | – | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 3 | |
| Remove | – | – | – | 0 | 1 | 2 | 3 | – | – | 0 | 1 | – | |
| Bring in | 0 | 1 | 2 | 3 | 0 | 1 | 4 | – | – | 2 | 3 | – | 9 |

**Table 6.2** FIFO Page-Removal Policy with a Four-Page Primary Device

| **Time** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Reference string** | **0** | **1** | **2** | **3** | **0** | **1** | **4** | **0** | **1** | **2** | **3** | **4** | |
| Primary device contents | – | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 4 | 3 | Pages brought in |
| | – | – | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | |
| | – | – | – | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | |
| | – | – | – | – | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | |
| Remove | – | – | – | – | – | – | 0 | 1 | 2 | 3 | 4 | 0 | |
| Bring in | 0 | 1 | 2 | 3 | – | – | 4 | 0 | 1 | 2 | 3 | 4 | 10 |

This unexpected increase of missing-page exception numbers with a larger primary device capacity is called *Belady's anomaly*, named after the author of the paper that first reported it. Belady's anomaly is not commonly encountered in practice, but it suggests that when comparing page-removal policies, what appears to be a better policy might actually be worse with a different primary device capacity. As we shall see, one way to simplify analysis is to avoid policies that can exhibit Belady's anomaly.

The objective of a multilevel memory management policy is to select for removal the page that will minimize the number of missing-page exceptions in the future. If we knew the future reference string, we could look ahead to see which pages are about to be touched. The optimal policy would always choose for removal the page not needed for the longest time. Unfortunately, this policy is unrealizable because it requires predicting the future. However, if we run a program and keep track of its reference string, afterwards we can review that reference string to determine how many missing-page exceptions would have occurred if we had used that optimal policy. That result can then be compared with the policy that was actually used to determine how close it is to the optimal one. This unrealizable policy is known as the *optimal (OPT) page-removal policy*. Tables 6.3 and 6.4 show the result of the OPT page-removal policy applied to the same reference string as before.

**Table 6.3** The OPT Page-Removal Policy with a Three-Page Primary Device

| **Time** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Reference string** | **0** | **1** | **2** | **3** | **0** | **1** | **4** | **0** | **1** | **2** | **3** | **4** | |
| Primary device contents | – | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 3 | Pages brought in |
| | – | – | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| | – | – | – | 2 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 4 | |
| Remove | – | – | – | 2 | – | – | 3 | – | – | 0 | 2 | – | |
| Bring in | 0 | 1 | 2 | 3 | – | – | 4 | – | – | 2 | 3 | – | 7 |

**Table 6.4** The OPT Page-Removal Policy with a Four-Page Primary Device

| **Time** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Reference string** | **0** | **1** | **2** | **3** | **0** | **1** | **4** | **0** | **1** | **2** | **3** | **4** | |
| Primary device contents | – | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | Pages brought in |
| | – | – | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| | – | – | – | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | |
| | – | – | – | – | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 4 | |
| Remove | – | – | – | – | – | – | 3 | – | – | – | 0 | – | |
| Bring in | 0 | 1 | 2 | 3 | – | – | 4 | – | – | – | 3 | – | 6 |

It is apparent from the number of pages brought in that, at least for this reference string, the OPT policy is better than FIFO. In addition, at least for this reference string, the OPT policy gets better when the primary device capacity is larger.

The design goal thus becomes to devise page-removal algorithms that (1) avoid Belady's anomaly, (2) have hit ratios not much worse than the optimal policy, and (3) are mechanically easy to implement.

Some easy-to-implement page-removal policies have an average performance on a wide class of applications that is close enough to the optimal policy to be effective. A popular one is the *least-recently-used (LRU) page-removal policy.* LRU is based on the observation that, more often than not, the recent past is a fairly good predictor of the immediate future. The LRU prediction is that the longer the time since a page has been used, the less likely it will be needed again soon. So LRU selects as its victim the page in the primary device that has not been used for the longest time (that is, the "least-recently-used" page). Let's see how LRU fares when it tackles our example reference string:

**Table 6.5** The LRU Page-Removal Policy with a Three-Page Primary Device

| **Time** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Reference string** | **0** | **1** | **2** | **3** | **0** | **1** | **4** | **0** | **1** | **2** | **3** | **4** | |
| Primary device contents | – | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | Pages brought in |
| | – | – | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| | – | – | – | 2 | 3 | 3 | 3 | 4 | 4 | 4 | 2 | 2 | |
| Remove | – | – | – | 1 | – | 2 | 3 | – | – | 4 | 0 | 1 | |
| Bring in | 0 | 1 | 2 | 3 | – | 1 | 4 | – | – | 2 | 3 | 4 | 9 |

**Table 6.6** The LRU Page-Removal Policy with a Four-Page Primary Device

| **Time** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Reference string** | **0** | **1** | **2** | **3** | **0** | **1** | **4** | **0** | **1** | **2** | **3** | **4** | |
| Primary device contents | – | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Pages brought in |
| | – | – | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| | – | – | – | 2 | 2 | 2 | 2 | 4 | 2 | 2 | 2 | 2 | |
| | – | – | – | – | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 3 | |
| Remove | – | – | – | – | – | – | 2 | – | – | – | 4 | 0 | |
| Bring in | 0 | 1 | 2 | 3 | – | – | 4 | – | – | – | 3 | 4 | 8 |

For this reference string, LRU is better than FIFO for a primary memory device of size 4 but not as good as the OPT policy. And for both LRU and the OPT policy the number of page movements is monotonically non-decreasing with primary device size; these two algorithms avoid Belady's anomaly, for a non-obvious reason that will be explained in Section 6.2.7.

Most useful algorithms require that the new page be the only page that moves in and that only one page move out. Algorithms that have this property are called *demand algorithms.* FIFO, LRU, and some algorithms that implement the OPT policy are demand algorithms. If any other page moves in to primary memory, the algorithm is said to use *prepaging*, one of the topics of Section 6.2.9.

As seen above, LRU is not as good as the OPT policy. Because it looks at history rather than the future, it sometimes throws out exactly the wrong page (the page movement at reference #11 in the four-page memory provides an example). For a more extreme example, a program that runs from top to bottom through a virtual memory that is larger than the primary device will always evict exactly the wrong page. Consider a primary device with capacity of four pages that is part of a virtual

memory that contains five pages being managed with LRU (the letter "F" means that this reference causes a missing-page exception):

| **Reference string** | **0** | **1** | **2** | **3** | **4** | **0** | **1** | **2** | **3** | **4** | **0** | **1** | **2** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4-page primary device | F | F | F | F | F | F | F | F | F | F | F | F | F |

If the application repeatedly cycles through the virtual memory from one end to the other, each reference to a page will result in a page movement. If we start with an empty primary device, references to page 0 through 3 will result in page movements. The reference to page 4 will also result in a page movement, in which LRU will remove page 0, since page 0 has been used least recently. The next reference, to page 0, will also result in a page movement, which leads LRU to remove page 1, since it has been used least recently. As a consequence, the next reference, to page 1, will result in a page movement, replacing page 2, and so on. In short, every access to a page will result in a page movement.

For such an application, a *most-recently-used (MRU) page-removal policy* would be better. MRU chooses as the victim the most recently used page.

Let's see how MRU fares on the contrived example that gave LRU so much trouble:

| **Reference string** | **0** | **1** | **2** | **3** | **4** | **0** | **1** | **2** | **3** | **4** | **0** | **1** | **2** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4-page primary memory | F | F | F | F | F | | | | F | | | | F |

The initial references to pages 0 through 3 result in page movements that fill the empty primary device. The first reference to page 4 will also result in a page movement, replacing page 3, since page 3 has been used most recently. The next reference, to page 0, will *not* result in a missing-page exception since page 0 is still in the primary device. Similarly, the succeeding references to page 1 and 2 will not result in page movements. The second reference to page 3 will result in a page movement, replacing page 2, but then there will be three references that do not require page movements. Thus, with the MRU page-removal policy, our contrived application will experience fewer missing-page exceptions than with the LRU page-removal policy: once in steady state, MRU will result in one page movement per loop iteration.

In practice, however, LRU is surprisingly robust because past references frequently are a reasonable predictor of future references; examples in which MRU does better are uncommon. A secondary reason why LRU works well is that programmers assume that the multilevel memory system uses LRU or some close approximation as the removal policy and they design their programs to work well under that policy.

### 6.2.7 Comparative Analysis of Different Policies

Once an overall system architecture that includes a multilevel memory system has been laid out, the designer needs to decide two things that will affect performance:

- How large the primary memory device should be
- Which page removal policy to use

These two decisions can be—and in practice often are—supported by an analysis that begins by instrumenting a hardware processor or an emulator of a processor to maintain a trace of the reference string of a running program. After collecting several such traces of typical programs that are to be run on the system under design, these traces can then be used to simulate the operation of a multilevel memory with various primary device sizes and page-removal policies. The usual measure of a multilevel memory's performance is the hit ratio because it is a pure number whose value depends only on the size of the primary device and the page-removal policy. Given the hit ratio and the latency of the primary and secondary memory devices, one can immediately estimate the performance of the multilevel memory system by using Equation 6.2.

In the early 1970s, a team of researchers at the IBM Corporation developed a rapid way of doing such simulations to calculate hit ratios for one class of page-removal policies. If we look more carefully at the "primary device contents" rows of Tables 6.3 and 6.4, we notice that at all times the optimal policy keeps in the three-page memory a subset of the pages that it keeps in the four-page memory. But in FIFO Tables 6.1 and 6.2, at times 8, 9, 11, and 12, this *subset property* does not hold. This difference is no accident; it is the key to understanding how to avoid Belady's anomaly and how to rapidly analyze a reference string to see how a particular policy will perform for any primary device size.

If a page-removal policy can somehow maintain this subset property at all times and for every possible primary device capacity, then a larger primary device can never have more missing-page exceptions than a smaller one. Moreover, if we consider a primary device of capacity $n$ pages and a primary device of capacity $n + 1$ pages, the subset property ensures that the larger primary device contains exactly one page that is not in the smaller primary device. Repeating this argument for every possible primary device size $n$, we see that the subset property creates a total ordering of all the pages of the multilevel memory system. For example, suppose a memory of size 1 contains page $A$. A memory of size 2 must also contain page $A$, plus one other page, perhaps $B$. A memory of size 3 must then contain pages $A$ and $B$ plus one other page, perhaps $C$. Thus, the subset property creates the total ordering $\{A, B, C\}$. This total ordering is independent of the actual capacity chosen for the primary memory device.

The IBM research team called this ordering a "stack" (in a use of that word that has no connection with push-down stacks), and page-removal policies that maintain the subset property have since become known as *stack algorithms*. Although requiring the subset property constrains the range of algorithms, there are still several different, interesting, and practical algorithms in the class. In particular, the OPT policy, LRU, and MRU all turn out to be stack algorithms. When a stack algorithm is in use, the virtual memory system keeps just the pages from the front of the ordering in the primary device; it relegates the remaining pages to the secondary device. As a consequence, if $m < n$, the set of pages in a primary device of capacity $m$ is always a subset of the set of pages in a primary device of capacity $n$. Thus a larger memory will always be able to satisfy all of the requests that a smaller memory could—and with luck some additional requests. Put another way, the total ordering ensures that if a particular reference hits in a primary memory of size $n$, it will also hit in every memory larger than $n$. When a

**Table 6.7** Simulation of the LRU Page-Removal Policy for Several Primary Device Sizes

| **Time** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Reference string** | **0** | **1** | **2** | **3** | **0** | **1** | **4** | **0** | **1** | **2** | **3** | **4** | |
| Stack contents after reference | 0<br>–<br>–<br>–<br>– | 1<br>0<br>–<br>–<br>– | 2<br>1<br>0<br>–<br>– | 3<br>2<br>1<br>0<br>– | 0<br>3<br>2<br>1<br>– | 1<br>0<br>3<br>2<br>– | 4<br>1<br>0<br>3<br>2 | 0<br>4<br>1<br>3<br>2 | 1<br>0<br>4<br>3<br>2 | 2<br>1<br>0<br>4<br>3 | 3<br>2<br>1<br>0<br>4 | 4<br>3<br>2<br>1<br>0 | Number of moves in |
| Size 1 in/out | 0/– | 1/0 | 2/1 | 3/2 | 0/3 | 1/0 | 4/1 | 0/4 | 1/0 | 2/1 | 3/2 | 4/3 | 12 |
| Size 2 in/out | 0/– | 1/– | 2/0 | 3/1 | 0/2 | 1/3 | 4/0 | 0/1 | 1/4 | 2/0 | 3/1 | 4/2 | 12 |
| Size 3 in/out | 0/– | 1/– | 2/– | 3/0 | 0/1 | 1/2 | 4/3 | –/– | –/– | 2/4 | 3/0 | 4/1 | 10 |
| Size 4 in/out | 0/– | 1/– | 2/– | 3/– | –/– | –/– | 4/2 | –/– | –/– | 2/3 | 3/4 | 4/0 | 8 |
| Size 5 in/out | 0/– | 1/– | 2/– | 3/– | –/– | –/– | 4/- | –/– | –/– | –/– | –/– | –/– | 5 |

stack algorithm is in use, the hit ratio in the primary device is thus guaranteed to be a non-decreasing function of increasing capacity. Belady's anomaly cannot arise.

The more interesting feature of the total ordering and the subset property is that for a given page-removal policy an analyst can perform a simulation of all possible primary memory sizes, with a single pass through a given reference string, by computing the total ordering associated with that policy. At each reference, some page moves to the top of the ordering, and the pages that were above it either move down or stay in their same place, as dictated by the page-removal policy. The simulation notes, for each primary memory device size of interest, whether or not these movements within the total ordering also correspond to movements between the primary and secondary memory devices. By counting those movements, when it reaches the end of the reference string the simulation can directly calculate the hit ratio for each potential primary memory size. Table 6.7 shows the result of this kind of simulation for the LRU policy when it runs with the reference string used in the previous examples. In this table, the "size *n* in/out" rows indicate which pages, if any, the LRU policy will choose to bring into and remove from primary memory in order to satisfy the reference above. Note that at every instant of time, the "stack contents after reference" are in order by time since last usage, which is exactly what intuition predicts for the LRU policy.

In contrast, when analyzing a non-stack algorithm such as FIFO, one would have to perform a complete simulation of the reference string for each different primary device capacity of interest and construct a separate table such as the one above for each memory size. It is instructive to try to create a similar table for FIFO.

In addition, since the reference string is available, its future is known, and the analyst can, with another simulation pass (running backward through the reference string), learn how the optimal page-removal policy would have performed on that same string for every memory size of interest. The analyst can then compare the OPT result with various realizable page-removal candidate policies.

**Table 6.8** The Optimal Page-Removal Policy for All Primary Memory Sizes

| **Time** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Reference string** | **0** | **1** | **2** | **3** | **0** | **1** | **4** | **0** | **1** | **2** | **3** | **4** | |
| Stack contents after reference | 0 | 1 | 2 | 3 | 0 | 1 | 4 | 0 | 1 | 2 | 3 | 4 | Number of pages removed |
| | – | 0 | 0 | 0 | 3 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | |
| | – | – | 1 | 1 | 1 | 3 | 1 | 1 | 4 | 4 | 4 | 3 | |
| | – | – | – | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 2 | 2 | |
| | – | – | – | – | – | – | 2 | 2 | 2 | 1 | 1 | 1 | |
| Size 1 victim | – | 0 | 1 | 2 | 3 | 0 | 1 | 4 | 0 | 1 | 2 | 3 | 11 |
| Size 2 victim | – | – | 1 | 2 | – | 3 | 1 | – | 1 | 2 | 3 | 4 | 10 |
| Size 3 victim | – | – | – | 2 | – | – | 4 | – | – | 2 | 3 | – | 7 |
| Size 4 victim | – | – | – | – | – | – | 4 | – | – | 2 | – | – | 6 |
| Size 5 victim | – | – | – | – | – | – | – | – | – | – | – | – | 5 |

**Sidebar 6.6 OPT is a Stack Algorithm and Optimal** To see that OPT is a stack algorithm, consider the following description of OPT, in terms of a total ordering:

1. Start with an empty primary device and an empty set that will become a total ordering. As each successive page is touched, note its depth $d$ in the total ordering (if it is not yet in the ordering, set $d$ to infinity) and move it to the front of the total ordering.
2. Then, move the page that was at the front down in the ordering. Move it down until it follows all pages already in the ordering that will be touched before this page is needed again, or to depth $d$, whichever comes first. This step requires knowing the future.
3. If $d > m$ (where $m$ is the size of the primary memory device), step 1 will require moving a page from the secondary device to the primary device, and step 2 will require moving a page from the primary device to the secondary device.

The result is that, if the algorithm removes a page from primary memory, it will always choose the page that will not be needed for the longest time in the future. Since the total ordering of all pages is independent of the capacity of the primary device, OPT is a stack algorithm. Therefore, for a particular reference string, the set of pages in a primary device of capacity $m$ is always a subset of the set of pages in a primary device of capacity $m + 1$. Table 6.8 illustrates this subset property.

Proof that the optimal page removal policy minimizes page movements, and that it can be implemented as an on-demand stack algorithm, is non-trivial. Table 6.8 illustrates that the statement is correct for the reference string of the previous examples. Sidebar 6.6 provides the intuition of why OPT is a stack algorithm and optimal. The

interested reader can find a detailed line of reasoning in the 1970 paper by the IBM researchers [Suggestions for Further Reading 6.1.2] who introduced stack algorithms and explained in depth how to use them in simulations.

### 6.2.8 Other Page-Removal Algorithms

Any algorithm based on the LRU policy requires updating recency-of-usage information on every memory reference, whether or not a page moves between the primary and secondary devices. For example, in a virtual memory system every instruction and data reference of the running program causes such an update. But manipulating the representation of this usage information may itself require several memory references, which escalates the cost of the original reference. For this reason, most multilevel memory designers look for algorithms that have approximately the same effect but are less costly to implement. One elegant approximation to LRU is the *clock page-removal algorithm.*

The clock algorithm is based on a modest hardware extension in which the virtual memory manager (implemented in the hardware of the processor) sets to TRUE a bit, called the *referenced* bit, in the page table entry for a page whenever the processor makes a reference that uses that page table entry. If at some point in time the multilevel memory manager clears the referenced bit for every page to FALSE, and then the application runs for a while, a survey of the referenced bits will reveal which pages that application used. The clock algorithm consists of a systematic survey of the referenced bits.

Suppose the physical block numbers of the primary device are arranged in numerical order in a ring (i.e., the highest block number is followed by block number 0), as illustrated in Figure 6.9. All referenced bits are initially set to FALSE, and the system begins running. A little later, in Figure 6.9, we find that the pages residing in blocks 0, 1, 2, 4, 6, and 7 have their referenced bits set to TRUE, indicating that some program touched them. Then, some program causes a missing-page exception, and the system invokes the clock algorithm to decide which resident page to evict in order to make room for the missing page. The clock algorithm maintains a pointer much like a clock arm (which is why it is called the clock algorithm). When the virtual memory system needs a free page, the algorithm begins moving the pointer clockwise, surveying the referenced bits as it goes:

**FIGURE 6.9**

Example operation of the clock page-removal policy.

1. If the clock arm comes to a block for which the referenced bit is TRUE, the algorithm sets the referenced bit to FALSE and moves the arm ahead to the next block. Thus,

the meaning of the referenced bit becomes "The processor has touched the page residing in this block since the last pass of the clock arm."

2. If the clock arm comes to a block for which the referenced bit is FALSE, that means that the page residing in this block has not been touched since the last pass of the clock arm. This page is thus a good candidate for removal, since it has been used less recently than any page that has its referenced bit set to TRUE. The algorithm chooses this page for eviction and leaves the arm pointing to this block for the next execution of the algorithm.

The clock algorithm thus removes the page residing in the first block that it encounters that has a FALSE referenced bit. If there are no such pages (that is, every block in the primary device has been touched since the previous pass of the clock arm), the clock will move all the way around once, resetting referenced bits as it goes, but at the end of that round it will come again to the first block it examined, which now has a FALSE referenced bit, so it chooses the page in that block. If the clock algorithm were run starting in the state depicted in Figure 6.9, it would choose to remove the page in block 3, since that is the first block in the clockwise direction that has a FALSE referenced bit.

The clock algorithm has a number of nice properties. Space overhead is small: just one extra bit per block of the primary device. The extra time spent per page reference is small: forcing a single bit to TRUE. Typically, the clock algorithm has to scan only a small fraction of the primary device blocks to find a page with a FALSE referenced bit. Finally, the algorithm can be run incrementally and speculatively. For example, if the designer of the virtual memory system wants to keep the number of free blocks above some threshold, it can run the policy ahead of demand, removing pages that haven't been used recently, and stop moving the arm as soon as it has met the threshold.

The clock algorithm provides only a rough approximation to LRU. Rather than strictly determining which page has been used least recently, it simply divides pages into two categories: (1) those used since the last sweep and (2) those not used since the last sweep. It then chooses as its victim the first page that the arm happens to encounter in the second category. This page has been used less recently than any of the pages in the first category, but is probably not the least-recently-used page. What seems like the worst-case scenario for the clock algorithm would be when all pages have their referenced bit set to TRUE; then the clock algorithm has no information on which to decide which pages have recently been used. On the other hand, if every page in the primary device has been used since the last sweep, there probably isn't a much better way of choosing a page to remove.

In multilevel memory systems that are completely implemented in hardware, even the clock algorithm may involve too much complexity, so designers resort to yet simpler policies. For example, some processors use a *random removal policy* for the translation look-aside buffer described in Chapter 5. Random removal can be quite effective in this application because

- its implementation requires minimal state to implement.
- if the look-aside buffer is large enough to hold the current working set of translations, the chance that a randomly chosen victim turns out to be a translation that is about to be needed is relatively small.
- the penalty for removing the wrong translation is also quite small—just one extra reference to a slightly slower random access memory.

Alternatively, some processor cache managers use a completely stateless policy called *direct mapping* in which the page chosen for eviction is the one located in block $n$ modulo $m$, where $n$ is the secondary device block number of the missing page and $m$ is the number of blocks in the primary device. If the compiler optimizer is aware that the processor uses a direct mapping policy, and it knows the size of the primary device, it can minimize the number of cache misses by carefully positioning instructions and data in the secondary device.

When page-removal policies are implemented in software, designers can use methods that maintain more state. One popular software policy is least-frequently-used, which tracks how often a page is used. Complete coverage of page-removal polices is beyond the scope of this book. The reader is encouraged to explore the large literature on this topic.

### 6.2.9 Other Aspects of Multilevel Memory Management

Page-removal policies are only one aspect of multilevel memory management. The designer of a multilevel memory manager must also provide a bring-in policy that is appropriate for the system load and, for some systems, may include measures to counter thrashing.

The bring-in policy of all of the paging systems described so far is that pages are moved to the primary device only when the application attempts to use them; such systems are called *demand paging* systems. The alternative method is known as *prepaging*. In a prepaging system, the multilevel memory manager makes a prediction about which pages might be needed and brings them in before the application demands them. By moving pages that are likely to be used before they are actually requested, the multilevel memory manager may be able to satisfy a future reference immediately instead of having to wait for the page to be retrieved from a slower memory. For example, when someone launches a new application or restarts one that hasn't been used for a while, none of its pages may be in the primary device. To avoid the delay that would occur from bringing in a large number of pages one at a time, the multilevel memory manager might choose to prepage as a single batch all of the pages that constitute the program text of the application, or all of the data pages that the application used on a previous execution.

Both demand paging and prepaging make use of speculation to improve performance. Demand paging speculates that the application will touch other bytes on the page just brought in. Prepaging speculates that the application will use the prepaged pages.

A problem that arises in a multiple-application system is that the working sets of the various applications may not all simultaneously fit in the primary device. When

that is case, the multilevel memory manager may have to resort to more drastic measures to avoid thrashing. One such drastic measure is *swapping*. When an application encounters a long wait, the multilevel memory manager moves all of its pages out of the primary device in a batch. A batch of writes to the disk can usually be scheduled to go faster than a series of single-block writes (Section 6.3.4 discusses this opportunity). In addition, swapping an application completely out immediately provides space for the other applications, so when they encounter a missing-page exception there is no need to wait to move some page out. However, to do swapping, the multilevel memory manager must be able to quickly identify which pages in primary memory are being used by the application being swapped out, and which of those pages are shared with other applications and therefore should not be swapped out.

Swapping is usually combined with prepaging. When a swapped-out application is restarted, the multilevel memory manager prepages the previous working set of that application, in the hope of later reducing the number of missing-page exceptions. This strategy speculates that when the program restarts, it will need the same pages that it was using before it was swapped out.

The trade-offs involved in swapping and prepaging are formidable, and they resist modeling analysis because reasonably accurate models of application program behavior are difficult to obtain. Fortunately, technology improvements have made these techniques less important for a large class of systems. However, they continue to be applicable to specialized systems that require the utmost in performance.

## 6.3 SCHEDULING

When a stage is temporarily overloaded in Figure 6.3, a queue of requests builds up. An important policy decision is to determine which requests from the queue to perform first. For example, if the disk has a queue of disk requests, in which order should the disk manager schedule them to minimize latency? For another example, should a stage schedule requests in the order they are received? That policy may result in high throughput, but perhaps in high average latency for individual requests because one client's expensive request may delay several inexpensive requests from other clients. These questions are all examples of the general question of how to schedule resources. This section provides an introduction to systematic answers to this general question. This introduction is sufficient to tackle resource scheduling problems that we encounter in later chapters but scratches only the surface of the literature on scheduling.

Because the technology underlying resources improves rapidly in computer systems, some scheduling decisions become irrelevant over time. For example, in the 1960s and 1970s when several users shared a single computer and the processor was a performance bottleneck, scheduling the processor among users was important. With the arrival of personal computers and the increase in processing power, processor scheduling became mostly irrelevant because it is no longer a performance bottleneck in most situations, and any reasonable policy is good enough. On the other hand, with massive Internet services handling millions of paying customers, the issue of scheduling has increased in importance. The Internet exposes Web sites to extreme

variations in load, which can result in more requests than a server can handle at an instant of time, and the service must make a choice in which order to handle the queued requests.

### 6.3.1 Scheduling Resources

Computer systems make scheduling decisions at different levels of abstraction. At a high level of abstraction, a Web site selling goods might allocate more memory and processor time to a user who always buys goods than to a user who never buys goods but just browses the catalog. At a lower level of abstraction, a bus arbiter must decide to which processor's memory reference to allocate a shared bus.

Although in these examples allocation decisions are made at different levels of abstraction, the scheduling problem is similar. From the perspective of scheduling, a computer system is a collection of entities that require the use of a set of resources, and *scheduling* is the set of policies and dispatch mechanisms to allocate resources to entities. Examples of entities include threads, address spaces, users, clients, services, and requests. Examples of resources include processor time, physical memory space, disk space, network capacity, and I/O-bus time. Policies to assign resources to entities include dividing the resources equally among the entities, giving one entity priority over another entity, and providing some minimum guarantee by performing admission control on the number of entities. The *scheduler* is the component that implements a policy.

Designing the right policy is difficult because there are usually gaps between the high-level goal and the available policy, between the chosen policy and mechanism to dispatch, and between the chosen mechanism and its actual implementation. We discuss each of these challenges in turn.

The desired scheduling policy might incorporate elements of the environment in which the computer system is used but that are difficult to capture in a computer system. For example, how can a Web site identify a high-value customer (that is, one who is likely to make a large purchase)? The high-value user might never have bought before at this site, or it may be difficult to associate an anonymous catalog-browsing request with a particular previous customer. Even if we could identify the request with a particular customer, the request may traverse several modules of the Web site, some of which may have no notion of users. For example, the database that contains information about prices and goods might be unable to prioritize requests from an important customer.

If we can construct the right policy, then there is the challenge of identifying the mechanism to implement the policy. One module might implement a scheduling policy, but because another module is not aware of it, the policy is ineffective. For example, we might desire to give the text editor high priority to provide a good interactive experience to users. We can easily change the thread scheduler to give the thread running the editor higher priority than any other runnable thread. However, how does the bus arbiter, shared file service, or disk scheduler know that a memory, file, or disk request on behalf of the editor should have higher priority than other disk or memory requests? Worse, the disk scheduler is likely to delay operations to batch

disk requests to achieve high throughput, but this decision may result in bad interactive performance for the text editor because its requests are delayed.

The final challenge is getting the actual implementation of the mechanism right. Sidebar 6.7 on receive livelock provides an example of how easy it is for two schedulers to interact badly. It illustrates that to design a computer system that doesn't collapse under overload is a challenge and requires that a designer carefully think through all implementation decisions.

The list of challenges in designing and implementing schedulers is formidable, but fortunately sophisticated schedulers are often not a requirement for computer systems. Airlines use sophisticated and complex scheduling algorithms because they deal with genuinely expensive and scarce resources (such as airplanes, landing slots, and fuel) and situations in which the peak load can be far larger than usual load (e.g., travel around family holidays). Usually, in a computer system few resources are truly scarce, and simple policies, mechanisms, and implementations suffice.

The rest of Section 6.3 introduces some common goals for a scheduler in a computer system, describes some basic policies to achieve these goals, and presents a case study of scheduling a disk arm. Along the way, the section points out a few scheduling pitfalls, such as receive livelock and priority inversion.

### 6.3.2 Scheduling Metrics

To appreciate possible goals for a scheduler, consider the thread scheduler from the previous chapter. It chooses a thread from a set of runnable threads. In the implementation of the thread manager in Figure 5.24, the scheduler picks the threads in the order in which they appear in the thread table. This scheduling policy is one of many possible policies.

By slightly restructuring the thread scheduler, it could implement different policies easily. A more general implementation of the thread manager would follow the design hint *separate mechanism from policy* (see Sidebar 6.5). This implementation would separate the dispatch mechanism (the mechanisms for suspending and resuming a thread) from scheduling policy (selecting which thread to run next) by putting them into their own procedures, so that a designer can change the policy without having to change the dispatch mechanism.

A designer may want to change the policy because there is no one single best scheduling policy. "Best" might mean different things in different situations. For example, there is tension between achieving good overall efficiency and providing good service to individual requests. From the system's perspective, the two important measures for "best" are throughput and utilization. With a good scheduler, throughput grows linearly with offered load until throughput hits the capacity of the system. A good scheduler will also ensure that a system doesn't collapse under overload conditions. Finally, a good scheduler is efficient: it doesn't consume many resources itself. A scheduler that needs 90% of the processor's time to do its job is not of much value.

Applications achieve high throughput by being immediately scheduled when a request arrives and processing it to completion, without being rescheduled.

**Sidebar 6.7 Receive Livelock** When a system is temporarily overloaded, it is important to have an effective response to the overload situation. The response doesn't have to be perfect, but it must ensure that the system doesn't collapse. For example, suppose that a Web news server can handle 1,000 requests per second, but a short time ago there was a big earthquake in Los Angeles and requests are arriving at the rate 10,000 per second. The goal is to successfully serve (perhaps a random) 10% of the load, but if the designer isn't careful, the server may end up serving 0%. The problem is called *receive livelock*, and it can arise if the server spends too much of its time saying "I'm too busy" and as a result never gets a chance to serve any of the requests. Consider the following simple interrupt-driven Web service with a bounded buffer:

When a request arrives on the network device, the device generates an interrupt, which causes the interrupt handler to run. The interrupt handler copies the request from the device into a bounded buffer and reenables interrupts so that it can receive the next request. The service has a single thread, which consumes requests from the bounded buffer. When the service is overloaded and requests arrive faster than the service can process them, then the system as described reaches a state where it serves no requests at all because it experiences receive livelock.

Consider what happens when requests arrive much faster than the service can process them. While the service thread is processing a request, the processor receives an interrupt from the network device and the interrupt handler runs. The interrupt handler copies the request into the buffer, notifies the service thread, and returns, reenabling interrupts. As soon as the handler reenables interrupts, the arrival of another request may interrupt the processor again, invoking the interrupt handler. The interrupt handler goes through the same sequence as before until the buffer fills up; then it has no other choice than to discard the request and return from the interrupt, reenabling interrupts. If the network device has another request available, it will interrupt the processor immediately again; the interrupt handler will throw the request away and return. This sequence of events continues indefinitely as long as requests arrive faster than the time for the interrupt handler to run. We have receive livelock: the service never runs, and as a result the number of requests processed by the service per second drops to zero; to users the Web site appears to be down!

The problem here is that the processor's internal scheduler interacts badly with the thread scheduler. Conceptually, the processor schedules the main thread and the

(*Sidebar continues*)

interrupt thread, and the thread manager schedules the main processor thread among the service thread and any other threads. The processor scheduler gives absolute priority to the interrupt thread, scheduling it as soon as an interrupt arrives; the main thread never gets a chance to run the thread manager, and as a result the service thread never receives the processor. This problem occurs when some processing must be performed *outside* of the interrupt handler. One could contemplate moving all processing into interrupt handlers. This approach has its own problems (as discussed in Section 5.6.4) and negates the modularity advantages of using threads. However, once the problem is stated as a scheduling problem, a solution is available.

The solution [Suggestions for Further Reading 6.4.2] is to modify the scheduling policy so that the service thread gets a chance to run when requests are available in the bounded buffer. This policy can be implemented with a slight modification to the interrupt handler. If the bounded buffer fills up, the interrupt handler should not reenable interrupts as it returns. When the service thread has drained the bounded buffer, say, to only half full, it should reenable interrupts. This policy ensures that the network device doesn't discard requests unless the buffer is full (i.e., there is an overload situation) and sees that the service thread gets a chance to process requests, avoiding livelock.

It is still possible that requests may be discarded. If the network device receives a request but it cannot generate an interrupt, the device has no other choice than to discard the next request. This situation is unavoidable: if the network can generate a higher load than the capacity of the service, the device must shed load. The good news is that under overload the system will at least process some requests rather than none at all.

For example, any time a thread scheduler starts a thread, but then preempts it to run another thread, it is delaying the preempted thread. Thus, for an application to achieve high throughput, a scheduler must minimize the number of preemptions and the number of scheduling decisions. Unfortunately, this system-level goal may conflict with the needs of individual threads.

Each individual request wants good service, which typically means good response: it starts soon and completes quickly. There are several ways of measuring a request's response:

- *Turnaround time*. The length of time from when a request arrives at a service until it completes.
- *Response time*. The length of time from when a request arrives at a service until it starts producing output. For interactive requests, this measure is typically more useful than turnaround time. For example, many Web browsers optimize for this metric. Typically, a browser displays an incomplete Web page as soon as the browser receives parts of it (e.g., the text) and fills in the remainder later (e.g., images).

- *Waiting time*. The length of time from when a request arrives at a service until the service starts processing the request. This measure is better than turnaround time, since it captures how long the thread must wait even though it is ready to execute. The ideal waiting time is zero seconds.

More sophisticated measures are also possible by combining the performance of all requests using some of these measures and some way of combining. For example, one can compute *average waiting time* as the average of waiting times of all requests. Similarly, one can calculate the sum of the waiting times, the variance in response time, and so on.

In an interactive computer system, many requests are on behalf of a human user sitting in front of a display. Therefore, the perception of the user is another measure of the goodness of the service that a request receives. For example, an interactive user may tend to perceive a high variance in response time to be more annoying than a high mean. On the other hand, a response time that is faster than the human reaction time may not improve the perception of goodness.

Sometimes a designer desires a scheduler that provides some degree of *fairness*, which means that each request obtains an equal share of the shared service. A scheduler that starves a request to serve other requests is an unfair scheduler. An unfair scheduler is not necessarily a bad scheduler; it may have higher throughput and better response time than a fair scheduler.

It is easy to convince oneself that designing a scheduler that optimizes for fairness, throughput, and response time all at the same time is an impossible task. As a result, there are many different scheduling algorithms; each one of them optimizes along different dimensions.

### 6.3.3 Scheduling Policies

To illustrate some basic scheduling algorithms, we present a number of them in the context of a thread manager. The objective is to share the processor efficiently among multiple threads. For example, when one thread is blocked waiting for I/O, we would like to run a different, runnable thread on the processor. These threads might be running different programs on a shared computer, or a number of threads that cooperate to implement a high-performance Web service on a dedicated computer.

Since threads typically go through a cycle of running and waiting (e.g., waiting for user input, a client request, or completion of disk request), it is useful to model a thread as a series of *jobs*. Each job corresponds to one burst of activity.

We survey a few different algorithms to schedule these jobs. Many textbooks, lecture notes, and papers explore these algorithms in more detail, and our description is based on this literature. Although the algorithms are described in the context of a thread manager for a single processor, the algorithms are generic and apply to other contexts as well. For example, they work equally well for multiprocessors and have the same pros and cons, but they are harder to illustrate when several jobs run concurrently. The algorithms also apply to disk-arm scheduling, which we shall discuss in Section 6.3.4.

#### 6.3.3.1 First-Come, First-Served

At a busy post office, customers may be asked to take a ticket with a number as they walk in and wait until the number on their ticket is called. Typically, the post office allocates the numbers in strict increasing order and calls the numbers in that order. This policy is called a *first-come, first-served (FCFS) scheduler* and some thread managers use it too.

A thread manager can implement the first-come, first-served policy by organizing the ready list as a first-in, first-out queue. The manager simply runs the first job on the queue until it finishes; then the manager runs the next job, which is now the first job, and so on. When a job becomes ready, the scheduler simply adds it to the end of the queue.

To illustrate and analyze the behavior of a scheduling policy, the literature uses sequences of job arrivals, in which each job has a specific amount of work. We adopt one particular sequence, which illustrates the differences between the scheduling algorithms that we cover. This sequence is the following:

| Job | Arrival Time | Amount of Work |
|---|---|---|
| A | 0 | 3 |
| B | 1 | 5 |
| C | 3 | 2 |

Given a specific sequence, one can draw a timeline that depicts when the thread manager dispatches jobs. For the above sequence and the first-come, first-served policy this timeline is as follows:

Given this timeline, one can fill out a table that includes finish time and waiting times, and make some observations about a policy. For the above timeline and the first-come, first-served policy this table is as follows:

| Job | Arrival Time | Amount of Work | Start Time | Finish Time | Waiting Time Till Job Starts | Wait Time Till Job is Done |
|---|---|---|---|---|---|---|
| A | 0 | 3 | 0 | 3 | 0 | 3 |
| B | 1 | 5 | 3 | 8 | 2 | 7 |
| C | 3 | 2 | 8 | 10 | 5 | 7 |
| | **Total waiting** | | | | 7 | |

From the table we can see that for the given job sequence, the first-come, first-served policy favors the long jobs A and B. Job C waits 5 seconds to start a job that takes 2 seconds. Relative to the amount of work, job C is punished the most.

Because first-come, first-served can favor long jobs over short jobs, a system can get into an undesirable state. Consider what happens if we have a system with one thread that periodically waits for I/O but mostly computes and several threads that perform mostly I/O operations. Suppose the scheduler runs the I/O-bound threads first. They will all quickly finish their jobs and go start their I/O operations, leaving the scheduler to run the processor-bound thread. After a while, the I/O-bound threads will finish their I/O and queue up behind the processor-bound thread, leaving all the I/O devices idle. When the processor-bound thread finishes its job, it initiates its I/O operation, allowing the scheduler to run the I/O-bound threads.

As before, the I/O-bound threads will quickly finish computation and initiate an I/O operation. Now we have the processor sitting idle, while all the threads are waiting for their I/O operations to complete. Since the processor-bound thread started its I/O first, it will likely finish first, grabbing the processor and making all the other threads wait before they can run. The system will continue this way, alternating between periods when the processor is busy and all the I/O devices are idle with periods when the processor is idle and all the threads are doing I/O in a convoy, which is why the literature sometimes refers to this case as a *convoy effect*. The main opportunity for having threads is missed, since in this convoy scenario the system never overlaps computation with I/O.

This scenario is unlikely to materialize in practice because workloads are unlikely to have exactly the right mix of computing and I/O threads that would produce a sequence of scheduling decisions that lead to a situation where I/O isn't overlapped at all with computation. Nevertheless, it has inspired researchers to think about policies other than first-come, first-served.

#### *6.3.3.2 Shortest-Job-First*

The undesirable scenario with the first-come first-served policy suggests another scheduler: a *shortest-job-first scheduler*. Whenever the time comes to dispatch a job, the scheduler chooses the job that has the shortest expected running time. Shortest-job-first requires that the scheduler has a prediction of the running time of a job before running it. In the general case, it is difficult to make predictions of the running time of a job, but in practice there are special cases that can work.

Let's assume we know the running time of a job beforehand and see how a shortest-job-first scheduler performs on the example sequence:

As we can see, job C runs before job B because when the scheduler runs after job A completes, it picks C instead of B, since job C has just entered the system and needs less time than job B. Here is the complete table for the shortest-job-first policy:

| Job | Arrival Time | Amount of Work | Start Time | Finish Time | Waiting Time Till Job Starts | Waiting Time Till Job is Done |
|---|---|---|---|---|---|---|
| A | 0 | 3 | 0 | 3 | 0 | 3 |
| B | 1 | 5 | 5 | 10 | 4 | 9 |
| C | 3 | 2 | 3 | 5 | 0 | 2 |
| | **Total waiting** | | | | 4 | |

Job B's waiting time has increased, but relative to the amount of work it has to do, it has to wait less than job C did under the first-come, first-served policy. The *total* amount of waiting time for the shortest-job-first policy decreased compared to the first-come, first-served policy (4 versus 7).

The shortest-job-first policy has one implementation challenge: how do we know the amount of work a job has to do? In some cases, we may be able to decide before running the job whether or not this is a short job. For example, if we have two requests for reading different sectors on the disk and the disk arm is close to one of them, then the request that requires moving the arm to the closer track is the shorter job.

If we cannot decide without executing a job whether or not the job is short, we can make some forward progress by assuming that jobs fall in different classes: a thread that is interactive has mostly short jobs, while a thread that is computationally intensive is likely to have mostly long jobs. This suggests that if we track the past behavior of a thread, then we might be able to predict its future behavior. For example, if a thread just completed a short job, we might predict that its next job also will be short. We can make this idea more precise by basing our prediction on all past jobs of a given thread. One way of doing so is using an *Exponentially Weighted Moving Average (EWMA)* (see Sidebar 7.6 [on-line]). Of course, past behavior may be a weak indicator of future behavior.

A disadvantage of the shortest-job-first policy versus the first-come first-served policy is that shortest-job-first may lead to *starvation*. Several threads that consist entirely of short jobs and that together present a load large enough to use up the available processors may prevent a long job from ever being run. In practice, as we will see in Sections 6.3.3.4 and 6.3.4, the shortest-job-first policy can be combined with other policies to avoid starvation.

#### 6.3.3.3 Round-Robin

One of the issues with shortest job first is identifying which jobs are short and which are long. One approach is to make all jobs short by breaking long jobs up into a number of smaller jobs using preemptive scheduling. A preemptive scheduling policy stops a job after a certain amount of time so that the scheduler can pick

another job, resuming the preempted one at some time later. As we discussed in Chapter 5, preemptive scheduling also has the benefit that it enforces modularity; a programming error cannot cause a job to never release the processor.

A simple preemptive scheduling policy is *round-robin scheduling.* A round-robin scheduler maintains a queue of runnable jobs as before. It selects the first job from this queue, as in the first-come first-serve policy, but stops the job after some period of time, and selects a new job. Some time later the scheduler will select the stopped job again and run it again for no longer than the fixed period of time, and so on, until the job completes.

Round-robin can be implemented as follows. Before running the job, the round-robin scheduler sets a timer with a fixed time value, called a *quantum*. When the timer expires, it causes an interrupt and the interrupt handler calls YIELD. This call gives control back to the scheduler, which moves the job to the end of the queue and selects a new job from the front of the queue. The quantum should be long enough that most short jobs complete without being interrupted, and it should be short enough that most long jobs do get interrupted so that short jobs can get to run sooner.

Let's look at how a round-robin scheduler with a quantum of 1 second performs on the example sequence:

At time 0, only A is in the queue of runnable jobs, so the scheduler selects it. At time 1, B is in the queue so the scheduler selects B and appends A to the end of the queue, since it is not done. At time 2, A is at the front, so the scheduler selects A and appends B to the end of the queue. At time 3, the scheduler appends C to the end of the queue after B. Then, the scheduler selects B, since it is at the front of the queue, and appends A after C. At time 4, the scheduler appends B to the end of the queue and selects C to run. At time 5, the scheduler appends C to the end of the queue and selects A. At time 6, A is done, and the scheduler selects B, and so on.

This timeline results in the following table:

| Job | Arrival Time | Amount of Work | Start Time | Finish Time | Waiting Time Till Job Starts | Waiting Time Till Job is Done |
|---|---|---|---|---|---|---|
| A | 0 | 3 | 0 | 6 | 0 | 6 |
| B | 1 | 5 | 1 | 10 | 1 | 9 |
| C | 3 | 2 | 5 | 8 | 2 | 5 |
| | **Total waiting** | | | | 3 | |

As can been seen in this example, compared to first-come, first-served and shortest-job-first, round-robin results in the worst performance to complete an individual job, measured in total time elapsed since start. This is not surprising because a round-robin scheduler forces long jobs to stop after a quantum of time.

Round-robin, however, has the shortest total waiting time because with round-robin jobs start earlier: every job runs no longer than a quantum before it is stopped and the scheduler selects another job.

Round-robin favors jobs that run for less than a quantum at the expense of jobs that are more than a quantum long, since the scheduler will stop a long job after one quantum and run the short one before returning the processor to the long one. Round-robin is found in many computer systems because many computer systems are interactive, have short jobs, and a quick response provides a good user experience.

#### *6.3.3.4 Priority Scheduling*

Some jobs are more important than others. For example, a system thread that performs minor housekeeping chores such as garbage collecting unused temporary files might be given lower priority than a thread that runs a user program. In addition, if a thread has been blocked for a long time, it might be better to give it higher priority over threads that have run recently.

A scheduler can implement such policies using a *priority scheduling policy*, which assigns each job a priority number. The dispatcher selects the job with the highest priority number. The scheduler must have some rule to break ties, but it doesn't matter much what the rule is, as long as it doesn't consistenly favor one job over another.

A scheduler can assign priority numbers in many different ways. The scheduler could use a predefined assignment (e.g., systems jobs have priority 1, and user jobs have priority 0) or the priority could be computed using a policy function provided by the system designer. Or the scheduler could compute priorities *dynamically*. For example, if a thread has been waiting to run for a long time, the scheduler could temporarily boost the priority number of the thread's job. This approach can be used, for example, to avoid the starvation problem of the shortest-job-first policy.

A priority scheduler may be preemptive or non-preemptive. In the preemptive version, when a high-priority job enters while a low-priority job is running, the scheduler may preempt the low-priority job and start the high-priority job immediately. For example, an interrupt may notify a high-priority thread. When the interrupt handler calls NOTIFY, a preemptive thread manager may run the scheduler, which may interrupt some other processor that is running a low-priority job. The non-preemptive version would not do any rescheduling or preemption at interrupt time, so the low-priority job would run to completion; when it calls AWAIT, the scheduler will switch to the newly runnable high-priority job.

As we make schedulers more sophisticated, we have to be on the alert for surprising interactions among different schedulers. For example, if a thread manager that provides priorities isn't carefully designed, it is possible that the highest priority thread obtains the least amount of processor time. Sidebar 6.8, which explains priority inversion, describes this pitfall.

**Sidebar 6.8 Priority Inversion** Priority inversion is a common pitfall in designing a scheduler with priorities. Consider a thread manager that implements a preemptive, priority scheduling policy. Let's assume we have three threads, $T_1$, $T_2$, and $T_3$, and threads $T_1$ and $T_3$ share a lock *l* that serializes references to a shared resource. Thread $T_1$ has a low priority (1), thread $T_2$ has a medium priority (2), and thread $T_3$ has a high priority (3).

The following timing diagram shows a sequence of events that causes the high-priority thread $T_3$ to be delayed indefinitely while the medium priority thread $T_2$ receives the processor continuously.

Let's assume that $T_2$ and $T_3$ are not runnable; for example, they are waiting for an I/O operation to complete. The scheduler will schedule $T_1$, and $T_1$ acquires lock *l*. Now the I/O operation completes, and the I/O interrupt handler notifies $T_2$ and $T_3$. The scheduler chooses $T_3$ because it has the highest priority. $T_3$ runs for a short time until it tries to acquire lock *l*, but because $T_1$ already holds that lock, $T_3$ must wait. Because $T_2$ is runnable and has higher priority than $T_1$, the thread scheduler will select $T_2$. $T_2$ can compute indefinitely; when $T_2$'s time quantum runs out, the scheduler will find two threads runnable: $T_1$ and $T_2$. It will select $T_2$ because $T_2$ has a higher priority than $T_1$. As long as $T_2$ doesn't call WAIT, $T_2$ will keep the processor. As long as $T_2$ is runnable, the scheduler won't run $T_1$, and thus $T_1$ will not be able to release the lock and $T_3$, the high priority thread, will wait indefinitely. This undesirable phenomenon is known as *priority inversion*.

The solution to this specific example is simple. When $T_3$ blocks on acquiring lock *l*, it should temporarily lend its priority to the holder of the lock (sometimes called *priority inheritance*)—in this case, $T_1$. With this solution, $T_1$ will run instead of $T_2$, and as soon as $T_1$ releases the lock its priority will return to its normal low value and $T_3$ will run. In essence, this example is one of interacting schedulers. The thread manager schedules the processor and locks schedule references to shared resources. A challenge in designing computer systems is recognizing schedulers and understanding the interactions between them.

(*Sidebar continues*)

The problem and solution have been "discovered" by researchers in the real-time system, database, and operating system communities, and are well documented by now. Nevertheless, it is easy to fall into the priority inversion pitfall. For example, in July 1997 the Mars Pathfinder spacecraft experienced total systems resets on Mars, which resulted in loss of experimental data collected. The software engineers traced the cause of the resets to a priority inversion problem*.

*Mike Jones. What really happened on Mars? *Risks Forum 19,* 49 (December 1997). The Web page http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html includes additional information, including a follow-up by Glenn Reeves, who led the software team for the Mars Pathfinder.

#### 6.3.3.5 Real-Time Schedulers

Certain applications have *real-time* constraints; they require delivery of results before a specified deadline. A chemical process controller, for instance, might have a valve that must be opened every 10 seconds because otherwise a container overflows. Such applications employ *real-time schedulers* to guarantee that jobs complete by the stated deadline.

For some systems, such as a chemical plant, a nuclear reactor, or a hospital intensive-care unit, missing a deadline might result in disaster. Such systems require a *hard real-time scheduler.* For these schedulers, designers must carefully determine the amount of resources each job takes and design the complete system to ensure that all jobs can be handled in a timely manner, even in the worst case. Determining the amount of resources necessary and the time that a job takes, however, is difficult. For example, a system with a cache might sometimes run a job fast (when the job's references hit in the cache) and sometimes slow (when the job's references miss in the cache). Therefore, designers of hard real-time systems make the time a job takes as predictable as possible, either by turning off performance-enhancing techniques (e.g., caches) or by assuming the worst case performance. Typically, designers turn off interrupts and poll devices so that they can carefully control when to interact with a device. These techniques combined increase the likelihood that the designer can estimate when jobs will arrive and for how long they will run. Once the amount of resources and time required for each job are estimated, the designer of a hard real-time system can compute the schedule for executing all jobs.

For other systems, such as a digital music system, missing a deadline occasionally might be just a minor annoyance; such systems can use a *soft real-time scheduler.* A soft real-time scheduler attempts to meet all deadlines but doesn't guarantee it; it may miss a deadline. If, for example, multiple jobs arrive simultaneously, all have 1 second of work, and all have a deadline in 1 second, all jobs except one will miss their deadlines. The goal of a soft real-time scheduler is to avoid missing deadlines but to accept that it might happen when there is more work than there is time before the deadline to do the work.

One popular heuristic for avoiding missing deadlines is the *earliest-deadline-first scheduler*, which keeps the queue of jobs sorted by deadline. The dispatcher runs the first job on the queue, which is always the one with the closest deadline. Most students and faculty follow this policy: work first on the homework or paper that has the earliest deadline. This scheduling policy minimizes the total (summed) lateness of all the jobs.

For soft real-time schedulers that have a given set of jobs that must execute at periodic intervals, we can develop scheduling algorithms instead of just heuristics. Systems with periodic jobs are quite common. For example, a digital video recorder must process a picture frame every 1/30th of a second to make the output look like a movie.

To develop a scheduler for such a system, the total amount of work to be done by the periodic jobs must be less than the capacity of the system. Consider a system with $n$ periodic jobs $i$ that happen with a period of $P_i$ seconds and that each requires $C_i$ seconds. The load of such a system can be handled only if:

$$\left(\sum_{i=1}^{n} \frac{C_i}{P_i}\right) \leq 1$$

If the total amount of work exceeds the system's capacity at any time, then the system will miss a deadline. If the total amount of work is less than the capacity, the system may still miss a deadline occasionally because for some short interval of time the total amount of work to be done is greater than the capacity of the system. For example, a periodic interrupt may arrive at the same time that a periodic task must run. Thus, the condition stated is a necessary condition but not a sufficient one.

A good algorithm for dynamically scheduling periodic jobs is the *rate monotonic scheduler.* In the design phase of the system, the designer assigns each job a priority that is proportional to the frequency of the occurrence of that job. For example, a job that needs to run every 100 milliseconds receives a priority 10, and a job that needs to run every 200 milliseconds receives a priority 5. At runtime, the scheduler always runs the highest priority job, preempting a running job if necessary.

### 6.3.4 Case Study: Scheduling the Disk Arm

Much work has been done on thread scheduling, but since processors are no longer a usual performance bottleneck, thread scheduling has become less important. As explained in Section 6.1, however, disk arm scheduling is important because the mechanical disk arm creates an I/O bottleneck. The typical goal of a disk arm scheduler is to optimize overall throughput as opposed to the delay for each individual request.

When a disk controller receives a batch of disk requests from the file system, it must decide the order in which to process these requests. At first glance, it might appear that first-come first-served is a fine choice for scheduling the requests, but unfortunately that choice is a bad one.

To see why, recall from Section 6.1 that if the controller moves the disk arm, it reduces the transfer rate of the disk because seeking from one track to another takes time. However, the time required to do a seek depends on how many tracks the arm must cross. A simple, but adequate, model is that a seek from one track to another track that is $n$ tracks away takes $n \times t$ seconds, where $t$ is roughly constant.

Consider a disk controller that is on track 0 and receives four requests that require seeks to the tracks 0 (the innermost track), 90, 5, and 100 (outermost track). If the disk controller performs the four requests in the order in which it received them (first-come first-served), then it will seek first to track 0, then to 90, back to 5, and then forward to 100, for a total seek latency of $270t$:

| Request | Movement | Time |
|---|---|---|
| Seek 1 | $0 \rightarrow 0$ | $0t$ |
| Seek 2 | $0 \rightarrow 90$ | $90t$ |
| Seek 3 | $90 \rightarrow 5$ | $85t$ |
| Seek 4 | $5 \rightarrow 100$ | $95t$ |
| **Total** | | $270t$ |

A much better algorithm is to sort the requests by track number and process them in the sorted order. The total seek latency for that algorithm is $100t$:

| Request | Movement | Time |
|---|---|---|
| Seek 1 | $0 \rightarrow 0$ | $0t$ |
| Seek 2 | $0 \rightarrow 5$ | $5t$ |
| Seek 3 | $5 \rightarrow 90$ | $85t$ |
| Seek 4 | $90 \rightarrow 100$ | $10t$ |
| **Total** | | $100t$ |

In practice, disk scheduling algorithms are more complex because new requests arrive while the disk controller is working on a set of requests. For example, if the disk controller is working on requests in the order of track number (0, 5, 90, and 100), it finishes 5, and receives a new request for track 1, which request should it perform next? It can go back and perform 1, or it can keep going and perform 90 and 100. The first choice is an algorithm that is called shortest seek first; the second choice is called the *elevator algorithm*, named after the algorithm that many elevators execute

to transport people from floor to floor in buildings. With shortest-seek-first, the total seek time is 108*t*:

| Request | Movement | Time |
|---|---|---|
| Seek 1 | $0 \rightarrow 0$ | 0*t* |
| Seek 2 | $0 \rightarrow 5$ | 5*t* |
| Seek 3 | $5 \rightarrow 1$ | 4*t* |
| Seek 4 | $1 \rightarrow 90$ | 89*t* |
| Seek 5 | $90 \rightarrow 100$ | 10*t* |
| **Total** | | 108*t* |

With the elevator algorithm, the total seek latency is 199*t*:

| Request | Movement | Time |
|---|---|---|
| Seek 1 | $0 \rightarrow 0$ | 0*t* |
| Seek 2 | $0 \rightarrow 5$ | 5*t* |
| Seek 3 | $5 \rightarrow 90$ | 85*t* |
| Seek 4 | $90 \rightarrow 100$ | 10*t* |
| Seek 5 | $100 \rightarrow 1$ | 99*t* |
| **Total** | | 199*t* |

Many disk controllers use a combination of the shortest-seek-first algorithm and the elevator algorithm. When processing requests, for a while they use the shortest-seek algorithm to choose requests, minimizing seek time, but then switch to the elevator algorithm to avoid starving requests for more distant tracks. For example, if the controller performs the request for track 1 first, starts seeking into the direction of 90, but at track 5 another request for track 1 comes in, then shortest-seek-first would go back to track 1. Since this sequence of events may repeat forever, the disk controller may never serve the request for tracks 90 and 100. By bounding the time that disk controllers perform shortest-seek-first and then switching to the elevator algorithm, requests for the distant tracks will also be served. This method is fine for disk systems, since the primary objective is to maximize total throughput, and thus delaying one request over another is acceptable. In a building, however, people do not want to have long delays, and therefore for buildings the elevator algorithm is better.

## EXERCISES

**6.1** Suppose a processor has a clock rate of 100 megahertz. The time required to retrieve a word from the cache is 1 nanosecond, and the time required to retrieve a word not in the cache is 101 nanoseconds.

**6.1a** Determine the hit rate needed such that the average memory latency equals the processor cycle time.

*1988-1-4a*

**6.1b** Keeping the same memory devices but considering processors with a higher clock rate, what is the maximum useful clock rate such that the average memory latency equals the processor cycle time, and to what hit rate does it correspond?

*1988-1-4b*

**6.2** A particular program uses 100 data objects, each $10^5$ bytes long. The objects are contiguously allocated in a two-level memory system using the LRU page replacement policy with a fast memory of $10^6$ bytes and a page size of $10^3$ bytes. The program always makes 1,000 accesses to randomly selected bytes in one object, then moves on to another randomly selected object (with probability 0.01 it could be the same object), makes 1,000 accesses to randomly selected bytes there, and so on.

**6.2a** Ignoring any memory accesses that might be needed for fetching instructions, if the program runs long enough to reach an equilibrium state, what will the hit ratio be?

*1987-1-5a*

**6.2b** Will the hit ratio go up or down if the page size is changed from $10^3$ words to $10^4$ words, with all other memory parameters unchanged?

*1987-1-5b*

**6.3** OutTel corporation has been delivering j786 microprocessors to the computer industry for some time, and Metoo systems has decided to get into the act by building a microprocessor called the "clone786+", which differs from the j786 by providing twice as many processor registers. Metoo has simulated many programs and concluded that this one change reduces the number of loads and stores to memory by an average of 30%, and thus should improve performance, assuming of course that all programs—including its popular microkernel operating system—are recompiled to take advantage of the extra registers. Why might Metoo find the performance improvement to be less than their simulations predict? If there is more than one reason, which one is likely to reduce performance the most?

*1994-1-6*

**6.4** Mike R. Kernel is designing the OutTel P97 computer system, which currently has one page table in hardware. The first tests with this design show excellent performance with one application, but with multiple applications, performance is awful. Suggest three design changes to Mike's system that would improve

performance with multiple applications, and explain your choices briefly. You cannot change processor speed, but any other aspect of the system is fair game.

*1996-1-3*

**6.5** Ben Bitdiddle gets really excited about remote procedure call and implements an RPC package himself. His implementation starts a new service thread for each arriving request. The thread performs the operation specified in the request, sends a reply, and terminates. After measuring the RPC performance, Ben decides that it needs some improvement, so Ben comes up with a brute-force solution: he buys a much faster network. The transit time of the new network is half as large as it was before. Ben measures the performance of small RPCs (meaning that each RPC message contains only a few bytes of data) on the new network. To his surprise, the performance is barely improved. What might be the reason that his RPCs are not twice as fast?

*1995-1-5c*

**6.6** Why might increasing the page size of a virtual memory system increase performance? Why might increasing the page size of a virtual memory system decrease performance?

*1993-2-4a*

**6.7** Ben Bitdiddle and Louis Reasoner are examining a 3.5-inch magnetic disk that spins at 7,500 RPM, with an average seek time of 6.5 milliseconds and a data transfer rate of 10 megabytes per second. Sectors contain 512 bytes of user data.

**6.7a** On average, how long does it take to read a block of eight contiguous sectors when the starting sector is chosen at random?

**6.7b** Suppose that the operating system maintains a one-megabyte cache in RAM to hold disk sectors. The latency of this cache is 25 nanoseconds, and for block transfers the data transfer rate from the cache to a different location in RAM is 160 megabytes per second. Explain how these two specifications can simultaneously be true.

**6.7c** Give a formula that tells the expected time to read 100 randomly chosen disk sectors, assuming that the hit ratio of the disk block cache is $h$.

**6.7d** Ben's workstation has 256 megabytes of RAM. To increase the cache hit ratio, Ben reconfigures the disk sector cache to be much larger than one megabyte. To his surprise he discovers that many of his applications now run slower rather than faster. What has Ben probably overlooked?

**6.7e** Louis has disassembled the disk unit to see how it works. Remembering that the centrifuge in the biology lab runs at 36,000 RPM, he has come up with a bright idea on how to reduce the rotational latency of the disk. He suggests speeding it up to

96,000 RPM. He calculates that the rotation time will now be 625 microseconds. Ben says this idea is crazy. Explain Ben's concern.

*1994-3-1*

**6.8** Ben Bitdiddle has proposed the simple neat and robust file system (SNARFS).* Ben's system has no on-disk data structures other than the disk blocks themselves, which are *self-describing*. Each 4-kilobyte disk block starts with the following 24 bytes of information:

- *fid* (File-ID): a 64-bit number that uniquely defines a file. A *fid* of zero implies that the disk block is free.
- *sn* (Sequence Number): a 64-bit number that identifies which block of a file this disk block contains.
- *t* (Time): The time this block was last updated.

In addition, the first block of a file contains the file name (string), version number, and the *fid* of its parent directory. The rest of the first block is filled with data. Setting the directory *fid* to zero marks the entire file free.

Directories are just files. Each directory should contain only the *fid* of its parent directory. However, as a "hint" directories may also include a table giving the mapping from name to *fid* and the mapping from *fid* to blocks for some of the files in the directory.

To allow fast access, three in-memory (virtual memory) structures are created each time the system is booted:

- MAP: an in-memory hash table that associates a (*fid*, *sn*) pair with the disk block containing that block of that file
- FREE: a free list that represents all of the free blocks on disk in a compact manner
- RECYCLE: a list of blocks that are available for reuse but have not yet been written with a *fid* of 0

**6.8a** Each read or write of a disk block results in one disk I/O. What is the minimum number of disk I/Os required in SNARFS to create a new file containing 2 kilobytes of data in an existing directory? If the system crashes (i.e., the contents of virtual memory are lost) after these I/Os are completed, the file should be present in the appropriate directory after recovery.

**6.8b** Ben argues that the in-memory structures can easily be rebuilt after a crash. Explain what actions are required to rebuild MAP, FREE, and RECYCLE at boot time.

*1995-3-4a…c*

**6.9** Ben Bitdiddle has written a "style checker" intended to uncover writing problems in technical papers. The program picks up one sentence at a time, computes intensely for a while to parse it into nouns, verbs, and the like, and then looks up

*Credit for developing *Ex.* 6.8 goes to William J. Dally.

the resulting pattern in an enormous database of linguistic rules. The database was so large that it was necessary to place it on a remote service and do the lookup with a remote procedure call.

Ben is distressed to find that the RPCs have such a long latency that his style checker runs much more slowly than he hoped. He wonders if adding multiple threads to the client could speed up his program.

**6.9a** Ben's checker is running on a single-processor workstation. Explain how multiple client threads could reduce the time to analyze a technical paper.

**6.9b** Ben implements a multithreaded style checker and runs a series of experiments with various numbers of threads. He finds that performance does indeed improve when he adds a second thread and again when he adds a third. But he finds that as he adds more and more threads the additional performance improvement diminishes, and finally adding more threads leads to reduced performance. Give an explanation for this behavior.

**6.9c** Suggest a way of improving the style checker's performance without introducing threads. (Ben is allowed to change only the client.)

*1994-1-4a…c*

**6.10** Threads in a new multithreaded Web browser periodically query a nearby World Wide Web server to retrieve documents. On average, a browser's thread performs a query every $N$ instructions. Each request to the server incurs an average round-trip time of $T$ milliseconds before the answer returns.

**6.10a** For $N = 2{,}000$ instructions and $T = 1$ millisecond, what is the smallest number of such threads that would be required to keep a single 100 million instructions every second (MIPS) processor 100% busy? Assume that the context switch between threads is instantaneous and that the scheduler is optimal.

**6.10b** But context switches are not instantaneous. Assume that a context switch takes $C$ instructions to perform. Recompute the answer to 6.10a for $C = 500$ instructions.

**6.10c** What property of the application threads might cause the answers of parts 6.10a and 6.10b to be incorrect? That is, why might more threads be required to keep the processor running the browser busy?

**6.10d** What property of the actual computer system might make the answers of 6.10a and 6.10b gross overestimates?

*1995-1-4a…d*

**6.11** What are the advantages of using the clock algorithm as compared with implementing LRU directly?

**A.** Only a single bit per object or page is required.

**B.** Clock is more efficient to execute.

**C.** The first object or page to be purged is the most recently used one.

*2001-1-4*

**6.12** Louis Reasoner found the mention of prepaging systems in Section 6.2.9 to be so intriguing that he has devised a version of OPT that uses prepaging. Here is a description of Louis's prepage-OPT:

- Knowing the reference string, create a total ordering of the pages in which each page is in the order in which the application will next make reference to it. Then, prepage the front of the stack into the primary memory.
- After each page reference, rearrange the ordering so that every page is again in the order in which the application will next make reference to it. Thus, in contrast with LRU, which maintains an ordering since most recent use, prepage-OPT maintains an ordering of next use.

To do this rearrangement requires moving exactly one page, the one that was just touched, down in the ordering to the depth $d$ where it will next be used. All of the pages that were above depth $d$ move up one position. A page that will never be used again is assigned a depth of infinity and moves to the bottom of the stack. This rearrangement scheme ensures that the first page of the ordering is always the page that will be used next.

- If $d > m$ (where $m$ is the size of the primary memory device), the operation of the second bullet will result in a page being moved from the secondary memory to the primary memory. Since the reference string has not yet demanded this page, this movement anticipates a future need, another example of prepaging.

**6.11a** Is prepage-OPT a stack algorithm? Why or why not?

**6.11b** For the reference string in the example of Table 6.3, develop a version of that table (or of Table 6.8 if that is more appropriate) that shows what page movements occur with prepage-OPT for each memory size. Assume that the first step of the run is to preload the primary memory with pages from the front of the ordering.

**6.11c** Is prepage-OPT better or worse than demand-OPT?

*2006-0-1*

**Additional exercises relating to Chapter 6 can be found in the problem sets beginning on page 425.**

# About Part II

Part II of this textbook continues a main theme of Part I—enforcing modularity—by introducing still stronger forms of modularity. Part I introduces the client/service model and virtualization, both of which help prevent accidental errors in one module from propagating to another. Part II introduces stronger forms of modularity that can help protect against component and system failures, as well as malicious attacks. Part II explores communication networks, constructing reliable systems from unreliable components, creating all-or-nothing and before-or-after transactions, and implementing security. In doing so, Part II also introduces additional design principles to guide a designer who needs to build computer systems that have stronger modularity. Following is a brief summary of the topics of those [on-line] chapters and supplementary materials. In addition, the Table of Contents for Part I lists the sections of the Part II chapters.

*Chapter 7 [on-line]*: ***Networks***. By running clients and services on different computers that are connected by a network, one can build computer systems that exploit geographic separation to tolerate failures and construct systems that can enable information sharing across geographic distances. This chapter approaches the network as a case study of a system and digs deeply into how networks are organized internally and how they work. After a discussion that offers insight into why networks are built the way they are, it introduces a three-layer model, followed by a major section on each layer. A discussion of congestion control helps bring together the complete picture of interaction among the layers. The chapter ends with a short collection of war stories about network design flaws.

*Chapter 8 [on-line]*: ***Fault tolerance***. This chapter introduces the basic techniques to build computer systems that, despite component failures, continue to provide service. It offers a systematic development of design principles and techniques for creating reliable systems from unreliable components, based on modularity and on generalization of some of the techniques used in the design of networks. The chapter ends with a case study of fault tolerance in memory systems and a set of war stories about fault tolerant systems that failed to be fault tolerant. This chapter is an unusual feature for an introductory text—this material, if it appears at all in a curriculum, is usually left to graduate elective courses—yet some degree of fault tolerance is a requirement for almost all computer systems.

*Chapter 9 [on-line]*: ***Atomicity***. This chapter deals with the problem of making flawless updates to data in the presence of concurrent threads and despite system failures. It expands on concepts introduced in Chapter 5, taking a cross-cutting approach to atomicity—making actions atomic with respect to failures and also with respect to concurrent actions—that recognizes that atomicity is a form of modularity that plays a fundamental role in operating systems, database management, and processor design. The chapter begins by laying the groundwork for intuition about how a designer achieves atomicity, and then it introduces an easy-to-understand atomicity scheme.

This basis sets the stage for straightforward explanations of instruction renaming, transactional memory, logs, and two-phase locking. Once an intuition is established about how to systematically achieve atomicity, the chapter goes on to show how database systems use logs to create all-or-nothing actions and automatic lock management to ensure before-or-after atomicity of concurrent actions. Finally, the chapter explores methods of obtaining agreement among geographically separated workers about whether or not to commit an atomic action. The chapter ends with case studies of atomicity in processor design and management of disk storage.

*Chapter 10 [on-line]*: ***Consistency***. This chapter discusses a variety of requirements that show up when data is replicated for performance, availability, or durability: cache coherence, replica management for extended durability, and reconciliation of usually disconnected databases (e.g., "hotsync" of a personal digital assistant or cell phone with a desktop computer). The chapter introduces the reader to the requirements and the basic mechanisms used to meet those requirements. Sometimes these topics are identified with the label "distributed systems".

*Chapter 11 [on-line]*: ***Security***. Earlier chapters gradually introduced more powerful and far-reaching methods of enforcing modularity. This chapter cranks up the enforcement level to maximum strength by introducing the techniques of ensuring that modularity is enforced even in the face of adversaries who behave malevolently. It starts with design principles and a security model, and it then applies that model both to enforcement of internal modular boundaries (traditionally called "protection") and to network security. An advanced topics section explains cryptographic techniques, which are the basis for most network security. A case study of the Secure Socket Layer (SSL) protocol and a set of war stories of protection system failures illustrate the range and subtlety of considerations involved in achieving security.

***Suggestions for further reading***. The suggested reading list in Part II is, apart from updates, the same as the one in this book.

***Problem sets***. The Part II collection of problem sets includes both the Part I problem sets and many additional problem sets for the Part II chapters.

***Glossary***. The on-line Glossary is identical to the one in this book. In addition to its primary purpose of supporting this textbook, the on-line Glossary also can serve as a reference source that workers in other specialties may find useful in coordinating their terminology with that of the field of systems.

***Comprehensive Index***. The on-line Index of Concepts provides page numbers for both Part I and Part II in a single alphabetic list.

CHAPTER

# The Network as a System and as a System Component

# 7

## CHAPTER CONTENTS


**Overview ..... 7–2**
**7.1 Interesting Properties of Networks ..... 7–3**
7.1.1 Isochronous and Asynchronous Multiplexing ..... 7–5
7.1.2 Packet Forwarding; Delay ..... 7–9
7.1.3 Buffer Overflow and Discarded Packets ..... 7–12
7.1.4 Duplicate Packets and Duplicate Suppression ..... 7–15
7.1.5 Damaged Packets and Broken Links ..... 7–18
7.1.6 Reordered Delivery ..... 7–19
7.1.7 Summary of Interesting Properties and the Best-Effort Contract 7–20
**7.2 Getting Organized: Layers ..... 7–20**
7.2.1 Layers ..... 7–23
7.2.2 The Link Layer ..... 7–25
7.2.3 The Network Layer ..... 7–27
7.2.4 The End-to-End Layer ..... 7–28
7.2.5 Additional Layers and the End-to-End Argument ..... 7–30
7.2.6 Mapped and Recursive Applications of the Layered Model ..... 7–32
**7.3 The Link Layer ..... 7–34**
7.3.1 Transmitting Digital Data in an Analog World ..... 7–34
7.3.2 Framing Frames ..... 7–38
7.3.3 Error Handling ..... 7–40
7.3.4 The Link Layer Interface: Link Protocols and Multiplexing ..... 7–41
7.3.5 Link Properties ..... 7–44
**7.4 The Network Layer ..... 7–46**
7.4.1 Addressing Interface ..... 7–46
7.4.2 Managing the Forwarding Table: Routing ..... 7–48
7.4.3 Hierarchical Address Assignment and Hierarchical Routing ..... 7–56
7.4.4 Reporting Network Layer Errors ..... 7–59
7.4.5 Network Address Translation (An Idea That Almost Works) ..... 7–61
**7.5 The End-to-End Layer ..... 7–62**
7.5.1 Transport Protocols and Protocol Multiplexing ..... 7–63
7.5.2 Assurance of At-Least-Once Delivery; the Role of Timers ..... 7–67
7.5.3 Assurance of At-Most-Once Delivery: Duplicate Suppression ..... 7–71

7.5.4 Division into Segments and Reassembly of Long Messages ..... 7–73
7.5.5 Assurance of Data Integrity ..... 7–73
7.5.6 End-to-End Performance: Overlapping and Flow Control ..... 7–75
7.5.6.1 Overlapping Transmissions ..... 7–75
7.5.6.2 Bottlenecks, Flow Control, and Fixed Windows ..... 7–77
7.5.6.3 Sliding Windows and Self-Pacing ..... 7–79
7.5.6.4 Recovery of Lost Data Segments with Windows..... 7–81
7.5.7 Assurance of Stream Order, and Closing of Connections ..... 7–82
7.5.8 Assurance of Jitter Control ..... 7–84
7.5.9 Assurance of Authenticity and Privacy ..... 7–85
**7.6 A Network System Design Issue: Congestion Control.....7–86**
7.6.1 Managing Shared Resources ..... 7–86
7.6.2 Resource Management in Networks ..... 7–89
7.6.3 Cross-layer Cooperation: Feedback ..... 7–91
7.6.4 Cross-layer Cooperation: Control ..... 7–93
7.6.5 Other Ways of Controlling Congestion in Networks ..... 7–94
7.6.6 Delay Revisited ..... 7–98
**7.7 Wrapping up Networks.....7–99**
**7.8 Case Study: Mapping the Internet to the Ethernet .....7–100**
7.8.1 A Brief Overview of Ethernet .....7–100
7.8.2 Broadcast Aspects of Ethernet .....7–101
7.8.3 Layer Mapping: Attaching Ethernet to a Forwarding Network ...7–103
7.8.4 The Address Resolution Protocol .....7–105
**7.9 War Stories: Surprises in Protocol Design .....7–107**
7.9.1 Fixed Timers Lead to Congestion Collapse in NFS .....7–107
7.9.2 Autonet Broadcast Storms .....7–108
7.9.3 Emergent Phase Synchronization of Periodic Protocols .....7–108
7.9.4 Wisconsin Time Server Meltdown .....7–109
**Exercises.....7–111**
**Glossary for Chapter 7 .....7–125**
**Index of Chapter 7 .....7–135**
**Last chapter page 7–139**

## 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 7.1, by identifying and investigating the interesting properties just mentioned, as well as methods of coping with those properties. Section 7.2 describes a three-layer reference model for a data communication network that is based on a best-effort contract, and Sections 7.3, 7.4, and 7.5 then explore more carefully a number of implementation issues and techniques for each of the three layers. Finally, Section 7.6 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 judgement.

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 at the end of this book include several good books and papers that provide wide-ranging treatments of all aspects of networks.

## 7.1 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.

But 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 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 7.1, 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

* The formula that relates signaling rate, signal power, noise level, and maximum data rate, known as *Shannon's capacity theorem,* appears on page 7–37.

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 radio-based 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.

### 7.1.1 Isochronous and Asynchronous Multiplexing

Sharing has significant consequences. Consider the simplified (and gradually becoming obsolescent) telephone network of Figure 7.1, 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 7.1**

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 7.2: 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 $704^{th}$ 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 $704^{th}$ one.

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.

**FIGURE 7.2**

Data flow on an isochronous multiplexed link.

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 7.3, 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 7.3**

A simple data communication network.

**FIGURE 7.4**

Data flow on an asynchronous multiplexed link.

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 7.4 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.

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 from a digital telephone, by breaking the stream up into segments. Doing so does create a problem that 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,

* 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.

**FIGURE 7.5**

A packet forwarding network.

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.

### 7.1.2 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 7.1) 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 7.5. 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.

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 7.5 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 7.1) to allow error detection or copying it to an output buffer that is somewhere else in memory, is known as processing delay.

**Sidebar 7.1: 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 8[on-line] 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 11[on-line], 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.

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.

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, using the queuing theory formula in Section 6.1.6. 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 $\rho$ is the utilization of the outgoing line, which can range from 0 to 1. When we plot this result in Figure 7.6 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 $\rho_{max}$. 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 7.1.3, below, the replacement is actually more subtle than that.

**FIGURE 7.6**

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 $\rho_{max}$, 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 $\rho_{max}$ 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 7.6 of this chapter we will see how the delays can be shifted all the way back to the entry point of the network.

### 7.1.3 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 $\rho$ 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 $(3 \times \sqrt{5} \cong 7)$ 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 7.5 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 to which source to send 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 7.6 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 (asynchronous or isochronous) for dealing with overload 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 7.1.7, 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 7.2 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, 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 e-mail 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.

### 7.1.4 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 7.7* 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 conventions for representation of timing diagrams were described in Sidebar 4.2.

**FIGURE 7.7**

Lost packet recovery.

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'*.

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 7.8. Client A can't tell the difference between this case and the case of Figure 7.7, 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.

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

**FIGURE 7.8**

ost packet recovery leading to duplicate request.

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 7.9. Again, this duplicate may or may not matter to A, but at minimum A must take steps not to be confused by the arrival of a 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

**FIGURE 7.9**

Network delay combined with recovery leading to duplicate response.

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 7.9, 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 on page 7–71 of this chapter.

Ensuring suppression of duplicates is a significant complication so, 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 9[on-line].

### 7.1.5 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 8[on-line]. 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 7.3.3 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 7.5. 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 7.4 of this chapter.

### 7.1.6 Reordered Delivery

When a packet-forwarding network has an interconnection topology like that of Figure 7.5, 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.

### 7.1.7 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 7.10 summarizes the differences in application characteristics and in response to overload between isochronous and asynchronous multiplexing.

Similarly, Figure 7.11 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.

## 7.2 Getting Organized: Layers

To deal with the interesting properties of networks that we identified in Section 7.1, 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.

| | | Application characteristics | | |
|---|---|---|---|---|
| | | Continuous stream (e.g., interactive voice) | Bursts of data (most computer-to-computer data) | Response to load variations |
| Network Type | isochronous (e.g., telephone network) | good match | wastes capacity | (hard-edged) either accepts or blocks call |
| | asynchronous (e.g., Internet) | variable latency upsets application | good match | (gradual)<br>1 variable delay<br>2 discards data<br>3 rate adaptation |

**FIGURE 7.10**

Isochronous versus asynchronous multiplexing.

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

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

**FIGURE 7.11**

A summary of the "interesting" properties of computer networks. The last group of bullets defines what is called the *best-effort contract*.

**FIGURE 7.12**

A remote procedure call.

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. An example of a protocol that we have already seen is that of the Network File System shown in Figure 4.10.

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 7.12. As described in Chapter 4, 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 7.12 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.

**FIGURE 7.13**

Two protocol layers

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.

### 7.2.1 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 7.13. 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 7.14.

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

**FIGURE 7.14**

Three protocol layers

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.

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 7.15, 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 7.14 in more detail, we will find it useful to impose fur-

**FIGURE 7.15**

A layered system.

ther 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 7.1 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.

### 7.2.2 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 7.16**

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 7.16 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

7.16, 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 (as Section 2.1.3 suggested), 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 7.16 the link layer calls a procedure named NETWORK_HANDLE in the layer above.

### 7.2.3 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 7.17, 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.

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

**FIGURE 7.17**

Relation between the network layer and the link layer.

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.

### 7.2.4 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 7.11 on page 7–21. 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 7.18 shows the remote procedure call of Figure 7.12 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-to-end 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 7.18. 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

**FIGURE 7.18**

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.

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.

### 7.2.5 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 7.14: 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 reimplement 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.

### 7.2.6 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 7.8, 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 7.2), 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 7.8 explains how the network

**Sidebar 7.2: 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 on page 7–65. 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 Section 7 of the Suggestions for Further Reading.

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.

*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 11, are used to implement what is commonly called a "virtual private network" (VPN).

Recursive composition need not be limited to two levels. Figure 7.19 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 typ-

**FIGURE 7.19**

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.

ical 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.

## 7.3 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.

### 7.3.1 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 volt-

**FIGURE 7.20**

A simple protocol for data communication.

ages 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.

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 7.20. 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 $\Delta t$, 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, $\Delta t$ 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 as in Figure 7.21. 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 7.22. 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.

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

**FIGURE 7.21**

Serial transmission.

**FIGURE 7.22**

Bit shape deterioration with distance.

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.

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.xx

**Sidebar 7.4: Shannon's capacity theorem**

$$C \leq W \cdot \log_2\left(1 + \frac{S}{NW}\right)$$

where:

$C$ = 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

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 7.4).* 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 chan-

**Sidebar 7.3: 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 bits (n, n + 1) or 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.

nel capacity, it turns out to be surprisingly useful for estimating capacities in the real world.

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 signalling 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.

### 7.3.2 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

* 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.

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  shows a simple pseudocode implementation of the procedure to send a frame with bit stuffing, and Figure 7.24 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.)

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.

```
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 7.23**

Sending a frame with bit stuffing.

### 7.3.3 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 8[online].

```
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 7.24**

Receiving a frame with bit stuffing.

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 page 7–12) 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.

### 7.3.4 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 7.16 on page 7–26. 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, *link_protocol* tells LINK_SEND which link protocol to use for *this_data*, and its addition leads to the protocol layering illustrated in Figure 7.25.

**FIGURE 7.25**

Layer composition with multiple link protocols.

**FIGURE 7.26**

Layer composition with multiple link protocols *and* link layer multiplexing to support multiple network layer 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 7.26. 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.

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 7.27. 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 7.24 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 7.24. The procedure

```
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 7.27**

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.

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.

These procedures also illustrate an important property of layering that was discussed on page 7–29. 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 7.2.5.

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 7.27 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.

### 7.3.5 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 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.

## 7.4 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.

### 7.4.1 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 7.28 at the left. 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 7.28**

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* tells the network layer at 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 7.3, 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 7.27. 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 7.29 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 7.30 shows this latest addition to the overhead of a packet.

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 7.27. Because in a network, any network attachment point can send a packet to any other, the last argument of GIVE_TO_END_LAYER, 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 7.27.

### 7.4.2 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:

```
    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 7.29**

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.

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 partic-

**FIGURE 7.30**

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).

**FIGURE 7.31**

Routing example.

ipates 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 7.31. 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.

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:

| 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:

| destination | link |
|---|---|
| A | 1 |
| B | 2 |
| 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*.

To illustrate the algorithm, suppose participant G starts with a path vector that contains just one item, an entry for itself, as in Figure 7.32. 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 Figure 7.33. This advertisement allows G to discover the names, which are in this case network addresses, of each of its neighbors.

| to | path |
|---|---|
| G | < > |

**FIGURE 7.32**

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

From A, via link 1

| to | path |
|---|---|
| A | < > |

From H, via link 2:

| to | path |
|---|---|
| H | < > |

From J, via link 3:

| to | path |
|---|---|
| J | < > |

From K, via link 4:

| to | path |
|---|---|
| K | < > |

**FIGURE 7.33**

Path vectors received by G in the first round.

path vector

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

forwarding table

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

**FIGURE 7.34**

First-round path vector and forwarding table for G.

From A, via link 1

| to | path |
|---|---|
| A | < > |
| G | <G> |

From H, via link 2:

| to | path |
|---|---|
| B | <B> |
| C | <C> |
| G | <G> |
| H | < > |
| J | <J> |
| K | <K> |

From J, via link 3:

| to | path |
|---|---|
| D | <D> |
| E | <E> |
| G | <G> |
| H | <H> |
| J | < > |
| K | <K> |

From K, via link 4:

| to | path |
|---|---|
| E | <E> |
| F | <F> |
| G | <G> |
| H | <H> |
| J | <J> |
| K | < > |

**FIGURE 7.35**

Path vectors received by G in the second round.

path vector

| 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> |

forwarding table

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

**FIGURE 7.36**

Second-round path vector and forwarding table for G.

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 Figure 7.34. 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.

Now, each participant advertises its new path vector. This time, G receives the four path vectors of Figure 7.35, 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.

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 Figure 7.36. 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 7.37.

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.

```
// 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 7.37**

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.

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.

### 7.4.3 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 7.28, 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 7.31 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 7.38 illustrates the use of a forwarding table for hierarchical addresses that is constructed of two sections.

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.

forwarding table in R1.B

region forwarding section

| to | link |
|---|---|
| R1 | local |
| R2 | 1 |
| R3 | 1 |
| R4 | 3 |

local forwarding section

| to | link |
|---|---|
| R1.A | 1 |
| R1.B | end-layer |
| R1.C | 2 |
| R1.D | 3 |

**FIGURE 7.38**

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.

### 7.4.4 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 7.5 of this chapter, 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 7.6.

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 page 7–54 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*.

### 7.4.5 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 11[on-line].)

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 11[on-line], it can help create what that chapter describes as "defense in depth".

## 7.5 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 commu-

nication 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.

### 7.5.1 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. As was explained in Chapter 4, 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. Chapter 3 described 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 7.27, on page 7–43. 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 7.5 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. 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 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.)

**Sidebar 7.5: 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*)

and 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.

[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.

### 7.5.2 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*, which 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, which has semantics similar to the at-least-once RPC introducd in Section 4.2.2. 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 10[on-line] investigates this issue further.

In summary, just as with at-least-once RPC, 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 appears 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.

There are two common ways to minimize the use of fixed timers, both of which are applicable only when a transport protocol sends a stream of data segments to the same destination: adaptive timers and negative acknowledgments.

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 7.6 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 7.5.6, below.

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

**Sidebar 7.6: 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

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

where $\alpha < 1$ and the subscript indicates the age of the measurement; the most recent being $M_0$. The multiplier $(1 - \alpha)$ at the end normalizes the result. This scheme has two advantages over a simple average. First, it gives more weight to recent measurements. The multiplier, $\alpha$, 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:

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

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

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

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.

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 $\alpha$) 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 request their retransmission by sending a *negative acknowledgment*, or *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.

### 7.5.3 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 7.1 of this chapter—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. This assurance is the one that would probably be wanted in an implementation of the Remote Procedure Call protocol of Chapter 4. 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 9.6.2[on-line] 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.

### 7.5.4 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 7.5.7, below.

### 7.5.5 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 7.3.3, 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 enroute; this threat is explored briefly in Section 7.5.9, below, and in more depth in Chapter 11[on-line].

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.

### 7.5.6 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 7.39. 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.

#### *7.5.6.1 Overlapping Transmissions*

To avoid the wait times, we can employ a pipelining technique related to the pipelining described in Section 6.1.5: 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 7.40. 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 7.39**

Lock-step transmission of multiple segments.

This diagram 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 7.7). 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.

First, one or more packets or acknowledgments may be lost along the way. The first step in coping with this problem is for the sender to maintain a list of segments sent. As each acknowledgment comes back, the sender checks that segment off its list. Then, after sending the last segment, the sender sets a timer to expire a little more than one network round-trip time in the future. If, upon receiving an acknowledgment, the list of missing acknowledgments becomes empty, the sender can turn off the timer, confident that the entire message has been delivered. If, on the other hand, the timer expires and there is still a list of unacknowledged segments, the sender resends each one in the list, starts another timer, and continues checking off acknowledgments. The sender repeats this sequence until either every segment is acknowledged or the sender exceeds its retry limit, in which case it reports a failure to the application that initiated this message. Each timer expiration at the sending side adds one more round-trip time of delay in completing the transmission, but if packets get through at all, the process should eventually converge.

**FIGURE 7.40**

Overlapped transmission of multiple segments.

**Sidebar 7.7: 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*.

#### *7.5.6.2 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 con-

cept 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 7.41. 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.

The window provided by the scheme of Figure 7.41 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

**FIGURE 7.41**

Flow control with a fixed window.

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. Exercises 7.13, on page 7–114, and 7.16, on page 7–115, explore the *bang-bang protocol* and *pacing*, two more variants on the fixed window idea.

#### *7.5.6.3 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,

$$\textit{window size} \geq \textit{round-trip time} \times \textit{bottleneck data rate}$$

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 bot-

tleneck, 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 enroute 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

* 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.

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.

#### *7.5.6.4 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 $10^{30}$ transmission 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. This model is familiar. A sliding window

is an example of the producer–consumer problem described in Chapter 5, and the cumulative total window sizes granted and used are examples of eventcounts.

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 7.1.4.

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 7.6 of this chapter.

### 7.5.7 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 7.5.3, 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 7.5.1 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.)

### 7.5.8 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 seg-

ments 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 $D_{\text{long}}$.
4. From the distribution determine the shortest delay time that is observed in practice. Call this value $D_{\text{short}}$.
5. Now, provide enough buffering to delay every arriving segment so that it appears to have arrived with delay $D_{\text{long}}$. The largest number of segments that would need to be buffered is

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

where $D_{\text{headway}}$ 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.

### 7.5.9 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 that someone hostile intercepts and maliciously modifies packets, or that some end-to-end layer participants violate 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 11[on-line], 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.

## 7.6 A Network System Design Issue: Congestion Control

### 7.6.1 Managing Shared Resources

Chapters 5 and 6 discussed shared resources and their management: a thread manager creates many virtual processors from a few real, shared processors that must then be scheduled, and a multilevel memory manager creates the illusion of large, fast virtual memories for several clients by combining a small and fast shared memory with large and slow storage devices. In both cases we looked at relatively simple management mechanisms because more complex mechanisms aren't usually needed. In the network context, 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, so we need to revisit the topic.

In Section 7.1.2 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 7.6 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 judgement, 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 7.42 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.

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.)

Chapter 6 suggested 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

**FIGURE 7.42**

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.

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.

### 7.6.2 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 time-scale 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 7.6.3 and 7.6.4 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.

### 7.6.3 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.*

### 7.6.4 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 7.8).

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.

**Sidebar 7.8: 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
[Suggestions for Further Reading 1.4.5]

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 7.5 of this chapter 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 7.5.6, 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 7.5.2). 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 7.5.6 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 the expression on page 7–79.

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 7.9.

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.

### 7.6.5 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

**Sidebar 7.9: 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:

(*Sidebar continues*)

- *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.

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 7.12 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 Section 7.4 in the Suggestions for Further Reading.

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 7.10. 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.

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,

**Sidebar 7.10: 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)

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.

### 7.6.6 Delay Revisited

Section 7.1.2 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 7.1.2 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.

## 7.7 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, the backplane bus described in Chapter 3 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 judgement 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.

## 7.8 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.

### 7.8.1 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 [see Suggestions for Further Reading 7.1.2]. 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:

| 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.

### 7.8.2 Broadcast Aspects of Ethernet

Section 7.3.5 of this chapter mentioned Ethernet as an example of a network that uses a broadcast link. As illustrated in Figure 7.43, 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 7.43**

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 that every receiving station 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.

### 7.8.3 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 7.31 on page 7–50, by making the Ethernet a sixth link on router *K* in that figure. This connection produces the configuration of Figure 7.44.

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 7.43, 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 on page 7–50. 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 7.44**

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 on page 7–34, 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 such as that of Figure 7.45, 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 well enough connected to pass the packet along to its destination. This table is just another example of a forwarding table like the ones in Section 7.4 of this chapter.

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

**FIGURE 7.45**

Forwarding table to connect upper and lowe layer addresses

### 7.8.4 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 at the right. The internet network handler of station 17 can now proceed with its originally requested send operation.

| 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 at the left. Station *L* can now send the packet to the router, which presumably knows how to forward the packet to its intended destination.

| 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 Figure 7.45. 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.

## 7.9 War Stories: Surprises in Protocol Design

### 7.9.1 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) described in the case study in Section 4.5. 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.*

### 7.9.2 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". Echos 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 echos. 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.

*Lesson: 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.*

### 7.9.3 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 co-worker 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 on page 7–36.

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.*

### 7.9.4 Wisconsin Time Server Meltdown

NE TGEAR®, 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.*

*Lesson(s): There are several. (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://www.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.

## Exercises

7.1 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?

*1996–1–1e*

7.2 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.

*1999–2–03*

7.3 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.

*1998–2–01*

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

*1995-1-5a*

7.5 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.

7.5a. Is this a good way to manage retransmission delays for Ethernet? Why or why not?
*1987–1–2a*

7.5b. Is this a good way to manage retransmission delays for a receive-and-forward network? Why or why not?
*1987–1–2b*

**7.6** Variable delay is an intrinsic problem of isochronous networks. True or False?
*1995–1–1f*

**7.7** 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 $\alpha$, 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.

7.7a. How long should A wait for a frame to be acknowledged before retransmitting it?
*1987–1–3a*

7.7b. What is the average number of times that A will have to send each frame?
*1987–1–3b*

**7.8** 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.

*2000–2–02*

**7.9** 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.

*1997–1–1e*

**7.10** 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?

*1994–2–6*

**7.11** 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.

*1980–3–3*

**7.12** 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.

7.12a. 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?

*1995–2–2a*

7.12b. 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 7.12a, how long would it take for the client to realize it has reached the maximum throughput?

*1995–2–2b*

7.12c. 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 7.12a, how long will it take for the client to know it has achieved the maximum throughput?

*1995–2–2c*

**7.13** 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.

7.13a. What is the minimum time required to send the file?

*1988–2–2a*

7.13b. 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 go-ahead 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?)

*1988–2–2b]*

7.13c. What now is the minimum time required to send the file?

*1988–2–2c*

**7.14** 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 end-to-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.

*2000–2–09*

**7.15** 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?

*1994–2–5*

**7.16** 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.

7.16a. 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).

7.16b. What is the maximum fraction of the available satellite capacity that can be used by this pacing scheme?

7.16c. 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?

*1982–3–4*

**7.17** Which are true statements about network address translators as described in Section 7.4.5?

**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.

*2001–2–01, 2002–2–02, and 2004–2–2*

**7.18** 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.

*1994–1–2*

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

*1994–1–3b*

**7.20** 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.

*1999–2–01, 2000–1–04*

**7.21** 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.

*1998-2-02*

**7.22** 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.

*2004–1–3*

**7.23** 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.

*1996–2–1a*

**7.24** *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.

7.24a. 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

* Credit for developing exercise 7.24 goes to Anant Agarwal.

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'.

7.24b. 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.

7.24c. Can Alyssa ever lose packets with this protocol? Explain. (Hint: think about the properties of the underlying network.)

7.24d. What additional steps can the home node take to ensure that the scheme in question 7.24c never loses packets?

*1996–2–2*

**7.25** 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.

7.25a. What is the maximum throughput for the Send-and-wait?

7.25b. What is the maximum throughput for Flow-control?

7.25c. 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.

7.25d. 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.

*1997–2–2*

**7.26** 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.

7.26a. If there are no lost packets, estimate the maximum data rate you can achieve.

7.26b. 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.

7.26c. 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.)

7.26d. 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.)

7.26e. 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?

*1994–1–5*

7.27 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: | $12.5 \times 10^{-3}$ seconds |
| D.C to Seattle transit time: | $12.5 \times 10^{-3}$ seconds |
| Channel bandwidth in each direction: | $1.5 \times 10^{6}$ 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<br>= 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: | $100 \times 10^{-6}$ seconds |
| Server computation time per request: | $50 \times 10^{-6}$ seconds |

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

7.27a. What is the transmission delay the client thread observes in sending an RPC request message)?

7.27b. 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?

7.27c. 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?

7.27d. 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.

*1998–1–2a…d*

**7.28** 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.

7.28a. 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)?

7.28b. 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?

7.28c. 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.)

7.28d. 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.

*1996–1–4a…d*

**Additional exercises relating to Chapter 7 can be found in problem sets *17* through *25*.**

# Glossary for Chapter 7

**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*. [Ch. 2]

**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*. [Ch. 7]

**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. [Ch. 7]

**advertise**—In a network-layer routing protocol, for a participant to tell other participants which network addresses it knows how to reach. [Ch. 7]

**any-to-any connection**—A desirable property of a communication network, that any node be able to communicate with any other. [Ch. 7]

**asynchronous**—1. 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*. [Ch. 7]

**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. [Ch. 4]

**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. [Ch. 4]

**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. [Ch. 7]

**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. [Ch. 7]

**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. [Ch. 7]

**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 $10^{10}$. [Ch. 7]

**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. [Ch. 7]

**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). [Ch. 7]

**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. [Ch. 7]

**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. [Ch. 7]

**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. [Ch. 7]

**client**—At the end-to-end layer of a network, the end that initiates actions. Compare with **service**. [Ch. 7]

**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. [Ch. 4]

**coheerence**—See *read/write coherence* or *cache coherence*.

In networks, an event when two stations attempt to send a message over the same physical medium at the same time. See also *Ethernet*. [Ch. 7]

**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.) [Ch. 7]

**congestion collapse**—When an increase in offered load causes a catastrophic decrease in useful work accomplished. [Ch. 7]

**connection**—A communication path that requires maintaining state between successive messages. See *set up* and *tear down*. [Ch. 7]

**connectionless**—Describes a communication path that does not require coordinated state and can be used without set up or tear down. See *connection*. [Ch. 7]

**control point**—An entity that can adjust the capacity of a limited resource or change the load that a source offers. [Ch. 7]

**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. [Ch. 7]

**data integrity**—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*. [Ch. 7]

**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. [Ch. 7]

**destination**—The network attachment point to which the payload of a packet is to be delivered. Sometimes used as shorthand for *destination address*. [Ch. 7]

**destination address**—An identifier of the destination of a packet, usually carried as a field in the header of the packet. [Ch. 7]

**duplex**—Describes a link or connection between two stations that can be used in both directions. Compare with *simplex*, *half-duplex,* and *full-duplex*. [Ch. 7]

**duplicate suppression**—A transport protocol mechanism for achieving at-most-once delivery assurance, by identifying and discarding extra copies of packets or messages. [Ch. 7]

**early drop**—A predictive strategy for managing an overloaded resource: the system refuses service to some customers before the queue is full. [Ch. 7]

**end-to-end**—Describes communication between network attachment points, as contrasted with communication between points within the network or across a single link. [Ch. 7]

**end-to-end layer**—The communication system layer that manages end-to-end communications. [Ch. 7]

**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. Compare with *error-detection code*. [Ch. 7]

**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*. Compare with *error-correction code* and *checksum*. [Ch. 7]

**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. [Ch. 7]

**exactly-once**—A protocol assurance that the intended operation or message delivery was performed both at-least-once and at-most-once. [Ch. 4]

**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. [Ch. 7]

**flow control**—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. [Ch. 7]

**forwarding table**—A table that tells the network layer which link to use to forward a packet, based on its destination address. [Ch. 7]

**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. [Ch. 7]

**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. [Ch. 7]

**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*. [Ch. 7]

**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*. [Ch. 7]

**header**—Information that a protocol layer adds to the front of a packet. [Ch. 7]

**hierarchical routing**—A routing system that takes advantage of hierarchically assigned network destination addresses to reduce the size of its routing tables. [Ch. 7]

**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. [Ch. 7]

**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*. [Ch. 7]

**International Organization for Standardization (ISO)**—An international non-governmental 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. [Ch. 7]

**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*. [Ch. 7]

**jitter**—In real-time applications, variability in the delivery times of successive data elements. [Ch. 7]

[Ch. 7]**link layer**—The communication system layer that moves data directly from one physical point to another. [Ch. 7]

**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*. [Ch. 7]

**Manchester code**—A particular type of phase encoding in which each bit is represented by two bits of opposite value. [Ch. 7]

**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. [Ch. 7]

**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*. [Ch. 7]

**MTU discovery**—A procedure that systematically discovers the smallest maximum transmission unit along the path between two network attachment points. [Ch. 7]

**multihomed**—Describes a single physical interface between the network layer and the end-to-end layer that is associated with more than one network attachment point, each with its own network-layer address. [Ch. 7]

**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. [Ch. 7]

**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. [Ch. 7]

**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*. [Ch. 7]

**network**—A communication system that interconnects more than two things. [Ch. 7]

**network address**—In a network, the identifier of the source or destination of a packet. [Ch. 7]

**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). [Ch. 7]

**network layer**—The communication system layer that forwards data through intermediate links to carry it to its intended destination. [Ch. 7]

**nonce**—A unique identifier that should never be reused. [Ch. 7]

**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.* [Ch. 7]

**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. [Ch. 7]

**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.* [Ch. 7]

**page fault**—See *missing-page exception.*

**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. [Ch. 7]

**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. [Ch. 7]

**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.) [Ch. 7]

**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. [Ch. 7]

**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. [Ch. 7]

**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. [Ch. 7]

**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.* [Ch. 7]

**point-to-point**—Describes a communication link between two stations, as contrasted with a broadcast or multipoint link. [Ch. 7]

**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. [Ch. 7]

**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*.

**presentation protocol**—A protocol that translates semantics and data of the network to match those of the local programming environment. [Ch. 7]

**presented load**—See *offered load*.

**processing delay**—In a communication network, that component of the overall delay contributed by computation that takes place in various protocol layers. [Ch. 7]

**propagation delay**—In a communication network, the component of overall delay contributed by the velocity of propagation of the physical medium used for communication. [Ch. 7]

**protocol**—An agreement between two communicating parties, for example on the messasges and format of data that they intend to exchange. [Ch. 7]

**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. [Ch. 7]

**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. [Ch. 7]

**random drop**—A strategy for managing an overloaded resource: the system refuses service to a queue member chosen at random. [Ch. 7]

**random early detection** *(RED)*—A combination of random drop and early drop. [Ch. 7]

**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. [Ch. 7]

**reassembly**—Reconstructing a message by arranging, in correct order, the segments it was divided into for transmission. [Ch. 7]

**reliable delivery**—A transport protocol assurance: it provides both at-least-once delivery and data integrity. [Ch. 7]

**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. [Ch. 7]

**router**—A packet forwarder that also participates in a routing algorithm. [Ch. 7]

**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. [Ch. 7]

**segment**—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*. [Ch. 7]

**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. [Ch. 7]

**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. [Ch. 7]

**service**—At the end-to-end layer of a network, the end that responds to actions initiated by the other end. Compare with *client*. [Ch. 7]

**set up**—The steps required to allocate storage space for and initialize the state of a connection. [Ch. 7]

**simplex**—Describes a link between two stations that can be used in only one direction. Compare with *duplex*, *half-duplex*, and *full-duplex*. [Ch. 7]

**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. [Ch. 7]

**source**—The network attachment point that originated the payload of a packet. Sometimes used as shorthand for *source address*. [Ch. 7]

**source address**—An identifier of the source of a packet, usually carried as a field in the header of the packet. [Ch. 7]

**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*. [Ch. 7]

**station**—A device that can send or receive data over a communication link. [Ch. 7]

**stop and wait**—A synonym for **lock step.** [Ch. 7]

**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. [Ch. 7]

**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. [Ch. 7]

**tail drop**—A strategy for managing an overloaded resource: the system refuses service to the queue entry that arrived most recently. [Ch. 7]

**tear down**—The steps required to reset the state of a connection and deallocate the space that was used for storage of that state. [Ch. 7]

**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. [Ch. 7]

**trailer**—Information that a protocol layer adds to the end of a packet. [Ch. 7]

**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*. [Ch. 7]

**transmission delay**—In a communication network, the component of overall delay contributed by the time spent sending a frame at the available data rate. [Ch. 7]

**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. [Ch. 7]

**virtual circuit**—A connection intended to carry a stream through a forwarding network, in some ways simulating an electrical circuit. [Ch. 7]

**window**—In flow control, the quantity of data that the receiving side of a transport protocol is prepared to accept from the sending side. [Ch. 7]

# Index of Chapter 7

Design principles and hints appear in *underlined italics*. Procedure names appear in SMALL CAPS. Page numbers in **bold face** are in the chapter Glossary.


## A

ACK (see acknowledgment)
acknowledgment 7–67, 7–77, 7–82, **7–125**
adaptive
  routing 7–49, **7–125**
  timer 7–69
additive increase 7–96
address
  destination **7–127**
  in networks 7–46, **7–130**
  resolution protocol 7–105, **7–125**
  source **7–132**
*adopt sweeping simplifications* 7–20
advertise 7–51, **7–125**
any-to-any connection 7–4, **7–125**
application protocol 7–23
ARP (see address resolution protocol)
asynchronous 7–7, **7–125**
at-least-once
  protocol assurance 7–68, **7–125**
at-most-once
  protocol assurance 7–71, **7–125**
attachment point (see network attachment point)
automatic rate adaptation 7–14, 7–93, **7–125**

## B

backoff
  exponential 7–70, **7–128**
bandwidth 7–37, **7–125**
bang-bang protocol 7–114
best effort 7–14, **7–125**
  contract 7–21
bit error rate 7–38, **7–126**
bit stuffing 7–39, **7–126**
blast protocol 7–119
bottleneck
  data rate 7–79
broadcast 7–45, 7–102, **7–126**
burst 7–7, **7–126**

## C

carrier sense multiple access 7–100, **7–127**
checksum 7–10, 7–74, **7–126**
circuit
  switch 7–9, **7–126**
  virtual 7–82, **7–133**
client 7–63, **7–126**
close-to-open consistency **7–126**
collision
  Ethernet 7–100, **7–127**
congestion 7–13, 7–87, **7–126**
  collapse 7–87, 7–88, **7–126**
connection 7–7, **7–126**
connectionless 7–8, **7–126**
consistency
  close-to-open **7–126**
control point 7–89, **7–126**
cross-layer cooperation 7–91, 7–93
CSMA/CD (see carrier sense multiple access)
cut-through 7–10, **7–126**

## D

data integrity
  in communications 7–73, **7–127**
data rate 7–4, **7–127**
datagram 7–8
decay
  factor 7–70
*decouple modules with indirection* 7–110

decrypt 7–86
delay 7–9, 7–98
  processing 7–10, 7–98, **7–131**
  propagation 7–3, 7–10, 7–99, **7–131**
  queuing 7–11, 7–99, **7–131**
  transmission 7–10, 7–99, **7–133**
*design principles*
  *adopt sweeping simplifications* 7–20
  *decouple modules with indirection* 7–110
  *end-to-end argument* 7–31
  *incommensurate scaling rule* 7–91
destination 7–8, 7–27, 7–46, **7–127**
  address **7–127**
discovery
  of maximum transmission unit 7–61, **7–129**
distance vector 7–54
duplex 7–45, **7–127**
duplicate suppression 7–17, 7–71, **7–127**

## E

early drop 7–92, **7–127**
echo request 7–60
encrypt 7–86
end-to-end **7–127**
  layer 7–25, 7–28, 7–62, **7–127**
*end-to-end argument* 7–31
error
  correction 7–40, **7–127**
  detection 7–40, **7–127**
Ethernet 7–100, **7–127**
exactly-once
  protocol assurance 7–73, **7–128**
exponential
  backoff 7–70, **7–128**
exponentially weighted moving average 7–70

## F

fast start 7–114
fingerprint 7–10
fixed
  timer 7–69
  window 7–78
flow control 7–77, **7–128**
forwarder 7–9
forwarding table 7–48, **7–128**
fragment **7–128**
frame 7–6, 7–8, 7–37, **7–128**
full-duplex 7–45, **7–128**

## G

guaranteed delivery 7–14

## H

half-duplex 7–45, **7–128**
hard-edged 7–6
header 7–26, **7–128**
hierarchy
  in routing 7–56, **7–128**
hop limit 7–54, **7–128**

## I

ICMP (see Internet control message protocol)
idempotent 7–18
*incommensurate scaling rule* 7–91
integrity (see data integrity)
intended load 7–88, **7–128**
International Organization for Standardization 7–30, **7–128**
Internet 7–32
  control message protocol 7–60
  protocol 7–32
invisible hand 7–98
IP (see Internet protocol)
ISO (see International Organization for Standardization)
isochronous 7–6, **7–129**

## J

jitter 7–84, **7–129**

## L

layer
  end-to-end 7–25, 7–28, 7–62, **7–127**
  link 7–25, 7–34, **7–129**
  network 7–25, 7–27, 7–46, **7–130**
link
  layer 7–25, 7–34, **7–129**

lock-step protocol 7–75, **7–129**

## M

Manchester code 7–36, **7–129**
maximum transmission unit 7–45, **7–129**
message 7–7, 7–33, **7–129**
message-sending protocol 7–63
mobile host 7–118
MTU (see maximum transmission unit)
MTU discovery 7–61, **7–129**
multihomed 7–46, **7–129**
multiplexing 7–5, 7–42, 7–47, 7–64, **7–129**
multiplicative decrease 7–96
multipoint 7–67, **7–129**

## N

NAK (see negative acknowledgment)
NAT (see network address translation)
negative acknowledgment 7–71, 7–83, **7–129**
network 7–2, **7–129**
  address 7–46, **7–130**
  address translation 7–61
  attachment point 7–9, 7–27, 7–46, **7–130**
  layer 7–25, 7–27, 7–46, **7–130**
  services access point **7–130**
nonce 7–17, 7–71, **7–130**
NSAP (see network services access point)

## O

offered load 7–88
OSI (see International Organization for Standardization)
overlay network 7–33
overprovisioning 7–94

## P

pacing 7–115
packet 7–8, 7–33, **7–130**
  forwarding 7–9, **7–130**
  forwarding network 7–9
  switch 7–9, **7–130**
parallel transmission 7–35, **7–130**
path 7–48
  selection 7–51, **7–130**
  vector 7–51
payload 7–26, **7–130**
persistent
  sender 7–67, **7–130**
phase encoding 7–36, **7–130**
phase-locked loop 7–36
piggybacking 7–77, **7–130**
pipeline **7–131**
point-to-point 7–44, **7–131**
port 7–64, **7–131**
prepaging **7–131**
presentation
  protocol 7–23, 7–67, **7–131**
  service 7–29
pricing 7–97
probe 7–60
processing delay 7–10, 7–98, **7–131**
propagation delay 7–3, 7–10, 7–99, **7–131**
protocol 7–21, **7–131**
  address resolution 7–105, **7–125**
  application 7–23
  bang-bang 7–114
  blast 7–119
  carrier sense multiple access 7–100, **7–127**
  Internet 7–32
  internet control message 7–60
  lock-step 7–75, **7–129**
  message-sending 7–63
  multiplexing 7–42
  presentation 7–23, 7–67, **7–131**
  ready/acknowledge 7–35, **7–131**
  real-time transport 7–67
  reliable message stream 7–66
  request/response 7–66
  routing 7–50
  simple network time service 7–109
  stream transport 7–82
  transmission control 7–65
  transport 7–23, 7–63, **7–133**

user datagram 7–65

## Q

quench 7–13, 7–91, **7–131**
queuing delay 7–11, 7–99, **7–131**

## R

random
drop 7–92, **7–131**
early detection 7–92, **7–131**
ready/acknowledge protocol 7–35, **7–131**
real time 7–84
real-time
transport protocol 7–67
reassembly 7–8, **7–131**
RED (see random early detection)
reliable
delivery 7–74, **7–131**
message stream protocol 7–66
request/response protocol 7–66
round-trip time 7–67, **7–131**
estimation 7–69, 7–80
route 7–9, 7–48
router 7–9, 7–50, **7–132**
routing 7–48
algorithm 7–49, **7–132**
protocol 7–50
RTP (see real-time transport protocol)

## S

segment
of a message 7–8, 7–33, **7–132**
self-pacing 7–80, **7–132**
serial transmission 7–35, **7–132**
service 7–63, **7–132**
time 7–87
session service 7–29
set up 7–7, **7–132**
Shannon's capacity theorem 7–37
sharing 7–5
sign 7–86
simple
network time service protocol 7–109
simplex 7–44, **7–132**
sliding window 7–79, **7–132**
slow start 7–95
SNTP (see protocol, simple network time service)
source 7–27, 7–46, **7–132**
address **7–132**
static
routing 7–49, **7–132**
station 7–50, 7–101, **7–132**
identifier 7–101
stop and wait (see lock-step protocol)
store and forward 7–14, **7–132**
stream 7–7, 7–33, **7–132**
transport protocol 7–82

## T

tail drop 7–92, **7–133**
TCP (see transmission control protocol)
TDM (see time-division multiplexing)
tear down 7–7, **7–133**
time-division multiplexing 7–6
timer
adaptive 7–69
fixed 7–69
tombstone 7–72, **7–133**
tragedy of the commons 7–93
trailer 7–26, **7–133**
transit time 7–9, **7–133**
transmission
control protocol 7–65
delay 7–10, 7–99, **7–133**
parallel 7–35, **7–130**
serial 7–35, **7–132**
transport
protocol 7–23, 7–63, **7–133**
service 7–29
tunnel (in networks) 7–33

## U

UDP (see user datagram protocol)
upcall 7–27
user
datagram protocol 7–65

## V

verify 7–86
virtual
  circuit 7–82, **7–133**

## W

well-known
  port 7–64
window 7–78, **7–133**
  fixed 7–78
  sliding 7–79, **7–132**
witness 7–10

CHAPTER

# Fault Tolerance: Reliable Systems from Unreliable Components

# 8

## CHAPTER CONTENTS


**Overview ..... 8–2**
**8.1 Faults, Failures, and Fault Tolerant Design ..... 8–3**
8.1.1 Faults, Failures, and Modules ..... 8–3
8.1.2 The Fault-Tolerance Design Process ..... 8–6
**8.2 Measures of Reliability and Failure Tolerance ..... 8–8**
8.2.1 Availability and Mean Time to Failure ..... 8–8
8.2.2 Reliability Functions ..... 8–13
8.2.3 Measuring Fault Tolerance ..... 8–16
**8.3 Tolerating Active Faults ..... 8–16**
8.3.1 Responding to Active Faults ..... 8–16
8.3.2 Fault Tolerance Models ..... 8–18
**8.4 Systematically Applying Redundancy ..... 8–20**
8.4.1 Coding: Incremental Redundancy ..... 8–21
8.4.2 Replication: Massive Redundancy ..... 8–25
8.4.3 Voting ..... 8–26
8.4.4 Repair ..... 8–31
**8.5 Applying Redundancy to Software and Data ..... 8–36**
8.5.1 Tolerating Software Faults ..... 8–36
8.5.2 Tolerating Software (and other) Faults by Separating State ..... 8–37
8.5.3 Durability and Durable Storage ..... 8–39
8.5.4 Magnetic Disk Fault Tolerance ..... 8–40
8.5.4.1 Magnetic Disk Fault Modes ..... 8–41
8.5.4.2 System Faults ..... 8–42
8.5.4.3 Raw Disk Storage ..... 8–43
8.5.4.4 Fail-Fast Disk Storage ..... 8–43
8.5.4.5 Careful Disk Storage ..... 8–45
8.5.4.6 Durable Storage: RAID 1 ..... 8–46
8.5.4.7 Improving on RAID 1 ..... 8–47
8.5.4.8 Detecting Errors Caused by System Crashes ..... 8–49
8.5.4.9 Still More Threats to Durability ..... 8–49
**8.6 Wrapping up Reliability ..... 8–51**

8.6.1 Design Strategies and Design Principles ..... 8–51
8.6.2 How about the End-to-End Argument? ..... 8–52
8.6.3 A Caution on the Use of Reliability Calculations ..... 8–53
8.6.4 Where to Learn More about Reliable Systems ..... 8–53
**8.7 Application: A Fault Tolerance Model for CMOS RAM .....8–55**
**8.8 War Stories: Fault Tolerant Systems that Failed.....8–57**
8.8.1 Adventures with Error Correction ..... 8–57
8.8.2 Risks of Rarely-Used Procedures: The National Archives ..... 8–59
8.8.3 Non-independent Replicas and Backhoe Fade ..... 8–60
8.8.4 Human Error May Be the Biggest Risk ..... 8–61
8.8.5 Introducing a Single Point of Failure ..... 8–63
8.8.6 Multiple Failures: The SOHO Mission Interruption ..... 8–63
**Exercises.....8–64**
**Glossary for Chapter 8 .....8–69**
**Index of Chapter 8 .....8–75**
**Last chapter page 8–77**

## 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 each of these ideas has 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 11[on-line], 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.

## 8.1 Faults, Failures, and Fault Tolerant Design

### 8.1.1 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.

### 8.1.2 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 8.3:

- Identify every potential fault.
- Estimate the risk of each fault, as described in Section 8.2.
- 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 8.4:

- 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 also familiar: the *safety margin principle*, described near the end of Section 1.3.2. 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 mind set 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 11[online] 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.

## 8.2 Measures of Reliability and Failure Tolerance

### 8.2.1 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:*

$$Availability = \frac{\text{time system was running}}{\text{time system should have been running}}$$

$$= \frac{\sum_{i=1}^{N} TTF_i}{\sum_{i=1}^{N} (TTF_i + TTR_i)} \qquad \text{Eq. 8–1}$$

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):

$$MTTF = \frac{1}{N}\sum_{i=1}^{N} TTF_i \qquad MTTR = \frac{1}{N}\sum_{i=1}^{N} TTR_i \qquad \text{Eq. 8–2}$$

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

$$Availability = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR} = \frac{MTBF - MTTR}{MTBF} \qquad \text{Eq. 8–3}$$

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

$$\text{Down time} = (1 - Availability) = \frac{MTTR}{MTBF} \qquad \text{Eq. 8–4}$$

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 8–2. 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 8.2.2, 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 8–9 on page 8–13 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 8.1 and defined more

carefully in Section 8.2.2, 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.

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.

**FIGURE 8.1**

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

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, then can 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 tells only about availability, without any information 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 8.3.

### 8.2.2 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

$$R(t) = Pr\left(\begin{array}{c}\text{the module has not yet failed at time t, given that} \\ \text{the module was operating at time 0}\end{array}\right) \qquad \text{Eq. 8–5}$$

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

$$f(t) = Pr(\text{module fails between t and t + dt}) \qquad \text{Eq. 8–6}$$

(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 8.1.) Once *f(t)* is at hand, one can directly calculate the MTTF:

$$MTTF = \int_0^\infty t \cdot f(t)dt \qquad \text{Eq. 8–7}$$

One must keep in mind that this MTTF is predicted from the failure rate function *f(t)*, in contrast to the MTTF of eq. 8–2, 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.

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:

$$R(t) = e^{-\left(\frac{t}{MTTF}\right)} \qquad \text{Eq. 8–8}$$

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

**Sidebar 8.1: 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 8–6, 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 8.1 reports 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

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

conditional failure rate does change with time. This misappropriation starts with a fallacy: an assumption that the MTTF, as defined in eq. 8–7, can be calculated by inverting the measured failure rate. The fallacy arises because in general,

$$E(1/t) \neq 1/E(t) \qquad \text{Eq. 8–9}$$

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 eq. 8–8. 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 8.2. Suppose, for example, that a pro-

duction 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.

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 to ∞, which is about $3.4 \times 10^{-6}$, 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 8.2. 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

**FIGURE 8.2**

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.

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.

### 8.2.3 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 8.4 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.

## 8.3 Tolerating Active Faults

### 8.3.1 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 11[on-line] 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 it 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 8.3.

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.

**FIGURE 8.3**

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.

### 8.3.2 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 8.1.2, 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 described in Section 4.4 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 8.7. This fault tolerance model employs error detection and masking techniques that are described below in Section 8.4 of this chapter, so the reader may prefer to delay detailed study of that section until completing Section 8.4.

## 8.4 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.

### 8.4.1 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, but it is not possible to figure out what the original data pattern was. Thus, 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 7[on-line] 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 $2^3 = 8$ possible patterns for this 3-bit quantity, only 4 of which represent legitimate data. As illustrated in Figure 8.4, 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 8.4**

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 8.5, 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 $P_1$, $P_2$, and $P_4$ 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 $P_4\ P_2\ P_1$ in order, representing the troubled bits as ONES and untroubled bits as ZEROS, 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.)

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 8.4, 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 8.5. 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

| | bit | $P_7$ | $P_6$ | $P_5$ | $P_4$ | $P_3$ | $P_2$ | $P_1$ |
|---|---|---|---|---|---|---|---|---|
| Choose $P_1$ so XOR of every other bit ($P_7 \oplus P_5 \oplus P_3 \oplus P_1$) is 0 | | $\oplus$ | | $\oplus$ | | $\oplus$ | | $\oplus$ |
| Choose $P_2$ so XOR of every other pair ($P_7 \oplus P_6 \oplus P_3 \oplus P_2$) is 0 | | $\oplus$ | $\oplus$ | | | $\oplus$ | $\oplus$ | |
| Choose $P_4$ so XOR of every other four ($P_7 \oplus P_6 \oplus P_5 \oplus P_4$) is 0 | | $\oplus$ | $\oplus$ | $\oplus$ | $\oplus$ | | | |

**FIGURE 8.5**

A single-error-correction code. In the table, the symbol $\oplus$ 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.

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 8.5. (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 (see Section 2.1.1.4) 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 8.9). 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 8.6. 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.

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 8.8 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 school children 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.

### 8.4.2 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.

**FIGURE 8.6**

Update of a sector on disk 2 of a five-disk RAID 4 system. The old parity sector contains parity ← data 1 ⊕ data 2 ⊕ data 3 ⊕ 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 XORS. 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

$$\text{new parity} \leftarrow \text{old parity} \oplus \text{old data 2} \oplus \text{new data 2}$$

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 8.7, 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.

### 8.4.3 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

**FIGURE 8.7**

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.

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 8.2.2, 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,

$$R_{supermodule} = R^3 + 3R^2(1-R) = 3R^2 - 2R^3 \qquad \text{Eq. 8–10}$$

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

$$(1-R_{supermodule}) = (1-3R^2+2R^3)$$

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 8.8. 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.*

The numerical effect of fail-vote TMR is impressive. If the reliability of a single module at time $T$ is 0.999, equation 8–10 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 8.9 illustrates the reliability of our hypo-

**FIGURE 8.8**

Triple-modular redundant supermodule, with three inputs, three voters, and three outputs.

thetical 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 8.10 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 8.2 elaborates.)

**FIGURE 8.9**

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 8.10**

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 8.2: Risks of manipulating MTTFs** The apparently casual manipulation of MTTFs in Sections 8.4.3 and 8.4.4 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 —> 1.5 weeks between failure

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 1/2 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.

MTTF's 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 fail-fast replicas and a cleverer voter, as described in Section 8.4.4 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,

$$MTTF_{system} = 1 + 1/2 + 1/3 + \ldots(1/N) \qquad \text{Eq. 8–11}$$

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*.

### 8.4.4 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 1 of the 2 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

$$Pr(\text{supermodule fails while waiting for repair}) = \frac{2 \times MTTR}{MTTF} \qquad \text{Eq. 8–12}$$

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):

$$MTTF_{\text{supermodule}} = \frac{MTTF_{replica}}{3} \times \frac{MTTF_{replica}}{2 \times MTTR_{replica}} = \frac{(MTTF_{replica})^2}{6 \times MTTR_{replica}} \qquad \text{Eq. 8–13}$$

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 eq. 8–13, the MTTF of the data is

$$\frac{(MTTF_{replica})^2}{6 \times MTTR_{replica}} = \frac{(5 \text{ years})^2}{6 \cdot (10 \text{ hours})/(8760 \text{ hours/year})} = 3650 \text{ years} \qquad \text{Eq. 8–14}$$

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 10[on-line] 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 8–13 we have $MTTR_{replica} = 1$ (in this application, equation 8–13 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 8.11 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.

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

**FIGURE 8.11**

Failure points of three different 5MR supermodule designs, if repair does not happen in time.

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 $N + 1$ *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 8.4.1, or some more general replica management system such as those discussed in Section 10.3.7.

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.

## 8.5 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: state whose integrity must be preserved and state that can casually be abandoned because it is easy to reconstruct.

### 8.5.1 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, described in Section 4.4). 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.

### 8.5.2 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 9[on-line].
- 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 10[on-line].

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 10.3[on-line]. The three techniques taken together make DNS quite fault tolerant.

### 8.5.3 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 (see Table 2.1 in Section 2.5.1) 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 short compared with the expected operational lifetime of non-volatile 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 non-volatile 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 7.5.3) 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 10.3[on-line] 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.

### 8.5.4 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 (see Sidebar 2.8) 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 8.8.1. 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.

#### *8.5.4.1 Magnetic Disk Fault Modes*

Sidebar 2.8 described the physical design of the magnetic disk, including platters, magnetic material, read/write heads, seek arms, tracks, cylinders, and sectors, but it did not make any mention of disk reliability. There are several considerations:

- 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 re-reading or re-writing 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*.

#### *8.5.4.2 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.

#### *8.5.4.3 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.

#### *8.5.4.4 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 it 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.)

#### *8.5.4.5 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 8.12 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.

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 GETs and PUTs that specify 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.

#### *8.5.4.6 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 prob-

```
procedure CAREFUL_GET (data, sector_number)
    for i from 1 to NTRIES do
        if FAIL_FAST_GET (data, sector_number) = OK then
            return OK
    return BAD

procedure CAREFUL_PUT (data, sector_number)
    for i from 1 to NTRIES do
        if FAIL_FAST_PUT (data, sector_number) = OK then
            return OK
    return BAD
```

**FIGURE 8.12**

Procedures that implement careful disk storage.

ability 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 of positioning errors because they were all masked by a lower layer.

One configuration of RAID (see Section 2.1.1.4), 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.

#### *8.5.4.7 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 8.1 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).

#### *8.5.4.8 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 8.3: Are disk system checksums a wasted effort?** From the adjacent 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 8.4.1. 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 8.1 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.

#### *8.5.4.9 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 9[on-line].

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 10[on-line] 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

| | 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 |

**Table 8.1:** 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 9[on-line]

system has been designed using an overly simple disk failure model and thus misses some—or even many—straightforward failure masking opportunities.

## 8.6 Wrapping up Reliability

### 8.6.1 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 8.1.2, 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 11[on-line], to security) comes from experience, as documented in the case studies of Section 8.8:

**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.

### 8.6.2 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 8.8.1 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*.

### 8.6.3 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.

### 8.6.4 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 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 8.4 describes the *heartbeat*, a popular technique for detecting failures of active processes.

This chapter has oversimplified some ideas. For example, the definition of availability proposed in Section 8.2 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, Chapters 9[on-line] and 10[on-line] 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 11[on-line], 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.

**Sidebar 8.4: 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.

## 8.7 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 $O(p^n)$ means terms involving $p^n$ and higher, presumably negligible, powers. Here are the possibilities and their associated probabilities:

**Fault tolerance model for raw CMOS random access memory**

| | | probability |
|---|---|---|
| error-free case: | all 32 bits are correct | $(1-p)^{32} = 1 - O(p)$ |
| errors: | | |
| untolerated: | one bit is in error: | $32p(1-p)^{31} = O(p)$ |
| untolerated: | two bits are in error: | $(31 \cdot 32/2)p^2(1-p)^{30} = O(p^2)$ |
| untolerated: | three or more bits are in error: | $(30 \cdot 31 \cdot 32/3 \cdot 2)p^3(1-p)^{29} + \ldots + p^{32} = O(p^3)$ |

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 8.4.1. 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:

**Fault tolerance model for CMOS memory with error correction**

| | | probability |
|---|---|---|
| error-free case: | all 32 bits are correct | $(1-p)^{32} = 1 - O(p)$ |
| errors: | | |
| tolerated: | one bit corrected: | $32p(1-p)^{31} = O(p)$ |
| detected: | two bits are in error: | $(31 \cdot 32/2)p^2(1-p)^{30} = O(p^2)$ |
| untolerated: | three or more bits are in error: | $(30 \cdot 31 \cdot 32/3 \cdot 2)p^3(1-p)^{29} + \ldots + p^{32} = O(p^3)$ |

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:

$$\begin{aligned} Prob(\text{decoded value is correct}) &= (1-p)^{32} + 32p(1-p)^{31} \\ &= (1 - 32p + (31 \cdot 32/2)p^2 + \ldots) + (32p + 31 \cdot \\ &= (1 - O(p^2)) \end{aligned}$$

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 on page 8–19. 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 $n$th 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 is yes.

## 8.8 War Stories: Fault Tolerant Systems that Failed

### 8.8.1 Adventures with Error Correction*

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. But, 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

* 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.

things. One of the first steps in that direction was to implement the first what-you-see-is-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 8.8.1a: 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 8.8.1b: 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 8.8.1c: 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 8.8.1d: Never assume that the hardware actually does what it says in the specifications.*
*Lesson 8.8.1e: 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 8.8.1f: 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 8.8.1g (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."*

### 8.8.2 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 and 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 house-

cleaning 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.*

*Lesson 8.8.2:* 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.*

### 8.8.3 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.

* George Lardner Jr. "Archives Loses 43,000 E-Mails; officials can't explain summer erasure; backup system failed." *The Washington Post,* Thursday, January 6, 2000, page A17.

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 cancelled because of the disruption, about the same number that are cancelled 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."*

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.†

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 8.8.3: 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.*

### 8.8.4 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

* Tony Kennedy. "Cut cable causes cancellations, delays for Northwest Airlines." *Minneapolis Star Tribune*, March 22, 2000.

† Peter G. Neumann. *Computer Related Risks* (Addison-Wesley, New York, 1995), page 14.

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.*

*Lesson 8.8.4a: 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 4:10pm 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."†

*Lesson 8.8.4b: 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 8.8.4a.*

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 11AM 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 10AM 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 11AM the smoldering protector triggered the VESDA® smoke sensing system

---

* Robert Uhlig. "Engineer pulls plug on secure bunker." *Electronic Telegraph,* (9 May 1997).

† 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.

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."*

*Lesson 8.8.4c: A failure masking system not only can fail, it can cause a bigger failure than the one it is intended to mask.*

### 8.8.5 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."†

*Lesson 8.8.5: 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.*

### 8.8.6 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."‡

---

* Private internal communication.

† 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.

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 8.8.6: 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.*

## Exercises

**8.1** 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.

*1999–3–01*

**8.2** 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

‡ 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>

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.

*2006–2–3*

**8.3** 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 Aand C.

*1997–0–01*

**8.4** 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?

*2005–2–5*

**8.5** (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?

*1995–3–1a*

**8.6** 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:

1. "Daisy Chain" 2. "Fully connected"

Each link in the network has a failure probability of $p$.

8.6a. What is the probability that the daisy chain network is connecting all the buildings?

8.6b. What is the probability that the fully connected network is connecting all the buildings?

8.6c. 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?

*1985–0–1*

**8.7** Figure 8.11 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.

*2001–3–05*

**8.8** 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 failing on any one operation of $10^{-12}$. (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 till 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.*

*1976–0–3*

**8.9** Draw a figure, similar to that of Figure 8.6, that shows the recovery procedure for one sector of a 5-disk RAID 4 system when disk 2 fails and is replaced.

*2005–0–1*

**8.10** 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 8.4.1. 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 modulo 5) = 4), while the parity block for sector 19 would be on

disk 5 (because 1 + (19 modulo 5) = 5). Louis is hoping you can help him understand why this idea might be a good one.

8.10a. 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, its just another advertising gimmick.

*1997–3–01*

8.10b. Is there any workload for which RAID 4 has better write performance than RAID 5?

*2000–3–01*

8.10c. Louis is also wondering about whether he might be better off using a RAID 1 system (see Section 8.5.4.6). How does the number of disks required compare between RAID 1 and RAID 5?

*1998–3–01*

8.10d. Which of RAID 1 and RAID 5 has better performance for a workload consisting of small reads and small writes?

*2000–3–01*

**8.11** 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?

*2002–3–01*

**Additional exercises relating to Chapter 8 can be found in problem sets *26* through *28*.**

# Glossary for Chapter 8

**active fault**—A fault that is currently causing an error. Compare with *latent fault*. [Ch. 8]

**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*. [Ch. 8]

**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*. [Ch. 8]

**Byzantine fault**—A fault that generates inconsistent errors (perhaps maliciously) that can confuse or disrupt fault tolerance or security mechanisms. [Ch. 8]

**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. [Ch. 4]

**coheerence**—See *read/write coherence* or *cache coherence*.

**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. [Ch. 8]

**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. [Ch. 8]

**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*. [Ch. 8]

**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*. [Ch. 8]

**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. [Ch. 8]

**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. [Ch. 8]

**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. [Ch. 8]

**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. [Ch. 8]

**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. [Ch. 8]

**error correction**—A scheme to set to the correct value a data value or control signal that is in error. Compare with *error detection*. [Ch. 8]

**error detection**—A scheme to discover that a data value or control signal is in error. Compare with *error correction*. [Ch. 8]

**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*. [Ch. 8]

**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. [Ch. 8]

**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*. [Ch. 8]

**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. [Ch. 8]

**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. [Ch. 8]

**fail-vote**—Describes an *N*-modular redundancy system with a majority voter. [Ch. 8]

**failure**—The outcome when a component or system does not produce the intended result at its interface. Compare with **fault**. [Ch. 8]

**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. [Ch. 8]

**fault**—A defect in materials, design, or implementation that may (or may not) cause an error and lead to a failure. (Compare with *failure*.) [Ch. 8]

**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*. [Ch. 8]

**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. [Ch. 8]

**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*. [Ch. 8]

**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. [Ch. 8]

**hot swap**—To replace modules in a system while the system continues to provide service. [Ch. 8]

**intermittent fault**—A persistent fault that is active only occasionally. Compare with *transient fault*. [Ch. 8]

**latency**—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. [Ch. 8]

**latent fault**—A fault that is not currently causing an error. Compare with *active fault*. [Ch. 8]

**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. [Ch. 8]

**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*. [Ch. 8]

**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. [Ch. 8]

**mean time between failures (MTBF)**—The sum of MTTF and MTTR for the same component or system. [Ch. 8]

**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. [Ch. 8]

**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. [Ch. 8]

**memoryless**—A property of some time-dependent probabilistic processes, that the

probability of what happens next does not depend on what has happened before. [Ch. 8]

**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." [Ch. 8]

**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. [Ch. 8]

**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. [Ch. 8]

**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. [Ch. 8]

**page fault**—See *missing-page exception*.

**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-and-compare 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.** [Ch. 8]

**pair-and-spare**—See *pair-and-compare*.

**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. [Ch. 8]

**persistent fault**—A fault that cannot be masked by retry. Compare with *transient fault* and *intermittent fault*. [Ch. 8]

**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*.

**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. [Ch. 8]

**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. [Ch. 8]

**redundancy**—Extra information added to detect or correct errors in data or control signals. [Ch. 8]

**reliability**—A statistical measure, the probability that a system is still operating at time t,

given that it was operating at some earlier time $t_0$. [Ch. 8]

**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. [Ch. 8]

**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. [Ch. 8]

**replication**—The technique of using multiple replicas to achieve fault tolerance. [Ch. 8]

**single-event upset**—A synonym for *transient fault*. [Ch. 8]

**soft state**—State of a running program that the program can easily reconstruct if it becomes necessary to abruptly terminate and restart the program. [Ch. 8]

**supermodule**—A set of replicated modules interconnected in such a way that it acts like a single module. [Ch. 8]

**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*. [Ch. 8]

**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 *single-event upset*. Compare with *persistent fault* and **intermittent fault**. [Ch. 8]

**triple-modular redundancy (TMR)**—$N$-modular redundancy with $N = 3$. [Ch. 8]

**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*. [Ch. 8]

**valid construction**—The term used by software designers for *fault avoidance*. [Ch. 8]

**voter**—A device used in some NMR designs to compare the output of several nominally identical replicas that all have the same input. [Ch. 8]

# Index of Chapter 8

Design principles and hints appear in *underlined italics*. Procedure names appear in SMALL CAPS. Page numbers in **bold face** are in the chapter Glossary.


## A

accelerated aging 8–12
active fault 8–4, **8–69**
*adopt sweeping simplifications* 8–8, 8–37, 8–51
availability 8–9, **8–69**
*avoid rarely used components* 8–51, 8–60

## B

backward error correction 8–22, **8–69**
bathtub curve 8–10
*be explicit* 8–7
burn in, burn out 8–11
Byzantine fault 8–53, **8–69**

## C

careful storage 8–45
close-to-open consistency **8–69**
coding 8–21
conditional failure rate function 8–14
consistency
  close-to-open **8–69**
continuous operation 8–35, **8–69**

## D

decay 8–41
  set 8–42, **8–69**
defense in depth 8–3
*design for iteration* 8–8, 8–11, 8–15, 8–37
*design principles*
  *adopt sweeping simplifications* 8–8, 8–37, 8–51
  *avoid rarely used components* 8–51, 8–60
  *be explicit* 8–7
  *design for iteration* 8–8, 8–11, 8–15, 8–37
  *end-to-end argument* 8–49, 8–52
  *keep digging principle* 8–8, 8–64
  *robustness principle* 8–15
  *safety margin principle* 8–8, 8–16, 8–58
detectable error 8–17, **8–69**
Domain Name System
  fault tolerance of 8–36, 8–39
down time 8–9, **8–69**
durability 8–39
durable storage 8–38, 8–46, **8–69**

## E

*end-to-end argument* 8–49, 8–52
erasure 8–23, **8–69**
ergodic 8–10, **8–70**
error 8–4, **8–70**
  containment 8–2, 8–5, **8–70**
  correction 8–2, 8–57, **8–70**
  detection 8–2, **8–70**

## F

fail-
  fast 8–5, 8–17, **8–70**
  safe 8–17, **8–70**
  secure 8–17, **8–70**
  soft 8–17, **8–70**
  stop 8–5, **8–70**
  vote 8–27, **8–70**
failure 8–4, **8–70**
  tolerance 8–16, **8–70**
fault 8–3, **8–70**
  avoidance 8–6, **8–71**
  tolerance 8–5, **8–71**
  tolerance design process 8–6
  tolerance model 8–18
forward

error correction 8–21, **8–71**

## H

Hamming distance 8–21, **8–71**
hard error 8–5
hazard function 8–14
heartbeat 8–54
hot swap 8–35, **8–71**

## I

incremental
redundancy 8–21
infant mortality 8–11
intermittent fault 8–5, **8–71**

## K

*keep digging principle* 8–8, 8–64

## L

latency 8–5, **8–71**
latent fault 8–4, **8–71**

## M

margin 8–20, **8–71**
maskable error 8–18, **8–71**
masking 8–2, 8–17, **8–71**
massive redundancy 8–25
mean time
between failures 8–9, **8–71**
to failure 8–9, **8–71**
to repair 8–9, **8–71**
memoryless 8–13, **8–71**
mirror **8–72**
module 8–2
MTBF (see mean time between failures)
MTTF (see mean time to failure)
MTTR (see mean time to repair)

## N

N + 1 redundancy 8–35, **8–72**
N-modular redundancy 8–26, **8–72**
N-version programming 8–36, **8–72**
NMR (see N-modular redundancy)

## P

pair-and-compare 8–33, **8–72**
pair-and-spare **8–72**
partition 8–34, **8–72**
persistent
fault 8–5, **8–72**
prepaging **8–72**
preservation 8–40
preventive maintenance 8–12, **8–72**
purging 8–33, **8–72**

## Q

quad component 8–26

## R

RAID
RAID 1 8–47
RAID 4 8–24
RAID 5 8–67
raw storage 8–42
recovery 8–38
recursive
replication 8–27
redundancy 8–2, **8–72**
reliability 8–13, **8–72**
repair 8–31, **8–73**
replica 8–26, **8–73**
replication **8–73**
recursive 8–27
revectoring 8–46
*robustness principle* 8–15

## S

*safety margin principle* 8–8, 8–16, 8–58
safety-net approach 8–7
single
-event upset 8–5, **8–73**
point of failure 8–63
Six sigma 8–15
soft
error 8–5
state **8–73**
storage
careful 8–45

durable 8–38, 8–46, **8–69**
fail-fast 8–43
raw 8–42
supermodule 8–27, **8–73**

## T

Taguchi method 8–16
TMR (see triple-modular redundancy)
tolerated error 8–18, **8–73**
transient fault 8–5, **8–73**
triple-modular redundancy 8–26, **8–73**

## U

untolerated error 8–18, **8–73**

## V

valid construction 8–37, **8–73**
validation (see valid construction)
voter 8–26, **8–73**

## W

watchdog 8–54

## Y

yield (in manufacturing) 8–11

CHAPTER

# Atomicity: All-or-Nothing and Before-or-After

# 9

## CHAPTER CONTENTS


**Overview ..... 9–2**
**9.1 Atomicity ..... 9–4**
9.1.1 All-or-Nothing Atomicity in a Database ..... 9–5
9.1.2 All-or-Nothing Atomicity in the Interrupt Interface ..... 9–6
9.1.3 All-or-Nothing Atomicity in a Layered Application ..... 9–8
9.1.4 Some Actions With and Without the All-or-Nothing Property ..... 9–10
9.1.5 Before-or-After Atomicity: Coordinating Concurrent Threads ..... 9–13
9.1.6 Correctness and Serialization ..... 9–16
9.1.7 All-or-Nothing and Before-or-After Atomicity ..... 9–19
**9.2 All-or-Nothing Atomicity I: Concepts ..... 9–21**
9.2.1 Achieving All-or-Nothing Atomicity: ALL_OR_NOTHING_PUT ..... 9–21
9.2.2 Systematic Atomicity: Commit and the Golden Rule ..... 9–27
9.2.3 Systematic All-or-Nothing Atomicity: Version Histories ..... 9–30
9.2.4 How Version Histories are Used ..... 9–37
**9.3 All-or-Nothing Atomicity II: Pragmatics ..... 9–38**
9.3.1 Atomicity Logs ..... 9–39
9.3.2 Logging Protocols ..... 9–42
9.3.3 Recovery Procedures ..... 9–45
9.3.4 Other Logging Configurations: Non-Volatile Cell Storage ..... 9–47
9.3.5 Checkpoints ..... 9–51
9.3.6 What if the Cache is not Write-Through? (Advanced Topic) ..... 9–53
**9.4 Before-or-After Atomicity I: Concepts ..... 9–54**
9.4.1 Achieving Before-or-After Atomicity: Simple Serialization ..... 9–54
9.4.2 The Mark-Point Discipline ..... 9–58
9.4.3 Optimistic Atomicity: Read-Capture (Advanced Topic) ..... 9–63
9.4.4 Does Anyone Actually Use Version Histories for Before-or-After Atomicity? ..... 9–67
**9.5 Before-or-After Atomicity II: Pragmatics ..... 9–69**
9.5.1 Locks ..... 9–70
9.5.2 Simple Locking ..... 9–72
9.5.3 Two-Phase Locking ..... 9–73

9.5.4 Performance Optimizations ..... 9–75
9.5.5 Deadlock; Making Progress ..... 9–76
**9.6 Atomicity across Layers and Multiple Sites.....9–79**
9.6.1 Hierarchical Composition of Transactions ..... 9–80
9.6.2 Two-Phase Commit ..... 9–84
9.6.3 Multiple-Site Atomicity: Distributed Two-Phase Commit ..... 9–85
9.6.4 The Dilemma of the Two Generals ..... 9–90
**9.7 A More Complete Model of Disk Failure (Advanced Topic) .....9–92**
9.7.1 Storage that is Both All-or-Nothing and Durable ..... 9–92
**9.8 Case Studies: Machine Language Atomicity.....9–95**
9.8.1 Complex Instruction Sets: The General Electric 600 Line ..... 9–95
9.8.2 More Elaborate Instruction Sets: The IBM System/370 ..... 9–96
9.8.3 The Apollo Desktop Computer and the Motorola M68000 Microprocessor ..... 9–97
**Exercises.....9–98**
**Glossary for Chapter 9 .....9–107**
**Index of Chapter 9 .....9–113**
**Last chapter page 9–115**

## 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 8[on-line] introduced failure masking, but did not show how to mask failures of running programs. Chapter 5 introduced coordination of concurrent activities, and presented solutions to several specific problems, but it did not explain any systematic way to ensure that actions have the before-or-after property. 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 happen 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—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 9.1: 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 10[on-line] 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 10[on-line] 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.

## 9.1 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.

| Area | All-or-nothing atomicity | Before-or-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 |

### 9.1.1 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 8.5.4, 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 9.2.1 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 PUTs. 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 9.2.3 we will devise a TRANSFER procedure that has the all-or-nothing property, and in Section 9.3 we will see some additional ways of providing the property.

### 9.1.2 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. Recall from Chapters 2 and 5 that 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, as described in Section 5.6.4, 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 9.2 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?

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 program-

**Sidebar 9.2: 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…
   - 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.

mer 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 9.1.3 and 9.1.4 explore the case in which the exception terminates the running thread, thus creating a fault. Section 9.1.5 examines the case in which the interrupted thread continues, oblivious (one hopes) to the interruption.

### 9.1.3 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, recall the layered structure of the calendar management program of Chapter 2, reproduced in Figure 9.19.1 (that figure may seem familiar—it is a copy of Figure 2.10). 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 9.1**

An application system with three layers of interpretation. 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:

---

**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.

### 9.1.4 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 9.8 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). Section 5.3.4 pointed out that the SVC instruction, which 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), 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-thing 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 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 "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.

One of the appeals of the client/server model introduced in Chapter 4 is that it tends to force the all-or-nothing property out onto the design table. Because servers can fail independently of clients, it is necessary for the client to think through a plan for recovery

from server failure, and a natural model to use is to make every action offered by a server all-or-nothing.

### 9.1.5 Before-or-After Atomicity: Coordinating Concurrent Threads

In Chapter 5 we learned how to 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 9.1.2 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, Chapter 5 introduced 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 the eventcounts of Chapter 5.

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:

---

**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.**

---

In Chapter 5 we saw how before-or-after actions can be created with explicit locks and a thread manager that implements the procedures ACQUIRE and RELEASE. Chapter 5 showed some examples of before-or-after actions using locks, and emphasized that programming correct before-or-after actions, for example 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 "$X \leftarrow X + Y$" 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 9.2, which illustrates several possible time sequences of the READ and WRITE steps of the two threads with respect to variable $B$.

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*.

**FIGURE 9.2**

Six possible histories of variable *B* if two threads that share *B* do not coordinate their concurrent activities.

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.

### 9.1.6 Correctness and Serialization

The notion that the first two sequences of Figure 9.2 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.

The reasoning behind this concept of correctness involves several steps. Consider Figure 9.3,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:

**FIGURE 9.3**

A single action takes a system from one state to another state.

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 9.4, 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 9.5. 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.

**FIGURE 9.4**

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.

So long as the only coordination requirement is before-or-after atomicity, any serialization will do.

Moreover, we do not even need to insist that the system actually traverse the intermediate states along any particular path of Figure 9.5—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.

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

**FIGURE 9.5**

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*.

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 9.2, 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 9.2, 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 Sections 9.4 and 9.5 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, 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

[*] The general question of whether or not a collection of existing transactions is serializable is an advanced topic that is addressed in database management. Problem set *36* explores one method of answering this question.

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

*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.

### 9.1.7 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:

**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 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.

## 9.2 All-or-Nothing Atomicity I: Concepts

Section 9.1 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 introduced in Chapter 5 called *bootstrapping*, a method that resembles inductive proof. To review, 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. Recall that Chapter 5 tackled the general problem of creating before-or-after actions from arbitrary sequences of code by implementing a procedure named ACQUIRE that itself required before-or-after atomicity of two or three lines of code where it reads and then sets a lock value. It then implemented that before-or-after action with the help of a special hardware feature that directly makes a before-or-after action of the read and set sequence, and it also exhibited a software implementation (in Sidebar 5.2) that relies only on the hardware performing ordinary LOADS and STORES as before-or-after actions. 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 9.3 work.

### 9.2.1 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 8[on-line], 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 7[on-line]. 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 PUTS to the disk.
- untolerated error: Something fails in the hardware or operating system. The processor muddles along and PUTS 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 8.5.4.2: 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 8[on-line]—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 8.5.4.5), 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 9.7, 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 situa-

```
procedure ALMOST_ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)
    CAREFUL_PUT (data, all_or_nothing_sector.S1)
    CAREFUL_PUT (data, all_or_nothing_sector.S2)          // Commit point.
    CAREFUL_PUT (data, all_or_nothing_sector.S3)

procedure ALL_OR_NOTHING_GET (reference data, all_or_nothing_sector)
    CAREFUL_GET (data1, all_or_nothing_sector.S1)
    CAREFUL_GET (data2, all_or_nothing_sector.S2)
    CAREFUL_GET (data3, all_or_nothing_sector.S3)
    if data1 = data2 then data ← data1                    // Return new value.
    else data ← data3                                     // Return old value.
```

**FIGURE 9.6**

Algorithms for ALMOST_ALL_OR_NOTHING_PUT and ALL_OR_NOTHING_GET.

tion: 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 9.6 shows this almost correct pseudocode.

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,

```
procedure ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)
    CHECK_AND_REPAIR (all_or_nothing_sector)
    ALMOST_ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)

procedure CHECK_AND_REPAIR (all_or_nothing_sector) // Ensure copies match.
    CAREFUL_GET (data1, all_or_nothing_sector.S1)
    CAREFUL_GET (data2, all_or_nothing_sector.S2)
    CAREFUL_GET (data3, all_or_nothing_sector.S3)
    if (data1 = data2) and (data2 = data3) return     // State 1 or 7, no repair
    if (data1 = data2)
        CAREFUL_PUT (data1, all_or_nothing_sector.S3) return    // State 5 or 6.
    if (data2 = data3)
        CAREFUL_PUT (data2, all_or_nothing_sector.S1) return    // State 2 or 3.
    CAREFUL_PUT (data1, all_or_nothing_sector.S2)       // State 4, go to state 5
    CAREFUL_PUT (data1, all_or_nothing_sector.S3)       // State 5, go to state 7
```

**FIGURE 9.7**

Algorithms for ALL_OR_NOTHING_PUT and CHECK_AND_REPAIR.

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 9.7. CHECK_AND_REPAIR simply compares the three copies and, if they are not identical, it forces them to be identical. 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 | 2 | 3 | 4 | 5 | 6 | 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 9.7), 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 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 8.5. 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 8[on-line] 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 9.7, 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.

### 9.2.2 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_PUTS 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 9.8, 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.

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 9.9. 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 9.8**

Imaginary semantics for painless programming of all-or-nothing actions.

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

**FIGURE 9.9**

The commit point of an all-or-nothing action.

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 9.3 explores that possibility.)

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 9.7 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.

**Sidebar 9.3: 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 10[online] introduces a simple version of cascaded abort.

- 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 such as the ones described for the UNIX version of RENAME in Section 2.5.8.
- Post-commit: Release the space that was occupied by the original.

The ALL_OR_NOTHING_PUT algorithm of Figure 9.7 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 structure ensures that the implementation follows the golden rule.

### 9.2.3 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 9.3 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 Sections 9.4 and 9.5 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 8[on-line] 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 1950's, 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 9.10 illustrates. Whenever any action

**FIGURE 9.10**

Version history of a variable in journal storage.

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 9.11 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.

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 9.12. No other state transitions are possible, except to discard the outcome record once

**FIGURE 9.11**

Interface to and internal organization of an all-or-nothing storage system based on version histories and journal storage.

**FIGURE 9.12**

The allowed state transitions of an outcome record.

```
procedure NEW_ACTION ()
    id ← NEW_OUTCOME_RECORD ()
    id.outcome_record.state ← PENDING
    return id

procedure COMMIT (reference id)
    id.outcome_record.state ← COMMITTED

procedure ABORT (reference id)
    id.outcome_record.state ← ABORTED
```

**FIGURE 9.13**

The procedures NEW_ACTION, COMMIT, and ABORT.

there is no further interest in its state. Figure 9.13 illustrates implementations of the three 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 9.14. Now,

**FIGURE 9.14**

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.

```
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
                   else signal ("Tried to write outside of an all-or-nothing action!")
```

**FIGURE 9.15**

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.

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.

The procedures READ_CURRENT_VALUE and WRITE_NEW_VALUE thus follow the algorithms of Figure 9.15. 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

```
procedure TRANSFER (reference debit_account, reference credit_account,
                                                                  amount)
    my_id ← NEW_ACTION ()
    xvalue ← READ_CURRENT_VALUE (debit_account, my_id)
    xvalue ← xvalue - amount
    WRITE_NEW_VALUE (debit_account, xvalue, my_id)
    yvalue ← READ_CURRENT_VALUE (credit_account, my_id)
    yvalue ← yvalue + amount
    WRITE_NEW_VALUE (credit_account, yvalue, my_id)
    if xvalue > 0 then
        COMMIT (my_id)
    else
        ABORT (my_id)
        signal("Negative transfers are not allowed.")
```

**FIGURE 9.16**

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.)

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 9.2.1, 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.

Figure 9.16 shows the TRANSFER procedure of Section 9.1.5 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 9.7. (If there is concurrency, updates to the internal structures of the version history also need before-or-after atomicity. Section 9.4 will explore methods of providing it.)

### 9.2.4 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 9.4 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.

## 9.3 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 9.4 describes.

### 9.3.1 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 9.11. Figure 9.17 illustrates this two-step implementation.

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 9.18. In each of these three configurations, the log resides in non-volatile storage. For the *in-memory*

* A hardware architect would say "…it *graduates* the change to cell storage". This text, somewhat arbitrarily, chooses to use the database management term "install" .

**Sidebar 9.4: 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.

**FIGURE 9.17**

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.

**FIGURE 9.18**

Three common logging configurations. Arrows show data flow as the application reads, logs, and installs data.

*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.

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 9.7, which used a recovery procedure named CHECK_AND_REPAIR.)

### 9.3.2 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 9.17. 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 9.17, LOG and INSTALL are the logging implementation of the WRITE_NEW_VALUE part of the interface of Figure 9.11, and READ_CURRENT_VALUE is simply a READ from cell storage. We also need a logging implementation of the remaining parts of the Figure 9.11 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 9.19 shows our by now familiar TRANSFER action 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 9.20 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.

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

```
procedure TRANSFER (debit_account, credit_account, amount)
    my_id ← LOG (BEGIN_TRANSACTION)
    dbvalue.old ← GET (debit_account)
    dbvalue.new ← dbvalue.old - amount
    crvalue.old ← GET (credit_account, my_id)
    crvalue.new ← crvalue.old + amount
    LOG (CHANGE, my_id,
        "PUT (debit_account, dbvalue.new)",                 //redo action
        "PUT (debit_account, dbvalue.old)" )                //undo action
    LOG ( CHANGE, my_id,
        "PUT (credit_account, crvalue.new)"                 //redo action
        "PUT (credit_account, crvalue.old)")                //undo action
    PUT (debit_account, dbvalue.new)                        // install
    PUT (credit_account, crvalue.new)                       // install
    if dbvalue.new > 0 then
        LOG ( OUTCOME, COMMIT, my_id)
    else
        LOG (OUTCOME, ABORT, my_id)
        signal("Action not allowed. Would make debit account negative.")
    LOG (END_TRANSACTION, my_id)
```

**FIGURE 9.19**

An all-or-nothing TRANSFER procedure, implemented with logging.

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 proce-

**FIGURE 9.20**

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.

dure 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 9.21 illustrates.

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.

### 9.3.3 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 9.18. 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

```
procedure ABORT (action_id)
    starting at end of log repeat until beginning
        log_record ← previous record of log
        if log_record.id = action_id then
            if (log_record.type = OUTCOME)
                then signal ("Can't abort an already completed action.")
            if (log_record.type = CHANGE)
                then perform undo_action of log_record
            if (log_record.type = BEGIN)
                then break repeat
    LOG (action_id, OUTCOME, ABORTED)                    // Block future undos.
    LOG (action_id, END)
```

**FIGURE 9.21**

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.

```
procedure RECOVER () // Recovery procedure for a volatile, in-memory database.
    winners ← NULL
    starting at end of log repeat until beginning
        log_record ← previous record of log
        if (log_record.type = OUTCOME)
            then winners ← winners + log_record                    // Set addition.

    starting at beginning of log repeat until end
        log_record ← next record of log
        if (log_record.type= CHANGE)
            and (outcome_record ← find (log_record.action_id) in winners)
            and (outcome_record.status = COMMITTED) then
            perform log_record.redo_action
```

**FIGURE 9.22**

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.

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 REDOs 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 9.22 illustrates.

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 9.21 will satisfy this requirement if the individual undo actions are themselves idempotent.

### 9.3.4 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 9.18. 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 9.18. 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 9.3.6, 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 9.18 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 9.22. 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 9.23. 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.

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

```
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 9.23**

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.

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

```
procedure RECOVER ()            // Recovery procedure for rollback recovery.
    completeds ← NULL
    losers ← NULL
    starting at end of log repeat until beginning          // Perform undo scan.
        log_record ← previous record of log
        if (log_record.type = OUTCOME)
            then completeds ← completeds + log_record      // Set addition.
        if (log_record.action_id is not in completeds) then
            losers ← losers + log_record                   // New loser.
            if (log_record.type = CHANGE) then
                perform log_record.undo_action

    for each log_record in losers do
        log (log_record.action_id, OUTCOME, ABORT)         // Block future undos.
```

**FIGURE 9.24**

An idempotent undo-only recovery procedure for rollback logging.

redo installs for any committed action that has logged its END record. A useful exercise is to modify the procedure of Figure 9.23 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 9.24. 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.

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 non-volatile 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.

### 9.3.5 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 9.23 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.

### 9.3.6 What if the Cache is not Write-Through? (Advanced Topic)

Between the log and the write-through cache, the logging configurations just described require, for every data update, two synchronous writes to non-volatile storage, 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 non-volatile 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.

## 9.4 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 mechanism 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. Thus, 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 9.1.5 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. So we are looking for techniques that guarantee to produce 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 9.4 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 9.5 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.

### 9.4.1 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 9.25 shows this version of BEGIN_TRANSACTION. The scheme forces all transactions to execute in the serial order that threads happen to invoke BEGIN_TRANSACTION. Since that

```
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 9.25**

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 9.30.

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.

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 9.26 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.

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 it

**FIGURE 9.26**

Version history of a banking system.

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 9.26 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.

**FIGURE 9.27**

System state history with unchanged values shown.

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 9.26 represents the state history of the entire system in serialization order, but the slightly different representation of Figure 9.27 makes that state history more explicit. In Figure 9.27 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.

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 9.27, 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 unim-

portant 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 9.27 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 9.27 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*.

### 9.4.2 The Mark-Point Discipline

Concurrent threads that invoke READ_CURRENT_VALUE as implemented in Figure 9.15 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 9.28. 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.

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 a transaction needs as inputs, 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 9.27, 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 9.15 into two parts, named NEW_VERSION and WRITE_VALUE, as in Figure 9.29. 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.

Finally, we enforce the mark point discipline by putting a test and, depending on its outcome, a wait in BEGIN_TRANSACTION, as in Figure 9.30, 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 9.30 also illustrates an implementation of MARK_POINT_ANNOUNCE. No changes are needed in procedures ABORT and COMMIT as shown in Figure 9.13, so they are not repeated here.

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

```
procedure READ_CURRENT_VALUE (data_id, this_transaction_id)
    starting at end of data_id repeat until beginning
        v ← previous version of data_id
        last_modifier ← v.action_id
        if last_modifier ≥ this_transaction_id then skip v        // Keep searching
        wait until (last_modifier.outcome_record.state ≠ PENDING)
        if (last_modifier.outcome_record.state = COMMITTED)
            then return v.state
            else skip v                                           // Resume search
    signal ("Tried to read an uninitialized variable")
```

**FIGURE 9.28**

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.

```
procedure NEW_VERSION (reference data_id, this_transaction_id)
    if this_transaction_id.outcome_record.mark_state = MARKED then
        signal ("Tried to create new version after announcing mark point!")
    append new version v to data_id
    v.value ← NULL
    v.action_id ← transaction_id

procedure WRITE_VALUE (reference data_id, new_value, this_transaction_id)
    starting at end of data_id repeat until beginning
        v ← previous version of data_id
        if v.action_id = this_transaction_id
            v.value ← new_value
            return
    signal ("Tried to write without creating new version!"))
```

**FIGURE 9.29**

Mark-point discipline versions of NEW_VERSION and WRITE_VALUE.

```
procedure BEGIN_TRANSACTION ()
    id ← NEW_OUTCOME_RECORD (PENDING)
    previous_id ← id - 1
    wait until (previous_id.outcome_record.mark_state = MARKED)
        or (previous_id.outcome_record.state ≠ PENDING)
    return id

procedure NEW_OUTCOME_RECORD (starting_state)
    ACQUIRE (outcome_record_lock)        // Make this a before-or-after action.
    id ← TICKET (outcome_record_sequencer)
    allocate id.outcome_record
    id.outcome_record.state ← starting_state
    id.outcome_record.mark_state ← NULL
    RELEASE (outcome_record_lock)
    return id

procedure MARK_POINT_ANNOUNCE (reference this_transaction_id)
    this_transaction_id.outcome_record.mark_state ← MARKED
```

**FIGURE 9.30**

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 in the *starting_state* COMMITTED and immediately calling MARK_POINT_ANNOUNCE for the empty transaction.

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 9.30 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 of Section 5.6.3 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 9.15 to be atomic under both failure and concurrent activity, as in Figure 9.31. 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.

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

```
procedure TRANSFER (reference debit_account, reference credit_account,
                                                              amount)
    my_id ← BEGIN_TRANSACTION ()
    NEW_VERSION (debit_account, my_id)
    NEW_VERSION (credit_account, my_id)
    MARK_POINT_ANNOUNCE (my_id);
    xvalue ← READ_CURRENT_VALUE (debit_account, my_id)
    xvalue ← xvalue - amount
    WRITE_VALUE (debit_account, xvalue, my_id)
    yvalue ← READ_CURRENT_VALUE (credit_account, my_id)
    yvalue ← yvalue + amount
    WRITE_VALUE (credit_account, yvalue, my_id)
    if xvalue > 0 then
        COMMIT (my_id)
    else
        ABORT (my_id)
        signal("Negative transfers are not allowed.")
```

**FIGURE 9.31**

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.

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 9.5 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 disci-

pline 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 9.29.

### 9.4.3 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 makaing 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 9.27 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 9.32, which has the same version history as did Figure 9.27.

**FIGURE 9.32**

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 9.32. 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.

```
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 9.33**

Read-capture forms of READ_CURRENT_VALUE, NEW_VERSION, and WRITE_VALUE.

Read-capture is relatively easy to implement in a version history system. We start, as shown in Figure 9.33, 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.

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.

### 9.4.4 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 the x86 instruction set architecture described in Section 5.7, 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 9.29. 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 9.34 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.

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 [Suggestions for Further Reading 1.1.1].

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

**FIGURE 9.34**

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 $n + 1$.

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 9.4.3 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. Before-or-after atomicity by using locks systematically is the subject of the next section of this chapter.

## 9.5 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 introduced in Chapter 5—*locks*—for before-or-after atomicity. However, as Chapter 5 pointed out, 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.

### 9.5.1 Locks

To review, 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 the single-acquire protocol of Chapter 5: 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 9.5.4 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 9.5). 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 9.4. 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 9.16, 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*.

### 9.5.2 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 to which it adds each transaction identifier as soon as the transaction reaches its lock point and removes it 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 9.16, which needs to lock only two records, both of which are known at the outset) simple locking can be effective.

### 9.5.3 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. Further, 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 [Suggestions for Further Reading 1.1.5].

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 *Y*. Because the lock sets of $T_1$ and $T_2$ intersect at variable *Y*, 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 *Y*
$T_1$: 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. 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 non-overlapping lock sets at the moment that the lock values vanished. The recovery algorithm of Figure 9.23 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.

### 9.5.4 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 9.3.6 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.

### 9.5.5 Deadlock; Making Progress

Section 5.2.5 of Chapter 5 introduced the emergent problem of *deadlock*, the wait-for graph as a way of analyzing deadlock, and lock ordering as a way of preventing deadlock. With transactions and the ability to undo individual actions or even abort a transaction completely we now have more tools available to deal with deadlock, so it is worth revisiting that discussion.

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). As suggested in Chapter 5, 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 it, since the transaction that previously acquired it cannot be waiting for any locks that this transaction has already acquired—all those locks are lower in number than this one. 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 (as described in Section 5.2.5) 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 the emergent property that Chapter 5 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 Chapter 7[on-line]: *exponential random backoff*. When a timer expiration leads to an abort, the lock manager, after clearing the locks, 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.

## 9.6 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

```
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 9.35**

An example of two procedures, one of which calls the other, yet each should be individually atomic.

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*.

### 9.6.1 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 9.1.5 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 9.35. 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.

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 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 because 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 TRANSFERs 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 TRANSFERs 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 9.4.2) examines the sta-

tus 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, if necessary all the way to the highest-layer outcome record. 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 9.4.2, doesn't distinguish between reads arising within the same group of nested transactions and reads from some

completely unrelated transaction. Figure 9.36 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

**FIGURE 9.36**

Transaction $\text{TRANSFER}_2$, nested in transaction $\text{PAY_INTEREST}_2$, which is nested in transaction MONTH_END_INTEREST, wants to read the current value of account *bank*. But *bank* was last written by transaction $\text{TRANSFER}_1$, which is nested in COMMITTED transaction $\text{PAY_INTEREST}_1$, 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 $\text{TRANSFER}_1$. However, $\text{TRANSFER}_1$ and $\text{TRANSFER}_2$ 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.

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.

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 9.36, 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.

### 9.6.2 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 either COMMITs or ABORTs 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 9.36 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.

### 9.6.3 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 4 we learned of a method, known as Remote Procedure Call (RPC) for performing an action at another site. In Chapter 7[on-line] we learned how to design protocols such as RPC 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 9.37: 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.

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 Chapter 7[on-line] 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

**FIGURE 9.37**

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.

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, can *not* 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*.

### 9.6.4 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 well-enough defended 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

* 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.

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 8[on-line] 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.

## 9.7 A More Complete Model of Disk Failure (Advanced Topic)

Section 9.2 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.

### 9.7.1 Storage that is Both All-or-Nothing and Durable

In Chapter 8[on-line] 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 8[on-line]. That method would certainly work but it is a bit heavy-handed: with a replication count of just two, it would lead to allo-

cating 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 8[on-line] 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 9.38 shows the algorithm and the two replicas of the data, here named *D0* and *D1*.

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 9.6 and 9.7, 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 8[on-line], 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.

```
procedure ALL_OR_NOTHING_DURABLE_GET (reference data, atomic_sector)
    ds ← CAREFUL_GET (data, atomic_sector.D0)
    if ds = BAD then
        ds ← CAREFUL_GET (data, atomic_sector.D1)
    return ds

procedure ALL_OR_NOTHING_DURABLE_PUT (new_data, atomic_sector)
    SALVAGE(atomic_sector)
    ds ← CAREFUL_PUT (new_data, atomic_sector.D0)
    ds ← CAREFUL_PUT (new_data, atomic_sector.D1)
    return ds

procedure SALVAGE(atomic_sector)        //Run this program every T_d seconds.
    ds0 ← CAREFUL_GET (data0, atomic_sector.D0)
    ds1 ← CAREFUL_GET (data1, atomic_sector.D1)
    if ds0 = BAD then
        CAREFUL_PUT (data1, atomic_sector.D0)
    else if ds1 = BAD then
        CAREFUL_PUT (data0, atomic_sector.D1)
    if data0 ≠ data1 then
        CAREFUL_PUT (data0, atomic_sector.D1)
```

$D_0$: [ $data_0$ ]  $D_1$: [ $data_1$ ]

**FIGURE 9.38**

Data arrangement and algorithms to implement all-or-nothing durable storage on top of the careful storage layer of Figure 8.12.

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 $2 \times 10^7$ 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 back 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 1.8.7]. It raises one minor concern: 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.

## 9.8 Case Studies: Machine Language Atomicity

### 9.8.1 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 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 it was actually 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 "half-way 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.

### 9.8.2 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 and 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.

### 9.8.3 The Apollo Desktop Computer and the Motorola M68000 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.

## Exercises

**9.1** *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;
```

9.1a. 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*.

9.1b. Add calls to ACQUIRE and RELEASE that obey the *two-phase locking protocol*, and in addition postpone all ACQUIREs as late as possible and do all RELEASEs 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
```

9.1c. As usual, Louis has misunderstood some aspect of the problem. Give a brief explanation of what is wrong with this idea.

*1995–3–2a…c*

**9.2** 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?

*2006-0-1*

**9.3** 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 9.30 to create a version of NEW_OUTCOME_RECORD that doesn't need the ticket primitive.

**9.4** 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.

*1997–0–02*

**9.5** 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 $y \leftarrow 4$.

9.5a. After both transactions complete, what are the values of *x* and *y*? In what sense is this answer wrong?

*1982–3–3a*

9.5b. In the mid-1970's, 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.

*1982–3–3b*

**9.6** 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 value *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, 5, 9>
<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 **************
```

9.6a. Assume that the system is using a rollback recovery procedure. How much farther back in the log should the recovery procedure scan?

9.6b. Assume that the system is using a roll-forward recovery procedure. How much farther back in the log should the recovery procedure scan?

9.6c. Which operations mentioned in this part of the log are winners and which are losers?

9.6d. What are the values of *x* and *y* immediately after the recovery procedure finishes? Why?

*1994–3–3*

**9.7** The log of exercise 9.6 contains (perhaps ambiguous) evidence that someone didn't follow coordination rules. What is that evidence?

*1994–3–4*

**9.8** 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.

*1994–3–5*

**9.9** 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.

*1997–3–03*

**9.10** Pat, Diane, and Quincy are having trouble using e-mail to schedule meetings. Pat suggests that they take inspiration from the 2-phase commit protocol.

9.10a. Which of the following protocols most closely resembles 2-phase commit?

```
I.  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.

II. a–c, as in protocol I.
    d. Everyone, if they received the second message,
            acknowledge receipt.
        Otherwise
            send a message to Pat asking what happened.

III a–c, as in protocol I.
    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.

IV. a–f, as in protocol III.
    g. Pat sends a message telling everyone that everyone has confirmed.
    h. Everyone acknowledges the confirmation.
```

9.10b. For the protocol you selected, which step commits the meeting time?

*1994–3–7*

**9.11** Alyssa P. Hacker needs a transaction processing system for updating information about her collection of 97 cockroaches.*

9.11a. 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?

9.11b. *I*n 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:

Transactions T2, T3, and T5 committed before the crash, but T1 and T4 were still pending.

9.11c. 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,

* Credit for developing exercise 9.11 goes to Eddie Kohler.

mark off in the table whether that transaction needs to be undone, redone, or neither.

| | Undone | Redone | Neither |
|---|---|---|---|
| T1 | | | |
| T2 | | | |
| T3 | | | |
| T4 | | | |
| T5 | | | |

9.11d. 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 | | | |

*1996–3–3*

**9.12** 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.

*2008–3–11*

**Additional exercises relating to Chapter 9 can be found in problem sets *29* through *40*.**

# Glossary for Chapter 9

**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*. [Ch. 9]

**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*. [Ch. 9]

**archive**—A record, usually kept in the form of a log, of old data values, for auditing, recovery from application mistakes, or historical interest. [Ch. 9]

**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 before-or-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-or-nothing, 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*. [Ch. 9]

**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*. [Ch. 9]

**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. Compare with *all-or-nothing atomicity* and *atomic*. [Ch. 9]

**blind write**—An update to a data value *X* by a transaction that did not previously read *X*. [Ch. 9]

**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*. [Ch. 9]

**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. [Ch. 9]

**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. [Ch. 4]

**coheerence**—See *read/write coherence* or *cache coherence*.

**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-or-nothing 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*. [Ch. 9]

**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. [Ch. 9]

**do action**—(n.) Term used in some systems for a *redo action*. [Ch. 9]

**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. [Ch. 9]

**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 write-through usually have a feature that allows the invoker to force some or all of their contents to the secondary storage medium. [Ch. 9]

**install**—In a system that uses logs to achieve all-or-nothing atomicity, to write data to cell storage. [Ch. 9]

**journal storage**—Storage in which a WRITE or PUT appends a new value, rather than overwriting a previously stored value. Compare with *cell storage*. [Ch. 9]

**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. [Ch. 9]

**lock set**—The collection of all locks acquired during the execution of a before-or-after action. [Ch. 9]

**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. [Ch. 9]

**logical locking**—Locking of higher-layer data objects such as records or fields of a database. Compare with *physical locking*. [Ch. 9]

**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. [Ch. 9]

**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*. [Ch. 9]

**page fault**—See *missing-page exception*.

**pair-and-spare**—See *pair-and-compare*.

**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. [Ch. 9]

**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*. [Ch. 9]

**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*. [Ch. 9]

**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. [Ch. 9]

**presented load**—See *offered load*.

**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. [Ch. 9]

**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*.) [Ch. 9]

**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*. [Ch. 9]

**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*. [Ch. 9]

**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. [Ch. 9]

**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. [Ch. 9]

**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*. [Ch. 9]

**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. [Ch. 9]

**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. [Ch. 9]

**transactional memory**—A memory model in which multiple references to primary memory are both all-or-nothing and before-or-after. [Ch. 9]

**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. [Ch. 9]

**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*. [Ch. 9]

**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*. [Ch. 9]

**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*. [Ch. 9]

**version history**—The set of all values for an object or variable that have ever existed, stored in journal storage. [Ch. 9]

**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. [Ch. 9]

**write tearing**—See *atomic storage*.

# Index of Chapter 9

Design principles and hints appear in *underlined italics*. Procedure names appear in SMALL CAPS. Page numbers in **bold face** are in the chapter Glossary.


## A

abort 9–27, **9–107**
ACQUIRE 9–70
action 9–3
*adopt sweeping simplifications* 9–3, 9–29, 9–30, 9–47
all-or-nothing atomicity 9–21, **9–107**
archive 9–37, **9–107**
  log 9–40
atomic **9–107**
  action 9–3, **9–107**
  storage **9–107**
atomicity 9–3, 9–19, **9–107**
  all-or-nothing 9–21, **9–107**
  before-or-after 9–54, **9–107**
  log 9–40

## B

backoff
  exponential random 9–78, **9–107**, **9–108**
before-or-after atomicity 9–54, **9–107**
blind write 9–49, 9–66, **9–107**
blocking read 9–11
bootstrapping 9–21, 9–43, 9–61, 9–80

## C

cell
  storage 9–31, **9–107**, **9–108**
checkpoint 9–51, **9–107**, **9–108**
close-to-open consistency **9–107**, **9–108**
commit 9–27, **9–107**, **9–108**
  two-phase 9–84, **9–107**, **9–111**
compensation **9–107**, **9–108**
consistency
  close-to-open **9–107**, **9–108**
  external time 9–18
  sequential 9–18

## D

deadlock 9–76
dependent outcome record 9–81
*design principles*
  *adopt sweeping simplifications* 9–3, 9–29, 9–30, 9–47
  *end-to-end argument* 9–79
  *golden rule of atomicity* 9–26, 9–42
  *law of diminishing returns* 9–53
dilemma of the two generals 9–90, **9–107**, **9–111**
*diminishing returns, law of* 9–53
discipline
  simple locking 9–72, **9–107**, **9–110**
  systemwide locking 9–70
  two-phase locking 9–73, **9–107**, **9–111**
do action (see redo action)
dry run 9–97
durability
  log 9–40

## E

*end-to-end argument* 9–79
exponential
  random backoff 9–78, **9–107**, **9–108**
external time consistency 9–18

## F

force 9–53, **9–107**, **9–108**

## G

*golden rule of atomicity* 9–26, 9–42
granularity 9–71

**H**
high-water mark 9–65
*hints*
*optimize for the common case* 9–39

**I**
idempotent 9–47
IMS (see Information Management System)
in-memory database 9–39
Information Management System 9–100
install 9–39, **9–107**, **9–108**

**J**
journal storage 9–31, **9–107**, **9–108**

**L**
*law of diminishing returns* 9–53
livelock 9–78
lock 9–69
compatibility mode 9–76
manager 9–70
point 9–72, **9–107**, **9–108**
set 9–72, **9–107**, **9–109**
locking discipline
simple 9–72, **9–107**, **9–110**
systemwide 9–70
two-phase 9–73, **9–107**, **9–111**
log 9–39, **9–107**, **9–109**
archive 9–40
atomicity 9–40
durability 9–40
performance 9–40
record 9–42
redo 9–50, **9–107**, **9–110**
sequence number 9–53
undo 9–50, **9–107**, **9–110**
write-ahead 9–42, **9–107**, **9–111**
logical
locking 9–75, **9–107**, **9–109**

**M**
mark point 9–58, **9–107**, **9–109**
memory
transactional 9–69, **9–107**, **9–111**
multiple
-reader, single-writer protocol 9–76

**N**
nested outcome record 9–86
non-blocking read 9–12

**O**
optimistic concurrency control 9–63, **9–107**, **9–109**
optimize for the common case 9–45
*optimize for the common case* 9–39
outcome record 9–32

**P**
pending 9–32, **9–107**, **9–109**
performance log 9–40
pessimistic concurrency control 9–63, **9–107**, **9–109**
physical
locking 9–75, **9–107**, **9–109**
prepaging **9–107**, **9–109**
PREPARED
message 9–87
state **9–107**, **9–109**
presumed commit 9–88
progress 9–77, **9–107**, **9–110**
protocol
two-phase commit 9–84, **9–107**, **9–111**

**R**
random
backoff, exponential 9–78, **9–107**, **9–108**
read-capture 9–63
redo
action 9–43, **9–107**, **9–110**
log 9–50, **9–107**, **9–110**
register renaming 9–67
RELEASE 9–70
reorder buffer 9–67
representations
version history 9–55
roll-forward recovery 9–50, **9–107**, **9–110**

rollback recovery 9–50, **9–107**, **9–110**

## S

sequence coordination 9–13
sequential consistency 9–18
serializable 9–18, **9–107**, **9–110**
shadow copy 9–29, **9–107**, **9–110**
simple
  locking discipline 9–72, **9–107**, **9–110**
  serialization 9–54, **9–107**, **9–110**
single-writer, multiple-reader protocol 9–76
snapshot isolation 9–68
storage
  atomic **9–107**
  cell 9–31, **9–107**, **9–108**
  journal 9–31, **9–107**, **9–108**
sweeping simplifications
  (see *adopt sweeping simplifications)*
systemwide lock 9–70

## T

tentatively committed 9–82
transaction 9–3, 9–4, **9–107**, **9–110**
transactional memory 9–69, **9–107**, **9–111**
TRANSFER operation 9–5
two generals dilemma 9–90, **9–107**, **9–111**
two-phase
  commit 9–84, **9–107**, **9–111**
  locking discipline 9–73, **9–107**, **9–111**

## U

undo
  action 9–43, **9–107**, **9–111**
  log 9–50, **9–107**, **9–110**

## V

version history 9–30, **9–107**, **9–111**

## W

WAL (see write-ahead log)
write-ahead log 9–42, **9–107**, **9–111**
write tearing **9–107**

CHAPTER

# Consistency

# 10

## CHAPTER CONTENTS


**Overview** ..... **10–2**
**10.1 Constraints and Interface Consistency** ..... **10–2**
**10.2 Cache Coherence** ..... **10–4**
10.2.1 Coherence, Replication, and Consistency in a Cache ..... 10–4
10.2.2 Eventual Consistency with Timer Expiration ..... 10–5
10.2.3 Obtaining Strict Consistency with a Fluorescent Marking Pen .. 10–7
10.2.4 Obtaining Strict Consistency with the Snoopy Cache ..... 10–7
**10.3 Durable Storage Revisited: Widely Separated Replicas** ..... **10–9**
10.3.1 Durable Storage and the Durability Mantra ..... 10–9
10.3.2 Replicated State Machines ..... 10–11
10.3.3 Shortcuts to Meet more Modest Requirements ..... 10–13
10.3.4 Maintaining Data Integrity ..... 10–15
10.3.5 Replica Reading and Majorities ..... 10–16
10.3.6 Backup ..... 10–17
10.3.7 Partitioning Data ..... 10–18
**10.4 Reconciliation** ..... **10–19**
10.4.1 Occasionally Connected Operation ..... 10–20
10.4.2 A Reconciliation Procedure ..... 10–22
10.4.3 Improvements ..... 10–25
10.4.4 Clock Coordination ..... 10–26
**10.5 Perspectives** ..... **10–26**
10.5.1 History ..... 10–27
10.5.2 Trade-Offs ..... 10–28
10.5.3 Directions for Further Study ..... 10–31
**Exercises** ..... **10–32**
**Glossary for Chapter 10** ..... **10–35**
**Index of Chapter 10** ..... **10–37**
**Last chapter page 10–38**

## 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 10.1 introduces constraints and discusses how transactions can be used to maintain invariants and implement memory models that provide interface consistency. Sections 10.2 and 10.3 develop techniques used in two different application areas, *caching* and *geographically distributed replication*, to achieve higher performance and greater durability, respectively. Section 10.4 discusses *reconciliation*, which is a way of restoring the constraint that replicas be identical if their contents should drift apart. Finally, Section 10.5 considers some perspectives relating to Chapters 9[on-line] and 10.

## 10.1 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[on-line], 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, and it reopens the discussion of memory coherence and data consistency models introduced in Section 2.1.1.1. 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, for example in banking and safety-critical systems. Section 9.1.6 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 cat-

alogs 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 10.2 provides the details. The Internet Domain Name System, described in Section 4.4 and revisited in Section 10.2.2, 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 10.3 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 10.4, 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.

## 10.2 Cache Coherence

### 10.2.1 Coherence, Replication, and Consistency in a Cache

Chapter 6 described the cache as an example of a multilevel memory system. A cache 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, as defined in Section 2.1.1.1, for the entire name space of the cache:

- **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.

### 10.2.2 Eventual Consistency with Timer Expiration

The Internet Domain Name System, whose basic operation was described in Section 4.4, 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. Recall that 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.

### 10.2.3 Obtaining Strict Consistency with a Fluorescent Marking 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.

### 10.2.4 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. But 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 10.1.1]—each cache is snooping on bus activity. Figure 10.1 illustrates the snoopy cache.

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 [Suggestions for Further Reading 1.1.1].

## 10.3 Durable Storage Revisited: Widely Separated Replicas

### 10.3.1 Durable Storage and the Durability Mantra

Chapter 8[on-line] demonstrated how to create durable storage using a technique called *mirroring*, and Section 9.7[on-line] 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

**FIGURE 10.1**

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).

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…*

Sidebar 4.5 referred to Ross Anderson's Eternity Service, a system that makes use of this design principle. Another formulation of the durability mantra is "lots of copies keep stuff safe" [Suggestions for Further Reading 10.2.3]. 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

* 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.*

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.

### 10.3.2 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 10.2. To maintain identical data replicas, co-locate with each of those replicas a replica of the state machine, and send 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.

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 10.2, 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.

**FIGURE 10.2**

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 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.

### 10.3.3 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 9[on-line].)
- If the database is large, partition it into small regions, each of which can be updated independently. Section 10.3.7, 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 e-mail 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 e-mail 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.

### 10.3.4 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 on page 9–94 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 11.7) 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 (page 7–31) and in the link layer (page 7–40); 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 11[on-line] we will see that by separately protecting a witness one can also counter threats to data integrity that are posed by an adversary.

### 10.3.5 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 9[on-line].

### 10.3.6 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.)

### 10.3.7 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 compro-

mises 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.

## 10.4 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.

### 10.4.1 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 on page 9–63 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.

### 10.4.2 A Reconciliation Procedure

To illustrate some of the issues involved in reconciliation, Figure 10.3 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.

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*),

```
procedure RECONCILE (reference left, reference right,
                                    reference last_reconcile_time)
    quiesce all activity on left and right    // Shut down all file-using applications
    ALL_OR_NOTHING_GET (last_reconcile_time, reconcile_time_sector)
    left_list ← enumerate(left)
    right_list ← enumerate(right)
    common_list ← intersect(left_list, right_list)
    left_only_list ← remove members of common_list from left_list
    right_only_list ← remove members of common_list from right_list
    conflict_list ← NIL

    for each named_file in common_list do   // Reconcile files found both sides
        left_new ← (MODIFICATION_TIME (left.named_file) > last_reconcile_time)
        right_new ← (MODIFICATION_TIME (right.named_file) > last_reconcile_time)
        if left_new and right_new then
            add named_file to conflict_list
        else if left_new then
            copy named_file from left to right
        else if right_new then
            copy named_file from right to left
        else if MODIFICATION_TIME (left.named_file) ≠
                    (MODIFICATION_TIME (right.named_file)
            then TERMINATE ("Something awful has happened.")

    for each named_file in left_only_list do   // Reconcile files found one side
        if MODIFICATION_TIME (left.named_file) > last_reconcile_time then
            copy named_file from left to right
        else
            delete left.named_file
    for each named_file in right_only_list do
        if MODIFICATION_TIME (right.named_file) > last_reconcile_time then
            copy named_file from right to left
        else
            delete right.named_file

    for each named_file in conflict_list do          // Handle conflicts
        MANUALLY_RESOLVE (right.named_file, left.named_file)
    last_reconcile_time ← NOW ()
    ALL_OR_NOTHING_PUT (last_reconcile_time, reconcile_time_sector)
    Allow activity to resume on left and right
```

**FIGURE 10.3**

A simple reconciliation algorithm.

- 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.

### 10.4.3 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.

### 10.4.4 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.

## 10.5 Perspectives

In [on-line] Chapters 9 and 10 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.

### 10.5.1 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 time-sharing 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 on-line 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.

### 10.5.2 Trade-Offs

An interesting set of trade-offs applies to techniques for coordinating concurrent activities. Figure 10.4 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.)

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 [on-line] Chapters 9 and 10 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 inter-

ference 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 9.5.4[on-line] 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

**FIGURE 10.4**

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.

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 10.4 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 1950's and 1960's, the mini-computers of the 1970's, and the one-chip microprocessors of the 1980's) 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.

### 10.5.3 Directions for Further Study

Chapters 9 and 10 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 10.4, 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, for example 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 11[on-line].

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.

## Exercises

**10.1** 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.

10.1a. 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?

10.1b. Describe a modification that solves this problem.

10.1c. 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.

*1994–3–2*

**10.2** What assumptions does the design of the RECONCILE procedure of Section 10.4.2 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.

*1999–3–04*

**10.3** 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 e-mail inbox, reading and deleting messages, and preparing replies, which go into an e-mail 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.

*1997–0–03*

**10.4** Which of the following statements are true of the RECONCILE program of Figure 10.3?

**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.

*2008–3–2*

**10.5** Here is a proposed invariant for a file reconciler such as the program RECONCILE of Figure 10.3: 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.

**Additional exercises relating to Chapter 10 can be found in problem sets *40* through *42*.**

# Glossary for Chapter 10

**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*. [Ch. 10]

**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. [Ch. 10]

**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. [Ch. 4]

**coheerence**—See *read/write coherence* or *cache coherence*.

**consensus**—Agreement at separated sites on a data value despite communication failures. [Ch. 10]

**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*. [Ch. 10]

**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. [Ch. 10]

*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. [Ch. 10]

**fingerprint**—Another term for a *witness*. [Ch. 10]

**incremental backup**—A backup copy that contains only data that has changed since making the previous backup copy. [Ch. 10]

**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. [Ch. 10]

**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. [Ch. 10]

**master**—In a multiple-site replication scheme, the site to which updates are directed. Compare with *slave*. [Ch. 10]

**page fault**—See *missing-page exception*.

**pair-and-spare**—See *pair-and-compare*.

**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*. [Ch. 10]

**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*.

**presented load**—See *offered load*.

**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*.) [Ch. 10]

**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. [Ch. 10]

**reconciliation**—A procedure that compares replicas that are intended to be identical and repairs any differences. [Ch. 10]

**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. [Ch. 10]

**slave**—In a multiple-site replication scheme, a site that takes update requests from only the master site. Compare with *master*. [Ch. 10]

**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. [Ch. 10]

**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*. [Ch. 10]

**witness**—A (usually cryptographically strong) hash value that attests to the content of a file. Another widely used term for this concept is **fingerprint**. [Ch. 10]

# Index of Chapter 10

Design principles and hints appear in *underlined italics*. Procedure names appear in SMALL CAPS. Page numbers in **bold face** are in the chapter Glossary.


**A**

*adopt sweeping simplifications* 10–11

**B**

backup copy 10–10, **10–35**

**C**

cache
  coherence 10–4, **10–35**
  snoopy 10–8, **10–36**
close-to-open consistency **10–35**
coherence
  cache 10–4, **10–35**
compensation 10–31
conflict 10–19
consensus 10–11, **10–35**
  the consensus problem 10–11
consistency **10–35**
  close-to-open **10–35**
  eventual 10–3
  strict 10–3, **10–36**
  strong (see consistency, strict)
constraint 10–2, **10–35**
cursor stability 10–30

**D**

data integrity
  in storage 10–15
*design principles*
  *adopt sweeping simplifications* 10–11
  *durability mantra* 10–10
  *end-to-end argument* 10–30
Domain Name System
  eventual consistency in 10–5
*durability mantra* 10–10

**E**

*end-to-end argument* 10–30
eventual consistency 10–3, **10–35**

**F**

fingerprint **10–35**

**I**

incremental
  backup 10–18, **10–35**
invalidate 10–7, **10–35**

**L**

logical
  copy 10–10, **10–35**

**M**

master 10–10, **10–35**
mirror 10–9

**O**

occasionally connected 10–20

**P**

partition 10–18
physical
  copy 10–10, **10–35**
prepaging **10–36**
primary
  copy 10–10, **10–36**

**Q**

quorum 10–16, **10–36**

**R**

reconciliation 10–12, 10–19, **10–36**

replicated state machine 10–11, **10–36**

## S

single
  state machine 10–13
slave 10–10, **10–36**
snoopy cache 10–8, **10–36**
stability
  cursor 10–30
strict consistency 10–3, **10–36**
strong consistency (see strict consistency)

## T

temporal
  database 10–28
time domain addressing 10–28
time-to-live 10–6
TTL (see time-to-live)

## W

witness 10–21, **10–36**

CHAPTER

# Information Security

# 11

**Information 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.

— *Information Operations.* Joint Chiefs of Staff of the United States Armed Forces, Joint Publication 3-13 (13 February 2006).

## CHAPTER CONTENTS


**Overview ..... 11–4**
**11.1 Introduction to Secure Systems ..... 11–5**
11.1.1 Threat Classification ..... 11–7
11.1.2 Security is a Negative Goal ..... 11–9
11.1.3 The Safety Net Approach ..... 11–10
11.1.4 Design Principles ..... 11–13
11.1.5 A High d(technology)/dt Poses Challenges For Security ..... 11–17
11.1.6 Security Model ..... 11–18
11.1.7 Trusted Computing Base ..... 11–26
11.1.8 The Road Map for this Chapter ..... 11–28
**11.2 Authenticating Principals ..... 11–28**
11.2.1 Separating Trust from Authenticating Principals ..... 11–29
11.2.2 Authenticating Principals ..... 11–30
11.2.3 Cryptographic Hash Functions, Computationally Secure, Window of Validity ..... 11–32
11.2.4 Using Cryptographic Hash Functions to Protect Passwords ..... 11–34
**11.3 Authenticating Messages ..... 11–36**
11.3.1 Message Authentication is Different from Confidentiality ..... 11–37
11.3.2 Closed versus Open Designs and Cryptography ..... 11–38
11.3.3 Key-Based Authentication Model ..... 11–41
11.3.4 Properties of SIGN and VERIFY ..... 11–41

11.3.5 Public-key versus Shared-Secret Authentication .....11–44
11.3.6 Key Distribution .....11–45
11.3.7 Long-Term Data Integrity with Witnesses .....11–48
**11.4 Message Confidentiality .....11–49**
11.4.1 Message Confidentiality Using Encryption .....11–49
11.4.2 Properties of ENCRYPT and DECRYPT .....11–50
11.4.3 Achieving both Confidentiality and Authentication .....11–52
11.4.4 Can Encryption be Used for Authentication? .....11–53
**11.5 Security Protocols .....11–54**
11.5.1 Example: Key Distribution .....11–54
11.5.2 Designing Security Protocols .....11–60
11.5.3 Authentication Protocols .....11–63
11.5.4 An Incorrect Key Exchange Protocol .....11–66
11.5.5 Diffie-Hellman Key Exchange Protocol .....11–68
11.5.6 A Key Exchange Protocol Using a Public-Key System .....11–69
11.5.7 Summary .....11–71
**11.6 Authorization: Controlled Sharing .....11–72**
11.6.1 Authorization Operations .....11–73
11.6.2 The Simple Guard Model .....11–73
11.6.2.1 The Ticket System.....11–74
11.6.2.2 The List System .....11–74
11.6.2.3 Tickets Versus Lists, and Agencies.....11–75
11.6.2.4 Protection Groups .....11–76
11.6.3 Example: Access Control in UNIX .....11–76
11.6.3.1 Principals in UNIX.....11–76
11.6.3.2 ACLs in UNIX .....11–77
11.6.3.3 The Default Principal and Permissions of a Process.....11–78
11.6.3.4 Authenticating Users .....11–79
11.6.3.5 Access Control Check.....11–79
11.6.3.6 Running Services .....11–80
11.6.3.7 Summary of UNIX Access Control .....11–80
11.6.4 The Caretaker Model .....11–80
11.6.5 Non-Discretionary Access and Information Flow Control .....11–81
11.6.5.1 Information Flow Control Example.....11–83
11.6.5.2 Covert Channels .....11–84
**11.7 Advanced Topic: Reasoning about Authentication .....11–85**
11.7.1 Authentication Logic .....11–86
11.7.1.1 Hard-wired Approach.....11–88
11.7.1.2 Internet Approach.....11–88
11.7.2 Authentication in Distributed Systems .....11–89
11.7.3 Authentication across Administrative Realms .....11–90
11.7.4 Authenticating Public Keys .....11–92
11.7.5 Authenticating Certificates .....11–94
11.7.6 Certificate Chains .....11–97
11.7.6.1 Hierarchy of Central Certificate Authorities .....11–97

11.7.6.2 Web of Trust ..... 11–98
**11.8 Cryptography as a Building Block (Advanced Topic) ..... 11–99**
11.8.1 Unbreakable Cipher for Confidentiality (*One-Time Pad*) ..... 11–99
11.8.2 Pseudorandom Number Generators ..... 11–101
11.8.2.1 Rc4: A Pseudorandom Generator and its Use ..... 11–101
11.8.2.2 Confidentiality using RC4 ..... 11–102
11.8.3 Block Ciphers ..... 11–103
11.8.3.1 Advanced Encryption Standard (AES) ..... 11–103
11.8.3.2 Cipher-Block Chaining ..... 11–105
11.8.4 Computing a Message Authentication Code ..... 11–106
11.8.4.1 MACs Using Block Cipher or Stream Cipher ..... 11–107
11.8.4.2 MACs Using a Cryptographic Hash Function ..... 11–107
11.8.5 A Public-Key Cipher ..... 11–109
11.8.5.1 Rivest-Shamir-Adleman (RSA) Cipher ..... 11–109
11.8.5.2 Computing a Digital Signature ..... 11–111
11.8.5.3 A Public-Key Encrypting System ..... 11–112
**11.9 .Summary ..... 11–112**
**11.10 Case Study: Transport Layer Security (TLS) for the Web . 11–116**
11.10.1 The TLS Handshake ..... 11–117
11.10.2 Evolution of TLS ..... 11–120
11.10.3 Authenticating Services with TLS ..... 11–121
11.10.4 User Authentication ..... 11–123
**11.11 War Stories: Security System Breaches ..... 11–125**
11.11.1 Residues: Profitable Garbage ..... 11–126
11.11.1.1 1963: Residues in CTSS ..... 11–126
11.11.1.2 1997: Residues in Network Packets ..... 11–127
11.11.1.3 2000: Residues in HTTP ..... 11–127
11.11.1.4 Residues on Removed Disks ..... 11–128
11.11.1.5 Residues in Backup Copies ..... 11–128
11.11.1.6 Magnetic Residues: High-Tech Garbage Analysis ..... 11–129
11.11.1.7 2001 and 2002: More Low-tech Garbage Analysis ..... 11–129
11.11.2 Plaintext Passwords Lead to Two Breaches ..... 11–130
11.11.3 The Multiply Buggy Password Transformation ..... 11–131
11.11.4 Controlling the Configuration ..... 11–131
11.11.4.1 Authorized People Sometimes do Unauthorized Things 11–132
11.11.4.2 The System Release Trick ..... 11–132
11.11.4.3 The Slammer Worm ..... 11–132
11.11.5 The Kernel Trusts the User ..... 11–135
11.11.5.1 Obvious Trust ..... 11–135
11.11.5.2 Nonobvious Trust (Tocttou) ..... 11–136
11.11.5.3 Tocttou 2: Virtualizing the DMA Channel. ..... 11–136
11.11.6 Technology Defeats Economic Barriers ..... 11–137
11.11.6.1 An Attack on Our System Would be Too Expensive ..... 11–137
11.11.6.2 Well, it Used to be Too Expensive ..... 11–137
11.11.7 Mere Mortals Must be Able to Figure Out How to Use it ..... 11–138

11.11.8 The Web can be a Dangerous Place ..... 11–139
11.11.9 The Reused Password ..... 11–140
11.11.10 Signaling with Clandestine Channels ..... 11–141
11.11.10.1 Intentionally I: Banging on the Walls..... 11–141
11.11.10.2 Intentionally II ..... 11–141
11.11.10.3 Unintentionally ..... 11–142
11.11.11 It Seems to be Working Just Fine ..... 11–142
11.11.11.1 I Thought it was Secure..... 11–143
11.11.11.2 How Large is the Key Space…Really? ..... 11–144
11.11.11.3 How Long are the Keys? ..... 11–145
11.11.12 Injection For Fun and Profit ..... 11–145
11.11.12.1 Injecting a Bogus Alert Message to the Operator..... 11–146
11.11.12.2 CardSystems Exposes 40,000,000 Credit Card Records to SQL Injection..... 11–146
11.11.13 Hazards of Rarely-Used Components ..... 11–148
11.11.14 A Thorough System Penetration Job ..... 11–148
11.11.15 Framing Enigma ..... 11–149
**Exercises.....11–151**
**Glossary for Chapter 11 .....11–163**
**Index of Chapter 11 .....11–169**
**Last chapter page 11–171**

## 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 8[on-line], *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**

*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 11.1.1), why security is a negative goal (Section 11.1.2), presents the safety net approach (Section 11.1.3), lays out principles for designing secure computer systems (Section 11.1.4), the basic model for structuring secure computer systems (Section 11.1.6), an implementation strategy based on minimizing the trusted computing base (Section 11.1.7), and concludes with a road map for the rest of this chapter (Section 11.1.8). The rest of the chapter works the ideas introduced in the next section 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.

## 11.1 Introduction to Secure Systems

In Chapter 4 we saw how to divide a computer system into modules so that errors don't propagate from one module to another. In the presentation, we assumed that errors happen *unintentionally*: modules fail to adhere to their contracts because users make mistakes or hardware fails accidently. 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 11.1). 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 11.1: 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 1.1.6]. Some privacy advocates (see for example Suggestions for Further Reading 11.1.2) 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, on-line 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.

### 11.1.1 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 Suggestions for Further Reading 1.1.7 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.

### 11.1.2 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 11.1, the possible states and links form a graph, with the

* 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.

**FIGURE 11.1**

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.

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.

### 11.1.3 The Safety Net Approach

To successfully design systems that satisfy negative goals, this chapter adopts the safety net approach of Chapter 8[on-line], 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. Section 11.1.7 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 11.5.3].

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.

### 11.1.4 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 11.11 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 11.2 summarizes some of the arguments.

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

**Sidebar 11.2: 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.

(*sidebar continues*)

---

* A.C Hobbs (Charles Tomlinson, ed.), *Locks and Safes: The Construction of Locks*. Virtue & Co., London, 1853 (revised 1868).

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 customer priviate 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 vunerability, 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.

finalized and security researchers began to examine the standard, they immediately found several problems, one of which is described on page 11–51.

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 comprised, 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 on page 11–5, 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 objec-

tives 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 opportunies 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).

### 11.1.5 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 11.2 and Section 11.1) 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 cryptogra-

phy. 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 on-line. 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 on-line 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 11.3). 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.

### 11.1.6 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

* 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 11.6.6.

**Sidebar 11.3: 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 [Suggestions for Further Reading 1.3.5]. 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 e-mail (SPAM) on behalf of paying customers, which generates a revenue stream for the adversary [Suggestions for Further Reading 11.6.5].*

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 naïvety 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 e-mail message with a malicious executable attachment. The adversary sent the e-mail to a few recipients. When a recipient opened the executable e-mail (attracted by "ILOVEYOU" in the e-mail'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 e-mail that it spread rapidly and overwhelmed e-mail servers at many institutions.

The Morris *worm* [Suggestions for Further Reading 11.6.1], 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 11.4) in an e-mail 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.

---

* Problem set *47* explores a potential stamp-based solution.

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 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 11.2). 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

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.

**FIGURE 11.2**

The security model based on complete mediation. The authenticity question includes both verifying the integrity and the source of 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 on-line 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 11.4), 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.

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 e-mail 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) maintain-

**Sidebar 11.4: 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*.

(*Sidebar continues*)

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. O*ne 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 as described in Section 5.4.5, 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

(*Sidebar continues*)

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.

ing 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.

The archives and the audit trail can be used to recover from a security breach. If an inspection of the service reveals that something bad has happened, the archived copies can be used to restore the data. The audit trail may help in figuring out what happened (e.g., what data has been damaged) and which principal did it. As mentioned earlier, the audit trail might also be useful as a proof in court to punish adversaries. These measures can be viewed as an example of defense in depth—if the first line of defense fails, one hopes that the next measure will help.

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 11.6.4 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 10.3[on-line]) 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.

### 11.1.7 Trusted Computing Base

Implementing the security model of Section 11.1.6 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 page 11–77), 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 11.1.4 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 11.5) 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 11.3.3 and 11.3.4].

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 and 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 design method also implies that when a bug is detected and repaired, the designer must review the assumptions to see which ones were wrong or missing, repair the assumptions, and repeat this process until sufficient confidence in the security of the system has been obtained. This approach flushes out any fuzzy thinking, makes the system more reliable, and slowly builds confidence that the system is correct.

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.

### 11.1.8 The Road Map for this Chapter

The rest of this chapter follows the security model of Figure 11.2. Section 11.2 presents techniques for authenticating principals. Section 11.2 explains how to authenticate messages by using a pair of procedures named SIGN and VERIFY. Section 11.4 explains how to keep messages confidential using a pair of procedures named ENCRYPT and DECRYPT. Section 11.5 explains how to set up, for example, an authenticated and secure communication link using security protocols. Section 11.6 discusses different designs for an authorization service. Because authentication is the foundation of security, Section 11.5 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 11.8. The case study in Section 11.10 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 11.11 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.

## 11.2 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 11.3. 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 11.1, 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.

**FIGURE 11.3**

Authentication model.

### 11.2.1 Separating Trust from Authenticating Principals

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 real-world 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.

### 11.2.2 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 11.3, 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 driving license. In that case, the administrator relies on the agency that issued the passport or driving 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.

### 11.2.3 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 *cryptographic 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 11.8 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 referrred to as "SHA-0" and which produces a 160-bit value for $V$ was first published in 1993, and 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.

### 11.2.4 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 11.2.3, 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;

---

* 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.

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 11.4, and the design of security protocols in Section 11.5. With these mechanisms one can design among many other things a secure password protocol.

## 11.3 Authenticating Messages

Returning to Figure 11.3, 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 11.2.

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.

### 11.3.1 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.

**FIGURE 11.4**

A closed design for authentication relies on the secrecy of an algorithm.

- 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*.

### 11.3.2 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 11.4) 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 11.4 shows how one can implement a secure channel that also provides confidentiality.)

Before diving in the details of how to implement SIGN and VERIFY, lets 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.

C*ryptographic 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 11.8), 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 with that idea that if the adversary doesn't understand how SIGN and VERIFY work, it will be difficult to break the tag. Auguste Kerchkoffs more than a century ago* observed 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 11.5 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)

* "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).

**FIGURE 11.5**

An open design for authentication relies on the secrecy of keys.

it is easy to change the secret parts (i.e., the two keys) without having to reanalyze the system's strength.

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*.

### 11.3.3 Key-Based Authentication Model

Returning to Figure 11.5, 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 it 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 11.5, 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.

### 11.3.4 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. VERIFY $(M', T', K)$ returns ACCEPT if $M' = M$, $T' =$ SIGN $(M, K)$
2. Without knowing $K$, it is difficult for an adversary to compute an $M'$ and $T'$ such that 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. VERIFY $(M', T', K_2)$ returns ACCEPT if $M' = M$, $T' =$ SIGN $(M, K_1)$
2. Without knowing $K_1$, it is difficult for an adversary to compute an $M'$ and $T'$ such that 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 11.8) 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 11.5).

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 11.8 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

* A. Sotirov et al. MD5 considered harmful: creating a rogue CA certificate. *25th Annual Chaos Communication Congress*, Berlin, December 2008.

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 11.5.1 and 11.7.4, and Section 11.10 is a case study of TLS.)

### 11.3.5 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.

### 11.3.6 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 11.5.

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 11.5, about the *resurrecting duckling* provides a solution that allows key distribution to be performed locally, without a central principal involved.

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

**Sidebar 11.5: 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 11.4.2], 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).

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 11.5, securely distributing keys such that a masquerading attack is impossible is a challenging problem.

### 11.3.7 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 7.1[on-line] 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 10.3.4[on-line]). 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 finger print of the replica, verifying the witness using the public key of the archiver, and then comparing the finger print of the witness against the finger print 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.

## 11.4 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 (see Section 5.3.5), 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.

### 11.4.1 Message Confidentiality Using Encryption

The setup for providing confidentiality over untrusted networks is shown in Figure 11.6. 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, e-mail 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.)

**FIGURE 11.6**

Providing confidentiality using ENCRYPT and DECRYPT over untrusted networks.

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 ($K_{\text{Bpub}}$), 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 $K_{\text{Apub}}$ and $K_{\text{Apriv}}$ for Alice, and Bob needs to learn Alice's public key.

### 11.4.2 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.

---

* N. Borisov, I. Goldberg, and D. Wagner, *Intercepting mobile communications: the insecurity of 802.11*, MOBICOM '01, Rome, Italy, July 2001.

Section 11.8 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.

### 11.4.3 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 ($M$, $K_{encrypt}$), $K_{sign}$)), 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 11.8 gives an example of exploitation of repetition in an otherwise unbreakable encryption system known as the one-time pad. Problem set *44* and *46* also explores one-time pads to setup a secure communicaiton 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 11.5.

### 11.4.4 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 respliced, 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 11.8, 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 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.

## 11.5 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 e-mail, anonymous e-mail, 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 7[on-line] 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.

### 11.5.1 Example: Key Distribution

To illustrate the need for security protocols, let's study two protocols for key distribution. In Section 11.3.6, 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

* 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.

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 7[on-line] are not authenticated.
For this message, Charles doesn't really care who sent it, so he replies:

1. 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 that he 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 7[on-line] 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 $K_{\mathrm{Lpub}}$ 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 $K_{\mathrm{Lpub}}$ and Lucifer has $K_{\mathrm{Lpriv}}$. This attack would be impossible with message 2 because Alice would notice that it was Lucifer's, rather than Bob's key.

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' 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) 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: $\{\text{"Use temporary authentication key } Ak_{AB} \text{ and temporary encryption key } Ek_{AB} \text{ to talk to Bob."}\}^{Ek_{AC}}_{Ak_{AC}}$

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' 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 is that message 2 is both authenticated with Charles' 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: $\{\text{"Use the temporary keys } Ak_{AB} \text{ and } Ek_{AB} \text{ to talk to Alice."}\}^{Ek_{BC}}_{Ak_{BC}}$

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 $k_{AB}$'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 keys $k_{AB}$'s. 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 11.6[on-line] 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.

---

* 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 networks of computers. Communications of the ACM 21, 12 (December, 1978), pages 993–999.

**Sidebar 11.6: The Kerberos authentication system** Kerberos* was developed in the late 1980's for project Athena, a network of engineering workstations and servers designed to support undergraduate education at M.I.T.† The first version in wide-spread 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, some consequences of which were discussed in Section 3.3.4. 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 $S$:

$$\text{Alice} \Rightarrow \text{KDDS}: \{\text{"alice@Scholarly.edu"}, S, T_{\text{current}}\}$$

and the service responds with a *ticket* identifying the user:

$$\text{KKDS} \Rightarrow \text{Alice}: \{K_{\text{tmp}}, S, \mathit{Lifetime}, T_{\text{current}}, \mathit{ticket}\}^{K_{\text{alice}}}$$

The service encrypts this response with the user's shared secret. The verification step occurs when the user decrypts the encrypted response. If $T_{\text{current}}$ 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:

$$\mathit{ticket} = \{K_{\text{tmp}}, \text{"alice@Scholarly.edu"}, S, T_{\text{current}}, \mathit{Lifetime}\}^{K_S}$$

(*Sidebar continues*)

* 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.

† 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.

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).*

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 on page 11–57). 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 11.4.4 on page 11–53, this property makes the protocols difficult to analyze.

---

* 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 XORs). 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 on-line at the same time (e.g., e-mail), 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 11.2).

### 11.5.2 Designing Security Protocols

Security protocols are vulnerable to several attacks in addition to the ones described in Section 11.3.4 (page 11–41) and 11.4.2 (page 11–50) 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 11.5.1. 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 11.5.1 might look more like:

*1* Alice ⇒ Charles: {"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?"}$_{Apriv}$

*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.

### 11.5.3 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* 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 e-mail from an e-mail 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†:

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

---

* 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)

† Myers and M. Rose, *Post Office Protocol Version 3*, Internet Engineering Task Force Request For Comments (RFC) 1939, May 1996.

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 e-mail 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 user name *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 e-mail.

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 11.10). Then, the client and server can run the challenge-response protocol over this connection. The server can also send the e-mail 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*. After authenticating three times unsuccessfully using the challenge-response protocol, the server allowed the fourth attempt to

* 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.

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.

### 11.5.4 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}$_{Apriv}$ }$^{Bpub}$

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* Alice ⇒ Bob: {"Here is my message:", ..... T}$^{KAB}$

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:

Alice ⇒ 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* Bob ⇒ CA: {"Give me the certificates for Bob and Charles"}

*2* CA ⇒ Bob: {"Here are the certificates:",
$\{Bob, B_{pub}, T'\}_{CApriv}$, $\{Charles, C_{pub}, T'\}_{CApriv}\}$

*3* Bob ⇒ Charles: {"Here is my certificate and a proposed key":,
$\{Alice, A_{pub}, T\}_{CApriv}$, $\{K_{AB}, T\}_{Apriv}\}^{Cpub}$

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* Bob ⇒ Charles: {"Please send me the secret business plan. Yours truly, Alice."}$^{KAB}$

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

* D. Denning and G. Sacco. Timestamps in key distribution protocols. *Communication of the ACM 24,* 8, pages 533–535, 1981.

(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, 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., $\{\text{Alice, Bob, } K_{AB}, T\}_{Apriv}$).

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.

### 11.5.5 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 ground-breaking paper *New Directions in cryptography* [Suggestions for Further Reading 1.8.5], 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 $n = g^k$ (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 $\{1, ..., p\text{-}2\}$. Alice sends to Bob: $g^a$ (modulo $p$), and Bob sends to Alice: $g^b$ (modulo $p$).

On receiving these messages, Alice computes $g^{ab} = (g^b)^a$ (modulo $p$), and Bob computes $g^{ba} = (g^a)^b$ (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 $k = g^{ab}$ (modulo $p$) given the two public values $g^a$ (modulo $p$) and $g^b$ (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 11.5.4. 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.

### 11.5.6 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 11.4.3]; a similar protocol is also used in the Taos distributed operating system [Suggestions for Further Reading 11.3.2]. Web clients and servers use the more complex SSL/TLS protocol, which is described in Section 11.10.

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:

Client ⇒ service: {"Here is a temporary public key $T_{pub}$ and two key halves encrypted with your public key:", $\{K_{c\text{-}cs}, K_{c\text{-}sc}\}^{Spub}$}

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 $K_{c\text{-}cs}'$ 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 $T_{pub}'$:

Service ⇒ client: {"Here are two key halves encrypted with your temporary public key:", $\{K_{s\text{-}cs}, K_{s\text{-}sc}\}^{Tpub}$}

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

- $K_{cs} \leftarrow$ HASH ( "client to server", $S_{pub}$, $T_{pub}$, $K_{s\text{-}cs}'$, $K_{c\text{-}cs}$)
- $K_{sc} \leftarrow$ HASH ( "server to client", $S_{pub}$, $T_{pub}$, $K_{s\text{-}sc}'$, $K_{c\text{-}sc}$)

and the server calculates

- $K_{cs}' \leftarrow$ HASH ( "client to server", $S_{pub}$, $T_{pub}'$, $K_{s\text{-}cs}$, $K_{c\text{-}cs}'$)
- $K_{sc}' \leftarrow$ HASH ( "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:

$$\{M\}^{K_{sse\text{-}cs}}_{K_{ssa\text{-}cs}}$$

to the server. 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.

### 11.5.7 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 imporant whenever the participants are separated by a network. Section 11.10 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 e-mail 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 e-mail 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 11.5.6]. 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.

## 11.6 Authorization: Controlled Sharing

Some data must stay confidential. For example, users require that their private authentication key stay confidential. Users wish to keep their password 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.

### 11.6.1 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.

The guard is distinct from, but operates on behalf of the authority, making mediation decisions by checking the permissions, and denying or allowing a request based on the permissions.

We discuss three models that differ in the way the service keeps track of who is authorized and who isn't: (1) the simple guard model, (2) the caretaker model, and (3) the flow-control model. The simple guard model is the simplest one, while flow control is the most complex model and is used primarily in heavy-duty security systems.

### 11.6.2 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 non-discretionary access control in Section 11.6.5.

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.

#### *11.6.2.1 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.

#### *11.6.2.2 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)*.

**Table 11.1:** Comparison of access control systems

| System | Advantage | Disadvantage |
|---|---|---|
| Ticket | Quick access check | Revocation is difficult |
| | Tickets can be passed around | Tickets can be passed around |
| List | Revocation is easy | Access check requires searching a list |
| | Audit possible | |
| Agency | List available | Revocation might be hard |

#### *11.6.2.3 Tickets Versus Lists, and Agencies*

Ticket and list systems each have advantages over the other. Table 11.1 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).

#### 11.6.2.4 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.

### 11.6.3 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 (see Section 2.5). UNIX was originally designed for a computer shared among multiple users, and therefore had to support access control. As described in Section 4.4, 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. The version of the UNIX system described in Section 2.5 didn't provide networking and didn't support servers well; modern UNIX systems, however, do, which further reenforces 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.

#### 11.6.3.1 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.

#### *11.6.3.2 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 (see page 2.5.11). 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 discus-

sion of the TCB on page 11–26. 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 [Suggestions for Further Reading 4.2.3], 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.

#### *11.6.3.3 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 e-mail delivery program that receives an e-mail 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 e-mail 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.

#### *11.6.3.4 Authenticating Users*

When a UNIX computer starts, it boots the kernel (see Sidebar 5.3). 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 on page 11–32) 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*.

#### *11.6.3.5 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.

#### *11.6.3.6 Running Services*

In addition to starting the login program, the init program usually starts several services (e.g., a Web server, an e-mail 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 did 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 (see Section 2.5.1) so that it can name only the files in its corner of 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*.

#### *11.6.3.7 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.

### 11.6.4 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 pm, 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.

### 11.6.5 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.

**FIGURE 11.7**

Confining a program within a compartment.

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 11.7). 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.

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*.

#### *11.6.5.1 Information Flow Control Example*

To make information flow control more concrete, consider a company that has information divided in two compartment:

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 $T_{\text{labelsseen}}$. Furthermore, the system remembers the maximum compartment label of data the thread has seen, $T_{maxlabels}$. Now the information flow control rules can be implemented as follows. The read rule is:

- Before reading an object with labels $O_{labels}$, check that $O_{labels} \subseteq T_{\text{maxlabels}}$.
- If so, set $T_{labelsseen} \leftarrow T_{labelsseen} \cup C_{labels}$, 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 $O_{labels}$ only if $T_{labelsseen} \subseteq O_{labels}$

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 $T$ has read information in a compartment other than the ones listed in $L$ than 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 judgement. 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.

#### *11.6.5.2 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 11.11.10.1 for a concrete example and see problem set *43* 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.

---

* U.S.A. Department of Defense, *Department of Defense trusted computer system evaluation criteria*, Department of Defense standard 5200, December 1985.

## 11.7 Advanced Topic: Reasoning about Authentication

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 11.5. 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 bel ficate 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 Kpub 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.

### 11.7.1 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 11.3.1].

Consider the following example. Alice types at her workstation "Send me the quiz" (see Figure 11.8). 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."

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 saw 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

**FIGURE 11.8**

Authentication example.

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.

#### *11.7.1.1 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.

#### *11.7.1.2 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 $k$):

Alice ⇒ file service: $\{$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**.

### 11.7.2 Authentication in Distributed Systems

All of the authentication examples we have discussed so far have involved one service. Using the techniques from Section 11.6, 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 11.10. 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 ser-

vice 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 11.6). 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 providing support in their services for openID.

### 11.7.3 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 (see Section 4.4) and so we cannot use DNS names to authenticate services.
2. How can we name users securely? We could use e-mail addresses, such as bob@Scholarly.edu, to identify principals but e-mail 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 11.7.4 through 11.7.6.

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 $K_{Bpriv}$ (e.g., 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 next Sections 11.7.4 through 11.7.6.

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:

$\{K_{Charlespub}$ is a member of the group $K_{ISSstudentspub}\}_{K_{ISSstudentspriv}}$,

which in the BAN logic translates to:

$K_{Charlespub}$ **speaks for** $K_{ISSstudentspub}$.

that is, Alice 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 $K_{\mathrm{Bobpub}}$ 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.

### 11.7.4 Authenticating Public Keys

How do we authenticate that $K_{\mathrm{Bpub}}$ 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

Bob **says** ($K_{pubBob}$ **speaks for** Bob) (belief #1)

and by applying the delegation of authority rule, Alice can immediately conclude that she is safe in believing

$K_{pubBob}$ **speaks for** Bob (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, $M_1$:

Bob **says** $M_1$

signs it with his private key:

$\{M_1\}_{KprivBob}$

which, in authentication logic, can be described as

$K_{privBob}$ **says** (Bob **says** $M_1$)

and sends it to Alice. Since the message arrived via the Internet, Alice now wonders if she should believe

Bob **says** $M_1$ (?)

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* ← VERIFY ($\{M_1\}_{KprivBob}$, $K_{pubBob}$)

If *result* is ACCEPT then one might think that Alice is entitled to believe:

$K_{privBob}$ **says** (Bob **says** $M_1$) (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

$$(K_{privBob} \textbf{ speaks for } K_{pubBob}) \qquad \text{(belief \#4?)}$$

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

$$K_{privBob} \textbf{ speaks for } \text{Bob} \qquad \text{(belief \#5)}$$

and she can now use the use of delegated authority rule to beliefs #5 and #3 to conclude that

$$\text{Bob } \textbf{says } M_1 \qquad \text{(belief \#6)}$$

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.

### 11.7.5 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

$$\textit{result} \leftarrow \text{VERIFY } (M_2, \text{ SIGN } (M_2, K_{privCharles}), K_{pubCharles})$$

is ACCEPT, all she believes (again assuming that the cryptographic system is secure) is that

$$K_{privCharles} \textbf{ says } (\text{Charles } \textbf{says } M_2)$$

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:

{"Charles's public key is $K_{pubCharles}$"}$_{KprivTrustUs}$

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** ($K_{pubCharles}$ **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 $\kappa_{pubCharles}$).
- Charles has kept $K_{privCharles}$ secret.

and 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. Trust.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.

### 11.7.6 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.

#### *11.7.6.1 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 Versign 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, for example 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.

* D. Eastlake, *Domain Name System Security Extensions*, Internet Engineering Task Force Request For Comments (RFC 2535), Mach 1999.

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 Schlarly 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.

#### *11.7.6.2 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, $K_{Apub}$}$K_{Cpriv}$, {Alice, $K_{Apub}$}$K_{Dpriv}$, {Alice, $K_{Apub}$}$K_{Epriv}$. 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 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) [Suggestions for Further Reading 1.3.16] and a number of other systems use the web of trust approach.

## 11.8 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 1.2.4], or *Handbook of Applied Cryptography* by Menezes, van Oorschot, and Vanstore [Suggestions for Further Reading 1.3.13]. *Introduction to cryptography* by Buchmann provides a concise description of the number theory that underlies cryptography [Suggestions for Further Reading 1.3.14]. There are many subtle issues in designing secure implementations of the primitives, which are beyond the scope of this text.

### 11.8.1 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 $i = 1,\ldots,n$:

$$c_i = m_i \oplus k_i$$

Untransforming is just as simple, because:

$$m_i = c_i \oplus k_i = m_i \oplus k_i \oplus k_i = m_i$$

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.

Assuming that the key string is truly random, a one-time pad cannot be broken by the attacks discussed in Section 11.4, since the ciphertext does not give the adversary any information about the plaintext (other than the length of the message). Each bit in the ciphertext has an equal probability of being one or zero, assuming the key string consists of truly random bits. Patterns in the plaintext won't show up as patterns in the ciphertext. Knowing the value of any number of bits in the ciphertext doesn't allow the adversary to guess the bits of the plaintext or other bits in the ciphertext. To the adversary the ciphertext is essentially just a random string of the same length as the message, no matter what the message is.

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.

### 11.8.2 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.

#### *11.8.2.1 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*:

```
procedure RC4_GENERATE ()
    i ← (i + 1) modulo 256
    j ← (j + S[i]) modulo 256
    SWAP (S[i], S[j])
    t ← (S[i] + S[j]) modulo 256
    k ← S[t]
    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).

#### *11.8.2.2 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.†

## 11.8.3 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.

### *11.8.3.1 Advanced Encryption Standard (AES)*

*Advanced Encryption Standard (AES)*‡ has 128-bit (or longer) keys and 128-bit plaintext and ciphertext blocks. AES replaces *Data Encryption Standard (DES)***††, which is now regarded as too insecure for many applications, as distributed Internet computations or

---

* 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.

‡ Advanced Encryption Standard, *Federal Information Processing Standards Publications (FIPS PUBS) 197*, National Institute of Standards and Technology (NIST), Nov. 2001.

** 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).

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 11.5.2].

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:

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:

```
procedure AES (in, out, key)
    state ← in                          // copy in into state as described above
    AddRoundKey (state, key)            // mix key into state
    for r from 1 to 9 do
        SubBytes (state)                // substitute some bytes in state
        ShiftRows (state)               // shift rows of state cyclically
        MixColumns (state)              // mix the columns up
        AddRoundKey (state, key[r×4, (r+1)×4 − 1])     // expand key, mix in
    SubBytes (state)
    ShiftRows (state)
    AddRoundKey (state, key[10×4, 11×4 − 1])
    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:

$$[s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}, s_{4,c}] \leftarrow [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}, s_{4,c}] \oplus key_{r\times 4+c}, \text{ for } 0 \le c < 4.$$

That is, each word of *key* is added to the corresponding column in *state*.

†† 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.

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:

$s_{r,c} \leftarrow s_{r,(c+\text{shift}(r,\,4))}$ **modulo** 4, for $0 \le c < 4$

The value of SHIFT is dependent on the row number as follows:

$$\text{SHIFT}(1,4) = 1,\ \text{SHIFT}(2,4) = 2, \text{ and } \text{SHIFT}(3,4) = 3$$

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.

#### *11.8.3.2 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 11.9). A dummy, random, ciphertext block, called the initialization vector (or IV) is inserted at the beginning.

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:

$C_0 = IV$ and $C_i \leftarrow \text{BC}\ (M_i \oplus C_{i-1}, key)$ for $i = 1, 2, \ldots, n$

where BC is some block cipher (e.g., AES).

To implement DECRYPT, one computes:

$M_i \leftarrow C_{i-1} \oplus \text{BC}\ (C_i, key)$

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

**FIGURE 11.9**

Cipher-block chaining.

block, implies that two encryptions of the same message with the same key will result in entirely different-looking ciphertexts. The last ciphertext block $C_n$ is a complicated key-dependent 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 11.9 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.

### 11.8.4 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 11.2. 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.

#### *11.8.4.1 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.

#### *11.8.4.2 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*: HASH (*k* + *M*). Since the receiver knows *k*, the receiver can recompute 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 11.5.5], 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:

HASH ((*k* ⊕ *outerpad*) + HASH ((*k* ⊕ *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.

**Sidebar 11.7: Secure Hash Algorithm (SHA)** SHA* is a family of cryptographic hash algorithms. SHA-1 takes as input a message of any length smaller than $2^{64}$ 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):

$$W(t) = \begin{cases} M_t, & \text{for } t = 0 \text{ to } 15 \\ (W(t-3) \oplus (W(t-8) \oplus (W(t-14) \oplus (W(t-16) <<< 1 & \text{for } t = 16 \text{ to } 79 \end{cases}$$

where <<< 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

$$K(t) = \begin{cases} 5A827999_{hex}, & \text{for } t = 0 \text{ to } 19 \quad // \text{ 2.5/4 in hex} \\ 6ED9EBA1_{hex}, & \text{for } t = 20 \text{ to } 39 \quad // \text{ 3.5/4 in hex} \\ 8F1BBCDC_{hex}, & \text{for } t = 40 \text{ to } 59 \quad // \text{ 5.5/5 in hex} \\ CA62C1D6_{hex}, & \text{for } t = 60 \text{ to } 79 \quad // \text{ 10.5/4 in hex} \end{cases}$$

(*Sidebar continues*)

* Secure hash standard, *Federal Information Processing Standards Publications (FIPS PUBS) 180-1*, National Institute of Standards and Technology (NIST), April 1995.

HMAC can be used with any good cryptographic hash function. Sidebar 11.7 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 SHA-512. Some cryptographers are recommending that research on designing cryptographic hash functions should start over.

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:

```
for each 512-bit block of M do
    for t from 0 to 79 do
        x ← (a <<< 5) + F(t, b, c, d) + e + W(t) + K(t)
        e ← d
        d ← c
        c ← b <<< 30
        b ← a
        a ← x
    A ← A + a; B ← B + b; C ← C + c; D ← D + d; E ← E + e
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, F, and produces a 256-bit value.The justification for the SHA family of hashes is outside the scope of this text.

## 11.8.5 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.

### *11.8.5.1 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 11.5.1]. 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 $e \times d = 1$ (*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 $n$. (Sections 11.8.5.2 and 11.8.5.3 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 $e$: $P^e$ (modulo $n$). To decipher, we compute $C$ to the power $d$: $C^d$ (modulo $n$).

The reason this works is as follows. $C^d = P^{ed} = P^{k(p-1)(q-1)+1}$, since $e \times d = 1$ (*modulo* $z$). 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 $k(p - 1)(q - 1) = 0$ (*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 $p = 47$ and $q = 59$, then $e$ is 17 and $d = 157$ because $e \times d = 1$ (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, $C$ is $587 = 31^{17}$ (modulo 2773). To reverse the transform, we compute $587^{157} = 31$ (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 a 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

---

* 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.

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 number 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.

#### *11.8.5.2 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 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 $M_2$: $(M_3)^d = (M_1 \times M_2)^d = M_1{}^d \times M_2{}^d$ *(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.

#### *11.8.5.3 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 11.5, 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 the next section, 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 $(M^d)^e = (M^e)^d = M^{ed}$ (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 obtain $M$ 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

## 11.9 .Summary

Section 11.1 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 11.11 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 11.1, 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 11.10 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, as explained in Section 5.7, 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 (see Sidebar 11.4), 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 11.8 summarizes some of the interactions, and their consequences.
- 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

**Sidebar 11.8: 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 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.

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 11.1, 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 page 11–51) 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 11.10 describes a successful secure protocol (with some growing pains to get it right) and the examples in Section 11.11 illustrate many ways to get things wrong.

## 11.10 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.

### 11.10.1 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 11.10 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.

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 11.10.3.)

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

* Tim Dierks and Eric Rescorla. The Transport Layer Security (TLS) protocol Version 1.2. *RFC 4346*. November 2007.

**FIGURE 11.10**

Typical TLS exchange of handshake protocol messages.

*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 11.10.3, 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 11.8 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 11.8). 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:

---

* 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.

PRF (*master_secret*, *finish_label*, HASH (*handshake_messsages*))

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.

### 11.10.2 Evolution of TLS

The TLS handshake protocol is more complicated than some of the 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 11.5.4]. 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).

### 11.10.3 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 (an on-line book retailer), 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 11.5.

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

```
structure certificate
    version
    serial_number
    signature_cipher_identifier
    issuer_signature
    issuer_name
    subject_name
    subject_public_key_cipher_identifier
    subject_public_key
    validity_period
```

**FIGURE 11.11**

Some fields in version 3 of the X.509 certificate

key of the certificate authority. (The service sends the certificates in step 3 of the handshake protocol, described in Section 11.10.1.)

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. Figure 11.11 shows some of the fields in Version 3 of X.509 certificates (the standard specifies them in a different order). 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 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, 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 authories, certificates, etc., and the legal status of certificates [Suggestions for Further Reading 1.3.17].

### 11.10.4 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 a certificate. Instead, many servers

authenticate users based on the IP address of the client machine or based on 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 it's 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 us 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 *45* explores some of the issues in protecting and authenticating cookies.

Web sites use cookies in many ways. For example, many Web sites uses 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.

## 11.11 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.

* George Santayana, *The Life of Reason, Volume 1, Introduction and Reason in Common Sense* (Scribner's: 1905)

- 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.

### 11.11.1 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, second-hand hard disks, and recycled magnetic tapes as well as by pawing through piles of physical trash (popularly known as "dumpster diving").

#### *11.11.1.1 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, then it 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*: 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.

#### *11.11.1.2 1997: Residues in Network Packets*

If one sends a badly formed request to a Kerberos Version 4 server (Sidebar 11.6) 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*: As in example 11.11.1.1, above, use *fail-safe defaults.* The packet buffer should have been cleared between uses.

#### *11.11.1.3 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*: Apparently unimportant specifications, such as "return only as much data as is actually in the file" can sometimes be quite important.

---

* 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.

* Reported by L0pht Heavy Industries in 1997, after the system had been in production use for ten years.

† Reported Monday 17April 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/>.

#### *11.11.1.4 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*: 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.

#### *11.11.1.5 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.)†

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 e-mail messages in connection with the Iran-Contra affair. They apparently did not realize that the PROFS e-mail system used by the National Security Council maintained backup copies. The messages found on the backup tapes became

---

* James Bandler. Harvard ouster linked to porn; Divinity School dean questioned. *Boston Globe* (Wednesday 19 May 1999) City Edition, page B1, Metro/Region section.

† Incident ca. 1970, reported by Roger G. Needham.

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.* †

*Lesson*: there is a tension between reliability, which calls for maintaining multiple copies of data, and security, which is enhanced by minimizing extra copies.

#### *11.11.1.6 Magnetic Residues: High-Tech Garbage 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. ‡

#### *11.11.1.7 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 CD's 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 CD's found that in addition to the intended files, there was a remarkable, and occasionally embarrassing, collection of personal information there, too.

---

* 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.

† The context issue is highlighted in *Armstrong v. Bush*, 721 F. Supp. 343, 345 n.1 (D.D.C. 1989).

‡ *Remanence Security Guidebook.* Naval Staff Office Publication NAVSO P-5239-26 (September 1993:United States Naval Information Systems Management Center: Washington D.C.)

** David Winfrey. "Uncleared disk space and MSVC". *Risks Forum Digest 21*, 50 (12 July 2001).

*Lesson*: "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."*

### 11.11.2 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 user names. 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 user names 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*: Pay attention to the *least privilege principle*: don't store your lunch (in this case, the names of users) 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.‡

---

* 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.

† 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 password records in Multics, the successor system to CTSS. But see item 11.11.3, which follows.

*Lesson (not restricted to security)*: Removing interlocks can be risky because it is hard to track down every part of the system that depended on the interlock being there.

### 11.11.3 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*: 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.*

### 11.11.4 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.

---

* 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).

#### *11.11.4.1 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.*

#### *11.11.4.2 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*: *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.

#### *11.11.4.3 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

---

* Reported by Richard G. Mills, 1965.

† 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.

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/second, a machine with a 10 megabits/second path to the Internet could launch packets at a rate of 3,000/second and a high-powered server with a 155 megabits/second connection might be able to launch as many as 45,000 packets/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,

‡ 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)

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 thousand vulnerable ports scattered through a space of 4 billion addresses, the chance that any single packet hits a vulnerable port is one in 120 thousand. 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 thousand 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.

*Lessons*: 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 fail-safe defaults*). For implementers, it emphasizes the importance of diligent care (and paranoid design) in network listener implementations, especially on widely activated UDP ports.*

## 11.11.5 The Kernel Trusts the User

### *11.11.5.1 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*: A bit more attention to paranoid design would have avoided these problems.

---

* 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.

† Noticed by the author, exploit developed by Maxim G. Smith, 1963.

#### *11.11.5.2 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 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, as described in Section 2.5.7). 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 *filename* 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*: 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.

#### *11.11.5.3 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

---

‡ Allocation strategy reported by Fred Brooks in *The Mythical Man-Month*.[Suggestions for Further Reading 1.1.3

* 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).

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 self-modifying, 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: It can be a major challenge to apply *complete mediation* to a legacy hardware architecture.

## 11.11.6 Technology Defeats Economic Barriers

### *11.11.6.1 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*: Never underestimate the effect of technology improvement, and the effectiveness of the resources that a clever adversary may bring to bear.

### *11.11.6.2 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

---

* 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.

† Reported by F. J. Corbató, ca. 1975.

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*: As technology improves, so do the tools available for adversaries.

### 11.11.7 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 e-mail, but who had not previously tried to send secure e-mail. 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 e-mail 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*:

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.

---

* 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.

† 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.

### 11.11.8 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*: 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.*

### 11.11.9 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*: People aren't good at keeping secrets.

---

* Chris Rioux provided details on this collection of browser problems, and discovered the .ISP exploitation, in 1998.

† This anecdote was reported in the 1970's, but its source has been lost.

## 11.11.10 Signaling with Clandestine Channels

### *11.11.10.1 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 non-discretionary 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 signalling 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: *Minimize common mechanisms*. A common mechanism such as a shared virtual memory or a shared cache can provide an unintended communication path.

### *11.11.10.2 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 nearby, cheap, 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*: Paranoid design requires considering *all* access paths.

---

* 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 <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).

‡ Markus G. Kuhn and Ross J. Anderson. Soft Tempest: Hidden Data Transmission Using Electromagnetic Emanations. In David Aucsmith (Ed.): *Information Hiding 1998, Lecture Notes in Computer Science 1525*, pages 124–142 (1998: Springer-Verlag: Berlin and Heidelberg).

#### *11.11.10.3 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*: 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.

### 11.11.11 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.

---

* 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.

#### *11.11.11.1 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 pro-

gram 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*: The programmer who ignored the returned status value was not sufficiently paranoid in the implementation. Also, 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*,. That designer also did not apply the principle to *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.

#### *11.11.11.2 How Large is the Key Space...Really?*

When a client presents a Kerberos ticket to a service (see Sidebar 11.6 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

* Reported by Theodore T'so in 1997.

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*: 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.*

#### *11.11.11.3 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 on-line 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 11.11.11.2.

### 11.11.12 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 e-mail 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.

---

* 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.

† Jeremy Epstein. *Risks-Forum Digest 20,* 64 (Thursday 4 November 1999).

The countermeasure for injection is known as "sanitizing the input". In principle, santizing 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.

#### *11.11.12.1 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.*

#### *11.11.12.2 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,

* This vulnerability was noticed, and corrected, by staff programmers in the late 1960's. As far as is known, it was never actually exploited.

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."* 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…"†

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…"‡

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

---

* `http://www.wired.com/news/technology/0,67980-0.html`

† *The New York Times*, Tuesday, June 21, 2005.

‡ 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.

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."*

Visa and American Express cancelled their contracts with CardSystems, and the company is no longer in business.

*Lesson*: 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. The lesson following anecdote 11.11.1.7 seems to apply here, also.

### 11.11.13 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:* Most reliability design principles also apply to security: *avoid rarely-used components*.

### 11.11.14 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 com-

---

* United States Federal Trade Commission Complaint, Case 0523148, Docket C-4168, September 5, 2006.

† Karger and Schell, *op. cit.*, Section 3.2.2.

promising 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 slip-ups. 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*: *Complete mediation* includes verifying the authenticity, integrity, and authorization of the software development process, too.

### 11.11.15 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 11.12. 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.

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

* Schell, 1979 *op. cit.*, page 22.

† Karger and Schell, 1974 *op. cit.,* Sections 3.4.5 and 3.4.6.

Enigma Rotor with eight contacts

Side view, showing contacts.

Edge view, showing some connections.

Two Enigma Rotors with a reflector, showing an input-output path.

**FIGURE 11.12**

Enigma design concept (simplified for illustration).

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.)

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.*

## Exercises

**11.1** 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

* 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].

† Throughout the Problems and Solutions, 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.

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}, $K_{cs}$)
service ⇒ client: ENCRYPT ({nonce, response}, $K_{cs}$)

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 ({$K_{tn}$}, $K_{cs}$)
client ⇒ service: ENCRYPT ({nonce, procedure, arguments}, $K_{tn}$)
service ⇒ client: ENCRYPT ({nonce, response}, $K_{tn}$)

in which $K_{\mathrm{tn}}$ 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 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.

*1983-3-5b*

**11.2** 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.

11.2a. Assuming 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?

*1994–2–1a*

11.2b. How long will it take if Alice chooses an eight-character password that includes upper- and lower-case letters, numbers, and 16 special characters, 78 characters in all?

*1994–2–1b*

11.2c. 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?

*1994–2–1c*

**11.3** 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?

*[1983–2–4b]*

**11.4** 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 $k_2$,

$$\text{TRANSFORM}\ (\text{TRANSFORM}\ (M, K_a), K_b) = \text{TRANSFORM}\ (\text{TRANSFORM}\ (M, K_b), K_a)$$

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, ENCRYPT_C ($M$, $K_a$), to Bob.

- Bob chooses another random key, $K_b$, encrypts the already-encrypted message to produce ENCRYPT_C (ENCRYPT_C ($M$, $K_a$), $K_b$) and sends the doubly-encrypted result back to Alice.
- By commutativity, this message is identical to ENCRYPT_C (ENCRYPT_C ($M$, $K_b$), $K_a$), which is a message that Alice can decrypt with her key $K_a$. She does so, revealing ENCRYPT_C ($M$, $K_b$).
- 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.

11.4a. 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.

*1994–2–2a*

11.4b. 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.

*1994–2–2b*

**11.5** 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.

11.5a. 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?

*1995–2–3a*

11.5b. 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?

*1995–2–3b*

11.5c. 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.)

*1995–2–3c*

11.5d. 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.

*1995–2–3d*

**11.6** Alice wants to communicate with Bob over an insecure network. She learned about one-time pads in Section 11.8, 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 XORs the message $m$ with the key $r$ to produce a ciphertext $c$, and sends this on the network. Bob XORs $c$ with $r$ to retrieve $m$.

11.6a. 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 $m1$ to Bob, Alice picks a $k$-bit random number $r1$, computes ciphertext $c1 = m1 \oplus r1$, and sends $c1$ to Bob. Bob then picks his own $k$-bit random number $r2$, computes $c2 = c1 \oplus r2$, and sends $c2$ to Alice. Alice finally computes $c3 = c2 \oplus r1$ and sends $c3$ to Bob.

11.6b. Which of the following statements are correct of Alice's new scheme?

**A.** Bob can correctly decrypt Alice's message $m1$, without receiving $r1$ 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.

*2008-3-12-13*

**11.7** 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)

*1995–2–4a*

**11.8** 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.** Propagation.
**C.** Revocation.
**D.** Auditing.
**E.** Ease of use.

*1984–2–4*

**11.9** 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 $K_1$ to encrypt their communications and a second shared-secret key $K_2$ to authenticate their communications, using the following protocol:

Bank 1 ⇒ Bank 2$\{\{\text{"transfer from our Account Y"}\}_{K2}\}^{K1}$
Bank 1 ⇒ Bank 2$\{\{\text{"to your Account X"}\}_{K2}\}^{K1}$
Bank 1 ⇒ Bank 2$\{\{\text{"Amount Z"}\}_{K2}\}^{K1}$
Bank 2 ⇒ Bank 1$\{\{\text{"OK"}\}_{K2}\}^{K1}$

Alyssa immediately realizes that without knowing either *K1* or *K2* an intruder could

subvert the banks.

11.9a. With an Apple II in the manhole in middle of Main street describe how Alyssa could

**A.** Increase or decrease the amount of a transfer.
**B.** Cause a transfer to occur more than once.
**C.** Cause a transfer not to occur at all without arousing suspicion at the requesting bank.

*1984–2–3a*

11.9b. Design a new protocol that eliminates these problems and uses only two messages.

*1984–2–3b*

**11.10** 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-secret-key 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.

11.10a. Ben's intuition is that this can't possibly work, but he isn't sure why. Give Ben some help by explaining why.

*2002–0–1*

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:

$raw_message_i \leftarrow \{i, D_i, K_{i-2}\}$
$authtag_i \leftarrow \text{SIGN}\ (raw_message_i, K_i)$
$P_i \leftarrow \{raw_message_i, authtag_i\}$

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.

11.10b. 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.

*2002–0–2*

**11.11** 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.

*2002–2–04*

**11.12** 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.

*2003–3–5*

**11.13** 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 11.11.4.3. 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:

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.

11.13a. Give an estimate of the total number of machines that run the FOO server.

**A.** 100 machines
**B.** $7.2\times10^{18}$ machines
**C.** 25,600 machines
**D.** 8,000 machines

11.13b. 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

*2007-3-3-4*

**11.14** Ben Bitdiddle, the new head of Cyber Security for the Department of Homeland Security, studied the war story about the Slammer worm in Section 11.11.4.3 and 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.

11.14a. 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, remotely-configurable 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.

11.14b. 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.** 10000 worm probes/second
**E.** 100000 worm probes/second

*2008-3-6-7*

**11.15** 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 in e-mail from a spammer.

**11.16** 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, but 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. $K_{P_{xyz}}$ is the public key of principal xyz.

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; message 3 is shown here

only to place the PKPI protocol in context.

11.16a. 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' position, or an example demonstrating that Louis is mistaken.

11.16b. 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.

11.16c. 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?

*1995–2–5a…c*

**11.17** 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?

*1983–3–3*

**Additional exercises relating to Chapter 11 can be found in problem sets *43–49*.**

# Glossary for Chapter 11

**access control list (ACL)**—A list of principals authorized to have access to some object. [Ch. 11]

**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. [Ch. 11]

**authentication**—Verifying the identity of a principal or the authenticity of a message. [Ch. 11]

**authentication tag**—A cryptographically computed string, associated with a message, that allows a receiver to verify the authenticity of the message. [Ch. 11]

**authorization**—A decision made by an authority to grant a principal permission to perform some operation, such as reading certain information. [Ch. 11]

**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. [Ch. 11]

**certificate**—A message that attests the binding of a principal identifier to a cryptographic key. [Ch. 11]

**certificate authority (CA)**—A principal that issues and signs certificates. [Ch. 11]

**certify**—To check the accuracy, correctness, and completeness of a security mechanism. [Ch. 11]

**cipher**—Synonym for a *cryptographic transformation*. [Ch. 11]

**ciphertext**—The result of encryption. Compare with *plaintext*. [Ch. 11]

**cleartext**—Synonym for *plaintext*. [Ch. 11]

**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. [Ch. 4]

**coheerence**—See *read/write coherence* or *cache coherence*.

**confidentiality**—Limiting information access to authorized principals. *Secrecy* is a synonym. [Ch. 11]

**confinement**—Allowing a potentially untrusted program to have access to data, while ensuring that the program cannot release information. [Ch. 11]

**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. [Ch. 11]

**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. [Ch. 11]

**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. [Ch. 11]

**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. [Ch. 11]

**cryptography**—A discipline of theoretical computer science that specializes in the study of cryptographic transformations and protocols. [Ch. 11]

**data integrity**—Authenticity of the apparent content of a message or file. [Ch. 11]

**decrypt**—To perform a reverse cryptographic transformation on a previously encrypted message to obtain the plaintext. Compare with *encrypt*. [Ch. 11]

**digital signature**—An authentication tag computed with public-key cryptography. [Ch. 11]

**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*. [Ch. 11]

**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. [Ch. 11]

**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. [Ch. 11]

**flow control**—In security, a system that allows untrusted programs to work with sensitive data but confines all program outputs to prevent unauthorized disclosure. [Ch. 11]

**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. [Ch. 11]

**freshness**—A property of a message in a security protocol: if the message is fresh, it is assured not to be a replay. [Ch. 11]

**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. [Ch. 11]

**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. [Ch. 11]

**list system**—A design for an access control mechanism in which each protected object is associated with a list of authorized principals. [Ch. 11]

**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. [Ch. 11]

**message authentication**—The verification of the integrity of the origin and the data of a message. [Ch. 11]

**message authentication code (MAC)**—An authentication tag computed with shared-secret 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." [Ch. 11]

**name-to-key binding**—A binding between a principal identifier and a cryptographic key. [Ch. 11]

**non-discretionary access control**—A property of an access control system. In a non-discretionary 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*. [Ch. 11]

**origin authenticity**—Authenticity of the claimed origin of a message. Compare with *data integrity*. [Ch. 11]

**page fault**—See *missing-page exception*.

**pair-and-spare**—See *pair-and-compare*.

**password**—A secret character string used to authenticate the claimed identity of an individual. [Ch. 11]

**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*. [Ch. 11]

**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*.

**presented load**—See *offered load*.

**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. [Ch. 11]

**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. [Ch. 11]

**private key**—In public-key cryptography, the cryptographic key that must be kept secret. Compare with *public key*. [Ch. 11]

**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. [Ch. 11]

**protection group**—A principal that is shared by more than one user. [Ch. 11]

**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*. [Ch. 11]

**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. [Ch. 11]

**repudiate**—To disown an apparently authenticated message. [Ch. 11]

**secrecy**—Synonym for *confidentiality*. [Ch. 11]

**secure area**—A physical space or a virtual address space in which confidential information can be safely confined. [Ch. 11]

**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. [Ch. 11]

**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. [Ch. 11]

**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. [Ch. 11]

**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. [Ch. 11]

**shared-secret key**—The key used by a shared-secret cryptography system. [Ch. 11]

**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". [Ch. 11]

**speaks for**—A phrase used to express delegation relationships between principals. "A speaks for B" means that B has delegated some authority to A. [Ch. 11]

**threat**—A potential security violation from either a planned attack by an adversary or an unintended mistake by a legitimate user. [Ch. 11]

**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. [Ch. 11]

**trusted computing base (TCB)**—That part of a system that must work properly to make the overall system secure. [Ch. 11]

# Index of Chapter 11

Design principles and hints appear in *underlined italics*. Procedure names appear in SMALL CAPS. Page numbers in **bold face** are in the chapter Glossary.


## A

access control list 11–74, **11–163**
ACL (see access control list)
*adopt sweeping simplifications* 11–16
Advanced Encryption Standard (AES) 11–103
adversary 11–6, **11–163**
authentication 11–20, **11–163**
  key 11–41
  logic 11–86
  origin 11–37, **11–165**
  tag 11–41, **11–163**
authorization 11–21, 11–73, **11–163**
  matrix 11–73
*avoid rarely used components* 11–148

## B

*be explicit* 11–4, 11–10, 11–24, 11–26, 11–53, 11–55, 11–61, 11–67, 11–68
block
  cipher 11–103
bot 11–19
buffer overrun attack 11–22, 11–23

## C

CA (see certificate authority)
capability 11–74, **11–163**
cascading change propagation 11–105
CBC (see cipher-block chaining)
certificate 11–56, **11–163**
  authority 11–56, **11–163**
  self-signed 11–92
certify 11–11, **11–163**
cipher 11–99, **11–163**
cipher-block chaining 11–105
ciphertext 11–49, **11–163**
cleartext 11–38, **11–163**
close-to-open consistency **11–163**
collision
  hash 11–33
commutative cryptographic transformation 11–153
compartment 11–81
*complete mediation* 11–5, 11–15, 11–18, 11–25, 11–136
computationally secure 11–33
confidentiality 11–49, **11–163**
confinement 11–82, **11–163**
consistency
  close-to-open **11–163**
cookie 11–124
covert channel 11–84, **11–163**
cryptographic
  hash function 11–32, **11–164**
  key 11–39, **11–164**
  transformation 11–39, 11–99, **11–164**
  transformation, commutative 11–153
cryptography 11–22, **11–164**
  public key 11–40, **11–166**
  shared-secret 11–40, **11–166**

## D

data integrity
  in security assurance 11–36, **11–164**
declassify 11–84
decrypt 11–49, **11–164**
DECRYPT 11–49
defense in depth 11–12
delayed authentication 11–157
*design for iteration* 11–4, 11–10, 11–26
*design principles*
  *adopt sweeping simplifications* 11–16

*avoid rarely used components* 11–148
*be explicit* 11–4, 11–10, 11–24, 11–26, 11–53, 11–55, 11–61, 11–67, 11–68
*complete mediation* 11–5, 11–15, 11–18, 11–25, 11–136
*design for iteration* 11–4, 11–10, 11–26
*economy of mechanism* 11–16, 11–26
*end-to-end argument* 11–16
*fail-safe defaults* 11–16, 11–24, 11–126
*keep digging principle* 11–126
*least privilege principle* 11–17, 11–24, 11–39, 11–79, 11–80, 11–81, 11–130
*minimize common mechanism* 11–16, 11–141
*minimize secrets* 11–15, 11–34, 11–39
*open design principle* 11–13, 11–39, 11–64, 11–140
*principle of least astonishment* 11–15, 11–138
dictionary attack 11–34
digital signature 11–44, **11–164**
discretionary access control 11–74, 11–81, **11–164**

**E**

*economy of mechanism* 11–16, 11–26
encrypt 11–49, **11–164**
ENCRYPT 11–49
encryption key 11–49
*end-to-end argument* 11–16
explicitness 11–61, **11–164**

**F**

*fail-safe defaults* 11–16, 11–24, 11–126
flow control **11–164**
forward
secrecy 11–61, **11–164**
freshness 11–61, **11–164**

**H**

hashed MAC 11–107
*hints*
*separate mechanism from policy* 11–7, 11–84
HMAC (see hashed MAC)

**I**

information flow control 11–83

**K**

KDC (see key distribution center)
*keep digging principle* 11–126
key (see cryptographic key)
key distribution center 11–57, **11–165**
key-based cryptographic transformation 11–41, **11–164**

**L**

*least astonishment principle* 11–15, 11–138
*least privilege principle* 11–17, 11–24, 11–39, 11–79, 11–80, 11–81, 11–130
limited change propagation 11–100
list system 11–74, **11–165**

**M**

MAC
(see message authentication code)
malware 11–19
mediation 11–73, **11–165**
message
authentication 11–36, **11–165**
authentication code 11–44, **11–165**
*minimize common mechanism* 11–16, 11–141
*minimize secrets* 11–15, 11–34, 11–39

**N**

name-to-key binding 11–45, **11–165**
non-discretionary access control 11–74, 11–81, **11–165**

**O**

one-time pad 11–99
*open design principle* 11–13, 11–39, 11–64, 11–140

origin authenticity 11–37, **11–165**

## P

password 11–31, **11–165**
PGP (see protocol, pretty good privacy)
PKI (see public key infrastructure)
plaintext 11–38, 11–49, **11–165**
prepaging **11–165**
principal 11–20, **11–165**
*principle of least astonishment* 11–15, 11–138
privacy 11–6, **11–166**
private key 11–40, **11–166**
protection 11–6, **11–166**
  group 11–76, **11–166**
protocol
  challenge-response 11–64
  Diffie-Hellman key agreement 11–68
  Kerberos 11–58
  pretty good privacy 11–98
  secure shell 11–46
  secure socket layer 11–117
  security 11–36, 11–54, **11–166**
  transport layer security 11–116
pseudorandom number generator 11–101
public key 11–40, **11–166**
  cryptography 11–40, **11–166**
  infrastructure 11–93, 11–114

## R

random
  number generator 11–99
  pseudorandom number generator 11–101
RC4 cipher 11–101
reference
  monitor 11–20
repudiate **11–166**
revocation 11–73
Rivest, Shamir, and Adleman cipher 11–109
RSA (see Rivest, Shamir, and Adleman cipher)

## S

safety net approach 11–10
secrecy **11–166**
secure area **11–166**
secure channel 11–22, 11–116, **11–166**
Secure Socket Layer 11–117
security 11–6, **11–166**
  protocol 11–36, 11–54, **11–166**
seed 11–101
*separate mechanism from policy* 11–7, 11–84
shared-secret
  cryptography 11–40, **11–166**
  key 11–40, **11–166**
sign 11–41, **11–166**
speaks for 11–85, **11–166**
SSH (see protocol, secure shell)
SSL (see Secure Socket Layer)
stream
  cipher 11–99

## T

TCB (see trusted computing base)
threat 11–7, **11–167**
  insider 11–8
ticket system 11–74, **11–167**
tiger team 11–27
timed capability 11–156
TLS (see Transport Layer Security)
Transport Layer Security 11–116
trusted
  computing base 11–26, **11–167**

## V

verify 11–41
virus 11–19

## W

window
  of validity 11–33
witness 11–48
work factor 11–33
worm 11–19

APPENDIX

# The Binary Classification Trade-off

# A

A *binary classification trade-off* arises when we wish to classify a set of things into two categories (call them *In* and *Out*), but we do not have a direct way of doing the classifying. On the other hand, there is a *proxy* for those things that is relatively easy to classify. The problem is that the proxy is only approximate. Because it is only approximate, there are four classification outcomes:

- *True positive:* The proxy classifies things as *In* that should be *In*.
- *True negative:* The proxy classifies things as *Out* that should be *Out*.
- *False negative:* The proxy classifies things as *Out* that should be *In*.
- *False positive:* The proxy classifies things as *In* that should be *Out*.

The trade-off is that it may be possible to reduce the frequency of one of the false outcomes by adjusting some parameter of the proxy, but that adjustment will probably increase the frequency of the other false outcome*.

A common example is an e-mail spam filter, which is a proxy for the division between wanted e-mail and spam. The filter correctly classifies e-mail most of the time, but it occasionally misclassifies a wanted message as spam, with the undesirable outcome that you may never see that message. It may also misclassify some spam as wanted e-mail, with the undesirable outcome that the spam clutters up your mailbox. The trade-off appears when someone tries to adjust the spam filter. If the filter becomes more aggressive, more wanted e-mail is likely to end up misclassified as spam. If the spam filter becomes less aggressive, more spam is likely to end up in your mailbox.

Reducing both undesirable outcomes simultaneously usually requires discovering a better proxy, but a better one may be hard to find or may not exist at all.

***Representations:*** One can conveniently represent a binary classification trade-off with a $2 \times 2$ matrix such as the one on the next page by answering two questions: (1) What are the real categories? and (2) What are the proxy categories? The example describes a smoke detector. The real categories are {fire, no fire}. The proxy categories are {smoke detector signals, smoke detector is quiet}. A too-sensitive smoke detector

*In some areas, such as computer security and biometrics, the words "acceptance" and "rejection" replace "positive" and "negative", respectively.

| | | Real categories | |
|---|---|---|---|
| | | fire | no fire |
| Proxy categories | detector signals | TA: fire extinguished | FA: false alarm |
| | detector quiet | FR: house burns down | TR: all quiet |

may signal more false alarms, but an insensitive one may miss more real fires. When someone replaces the labels with numbers of actual events, this representation is called a *confusion matrix*.

A Venn diagram, such as the one below, can be another useful representation of a binary classification trade-off. Take, for example, document retrieval (e.g., a Google search) The real categories are wanted and unwanted documents. The proxy is a query, for which the categories are that the query matches or the query misses.

***Measures:*** Sometimes one can identify the true categorizations and compare them with the proxy classifications. When that is possible, it can be useful to calculate ratios to measure proxy quality. Unfortunately, there are too many possible ratios. The confusion matrix contains four numbers, which may be used singly or may be added up to use as either a numerator or a denominator in 14 ways, so it is possible to calculate $14 \times 13 = 182$ different ratios. Not all of these ratios are interesting, but one can usually find at least one ratio among the 182 that seems to support his or her position in a debate.

Nine of these ratios are popular enough to have names, although three of the nine are just complements of other named ratios. The information retrieval community uses one set of labels for these ratios, whereas the medical and bioinformatics communities use another, with other communities developing their own nomenclature. As will be seen, all of the labels can be confusing.

Suppose that there is a population of $In + Out = N$ items and that we have run the classifier and counted the number of true and false positives and negatives. Here are the nine ratios:

1. *Prevalance*: The fraction of the population that is *In*.

$$Prevalance = In/N$$

2. *Efficiency*, *Accuracy*, or *Hit Rate*: The fraction of the population the proxy classifies correctly.

$$Efficiency = (True\,Positives + True\,Negatives)/N$$

3. *Precision* (information retrieval) or *Positive Predictive Value* (medical): The fraction of things that the proxy classifies as *In* that are actually *In*.

$$Precision = (True\,Positives)/(True\,Positives + False\,Positives)$$

4. *Recall* (information retrieval), *Sensitivity* (medical), or *True acceptance rate* (biometrics): The fraction of things in the population that are *In* that the proxy classifies as *In*.

$$Recall = (True\,Positives)/In$$

5. *Specificity* (medical) or *True rejection rate* (biometrics): The fraction of things in the population that are *Out* that the proxy classifies as *Out*.

$$Specificity = (True\,Negatives)/Out$$

6. *Negative Predictive Value*: The fraction of things that the proxy classifies as *Out* that are actually *Out*.

$$Negative\,Predictive\,Value = \frac{True\,Negatives}{True\,Negatives + False\,Negatives}$$

7. *Misclassification Rate* or *Miss Rate*: The fraction of the population the proxy classifies wrong.

$$Miss\,Rate = (False\,Negatives + False\,Positives)/N = (1 - Efficiency)$$

8. *False Acceptance Rate*: The fraction of *Out* items that are falsely classified as *In*.

$$Fasle\ Acceptance\,Rate = (False\,Positives)/Out = (1 - Specificity)$$

9. *False Rejection Rate*: The fraction of *In* items that are falsely classified as *Out*.

$$Fasle\,Rejection\,Rate = (False\,Negatives)/In = (1 - Sensitivity)$$

# Suggestions for Further Reading

## TABLE OF CONTENTS


**Introduction** ..... **376**
**1 Systems** ..... **378**
1.1 Wonderful Books About Systems ..... 378
1.2 Really Good Books About Systems ..... 380
1.3 Good Books on Related Subjects Deserving Space on the Systems Bookshelf ..... 381
1.4 Ways of Thinking About Systems ..... 385
1.5 Wisdom About System Design ..... 386
1.6 Changing Technology and its Impact on Systems ..... 388
1.7 Dramatic Visions ..... 389
1.8 Sweeping New Looks ..... 390
1.9 Keeping Big Systems Under Control ..... 393
**2 Elements of Computer System Organization** ..... **394**
2.1 Naming Systems ..... 394
2.2 The UNIX® System ..... 395
**3 The Design of Naming Schemes** ..... **395**
3.1 Addressing Architectures ..... 396
3.2 Examples ..... 396
**4 Enforcing Modularity with Clients and Services** ..... **397**
4.1 Remote Procedure Call ..... 398
4.2 Client/Service Systems ..... 398
4.3 Domain Name System (DNS) ..... 399
**5 Enforcing Modularity with Virtualization** ..... **399**
5.1 Kernels ..... 399
5.2 Type Extension as a Modularity Enforcement Tool ..... 400
5.3 Virtual Processors: Threads ..... 401
5.4 Virtual Memory ..... 402
5.5 Coordination ..... 402
5.6 Virtualization ..... 404
**6 Performance** ..... **405**
6.1 Multilevel Memory Management ..... 405
6.2 Remote Procedure Call ..... 406
6.3 Storage ..... 406
6.4 Other Performance-Related Topics ..... 407
**7 The Network as a System and as a System Component** ..... **408**
7.1 Networks ..... 408
7.2 Protocols ..... 408

7.3 Organization for Communication ..... 410
7.4 Practical Aspects ..... 411
**8 Fault Tolerance: Reliable Systems from Unreliable Components ..... 411**
8.1 Fault Tolerance ..... 411
8.2 Software Errors ..... 412
8.3 Disk Failures ..... 412
**9 Atomicity: All-or-Nothing and Before-or-After ..... 413**
9.1 Atomicity, Coordination, and Recovery ..... 413
9.2 Databases ..... 413
9.3 Atomicity-Related Topics ..... 414
**10 Consistency and Durable Storage ..... 415**
10.1 Consistency ..... 415
10.2 Durable Storage ..... 417
10.3 Reconciliation ..... 418
**11 Information Security ..... 418**
11.1 Privacy ..... 418
11.2 Protection Architectures ..... 418
11.3 Certification, Trusted Computer Systems, and Security Kernels ..... 419
11.4 Authentication ..... 420
11.5 Cryptographic Techniques ..... 421
11.6 Adversaries (The Dark Side) ..... 422

## INTRODUCTION

The hardware technology that underlies computer systems has improved so rapidly and continuously for more than four decades that the ground rules for system design are constantly subject to change. It takes many years for knowledge and experience to be compiled, digested, and presented in the form of a book, so books about computer systems often seem dated or obsolete by the time they appear in print. Even though some underlying principles are unchanging, the rapid obsolescence of details acts to discourage prospective book authors, and as a result some important ideas are never documented in books. For this reason, an essential part of the study of computer systems is found in current—and, frequently, older—technical papers, professional journal articles, research reports, and occasional, unpublished memoranda that circulate among active workers in the field.

Despite that caveat, there are a few books, relatively recent additions to the literature in computer systems, that are worth having on the shelf. Until the mid-1980s, the books that existed were for the most part commissioned by textbook publishers to fill a market, and they tended to emphasize the mechanical aspects of systems rather than insight into their design. Starting around 1985, however, several very good books started to appear, when professional system designers became inspired to capture their insights. The appearance of these books also suggests that the concepts involved in computer system design are finally beginning to stabilize a bit. (Or it may just be

that computer system technology is beginning to shorten the latencies involved in book publishing.)

The heart of the computer systems literature is found in published papers. Two of the best sources are Association for Computing Machinery (ACM) publications: the journal *ACM Transactions on Computer Systems (TOCS)* and the bi-annual series of conference proceedings, the *ACM Symposium on Operating Systems Principles (SOSP)*. The best papers of each SOSP are published in a following issue of TOCS, and the rest—in recent years all—of the papers of each symposium appear in a special edition of *Operating Systems Review*, an ACM special interest group quarterly that publishes an extra issue in symposium years. Three other regular symposia are also worth following: the *European Conference on Computer Systems (EuroSys)*, the *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, and the USENIX *Symposium on Networked Systems Design and Implementation (NSDI)*. These sources are not the only ones—worthwhile papers about computer systems appear in many other journals, conferences, and workshops. Complete copies of most of the papers listed here, including many of the older ones, can be found on the World Wide Web by an on-line search for an author's last name and a few words of the paper title. Even papers whose primary listing requires a subscription are often posted elsewhere as open resources.

The following pages contain suggestions for further reading about computer systems, both papers and books. The list makes no pretensions of being complete. Instead, the suggestions have been selected from a vast literature to emphasize the best available thinking, best illustrations of problems, and most interesting case studies of computer systems. The readings have been reviewed for obsolescence, but it is often the case that a good idea is still best described by a paper from some time ago, where the idea was developed in a context that no longer seems very interesting. Sometimes that early context is much simpler than today's systems, thus making it easier to see how the idea works. Often, an early author was the first on the scene, so it was necessary to describe things more completely than do modern authors who usually assume significant familiarity with the surroundings and with all of the predecessor systems. Thus the older readings included here provide a very useful complement to current works.

By its nature, the study of the engineering of computer systems overlaps with other areas of computer science, particularly computer architecture, programming languages, databases, information retrieval, security, and data communications. Each of those areas has an extensive literature of its own, and it is often not obvious where to draw the boundary lines. As a general rule, this reading list tries to provide only first-level guidance on where to start in those related areas.

One thing the reader must watch for is that the terminology of the computer systems field is not agreed upon, so the literature is often confusing even to the professional. In addition, the quality level of the literature is quite variable, ranging from the literate through the readable to the barely comprehensible. Although the selections here try to avoid that last category, the reader must still be prepared for some papers, however important in their content, that do not explain their subject as well as they could.

In the material that follows, each citation is accompanied by a comment suggesting why that paper is worth reading—its importance, interest, and relation to other readings. When a single paper serves more than one area of interest, cross-references appear rather than repeating the citation.

## 1 Systems

As mentioned above, a few wonderful and several really good books about computer systems have recently begun to appear. Here are the must-have items for the reference shelf of the computer systems designer. In addition to these books, the later groupings of readings by topic include other books, generally of narrower interest.

### 1.1 Wonderful Books About Systems

**1.1.1** 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 very sophisticated, the book is always very 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.

**1.1.2** Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991. ISBN: 978–0–471–50336–1. 720 pages.

Much work on performance analysis of computer systems originates in academic settings and focuses on analysis that is mathematically tractable rather than on measurements that matter. This book is at the other end of the spectrum. It is written by someone with extensive industrial experience but an academic flair for explaining things. If you have a real performance analysis problem, it will tell you how to tackle it, how to avoid measuring the wrong thing, and how to step by other pitfalls.

**1.1.3** Frederick P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 20th Anniversary edition, 1995. ISBN: 978–0–201–83595–3 (paperback). 336 pages.

Well-written and full of insights, this reading is by far the most significant one on the subject of controlling system development. This is where you learn why adding more staff to a project that is behind schedule will delay it further. Although a few of the chapters are now a bit dated, much of the material here is timeless. Trouble in system development is also timeless, as evidenced by continual reports

of failures of large system projects. Most successful system designers have a copy of this book on their bookshelf, and some claim to reread it at least once a year. Most of the 1995 edition is identical to the first, 1974, edition; the newer edition adds Brooks' *No Silver Bullets* paper (which is well worth reading) and some summarizing chapters.

**1.1.4** Lawrence Lessig. *Code and Other Laws of Cyberspace, Version 2.0*. Basic Books, 2006. ISBN: 978-0-465-03914-28 (paperback). 432 pages; 978-0-465-03913-5 (paperback). 320 pages. Also available on-line at http://codev2.cc/

This book is an updated version of an explanation by a brilliant teacher of constitutional law of exactly how law, custom, market forces, and architecture together regulate things. In addition to providing a vocabulary to discuss many of the legal issues surrounding technology and the Internet, a central theme of this book is that because technology raises issues that were foreseen neither by law nor custom, the default is that it will be regulated entirely by market forces and architecture, neither of which is subject to the careful and deliberative thought that characterize the development of law and custom. If you have any interest in the effect of technology on intellectual property, privacy, or free speech, this book is required reading.

**1.1.5** 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.

**1.1.6** 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 its position in the U.S. legal framework.

**1.1.7** 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 is remarkable for the range of system security problems it considers, 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 on-line.

## 1.2 Really Good Books About Systems

**1.2.1** 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 tendency 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.

**1.2.2** Thomas P. Hughes. *Rescuing Prometheus*. Vintage reprint (paperback), originally published in 1998. ISBN: 978–0679739388. 372 pages.

A retired professor of history and sociology explains the stories behind the management of four large-scale, one-of-a-kind system projects: the Sage air defense system, the Atlas rocket, the Arpanet (predecessor of the Internet), and the design phase of the Big Dig (Boston Central Artery/Tunnel). The thesis of the book is that such projects, in addition to unique engineering, also had to develop a different kind of management style that can adapt continuously to change, is loosely coupled with distributed control, and can identify a consensus among many players.

**1.2.3** Henry Petroski. *Design Paradigms: Case Histories of Error and Judgment in Engineering*. Cambridge University Press, 1994. ISBN: 978–0–521–46108–5 (hardcover), 978–0–521–46649–3 (paperback). 221 pages.

This remarkable book explores how the mindset of the designers (in the examples, civil engineers) allowed them to make what in retrospect were massive design errors. The failures analyzed range from the transportation of columns in Rome through the 1982 collapse of the walkway in the Kansas City Hyatt Regency Hotel, with a number of famous bridge collapses in between. Petroski analyzes particularly well how a failure of a scaled-up design often reveals that the original design worked correctly, but for a different reason than originally thought. There is no mention of computer systems in this book, but it contains many lessons for computer system designers.

**1.2.4** 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 very 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 *1.3.13*.

**1.2.5** 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*.

**1.2.6** Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufman, fourth edition, 2007. ISBN: 978-0-12-370548-8. 848 pages.

This book provides a systems perspective on computer networks. It represents a good balance of why networks are they way they are and a discussion of the important protocols in use. It follows a layering model but presents fundamental concepts independent of layering. In this way, the book provides a good discussion of timeless ideas as well as current embodiments of those ideas.

### *1.3 Good Books on Related Subjects Deserving Space on the Systems Bookshelf*

There are several other good books that many computer system professionals insist on having on their bookshelves. They don't appear in one of the previous categories because their central focus is not on systems or because the purpose of the book is somewhat narrower.

**1.3.1** Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill, second edition, 2001. 1,184 pages. ISBN: 978-0-07-297054-8 (hardcover); 978-0-262-53196-2 (M.I.T. Press paperback, not sold in U.S.A.)

**1.3.2** Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996. 872 pages ISBN: 978-1-55860-348-6.

Occasionally, a system designer needs an algorithm. Cormen et al. and Lynch's books are the place to find that algorithm, together with the analysis necessary to decide whether or not it is appropriate for the application. In a reading list on

theory, these two books would almost certainly be in one of the highest categories, but for a systems list they are better identified as supplementary.

**1.3.3** Douglas K. Smith and Robert C. Alexander. *Fumbling the Future*. William Morrow and Company, 1988. ISBN: 978-0-688-06959-9 (hardcover), 978-1-58348266-7 (iuniverse paperback reprint). 274 pages.

The history of computing is littered with companies that attempted to add general-purpose computer systems to an existing business—for examples, Ford, Philco, Zenith, RCA, General Electric, Honeywell, A. T. & T., and Xerox. None has succeeded, perhaps because when the going gets tough the option of walking away from this business is too attractive. This book documents how Xerox managed to snatch defeat from the jaws of victory by inventing the personal computer, then abandoning it.

**1.3.4** Marshall Kirk McKusick, Keith Bostic, and Michael J. Karels. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley, second edition, 1996. ISBN: 978-0-201-54979-9. 606 pages.

This book provides a complete picture of the design and implementation of the Berkeley version of the UNIX® operating system. It is well-written and full of detail. The 1989 first edition, describing 4.3BSD, is still useful.

**1.3.5** 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 very 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.

**1.3.6** Deborah G. Johnson and Helen Nissenbaum. *Computers, Ethics & Social Values*. Prentice-Hall, 1995. ISBN: 978-0-13-103110-4 (paperback). 714 pages.

A computer system designer is likely to consider reading a treatise on ethics to be a terribly boring way to spend the afternoon, and some of the papers in this extensive collection do match that stereotype. However, among the many scenarios, case studies, and other reprints in this volume are a large number of interesting and thoughtful papers about the human consequences of computer system design. This collection is a good place to acquire the basic readings concerning privacy, risks, computer abuse, and software ownership as well as professional ethics in computer system design.

**1.3.7** Carliss Y. Baldwin and Kim B. Clark. *Design Rules: Volume 1, The Power of Modularity*. M.I.T. Press, 2000. ISBN: 978-0-262-02466-2. 471 pages.

This book focuses wholely on modularity (as used by the authors, this term merges modularity, abstraction, and hierarchy) and offers an interesting

representation of interconnections to illustrate the power of modularity and of clean, abstract interfaces. The work uses these same concepts to interpret several decades of developments in the computer industry. The authors, from the Harvard Business School, develop a model of the several ways in which modularity operates by providing design options and making substitution easy. By the end of the book, most readers will have seen more than they wanted to know, but there are some ideas here that are worth at least a quick reading. (Despite the "Volume 1" in the title, there does not yet seem to be a Volume 2.)

**1.3.8** Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, fourth edition, 2003. ISBN: 978–0–13–066102–9. 813 pages.

This book provides a thorough tutorial introduction to the world of networks. Like the same author's book on operating systems (see reading *1.2.1*), this one also tends to emphasize the mechanics. But again it is a storehouse of up-to-date street knowledge, this time about computer communications, that is needed to get into (or perhaps avoid the need to consult) the rest of the literature. The book includes a selective and thoughtfully annotated bibliography on computer networks. An abbreviated version of this same material, sufficient for many readers, appears as a chapter of the operating systems book.

**1.3.9** David L. Mills. *Computer Network Time Synchronization: The Network Time Protocol*. CRC Press/Taylor & Francis, 2006. ISBN: 978–0849358050. 286 pages.

A comprehensive but very readable explanation of the Network Time Protocol (NTP), an under-the-covers protocol of which most users are unaware: NTP coordinates multiple timekeepers and distributes current date and time information to both clients and servers.

**1.3.10** Robert G. Gallager. *Principles of Digital Communication*. Cambridge University Press, 2008. ISBN: 978–0–521–87907–1. 422 pages.

This intense textbook focuses on the theory that underlies the link layer of data communication networks. It is not for casual browsing or for those easily intimidated by mathematics, but its an excellent reference source for analysis.

**1.3.11** Daniel P. Siewiorek and Robert S. Swarz. *Reliable Computer Systems: Design and Evaluation*. A. K. Peters Ltd., third edition, 1998. ISBN: 978–1–56881–092–8. 927 pages.

This is probably the best comprehensive treatment of reliability that is available, with well-explained theory and reprints of several case studies from recent literature. Its only defect is a slight "academic" bias in that little judgment is expressed on alternative methods, and some examples are without warning of systems that were never really deployed. The first, 1982, edition, with the title *The Theory and Practice of Reliable System Design*, contains an almost completely different (and much older) set of case studies.

**1.3.12** Bruce Schneier. *Secrets & Lies/Digital Security in a Networked World*. John Wiley & Sons, 2000. ISBN: 978-0-471-25311-2 (hardcover), 978-0-471-45380-2 (paperback). 432 pages.

This overview of security from a systems perspective provides much motivation, many good war stories (though without citations), and a high-level outline of how one achieves a secure system. Being an overview, it provides no specific guidance on the mechanics, other than to rely on people who know what they are doing. This is excellent book particularly for the manager who wants to go beyond the buzzwords and get an idea of what achieving computer system security involves.

**1.3.13** 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 very complete handbook on putting cryptography to work. It lacks the background and perspective of Reading 1.2.4, 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.

**1.3.14** Johannes A. Buchman. *Introduction to Cryptography*. Springer, second edition, 2004. ISBN: 978-0-387-21156-56 (hardcover). 335 pages.

Buchman provides a nice, concise introduction to number theory for cryptography.

**1.3.15** 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.

**1.3.16** 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 very 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.

**1.3.17** 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 very 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.

### *1.4 Ways of Thinking About Systems*

Quite a few books try to generalize the study of systems. They tend to be so abstract, however, that it is hard to see how they apply to anything, so none of them are listed here. Instead, here are five old but surprisingly relevant papers that illustrate ways to think about systems. The areas touched are allometry, aerodynamics, hierarchy, ecology, and economics.

**1.4.1** J[ohn] B[urdon] S[anderson] Haldane (1892–1964). On being the right size. In *Possible Worlds and Other Essays*, pages 20–28. Harper and Brothers Publishers, 1928. Also published by Chatto & Windus, London, 1927, and recently reprinted in John Maynard Smith, editor, *On Being the Right Size and Other Essays*, Oxford University Press, 1985. ISBN: 0–19–286045–3 (paperback), pages 1–8.

This is the classic paper that explains why a mouse the size of an elephant would collapse if it tried to stand up. It provides lessons on how to think about incommensurate scaling in all kinds of systems.

**1.4.2** Alexander Graham Bell (1847–1922). The tetrahedral principle in kite structure. *National Geographic Magazine 14*, 6 (June 1903), pages 219–251.

This classic paper demonstrates that arguments based on scale can be quite subtle. This paper—written at a time when physicists were still debating the theoretical possibility of building airplanes—describes the obvious scale argument against heavier-than-air craft and then demonstrates that one can increase the scale of an airfoil in different ways and that the obvious scale argument does not apply to all those ways. (This paper is a rare example of unreviewed vanity publication of an interesting engineering result. The *National Geographic* was—and still is—a Bell family publication.)

**1.4.3** Herbert A. Simon (1916–2001). The architecture of complexity. *Proceedings of the American Philosophical Society 106*, 6 (December 1962), pages 467–482. Republished as Chapter 4, pages 84–118, of *The Sciences of the Artificial*, M.I.T. Press, Cambridge, Massachusetts, 1969. ISBN: 0–262–191051–6 (hardcover); 0–262–69023–3 (paperback).

This paper is a tour-de-force of how hierarchy is an organizing tool for complex systems. The examples are breathtaking in their range and scope—from watchmaking and biology through political empires. The style of thinking shown in this paper suggests that it is not surprising that Simon later received the 1978 Nobel Prize in economics.

**1.4.4** LaMont C[ook] Cole (1916–1978). Man's effect on nature. *The Explorer: Bulletin of the Cleveland Museum of Natural History 11*, 3 (Fall 1969), pages 10–16.

This brief article looks at the Earth as an ecological system in which the actions of humans lead both to surprises and to propagation of effects. It describes a classic example of the propagation of effects: attempts to eliminate malaria in North Borneo led to an increase in the plague and roofs caving in.

**1.4.5** Garrett [James] Hardin (1915–). The tragedy of the commons. *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.

This seminal paper explores a property of certain economic situations in which Adam Smith's "invisible hand" works against everyone's interest. It is interesting for its insight into how to predict things about otherwise hard-to-model systems. In revisiting the subject 30 years later, Hardin suggested that the adjective "unmanaged" should be placed in front of "commons". Rightly or wrongly, the Internet is often described as a system to which the tragedy of the (unmanaged) commons applies.

### *1.5 Wisdom About System Design*

Before reading anything else on this topic, one should absorb the book by Brooks, *The Mythical Man-Month*, reading *1.1.3*, and the essay by Simon, "The architecture of complexity", reading *1.4.3*. The case studies on control of complexity in Section 1.9 also are filled with wisdom.

**1.5.1** Richard P. Gabriel. Worse is better. Excerpt from LISP: good news, bad news, how to win BIG, *AI Expert 6*, 6 (June 1991), pages 33–35.

This paper explains why doing the thing expediently sometimes works out to be a better idea than doing the thing right.

**1.5.2** Henry Petroski. Engineering: History and failure. *American Scientist 80*, 6 (November–December 1992), pages 523–526.

Petroski provides insight along the lines that one primary way that engineering makes progress is by making mistakes, studying them, and trying again. Petroski also visits this theme in two books, the most recent being reading *1.2.3*.

**1.5.3** Fernando J. Corbató. On building systems that will fail. *Communications of the ACM 34*, 9 (September 1991), pages 72–81. (Reprinted in the book by Johnson and Nissenbaum, reading *1.3.6*.)

The central idea in this 1991 Turing Award Lecture is that all ambitious systems will have failures, but those that were designed with that expectation are more likely to eventually succeed.

**1.5.4** Butler W. Lampson. Hints for computer system design. *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, in *Operating Systems Review 17*, 5 (October 1983), pages 33–48. Later republished, but with less satisfactory copy editing, in *IEEE Software 1*, 1 (January 1984), pages 11–28.

This encapsulation of insights is expressed as principles that seem to apply to more than one case. It is worth reading by all system designers.

**1.5.5** Jon Bentley. The back of the envelope—programming pearls. *Communications of the ACM 27*, 3 (March 1984), pages 180–184.

One of the most important tools of a system designer is the ability to make rough but quick estimates of how big, how long, how fast, or how expensive a design will be. This brief note extols the concept and gives several examples.

**1.5.6** Jeffrey C. Mogul. Emergent (mis)behavior vs. complex software systems. *Proceedings of the First European Conference on Computer Systems* (EuroSys 2006, Leuven, Belgium), pages 293–304. ACM Press, 2006, ISBN: 1–59593–322–0. Also in *Operating Systems Review 40*, 4 (October 2006).

This paper explores in depth the concept of emergent properties described in Chapter 1, providing a nice collection of examples and tying together issues and problems that arise throughout computer and network system design. It also suggests a taxonomy of emergent properties, lays out suggestions for future research, and includes a comprehensive and useful bibliography.

**1.5.7** Pamela Samuelson, editor. Intellectual property for an information age. *Communications of the ACM 44*, 2 (February 2001), pages 67–103.

This work is a special section comprising several papers about the challenges of intellectual property in a digital world. Each of the individual articles is written by a member of a new generation of specialists who understand both technology and law well enough to contribute some thoughtful insights to both domains.

**1.5.8** Mark R. Chassin and Elise C. Becher. The wrong patient. *Annals of Internal Medicine 136* (June 2002), pages 826–833.

This paper is a good example, first, of how complex systems fail for complex reasons and second, of the value of the "keep digging" principle. The case study presented here centers on a medical system failure in which the wrong patient was operated on. Rather than just identifying the most obvious reason, the case study concludes that there were a dozen or more opportunities in which the error that led to the failure should have been detected and corrected, but for various reasons all of those opportunities were missed.

**1.5.9** P[hillip] J. Plauger. Chocolate. *Embedded Systems Programming* 7, 3 (March 1994), pages 81–84.

This paper provides a remarkable insight based on the observation that many failures in a bakery can be remedied by putting more chocolate into the mixture. The author manages, with only a modest stretch, to convert this observation into a more general technique of keeping recovery simple, so that it is likely to succeed.

### 1.6 Changing Technology and its Impact on Systems

**1.6.1** Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics 38*, 8 (April 19, 1965), pages 114–117. Reprinted in *Proceedings of the IEEE 86*, 1 (January 1998), pages 82–85.

This paper defined what we now call Moore's law. The phenomena Moore describes have driven the rate of technology improvement for more than four decades. This paper articulates why and displays the first graph to plot Moore's law, based on five data points.

**1.6.2** John L. Hennessy and Norman P. Jouppi. Computer technology and architecture: An evolving interaction. *IEEE Computer 24*, 9 (September 1991), pages 19–29.

Although some of the technology examples are a bit of out of date, the systems thinking and the paper's insights remain relevant.

**1.6.3** Ajanta Chakraborty and Mark R. Greenstreet. Efficient self-timed interfaces for crossing clock domains. *Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems*, IEEE Computer Society (May 2003), pages 78–88. ISBN: 0–7695–1898–2.

This paper addresses the challenge of having a fast, global clock on a chip by organizing the resources on a chip as a number of synchronous islands connected by asynchronous links. This design may pose problems for constructing perfect arbiters (see Section 5.2.8).

**1.6.4** Anant Agarwal and Markus Levy. The KILL Rule for multicore. *44th ACM/IEEE Conference on Design Automation* (June 2007), pages 750–753. ISBN: 978–1–59593–627–1.

This short paper looks ahead to multiprocessor chips that contain not just four or eight, but thousands of processors. It articulates a rule for power-efficient designs: Kill If Less than Linear. For example, the designer should increase the chip area devoted to a resource such as a cache only if for every 1% increase in area there is at least a 1% increase in chip performance. This rule focuses attention on those design elements that make most effective use of the chip area and from back-of-the-envelope calculations favors increasing processor count (which the paper assumes to provide linear improvement) over other alternatives.

**1.6.5** Stephen P. Walborn et al. Quantum erasure. *American Scientist 91*, 4 (July–August 2003), pages 336–343.

This paper was written by physicists and requires a prerequisite of undergraduate-level modern physics, but it manages to avoid getting into graduate-level quantum mechanics. The strength of the article is its clear identification of what is reasonably well understood and what is still a mystery about these phenomena. That identification seems to be of considerable value both to students of physics, who may be inspired to tackle the parts that are not understood, and to students of cryptography, because knowing what aspects of quantum cryptography are still mysteries may be important in deciding how much reliance to place on it.

### 1.7 Dramatic Visions

Once in a while a paper comes along that either has a dramatic vision of what future systems might do or takes a sweeping new look at some aspect of systems design that had previously been considered to be settled. The ideas found in the papers listed in reading sections 1.7 and 1.8 often become part of the standard baggage of all future writers in the area, but the reprises rarely do justice to the originals, which are worth reading if only to see how the mind of a visionary (or revisionist) works.

**1.7.1** Vannevar Bush. As we may think. A*tlantic Monthly 176*, 1 (July 1945), pages 101–108. Reprinted in Adele J. Goldberg, *A History of Personal Workstations*, Addison-Wesley, 1988, pages 237–247 and also in Irene Greif, ed., *Computer-Supported Cooperative Work: A Book of Readings*, Morgan Kaufman, 1988. ISBN: 0–934613–57–5.

Bush looked at the (mostly analog) computers of 1945 and foresaw that they would someday be used as information engines to augment the human intellect.

**1.7.2** John G. Kemeny, with comments by Robert M. Fano and Gilbert W. King. A library for 2000 A.D. In Martin Greenberger, editor, *Management and the Computer of the Future*, M.I.T. Press and John Wiley, 1962, pages 134–178. (Out of print.)

It has taken 40 years for technology to advance far enough to make it possible to implement Kemeny's vision of how the library might evolve when computers are used in its support. Unfortunately, the engineering that is required still hasn't been done, so the vision has not yet been realized, but Google has stated a similar vision and is making progress in realizing it; see reading *3.2.4*.

**1.7.3** [Alan C. Kay, with the] Learning Research Group. *Personal Dynamic Media*. Xerox Palo Alto Research Center Systems Software Laboratory Technical Report SSL–76–1 (undated, circa March 1976).

Alan Kay was imagining laptop computers and how they might be used long before most people had figured out that desktop computers might be a good idea. He gave many inspiring talks on the subject, but he rarely paused long enough to write anything down. Fortunately, his colleagues captured some of his thoughts in this technical report. An edited version of this report, with some

pictures accidentally omitted, appeared in a journal in the year following this technical report: Alan [C.] Kay and Adele Goldberg. Personal dynamic media. *IEEE Computer 10*, 3 (March 1977), pages 31–41. This paper was reprinted with omitted pictures restored in Adele J. Goldberg, *A history of personal workstations*, Addison-Wesley, 1988, pages 254–263. ISBN: 0–201–11259–0.

**1.7.4** Doug[las] C. Engelbart. *Augmenting Human Intellect:A Conceptual Framework*. Research Report AFOSR–3223, Stanford Research Institute, Menlo Park, California, October 1962. Reprinted in Irene Greif, ed., *Computer-Supported Cooperative Work:A Book of Readings*, Morgan Kaufman, 1988. ISBN: 0–934613–57–5.

In the early 1960s Engelbart saw that computer systems would someday be useful in myriad ways as personal tools. Unfortunately, the technology of his time, multimillion-dollar mainframes, was far too expensive to make his vision practical. Today's personal computers and engineering workstations have now incorporated many of his ideas.

**1.7.5** F[ernando] J. Corbató and V[ictor] A. Vyssotsky. Introduction and overview of the Multics system. A*FIPS 1965 Fall Joint Computer Conference 27*, part I (1965), pages 185–196.

Working from a few primitive examples of time-sharing systems, Corbató and his associates escalated the vision to an all-encompassing computer utility. This paper is the first in a set of six about Multics in the same proceedings, pages 185–247.

## 1.8 Sweeping New Looks

**1.8.1** Jack B. Dennis and Earl C. Van Horne. Programming semantics for multiprogrammed computations. *Communications of the ACM 9*, 3 (March 1966), pages 143–155.

This paper set the ground rules for thinking about concurrent activities, both the vocabulary and the semantics.

**1.8.2** J. S. Liptay. Structural aspects of the System/360 model 85: II. The cache. *IBM Systems Journal 7*, 1 (1968), pages 15–21.

The idea of a cache, look-aside, or slave memory had been suggested independently by Francis Lee and Maurice Wilkes some time around 1963, but it was not until the advent of LSI technology that it became feasible to actually build one in hardware. As a result, no one had seriously explored the design space options until the designers of the IBM System/360 model 85 had to come up with a real implementation. Once this paper appeared, a cache became a requirement for most later computer architectures.

**1.8.3** Claude E. Shannon. The communication theory of secrecy systems. *Bell System Technical Journal 28*, 4 (October 1949), pages 656–715.

This paper provides the underpinnings of the theory of cryptography, in terms of information theory.

**1.8.4** Whitfield Diffie and Martin E. Hellman. Privacy and authentication: An introduction to cryptography. *Proceedings of the IEEE 67*, 3 (March 1979), pages 397–427.

This is the first technically competent paper on cryptography since Shannon in the unclassified literature, and it launched modern unclassified study. It includes a complete and scholarly bibliography.

**1.8.5** 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.

**1.8.6** Charles T. Davies, Jr. Data processing spheres of control. *IBM Systems Journal 17*, 2 (1978), pages 179–198. Charles T. Davies, Jr. Recovery semantics for a DB/DC system. *1973 ACM National Conference 28* (August 1973), pages 136–141.

This pair of papers—vague but thought-provoking—gives a high-level discussion of "spheres of control", a notion closely related to atomicity. Everyone who writes about transactions mentions that they found these two papers inspiring.

**1.8.7** 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.

**1.8.8** Leonard Kleinrock. *Communication Nets: Stochastic Message Flow and Delay*. McGraw-Hill, 1964. Republished by Dover, 2007. ISBN: 0–486–45880–6. 224 pages.

**1.8.9** Paul Baran, S. Boehm, and J. W. Smith. *On Distributed Communications*. A series of 11 memoranda of the RAND Corporation, Santa Monica, California, August 1964.

Since the growth in the Internet's popularity, there has been considerable discussion about who first thought of packet switching. It appears that Leonard Kleinrock, working in 1961 on his M.I.T. Ph.D. thesis on more effective ways

of using wired networks, and Paul Baran and his colleagues at Rand, working in 1961 on survivable communications, independently proposed the idea of packet switching at about the same time; both wrote internal memoranda in 1961 describing their ideas. Neither one actually used the words "packet switching", however; that was left to Donald Davies of the National Physical Laboratory, who coined that label several years later.

**1.8.10** Lawrence G. Roberts and Barry D. Wessler. Computer network development to achieve resource sharing. *AFIPS Spring Joint Computer Conference 36* (May 1970), pages 543–549.

This paper and four others presented at the same conference session (pages 543–597) represent the first public description of the ARPANET, the first successful packet-switching network and the prototype for the Internet. Two years later, *AFIPS Spring Joint Computer Conference 40* (1972), pages 243–298, presented five additional, closely related papers. The discussion of priority concerning reading *1.8.8* and reading *1.8.9* is somewhat academic; it was Roberts's sponsorship of the ARPANET that demonstrated the workability of packet switching.

**1.8.11** V[inton G.] Cerf et al. Delay-Tolerant Networking Architecture. *Request for Comments RFC 4838*, Internet Engineering Task Force (April 1997).

This document describes an architecture that evolved from a vision for an Interplanetary Internet, an Internet-like network for interplanetary distances. This document introduces several interesting ideas and highlights some assumptions that people make in designing networks without realizing it. NASA performed its first successful tests of a prototype implementation of a delay-tolerant network.

**1.8.12** Jim Gray et al. *Terascale Sneakernet. Using Inexpensive Disks for Backup, Archiving, and Data Exchange*. Microsoft Technical Report MS-TR-02-54 (May 2002). http://arxiv.org/pdf/cs/0208011

Sneakernet is a generic term for transporting data by physically delivering a storage device rather than sending it over a wire. Sneakernets are attractive when data volume is so large that electronic transport will take a long time or be too expensive, and the latency until the first byte arrives is less important. Early sneakernets exchanged programs and data using floppy disks. More recently, people have exchanged data by burning CDs and carrying them. This paper proposes to build a sneakernet by sending hard disks, encapsulated in a small, low-cost computer called a storage brick. This approach allows one to transfer by mail terabytes of data across the planet in a few days. By virtue of including a computer and operating system, it minimizes compatibility problems that arise when transferring the data to another computer.

Several other papers listed under specific topics also provide sweeping new looks or have changed the way people think about systems: Simon, The architecture of complexity, reading *1.4.3*; Thompson, Reflections on trusting trust, reading *11.3.3*;

Lampson, Hints for computer system design, reading *1.5.4*; and Creasy's VM/370 paper, reading *5.6.1*.

### *1.9 Keeping Big Systems Under Control*

**1.9.1** F[ernando] J. Corbató and C[harles] T. Clingen. A managerial view of the Multics system development. In Peter Wegner, *Research Directions in Software Technology*, M.I.T. Press, Cambridge, Massachusetts, 1979, pages 139–158. ISBN: 0–262–23096–8.

**1.9.2** W[illiam A.] Wulf, R[oy] Levin, and C. Pierson. Overview of the Hydra operating system development. *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, in *Operating Systems Review 9*, 5 (November 1975), pages 122–131.

**1.9.3** Thomas R. Horsley and William C. Lynch. Pilot: A software engineering case study. *Fourth International Conference on Software Engineering* (September 1979), pages 94–99.

These three papers are early descriptions of the challenges of managing and developing large systems. They are still relevant and easy to read, and provide complementary insights.

**1.9.4** Effy Oz. When professional standards are lax: The CONFIRM failure and its lessons. *Communications of the ACM 37*, 10 (October 1994), pages 30–36.

CONFIRM is an airline/hotel/rental-car reservation system that never saw the light of day despite four years of work and an investment of more than $100M. It is one of many computer system developments that went out of control and finally were discarded without ever having been placed in service. One sees news reports of software disasters of similar magnitude a few times each year. It is difficult to obtain solid facts about system development failures because no one wants to accept the blame, especially when lawsuits are pending. This paper suffers from a shortage of facts and an oversimplistic recommendation that better ethics are all that are needed to solve the problem. (It seems likely that the ethics and management problems simply delayed recognition of the inevitable.) Nevertheless, it provides a sobering view of how badly things can go wrong.

**1.9.5** Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *Computer 26*, 7 (July 1993), pages 18–41. (Reprinted in reading *1.3.6*.)

This is another sobering view of how badly things can go wrong. In this case, the software controller for a high-energy medical device was inadequately designed; the device was placed in service, and lethal injuries ensued. This paper manages to inquire quite deeply into the source of the problems. Unfortunately, similar mistakes have been made since; see, for example, United States Nuclear Regulatory Commission Information Notice 2001–8s1 (June 2001), which describes radiation therapy overexposures in Panama.

**1.9.6** Joe Morgenstern. City perils: The fifty-nine-story crisis. *The New Yorker 71*, 14 (May 29, 1995), pages 45–53.

This article discusses how an engineer responded to the realization that a skyscraper he had designed was in danger of collapsing in a hurricane.

**1.9.7** Eric S. Raymond. The cathedral and the bazaar. In *The Cathedral and The Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, pages 19–64. O'Reilly Media Inc., 2001. ISBN: 978–0596001087, 241 pages.

The book is based on a white paper of the same title that compares two styles of software development: the Cathedral model, which is used mostly by commercial software companies and some open-source projects such as the BSD operating system; and the Bazaar model, which is exemplified by development of the GNU/Linux operating system. The work argues that the Bazaar model leads to better software because the openness and independence of Bazaar allow anyone to become a participant and to look at anything in the system that seems of interest: "Given enough eyeballs, all bugs are shallow".

**1.9.8** Philip M Boffey. Investigators agree N.Y. blackout of 1977 could have been avoided. *Science 201*, 4360 (September 15, 1978), pages 994–996.

This is a fascinating description of how the electrical generation and distribution system of New York's Consolidated Edison fell apart when two supposedly tolerable faults occurred in close succession, recovery mechanisms did not work as expected, attempts to recover manually got bogged down by the system's complexity, and finally things cascaded out of control.

## 2 Elements of Computer System Organization

To learn more about the basic abstractions of memory and interpreters, the book *Computer Architecture* by Patterson and Hennessy (reading 1.1.1) is one of the best sources. Further information about the third basic abstraction, communication links, can be found in readings section 7.

### *2.1 Naming Systems*

**2.1.1** Bruce [G.] Lindsay. Object naming and catalog management for a distributed database manager. *Proceedings of the Second International Conference on Distributed Computing Systems*, Paris, France (April 1981), pages 31–40. Also IBM San Jose Research Laboratory Technical Report RJ2914 (August 1980). 17 pages.

This paper, a tutorial treatment of names as used in database systems, begins with a better-than-average statement of requirements and then demonstrates how those requirements were met in the R* distributed database management system.

**2.1.2** Yogen K. Dalal and Robert S. Printis. 48-bit absolute Internet and Ethernet host numbers. *Proceedings of the Seventh Data Communications Symposium*,

Mexico City, Mexico, (October 1981), pages 240–245. Also Xerox Office Products Division Technical Report OPD–T8101 (July 1981), 14 pages.

This paper describes how hardware addresses are handled in the Ethernet local area network.

**2.1.3** Theodor Holm Nelson. *Literary Machines, Ed. 87.1*. Project Xanadu, San Antonio, Texas, 1987. ISBN: 0–89347–056–2 (paperback). Various pagings.

Project Xanadu is an ambitious vision of a future in which books are replaced by information organized in the form of a naming network, in the form that today is called "hypertext". The book, being somewhat non-linear, is a primitive example of what Nelson advocates.

### 2.2 The UNIX® System

The following readings and the book by Marshall McKusick et al., reading *1.3.4*, are excellent sources on the UNIX system to follow up the case study in Section 2.5. A good, compact summary of its main features can be found in Tanenbaum's operating systems book [reading 1.2.1], which also covers Linux.

**2.2.1** Dennis M. Ritchie and Ken [L.] Thompson. The UNIX time-sharing system. *Bell System Technical Journal 57*, 6, part 2 (1978), pages 1905–1930.

This paper describes an influential operating system with very low-key, but carefully chosen and hard-to-discover, objectives. The system provides a hierarchical catalog structure and succeeds in keeping naming completely distinct from file management. An earlier version of this paper appeared in the *Communications of the ACM 17*, 7 (July, 1974), pages 365–375, after being presented at the *Fourth ACM Symposium on Operating Systems Principles*. The UNIX system evolved rapidly between 1973 and 1978, so the *BSTJ* version, though harder to find, contains significant additions, both in insight and in technical content.

**2.2.2** John Lions. *Lions' Commentary on UNIX 6th Edition with Source Code*. Peer-to-peer communications, 1977. ISBN: 978–1–57398–013–7, 254 pages.

This book contains the source code for UNIX Version 6, with comments to explain how it works. Although Version 6 is old, the book remains an excellent starting point for understanding how the system works from the inside, because both the source code and the comments are short and succinct. For decades, this book was part of the underground literature from which designers learned about the UNIX system, but now it is available to the public.

## 3 The Design of Naming Schemes

Almost any system has a naming plan, and many of the interesting naming plans can be found in papers that describe a larger system. Any reader interested in naming should study the Domain Name System, reading *4.3*, and the topic of Section 4.4.

### 3.1 Addressing Architectures

Several early sources still contain some of the most accessible explanations of designs that incorporate advanced naming features directly in hardware.

**3.1.1** Jack B. Dennis. Segmentation and the design of multiprogrammed computer systems. *Journal of the ACM 12*, 4 (October 1965), pages 589–602.

This is the original paper outlining the advantages of providing naming support in hardware architecture.

**3.1.2** 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.

**3.1.3** Elliott I. Organick. *Computer System Organization, The B5700/B6700 Series*. Academic Press, 1973. ISBN: 0–12–528250–8. 132 pages.

The Burroughs Descriptor system explained in this book is apparently the only example of a hardware-supported naming system actually implemented before the advent of microprogramming.

**3.1.4** Elliott I. Organick. *The Multics System: an Examination of its Structure*. M.I.T. Press, Cambridge, Massachusetts, 1972. ISBN: 0–262–15012–3. 392 pages.

This book explores every detail and ramification of the extensive naming mechanisms of Multics, both in the addressing architecture and in the file system.

**3.1.5** R[oger] M. Needham and A[ndrew] D. Birrell. The CAP filing system. *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, in *Operating Systems Review 11*, 5 (November 1977), pages 11–16.

The CAP file system is one of the few implemented examples of a genuine naming network.

### 3.2 Examples

**3.2.1** Paul J. Leach, Bernard L. Stumpf, James A. Hamilton, and Paul H. Levine. UIDs as internal names in a distributed file system. In *ACM SIGACT–SIGOPS Symposium on Principles of Distributed Computing*, Ottawa, Ontario (August 18–20, 1982), pages 34–41.

The Apollo DOMAIN system supports a different model for distributed function. It provides a shared primary memory called the Single Level Store, which extends transparently across the network. It is also one of the few systems to make substantial use of unstructured unique identifiers from a compact set as object names. This paper focuses on this latter issue.

**3.2.2** Rob Pike et al. Plan 9 from Bell Labs. *Computing Systems 8*, 3 (Summer 1995), pages 221–254. An earlier version by Rob Pike, Dave Presotto, Ken Thompson,

and Howard Trickey appeared in *Proceedings of the Summer 1990 UKUUG Conference* (1990), London, pages 1–9.

This paper describes a distributed operating system that takes the UNIX system idea that every resource is a file one step further by using it also for network and window system interactions. It also extends the file idea to a distributed system by defining a single file system protocol for access to all resources, whether they are local or remote. Processes can mount any remote resources into their name space, and to the user these remote resources behave just like local resources. This design makes users perceive the system as an easy-to-use time-sharing system that behaves like a single powerful computer, instead of a collection of separate computers.

**3.2.3** Tim Berners-Lee et al. The World Wide Web. *Communications of the ACM 37*, 8 (August 1994), pages 76–82.

Many of the publications about the World Wide Web are available only on the Web, with a good starting point being the home page of the World Wide Web Consortium at <http://w3c.org/>.

**3.2.4** Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Proceedings of the 7th WWW Conference*, Brisbane, Australia (April 1998). Also in *Computer Networks 30* (1998), pages 107–117.

This paper describes an early version of Google's search engine. It also introduces the idea of page rank to sort the results to a query in order of importance. Search is a dominant way in which users "name" Web pages.

**3.2.5** 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 involves material of Chapter 11 [on-line].

## 4 Enforcing Modularity with Clients and Services

Many systems are organized in a client/service style. A system that provides a good case study is the Network File System (see Section 4.5). The following papers provide some other examples.

### 4.1 Remote Procedure Call

**4.1.1** 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.

**4.1.2** Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, in *Operating Systems Review 27*, 5 (December 1993), pages 217–230.

This paper describes a programming language for distributed applications based on remote procedure calls, which hide most "distributedness" from the programmer.

**4.1.3** Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java™ system. *Computing Systems 9*, 4 (1996), pages 265–290. Originally published in *Proceedings of the Second USENIX Conference on Object-Oriented Technologies Volume 2* (1996).

This paper presents a remote procedure call system for the Java programming language. It provides a clear description of how an RPC system can be integrated with an object-oriented programming language and the new exception types RPC introduces.

### 4.2 Client/Service Systems

**4.2.1** Daniel Swinehart, Gene McDaniel, and David [R.] Boggs. WFS: A simple shared file system for a distributed environment. *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, in *Operating Systems Review 13*, 5 (December 1979), pages 9–17.

This early version of a remote file system opens the door to the topic of distribution of function across connected cooperating computers. The authors' specific goal was to keep things simple; thus, the relationship between mechanism and goal is much clearer than in more modern, but more elaborate, systems.

**4.2.2** Robert Scheifler and James Gettys. The X Window System. *ACM Transactions on Graphics 5*, 2 (April 1986), pages 79–109.

The X Window System is the window system of choice on practically every engineering workstation in the world. It provides a good example of using the client/service model to achieve modularity. One of the main contributions of the X Window System is that it remedied a defect that had crept into the UNIX system when displays replaced typewriters: the display and keyboard were the only hardware-dependent parts of the UNIX application programming interface. The X Window System allowed display-oriented UNIX applications to be completely

independent of the underlying hardware. In addition, the X Window System interposes an efficient network connection between the application and the display, allowing configuration flexibility in a distributed system.

**4.2.3** 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.

### 4.3 Domain Name System (DNS)

The domain name system is one of the most interesting distributed systems in operation. It is not only a building block in many distributed applications, but is itself an interesting case study, offering many insights for anyone wanting to build a distributed system or a naming system.

**4.3.1** Paul V. Mockapetris and Kevin J. Dunlap. Development of the Domain Name System. *Proceedings of the SIGCOMM 1988 Symposium*, pages 123–133. Also published in *ACM Computer Communications Review 18*, 4 (August 1988), pages 123–133, and republished in *ACM Computer Communications Review 25*, 1 (January 1995), pages 112–122.

**4.3.2** Paul [V.] Mockapetris. Domain names—Concepts and facilities. *Request for Comments RFC 1034*, Internet Engineering Task Force (November 1987).

**4.3.3** Paul [V.] Mockapetris. Domain names—Implementation and specification. *Request for Comments RFC 1035*, Internet Engineering Task Force (November 1987).

These three documents explain the DNS protocol.

**4.3.4** Paul Vixie. DNS Complexity. *ACM Queue 5*, 3 (April 2007), pages 24–29.

This paper uncovers many of the complexities of how DNS, described in the case study in Section 4.4, works in practice. The protocol for DNS is simple, and no complete, precise specification of the system exists. The author argues that the current descriptive specification of DNS is an advantage because it allows various implementations to evolve to include new features as needed. The paper describes many of these features and shows that DNS is one of the most interesting distributed systems in use today.

## 5 Enforcing Modularity with Virtualization

### 5.1 Kernels

The readings on the UNIX system (see readings section 2.2) are a good starting point for studying kernels.

**5.1.1** Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM 13*, 4 (April 1970), pages 238–241.

The RC–4000 was the first, and may still be the best explained, system to use messages as the primary concurrency coordination mechanism. It is also what would today be called a microkernel design.

**5.1.2** M. Frans Kaashoek et al. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, in *Operating Systems Review 31*, 5 (December 1997), pages 52–65.

The exokernel provides an extreme version of separation of policy from mechanism, sacrificing abstraction to expose (within protection constraints) all possible aspects of the physical environment to the next higher layer, giving that higher layer maximum flexibility in creating abstractions for its preferred programming environment, or tailored to its preferred application.

## 5.2 Type Extension as a Modularity Enforcement Tool

**5.2.1** 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.

**5.2.2** 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.

**5.2.3** Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *Operating Systems Review 41*, 2 (April 2007), pages 37–49.

Singularity is an operating system that uses type-safe languages to enforce modularity between different software modules, instead of relying on virtual-memory hardware. The kernel and all applications are written in a strongly typed programming language with automatic garbage collection. They run in a single address space and are isolated from each other by the language runtime. They

can interact with each other only through communication channels that carry type-checked messages.

### 5.3 Virtual Processors: Threads

**5.3.1** Andrew D. Birrell. *An introduction to programming with threads*. Digital Equipment Corporation Systems Research Center Technical Report #35, January 1989. 33 pages. (Also appears as Chapter 4 of Greg Nelson, editor, *Systems Programming with Modula-3*, Prentice-Hall, 1991, pages 88–118.) A version for the C# programming language appeared as Microsoft Research Report MSR-TR-2005-68.

This is an excellent tutorial, explaining the fundamental issues clearly and going on to show the subtleties involved in exploiting threads correctly and effectively.

**5.3.2** Thomas E. Anderson et al. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems 10*, 1 (February 1992), pages 53–79. Originally published in *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, in *Operating Systems Review 25*, 5 (December 1991), pages 95–109.

The distinction between user threads and kernel threads comes to the fore in this paper, which offers a way of getting the advantages of both by having the right kind of user/kernel thread interface. The paper also revisits the idea of a virtual processor, but in a multiprocessor context.

**5.3.3** David D. Clark. The structuring of systems using upcalls. *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, in *Operating Systems Review 19*, 5 (December 1985), pages 171–180.

Attempts to impose modular structure by strict layering sometimes manage to overlook the essence of what structure is most appropriate. This paper describes a rather different intermodule organization that seems to be especially effective when dealing with network implementations.

**5.3.4** Jerome H. Saltzer. *Traffic Control in a Multiplexed Computer System*. Ph.D. Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering, June 1966. Also available as Project MAC Technical Report TR-30, 1966.

This work describes what is probably the first systematic virtual processor design and thread package, the multiprocessor multiplexing scheme used in the Multics system. It defines the coordination primitives BLOCK and WAKEUP, which are examples of binary semaphores assigned one per thread.

**5.3.5** Rob Pike et al. Processor sleep and wakeup on a shared-memory multiprocessor. *Proceedings of the EurOpen Conference* (1991), pages 161–166.

This well-written paper does an excellent job of explaining how difficult it is to get preemptive multiplexing, handling interrupts, and implementing coordination primitives correct on shared-memory multiprocessor.

### 5.4 Virtual Memory

There are few examples of papers that describe a simple, clean design. The older papers (some can be found in reading section 3.1) get bogged down in technology constraints; the more recent papers (some of the them can be found in reading section 6.1 on multilevel memory management) often get bogged down in performance optimizations. The case study on the evolution of enforcing modularity with the Intel x86 (see Section 5.7 of Chapter 5) describes virtual memory support in the most widely used processor and shows how it evolved over time.

**5.4.1** A[ndre] Bensoussan, C[harles] T. Clingen, and R[obert] C. Daley. The Multics virtual memory: Concepts and design. *Communications of the ACM 15*, 5 (May 1972), pages 308–318.

This is a good description of a system that pioneered the use of high-powered addressing architectures to support a sophisticated virtual memory system, including memory-mapped files. The design was constrained and shaped by the available hardware technology (0.3 MIPS processor with an 18–bit address space), but the paper is a classic and easy to read.

### 5.5 Coordination

Every modern textbook covers the topic of coordination but typically brushes past the subtleties and also typically gives the various mechanisms more emphasis than they deserve. These readings either explain the issues much more carefully or extend the basic concepts in various directions.

**5.5.1** E[dsger] W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, NATO Advanced Study Institute, Villard-de-Lans, 1966. Academic Press, 1968, pages 43–112.

This paper introduces semaphores, the synchronizing primitive most often used in academic exercises, and is notable for its very careful, step-by-step development of the requirements for mutual exclusion and its implementation. Many modern treatments ignore the subtleties discussed here as if they were obvious. They aren't, and if you want to understand synchronization you should read this paper.

**5.5.2** E[dsger] W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM 8*, 9 (September 1965), page 569.

In this very brief paper, Dijkstra first reports Dekker's observation that multiprocessor locks can be implemented entirely in software, relying on the hardware to guarantee only that read and write operations have before-or-after atomicity.

**5.5.3** Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems 5*, 1 (February 1987), pages 1–11.

This paper presents a fast version of a software-only implementation of locks and gives an argument as to why this version is optimal.

**5.5.4** David P. Reed and Rajendra K. Kanodia. Synchronization with eventcounts and sequencers. *Communications of the ACM 22*, 2 (February 1979), pages 115–123.

This paper introduces an extremely simple coordination system that uses less powerful primitives for sequencing than for mutual exclusion; a consequence is simple correctness arguments.

**5.5.5** Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM 23*, 2 (February 1980), pages 105–117.

This is a nice discussion of the pitfalls involved in integrating concurrent activity coordination into a programming language.

**5.5.6** Stefan Savage et al. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems 15*, 4 (November 1997), pages 391–411. Also in the *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (October 1997).

This paper describes an interesting strategy for locating certain classes of locking mistakes: instrument the program by patching its binary data references; then watch those data references to see if the program violates the locking protocol.

**5.5.7** Paul E. McKenney et al. Read-copy update. *Proceedings of the Ottawa Linux Symposium*, 2002, pages 338–367.

This paper observes that locks can be an expensive mechanism for before-or-after atomicity for data structures that are mostly read and infrequently modified. The authors propose a new technique, read-copy update (RCU), which improves performance and scalability. The Linux kernel uses this mechanism for many of its data structures that processors mostly read.

**5.5.8** Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* 11, 1 (January 1991), pages 124–149.

This paper introduces the goal of wait-free synchronization, now often called non-blocking coordination, and gives non-blocking, concurrent implementations of common data structures such as sets, lists, and queues.

**5.5.9** Timothy L. Harris. A pragmatic implementation of non-blocking linked lists. *Proceedings of the fifteenth International Symposium on Distributed Computing*, (October 2001), pages 300–314.

This paper describes a practical implementation of a linked list in which threads can insert concurrently without blocking.

See also reading *5.1.1*, by Brinch Hansen, which uses messages as a coordination technique, and reading *5.3.1* by Birrell, which describes a complete set of coordination primitives for programming with threads.

## 5.6 Virtualization

**5.6.1** Robert J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development 25*, 5 (1981), pages 483–490.

This paper is an insightful retrospective about a mid-1960s project to virtualize the IBM 360 computer architecture and the development that led to VM/370, which in the 1970s became a popular virtual machine system. At the time, the unusual feature of VM/370 was its creation of a strict, by-the-book, hardware virtual machine, thus providing the ability to run any system/370 program in a controlled environment. Because it was a pioneer project, the author explained things particularly well, thus providing a good introduction to the concepts and problems in implementing virtual machines.

**5.6.2** Edouard Bugnion et al. Disco: running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems 15*, 14 (November 1997), pages 412–447.

This paper brought virtual machines back as a mainstream way of building systems.

**5.6.3** Carl Waldspurger. Memory resource management in VMware ESX server. *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation* (December 2002), pages 181–194.

This well-written paper introduces a nice trick (a balloon driver) to decide how much physical memory to give to guest operating systems.

**5.6.4** Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *Proceedings of the Twelfth Symposium on Architectural Support for Programming Languages and Operating Systems* (October 2006). ISBN: 1–59593–451–0. Also in *Operating Systems Review 40*, 5 (December 2006), pages 2–13.

This paper describes how one can virtualize the Intel x86 instruction set to build a high-performance virtual machine. It compares two implementation strategies: one that uses software techniques, such as binary rewriting, to virtualize the instruction set, and one that uses recent hardware additions to the x86 processor to make virtualizing easier. The comparison provides insights about implementing modern virtual machines and operating system support in modern x86 processors.

Also see the paper on the secure virtual machine monitor for the VAX machine, reading *11.3.5*.

## 6 Performance

### 6.1 Multilevel Memory Management

An excellent discussion of memory hierarchies, with special attention paid to the design space for caches, can be found in Chapter 5 of the book by Patterson and Hennessy, reading *1.1.1*. A lighter-weight treatment focused more on virtual memory, and including a discussion of stack algorithms, can be found in Chapter 3 of Tanenbaum's computer systems book, reading *1.2.1*.

**6.1.1** R[obert] A. Frieburghouse. Register allocation via usage counts. *Communications of the ACM 17*, 11 (November 1974), pages 638–642.

This paper shows that compiler code generators must do multilevel memory management and that they have the same problems as do caches and paging systems.

**6.1.2** R[ichard] L. Mattson, J. Gecsei, D[onald] R. Slutz, and I[rving] L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal 9*, 2 (1970), pages 78–117.

The original reference on stack algorithms and their analysis, this paper is well written and presents considerably more in-depth observations than the brief summaries that appear in modern textbooks.

**6.1.3** Richard Rashid et al. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers 37*, 8 (August 1988), pages 896–908. Originally published in *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems* (November 1987), pages 31–39.

This paper describes a design for a sophisticated virtual memory system that has been adopted by several operating systems, including several BSD operating systems and Apple's OS X. The system supports large, sparse virtual address spaces, copy-on-write copying of pages, and memory-mapped files.

**6.1.4** Ted Kaehler and Glenn Krasner. LOOM: Large object-oriented memory for Smalltalk–80 systems. In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, 1983, pages 251–271. ISBN: 0–201–11669–3.

This paper describes the memory-management system used in Smalltalk, an interactive programming system for desktop computers. A coherent virtual memory language support system provides for lots of small objects while taking into account address space allocation, multilevel memory management, and naming in an integrated way.

The paper on the Woodstock File System, by Swinehart et al., reading *4.2.1*, describes a file system that is organized as a multilevel memory management system. Also see reading *10.1.8* for an interesting application (shared virtual memory) using multilevel memory management.

## 6.2 Remote Procedure Call

**6.2.1** Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems 8*, 1 (February 1990), pages 1–17. Originally published in *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, in *Operating Systems Review 23*, 5 (December 1989), pages 102–113.

As a complement to the abstract discussion of remote procedure call in reading *4.1.1*, this paper gives a concrete, blow-by-blow accounting of the steps required in a particular implementation and then compares this accounting with overall time measurements. In addition to providing insight into the intrinsic costs of remote procedures, this work demonstrates that it is possible to do bottom-up performance analysis that correlates well with top-down measurements.

**6.2.2** Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems 8*, 1 (February 1990), pages 37–55. Originally published in *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, in *Operating Systems Review 23*, 5 (December 1989), pages 102–113.

**6.2.3** Jochen Liedtke. Improving IPC by kernel design. *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, in *Operating Systems Review 27*, 5 (December 1993), pages 175–187.

These two papers develop techniques to allow local kernel-based client/service modularity to look just like remote client/service modularity to the application designer, while at the same time capturing the performance advantage that can come from being local.

## 6.3 Storage

**6.3.1** Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *Computer 27*, 3 (March 1994), pages 17–28.

This paper is really two papers in one. The first five pages provide a wonderfully accessible explanation of how disk drives and controllers actually work. The rest of the paper, of interest primarily to performance modeling specialists, explores the problem of accurately simulating a complex disk drive, with measurement data to show the size of errors that arise from various modeling simplifications (or oversimplifications).

**6.3.2** Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems 2*, 3 (August 1984), pages 181–197.

The "fast file system" nicely demonstrates the trade-offs between performance and complexity in adding several well-known performance enhancement techniques, such as multiple block sizes and sector allocation based on adjacency, to a file system that was originally designed as the epitome of simplicity.

**6.3.3** 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.

**6.3.4** Andrew Birrell et al. A design for high-performance flash disks. *ACM Operating Systems Review 41*, 2 (April 2007), pages 88–93. (Also appeared as Microsoft Corporation technical report TR-2005-176.)

Flash (non-volatile) electronic memory organized to appear as a disk has emerged as a more expensive but very low-latency alternative to magnetic disks for durable storage. This short paper describes, in an easy-to-understand way, the challenges associated with building a high-performance file system using flash disks and proposes a design to address the challenges. This paper is a good start for readers who want to explore flash-based storage systems.

## 6.4 Other Performance-Related Topics

**6.4.1** Sharon E. Perl and Richard L. Sites. Studies of Windows NT performance using dynamic execution traces. *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation* (October 1996). Also in *Operating System Review 30*, SI (October 1996), pages 169–184.

This paper shows by example that any performance issue in computer systems can be explained. The authors created a tool to collect complete traces of instructions executed by the Windows NT operating system and applications. The authors conclude that pin bandwidth limits the achievable execution speed of applications and that locks inside the operating system can limit applications to scale to more than a moderate number of processors. The paper also discusses the impact of cache-coherence hardware (see Chapter 10 [on-line]) on application performance. All of these issues are increasingly important for multiprocessors on a single chip.

**6.4.2** Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *Transactions on Computer Systems 15*, 3 (August 1997), pages 217–252.

This paper introduces the problem of receive livelock (described in Sidebar 6.7) and presents a solution. Receive livelock is a possible undesirable situation when a

system is temporarily overloaded. It can arise if the server spends too much of its time saying "I'm too busy" and as a result has no time left to serve any of the requests.

**6.4.3** Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Proceedings of the Sixth USENIX Symposium on Operating Systems Design and Implementation* (December 2004), pages 137–150. Also in *Communications of the ACM 51*, 1 (January 2008), pages 1–10.

This paper is a case study of aggregating arrays (reaching into the thousands) of computers to perform parallel computations on large data sets (e.g., all the pages of the Web). It uses a model that applies when a composition of two serial functions (Map and Reduce) has no side-effects on the data sets. The charm of MapReduce is that for computations that fit the model, the runtime uses concurrency but hides it completely from the programmer. The runtime partitions the input data set, executes the functions in parallel on different parts of the data set, and handles the failures of individual computers.

## 7 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.

### 7.1 Networks

The book by Perlman on bridges and routers, reading *1.2.5*, explains how the network layer really works.

**7.1.1** 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.

**7.1.2** 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.

### 7.2 Protocols

**7.2.1** 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 familiar with current network technology will frequently exclaim, "So that's why the Internet works that way."

**7.2.2** 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.

**7.2.3** 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.

**7.2.4** 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.

**7.2.5** 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.

**7.2.6** 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.

**7.2.7** 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 7.9 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.

**7.2.8** 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.

**7.2.9** 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 author's 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.

One popular protocol, remote procedure call, is covered in depth in reading *4.1.1* by Birrell and Nelson, as well as Section 10.3 of Tanenbaum's *Modern Operating Systems,* reading *1.2.1.*

## 7.3 Organization for Communication

**7.3.1** Leonard Kleinrock. Principles and lessons in packet communications. *Proceedings of the IEEE 66*, 11 (November 1978), pages 1320–1329.

**7.3.2** 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.

**7.3.3** 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.

**7.3.4** 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 very good explanation of the problem.

### 7.4 Practical Aspects

For the complete word on the Internet protocols, check out the following series of books.

**7.4.1** 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.

## 8 Fault Tolerance: Reliable Systems from Unreliable Components

A plan for some degree of fault tolerance shows up in many systems. For an example of fault tolerance in distributed file systems, see the paper on Coda by Kistler and Satyanarayanan, reading *10.1.2*. See also the paper on RAID by Katz et al., reading *10.2.2*.

### 8.1 Fault Tolerance

Chapter 3 of the book by Gray and Reuter, reading *1.1.5*, provides a bedrock text on this subject.

**8.1.1** Jim [N.] Gray and Daniel P. Siewiorek. High-availability computer systems. *Computer 24*, 9 (September 1991), pages 39–48.

This is a very nice, easy-to-read overview of how high availability can be achieved.

**8.1.2** 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.

## 8.2 Software Errors

**8.2.1** 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.

**8.2.2** Michael M. Swift et al. Recovering device drivers. *Proceedings of the Sixth Symposium on Operating Systems 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.

## 8.3 Disk Failures

**8.3.1** 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 8.2, 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.

**8.3.2** 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.

## 9 Atomicity: All-or-Nothing and Before-or-After

### 9.1 Atomicity, Coordination, and Recovery

The best source on this topic is reading *1.1.5*, but Gray and Reuter's thousand-page book can be a bit overwhelming.

**9.1.1** 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.

**9.1.2** 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.

### 9.2 Databases

**9.2.1** 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.

**9.2.2** 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, commercial-quality database transaction system that uses write-ahead logging.

**9.2.3** 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 9.6), 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.

**9.2.4** 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 a off-the-shelf components, including a standard database system for storing the terabytes of data.

**9.2.5** 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 8.6 for the definition of Byzantine).

## 9.3 Atomicity-Related Topics

**9.3.1** 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.

**9.3.2** 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.

See also the paper by Lampson and Sturgis, reading *1.8.7* and the paper by Ganger and Patt, reading *6.3.3*.

## 10 Consistency and Durable Storage

### *10.1 Consistency*

**10.1.1** 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.

**10.1.2** 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 very well written.

**10.1.3** 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.

**10.1.4** 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.

**10.1.5** 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.

**10.1.6** 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.

**10.1.7** 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.

**10.1.8** Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems 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.

**10.1.9** 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 search.

**10.1.10** 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.

## 10.2 Durable Storage

**10.2.1** 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.

**10.2.2** 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.

**10.2.3** 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.

**10.2.4** 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.

### 10.3 Reconciliation

**10.3.1** Douglas B. Terry et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th 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.

**10.3.2** 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.

## 11 Information Security

### 11.1 Privacy

The fundamental book about privacy is reading *1.1.6* by Alan Westin.

**11.1.1** 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.

**11.1.2** 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 11.1 for a discussion of the paper's proposed extended definition.

### 11.2 Protection Architectures

**11.2.1** 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 30 years, this paper (an early version of the current Chapter 11) 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.

**11.2.2** 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 Fabry, reading *3.1.2*.

## *11.3 Certification, Trusted Computer Systems, and Security Kernels*

**11.3.1** 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.

**11.3.2** 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 *11.3.1* 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.

**11.3.3** 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 11.3.4 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.]

**11.3.4** 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 *11.3.3*). 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.

**11.3.5** 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 1970's, 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.

**11.3.6** 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.

**11.3.7** 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 11.1.4) 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.

See also reading *1.3.15* by Garfinkel and Spafford, reading *5.2.1* by Lampson and Sturgis, and reading *5.2.2* by Schroeder, Clark, and Saltzer.

## 11.4 Authentication

**11.4.1** 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 an historical development designed to simplify things for the reader, it describes the UNIX password security mechanism.

**11.4.2** 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 11.5).

**11.4.3** 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.

See also Sidebar 11.6 on Kerberos and reading *3.2.5*, which uses cryptographic techniques to secure a personal naming system.

## *11.5 Cryptographic Techniques*

The fundamental books about cryptography applied to computer systems are reading *1.2.4*, by Bruce Schneier, and reading *1.3.13*, by Alfred Menezes et al. In light of these two books, the first few papers from the 1970's listed below are primarily of historical interest. There is also a good, more elementary, treatment of cryptography in the book by Simson Garfinkel, reading *1.3.15*. 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 reading *1.3.14*.

**11.5.1** 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.

**11.5.2** 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 11.9.3.1).

**11.5.3** Ross J. Anderson. Why cryptosystems fail. *Communications of the ACM 37*, 11 (November 1994), pages 32–40.

Anderson presents a very nice analysis of what goes wrong in real-world cryptosystems—secure modules don't necessary 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.

**11.5.4** 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>.

**11.5.5** M[ihir] Bellare, R[an] Canetti, and H[ugo] Krawczyk. Keying hash functions for message authentication. *Proceedings of the 16th International Cryptography 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.

**11.5.6** 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.

## 11.6 Adversaries (The Dark Side)

Section 11.11 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.

**11.6.1** 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.

**11.6.2** 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.

**11.6.3** 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.

**11.6.4** 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.

**11.6.5** Chris Kanich et al. Spamalytics: An empirical analysis of spam marketing conversion. *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (October 2008), pages 3–14.

This paper describes the infrastructure that spammers use to send unsolicited e-mail 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.

**11.6.6** 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.

# Problem Sets

## TABLE OF CONTENTS

**PART I**

Introduction ..... 426
1 Bigger Files ..... 427
2 Ben's Stickr ..... 428
3 Jill's File System for Dummies ..... 430
4 EZ-Park ..... 432
5 Goomble ..... 436
6 Course Swap ..... 438
7 Banking on Local Remote Procedure Call ..... 443
8 The Bitdiddler ..... 446
9 Ben's Kernel ..... 448
10 A Picokernel-Based Stock-Ticker System ..... 453
11 Ben's Web Service ..... 458
12 A Bounded Buffer with Semaphores ..... 462
13 The Single-Chip NC ..... 464
14 Toastac-25 ..... 465
15 BOOZE: Ben's Object-Oriented Zoned Environment ..... 466
16 OutOfMoney.com ..... 469

**PART II** [On-line]

17 Quarria
18 PigeonExpress!.com I
19 Monitoring Ants
20 Gnutella: Peer-to-Peer Networking
21 The OttoNet
22 The Wireless EnergyNet
23 SureThing
24 Sliding Window
25 Geographic Routing
26 Carl's Satellite
27 RaidCo
28 ColdFusion
29 AtomicPigeon!.com
30 Sick Transit
31 The Bank of Central Peoria, Limited
32 Whisks

33 ANTS: Advanced "Nonce-ensical" Transaction System
34 KeyDB
35 Alice's Reliable Block Store
36 Establishing Serializability
37 Improved Bitdiddler
38 Speedy Taxi Company
39 Locking for Transactions
40 "Log"-ical Calendaring
41 Ben's Calendar
42 Alice's Replicas
43 JailNet
44 PigeonExpress!.com II
45 WebTrust.com (OutOfMoney.com, part II)
46 More ByteStream Products
47 Stamp out Spam
48 Confidential Bitdiddler
49 Beyond Stack Smashing

## INTRODUCTION

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. Here is a list of concepts that the exercises and problem sets introduce:

- *action graph* (Problem set *36*)
- *ad hoc wireless network* (Problem sets *19* and *21*)
- *bang-bang protocol* (Exercise *7.13*)
- *blast protocol* (Exercise *7.25*)
- *commutative cryptographic transformation* (Exercise *11.4*)
- *condition variable* (Problem set *13*)
- *consistent hashing* (Problem set *23*)
- *convergent encryption* (Problem set *48*)
- *cookie* (Problem set *45*)
- *delayed authentication* (Exercise *11.9*)
- *delegation forwarding* (Exercise *2.1*)
- *event variable* (Problem set *11*)

- *fast start* (Exercise *7.12*)
- *flooding* (Problem set *20*)
- *follow-me forwarding* (Exercise *2.1*)
- *Information Management System atomicity* (Exercise *9.5*)
- *mobile host* (Exercise *7.24*)
- *lightweight remote procedure call* (Problem set *7*)
- *multiple register set processor* (Problem set *9*)
- *object-oriented virtual memory* (Problem set *15*)
- *overlay network* (Problem set *20*)
- *pacing* (Exercise *7.16*)
- *peer-to-peer network* (Problem set *20*)
- *RAID 5,* with rotating parity (Exercise *8.8*)
- *restartable atomic region* (Problem set *9*)
- *self-describing storage* (Exercise *6.8*)
- *serializability* (Problem set *36*)
- *timed capability* (Exercise *11.7*)

Exercises for Chapter 7 and above are in on-line chapters, and problem sets numbered 17 and higher are in the on-line book of problem sets.

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.

## 1 Bigger Files*

*(Chapter 2)*

For his many past sins on previous exams, Ben Bitdiddle is assigned to spend eternity maintaining a PDP-11 running version 7 of the UNIX® operating system. Recently, one of his user's database applications failed after reaching the file size limit of 1,082,201,088 bytes (approximately 1 gigabyte). In an effort to solve the problem, he upgraded the computer with an old 4-gigabyte ($2^{32}$ byte) drive; the disk controller hardware supports 32-bit sector addresses, and can address disks up to 2 terabytes in size. Unfortunately, Ben is disappointed to find the file size limit unchanged after installing the new disk.

In this question, the term *block number* refers to the block pointers stored in inodes. There are 512 bytes in a block. In addition, Ben's version 7 UNIX system has a file system that has been expanded from the one described in Section 2.5: its inodes

*Credit for developing this problem set goes to Lewis D. Girod.

are designed to support larger disks. Each inode contains 13 block numbers of 4 bytes each; the first 10 block numbers point to the first 10 blocks of the file, and the remaining 3 are used for the rest of the file. The 11th block number points to an indirect block, containing 128 block numbers, the 12th block number points to a double-indirect block, containing 128 indirect block numbers, and the 13th block number points to a triple-indirect block, containing 128 double-indirect block numbers. Finally, the inode contains a four-byte file size field.

**Q 1.1** Which of the following adjustments will allow files larger than the current 1-gigabyte limit to be stored?

**A.** Increase just the file size field in the inode from a 32-bit to a 64-bit value.
**B.** Increase just the number of bytes per block from 512 to 2048 bytes.
**C.** Reformat the disk to increase the number of inodes allocated in the inode table.
**D.** Replace one of the direct block numbers in each inode with an additional triple-indirect block number.

*2008-1-5*

Ben observes that there are 52 bytes allocated to block numbers in each inode (13 block numbers at 4 bytes each), and 512 bytes allocated to block numbers in each indirect block (128 block numbers at 4 bytes each). He figures that he can keep the total space allocated to block numbers the same, but change the size of each block number, to increase the maximum supported file size. While the number of block numbers in inodes and indirect blocks will change, Ben keeps exactly one indirect, one double-indirect and one triple-indirect block number in each inode.

**Q 1.2** Which of the following adjustments (without any of the modifications in the previous question), will allow files larger than the current approximately 1-gigabyte limit to be stored?

**A.** Increasing the size of a block number from 4 bytes to 5 bytes.
**B.** Decreasing the size of a block number from 4 bytes to 3 bytes.
**C.** Decreasing the size of a block number from 4 bytes to 2 bytes.

*2008-1-6*

## 2 Ben's Stickr*

*(Chapter 4)*

Ben is in charge of system design for Stickr, a new Web site for posting pictures of bumper stickers and tagging them. Luckily for him, Alyssa had recently implemented

*Credit for developing this problem set goes to Samuel R. Madden.

a Triplet Storage System (TSS), which stores and retrieves arbitrary triples of the form {*subject, relationship, object*} according to the following specification:

**procedure** FIND (*subject*, *relationship*, *object*, *start*, *count*)
// returns OK + array of matching triples

**procedure** INSERT (*subject*, *relationship*, *object*)
// adds the triple to the TSS if it is not already there and returns OK

**procedure** DELETE (*subject*, *relationship*, *object*)
// removes the triple if it exists, returning TRUE, FALSE otherwise

Ben comes up with the following design:

As shown in the figure, Ben uses an RPC interface to allow the Web server to interact with the triplet storage system. Ben chooses *at-least-once* RPC semantics. Assume that the triplet storage system never crashes, but the network between the Web server and triplet storage system is unreliable and may drop messages.

**Q 2.1** Suppose that only a single thread on Ben's Web server is using the triplet storage system and that this thread issues just one RPC at a time. What types of incorrect behavior can the Web server observe?

**A.** The FIND RPC stub on the Web server sometimes returns no results, even though matching triples exist in the triplet storage system.
**B.** The INSERT RPC stub on the Web server sometimes returns OK without inserting the triple into the storage system.
**C.** The DELETE RPC stub on the Web server sometimes returns FALSE when it actually deleted a triple.
**D.** The FIND RPC stub on the Web server sometimes returns triples that have been deleted.

**Q 2.2** Suppose Ben switches to *at-most-once* RPC; if no reply is received after some time, the RPC stub on the Web server gives up and returns a "timer expired" error code. Assume again that only a single thread on Ben's Web server is using the triple storage system and that this thread issues just one RPC at a time. What types of incorrect behavior can the Web server observe?

**A.** Assuming it does not time out, the FIND RPC stub on the Web server can sometimes return no results when matching triples exist in the storage system.
**B.** Assuming it does not time out, the INSERT RPC stub on the Web server can sometimes return OK without inserting the triple into the storage system.
**C.** Assuming it does not time out, the DELETE RPC stub on the Web server can sometimes return FALSE when it actually deleted a triple.
**D.** Assuming it does not time out, the FIND RPC stub on the Web server can sometimes return triples that have been deleted.

## 3 Jill's File System for Dummies*

*(Chapter 4)*

Mystified by the complexity of NFS, Moon Microsystems guru Jill Boy decides to implement a simple alternative she calls File System for Dummies, or FSD. She implements FSD in two pieces:

1. An FSD server, implemented as a simple user application, which responds to FSD requests. Each request corresponds exactly to a UNIX file system call (e.g., READ, WRITE, OPEN, CLOSE, or CREATE) and returns just the information returned by that call (status, integer file descriptor, data, etc.).
2. An FSD client library, which can be linked together with various applications to substitute Jill's FSD implementations of file system calls like OPEN, READ, and WRITE for their UNIX counterparts. To avoid confusion, let's refer to Jill's FSD versions of these procedures as FSD_OPEN, and so on.

Jill's client library uses the standard UNIX calls to access local files but uses names of the form

```
/fsd/hostname/apath
```

to refer to the file whose absolute path name is /apath on the host named hostname. Her library procedures recognize operations involving remote files e.g.,

```
FSD_OPEN("/fsd/cse.pedantic.edu/foobar", READ_ONLY)
```

and translates them to RPC requests to the appropriate host, using the file name on that host e.g.,

```
RPC("/fsd/cse.pedantic.edu/foobar", "OPEN", "/foobar", READ_ONLY).
```

The RPC call causes the corresponding UNIX call e.g.,

```
OPEN("/foobar", READ_ONLY)
```

to be executed on the remote host and the results (e.g., a file descriptor) to be returned as the result of the RPC call. Jill's server code catches errors in the processing of each request and returns ERROR from the RPC call on remote errors.

Figure 1 describes pseudocode for Version 1 of Jill's FSD client library. The RPC calls in the code relay simple RPC commands to the server, using *exactly-once* semantics. Note that no data caching is done by either the server or the client library.

**Q 3.1** What does the above code indicate via an empty string ("") in an entry of handle to host table?

**A.** An unused entry of the table.
**B.** An open file on the client host machine.
**C.** An end-of-file condition on an open file.
**D.** An error condition.

*Credit for developing this problem set goes to Stephen A. Ward.

```
// Map FSD handles to host names, remote handles:
string handle_to_host_table[1000]                    // initialized to UNUSED
integer handle_to_rhandle_table[1000]                // handle translation table

procedure FSD_OPEN (string name, integer mode)
    integer handle ← FIND_UNUSED_HANDLE ()
    if name begins with "/fsd/" then
        host ← EXTRACT_HOST_NAME (name)
        filename ← EXTRACT_REMOTE_FILENAME (name)   // returns remote file handle or ERROR
        rhandle ← RPC (host, "OPEN", filename, mode)
    else
        host ← ""
        rhandle ← OPEN (name, mode)
    if rhandle ← ERROR then return ERROR
    handle_to_rhandle_table[handle] ← rhandle
    handle_to_host_table[handle] ← host
    return handle

procedure FSD_READ (integer handle, string buffer, integer nbytes)
    host ← handle_to_host_table[handle]
    rhandle ← handle_to_rhandle_table[handle]
    if host ← "" then return READ (rhandle, buffer, nbytes)
    // The following call sets "result" to the return value from
    // the read(...) on the remote host, and copies data read into buffer:
    result, buffer ← RPC (host, "READ", rhandle, nbytes)
    return result

procedure FSD_CLOSE (integer handle)
    host ← handle_to_host_table[handle]
    rhandle ← handle_to_rhandle_table[handle]
    handle_to_rhandle_table[handle] ← UNUSED
    if host ← "" then return CLOSE (rhandle)
    else return RPC (host, "CLOSE", rhandle)
```

**FIGURE 1**

Pseudocode for FSD client library, Version 1

Mini Malcode, an intern assigned to Jill, proposes that the above code be simplified by eliminating the *handle_to_rhandle_table* and simply returning the untranslated handles returned by OPEN on the remote or local machines. Mini implements her simplified client library, making appropriate changes to each FSD call, and tries it on several test programs.

**Q 3.2** Which of the following test programs will continue to work after Mini's simplification?

- **A.** A program that reads a single, local file.
- **B.** A program that reads a single remote file.
- **C.** A program that reads and writes many local files.
- **D.** A program that reads and writes several files from a single remote FSD server.
- **E.** A program that reads many files from different remote FSD servers.
- **F.** A program that reads several local files as well as several files from a single remote FSD server.

Jill rejects Mini's suggestions, insisting on the Version 1 code shown above. Marketing asks her for a comparison between FSD and NFS (see Section 4.5).

**Q 3.3** Complete the following table comparing NFS to FSD by circling yes or no under each of NFS and FSD for each statement:

| Statement | NFS | FSD |
|---|---|---|
| Remote handles include inode numbers | Yes/No | Yes/No |
| Read and write calls are idempotent | Yes/No | Yes/No |
| Can continue reading an open file after deletion (e.g., by program on remote host) | Yes/No | Yes/No |
| Requires mounting remote file systems prior to use | Yes/No | Yes/No |

Convinced by Moon's networking experts that a much simpler RPC package promising *at-least-once* rather than *exactly-once* semantics will save money, Jill substitutes the simpler RPC framework and tries it out. Although the new (Version 2) FSD works most of the time, Jill finds that an `FSD_READ` sometimes returns the wrong data; she asks you to help. You trace the problem to multiple executions of a single RPC request by the server and are considering

- A response cache on the client, sufficient to detect identical requests and returning a cached result for duplicates without resending the request to the server.
- A response cache on the server, sufficient to detect identical requests and returning a cached result for duplicates without reexecuting them.
- A monotonically increasing *sequence number* (nonce) added to each RPC request, making otherwise identical requests distinct.

**Q 3.4** Which of the following changes would you suggest to address the problem introduced by the *at-least-once* RPC semantics?

**A.** Response cache on each client.
**B.** Response cache on server.
**C.** Sequence numbers in RPC requests.
**D.** Response cache on client AND sequence numbers.
**E.** Response cache on server AND sequence numbers.
**F.** Response caches on both client and server.

*2007-2-7…10*

## 4 EZ-Park*

*(Chapter 5 in Chapter 4 setting)*

Finding a parking spot at Pedantic University is as hard as it gets. Ben Bitdiddle, deciding that a little technology can help, sets about to design the EZ-Park client/server

*Credit for developing this problem set goes to Hari Balakrishnan.

system. He gets a machine to run an EZ-Park server in his dorm room. He manages to convince Pedantic University parking to equip each car with a tiny computer running EZ-Park client software. EZ-Park clients communicate with the server using remote procedure calls (RPCs). A client makes requests to Ben's server both to find an available spot (when the car's driver is looking for one) and to relinquish a spot (when the car's driver is leaving a spot). A car driver uses a parking spot if, and only if, EZ-Park allocates it to him or her.

In Ben's initial design, the server software runs in one address space and spawns a new thread for each client request. The server has two procedures: FIND_SPOT () and RELINQUISH_SPOT (). Each of these threads is spawned in response to the corresponding RPC request sent by a client. The server threads use a shared array, *available*[], of size NSPOTS (the total number of parking spots). *available*[*j*] is set to TRUE if spot *j* is free, and FALSE otherwise; it is initialized to TRUE, and there are no cars parked to begin with. The NSPOTS parking spots are numbered from 0 through NSPOTS - 1. *numcars* is a global variable that counts the total number of cars parked; it is initialized to 0.

Ben implements the following pseudocode to run on the server. Each FIND_SPOT () thread enters a **while** loop that terminates only when the car is allocated a spot:

```
procedure FIND_SPOT ()                  // Called when a client car arrives
    while TRUE do
        for i ← 0 to NSPOTS do
            if available[i] = TRUE then
                available[i] ← FALSE
                numcars ← numcars + 1
                return i     // Client gets spot i

procedure RELINQUISH_SPOT (spot)        // Called when a client car leaves
    available[spot] ← TRUE
    numcars ← numcars − 1
```

Ben's intended correct behavior for his server (the "correctness specification") is as follows:

**A.** FIND_SPOT () allocates any given spot in [0, …, NSPOTS − 1] to at most one car at a time, even when cars are concurrently sending requests to the server requesting spots.

**B.** *numcars* must correctly maintain the number of parked cars.

**C.** If at any time (1) spots are available and no parked car ever leaves in the future, (2) there are no outstanding FIND_SPOT () requests, and (3) exactly one client makes a FIND_SPOT request, then the client should get a spot.

Ben runs the server and finds that when there are no concurrent requests, EZ-Park works correctly. However, when he deploys the system, he finds that sometimes multiple cars are assigned the same spot, leading to collisions! His system does not meet the correctness specification when there are concurrent requests.

Make the following assumptions:

1. The statements to update *numcars* are ***not*** atomic; each involves multiple instructions.
2. The server runs on a single processor with a preemptive thread scheduler.
3. The network delivers RPC messages reliably, and there are no network, server, or client failures.
4. Cars arrive and leave at random.
5. ACQUIRE and RELEASE are as defined in chapter 5.

**Q 4.1** Which of these statements is true about the problems with Ben's design?

**A.** There is a race condition in accesses to *available*[], which may violate one of the correctness specifications when two FIND_SPOT () threads run.
**B.** There is a race condition in accesses to *available*[], which may violate correctness specification A when one FIND_SPOT () thread and one RELINQUISH_SPOT () thread runs.
**C.** There is a race condition in accesses to *numcars*, which may violate one of the correctness specifications when more than one thread updates *numcars*.
**D.** There is no race condition as long as the average time between client requests to find a spot is larger than the average processing delay for a request.

Ben enlists Alyssa's help to fix the problem with his server, and she tells him that he needs to set some locks. She suggests adding calls to ACQUIRE and RELEASE as follows:

```
1      procedure FIND_SPOT ()                // Called when a client car wants a spot
2          while TRUE do
!→             ACQUIRE (avail_lock)
3                  for i ← 0 to NSPOTS do
4                      if available[i] = TRUE then
5                          available[i] ← FALSE
6                          numcars ← numcars + 1
!→                         RELEASE (avail_lock)
7                          return i  // Allocate spot i to this client
!→                     RELEASE (avail_lock)

8      procedure RELINQUISH_SPOT (spot)      // Called when a client car is leaving spot
!→         ACQUIRE (avail_lock)
9          available[spot] ← TRUE
10         numcars ← numcars − 1
!→         RELEASE (avail_lock)
```

**Q 4.2** Does Alyssa's code solve the problem? Why or why not?

**Q 4.3** Ben can't see any good reason for the RELEASE (*avail_lock*) that Alyssa placed after line 7, so he removes it. Does the program still meet its specifications? Why or why not?

Hoping to reduce competition for *avail_lock*, Ben rewrites the program as follows:

```
1      procedure FIND_SPOT ()                    // Called when a client car wants a spot
2          while TRUE do
3              for i ← 0 to NSPOTS do
!→                 ACQUIRE (avail_lock)
4                      if available[i] = TRUE then
5                          available[i] ← FALSE
6                          numcars ← numcars + 1
!→                         RELEASE (avail_lock)
7                          return i          // Allocate spot i to this client
!→                     else RELEASE (avail_lock)

8      procedure RELINQUISH_SPOT (spot)          // Called when a client car is leaving spot
!→         ACQUIRE (avail_lock)
9          available[spot] ← TRUE
10         numcars ← numcars − 1
!→         RELEASE (avail_lock)
```

**Q 4.4** Does that program meet the specifications?

Now that Ben feels he understands locks better, he tries one more time, hoping that by shortening the code he can really speed things up:

```
1      procedure FIND_SPOT ()                    // Called when a client car wants a spot
2          while TRUE do
!→             ACQUIRE (avail_lock)
3              for i ← 0 to NSPOTS do
4                  if available[i] = TRUE then
5                      available[i] ← FALSE
6                      numcars ← numcars + 1
7                      return i // Allocate spot i to this client

8      procedure RELINQUISH_SPOT (spot)          // Called when a client car is leaving spot
9          available[spot] ← TRUE
10         numcars ← numcars − 1
!→         RELEASE (avail_lock)
```

**Q 4.5** Does Ben's slimmed-down program meet the specifications?

Ben now decides to combat parking at a truly crowded location: Pedantic's stadium, where there are always cars looking for spots! He updates NSPOTS and deploys the system during the first home game of the football season. Many clients complain that his server is slow or unresponsive.

**Q 4.6** If a client invokes the FIND_SPOT ( ) RPC when the parking lot is full, how quickly will it get a response, assuming that multiple cars may be making requests?

**A.** The client will not get a response until at least one car relinquishes a spot.
**B.** The client may never get a response even when other cars relinquish their spots.

Alyssa tells Ben to add a client-side timer to his RPC system that expires if the server does not respond within 4 seconds. Upon a timer expiration, the car's driver may retry the request, or instead choose to leave the stadium to watch the game on TV. Alyssa warns Ben that this change may cause the system to violate the correctness specification.

**Q 4.7** When Ben adds the timer to his client, he finds some surprises. Which of the following statements is true of Ben's implementation?

- **A.** The server may be running multiple active threads on behalf of the same client car at any given time.
- **B.** The server may assign the same spot to two cars making requests.
- **C.** *numcars* may be smaller than the actual number of cars parked in the parking lot.
- **D.** *numcars* may be larger than the actual number of cars parked in the parking lot.

**Q 4.8** Alyssa thinks that the operating system running Ben's server may be spending a fair amount of time switching between threads when many RPC requests are being processed concurrently. Which of these statements about the work required to perform the switch is correct? Notation: PC = program counter; SP = stack pointer; PMAR = page-map address register. Assume that the operating system behaves according to the description in Chapter 5.

- **A.** On any thread switch, the operating system saves the values of the PMAR, PC, SP, and several registers.
- **B.** On any thread switch, the operating system saves the values of the PC, SP, and several registers.
- **C.** On any thread switch between two RELINQUISH_SPOT () threads, the operating system saves *only* the value of the PC, since RELINQUISH_SPOT () has no return value.
- **D.** The number of instructions required to switch from one thread to another is proportional to the number of bytes currently on the thread's stack.

## 5 Goomble*

*(Chapter 5)*

Observing that U.S. legal restrictions have curtailed the booming on-line gambling industry, a group of laid-off programmers has launched a new venture called Goomble. Goomble's Web server allows customers to establish an account, deposit funds using a credit card, and then play the Goomble game by clicking a button labeled **I FEEL LUCKY**. Every such button click debits their account by $1, until it reaches zero.

Goomble lawyers have successfully defended their game against legal challenges by arguing that there's no gambling involved: the Goomble "service" is entirely deterministic.

The initial implementation of the Goomble server uses a single thread, which causes all customer requests to be executed in some serial order. Each click on the **I FEEL LUCKY** button results in a procedure call to LUCKY (*account*), where *account*

*Credit for developing this problem set goes to Stephen A. Ward.

refers to a data structure representing the user's Goomble account. Among other data, the account structure includes an unsigned 32-bit integer *balance*, representing the customer's current balance in dollars.
The LUCKY procedure is coded as follows:

```
1    procedure LUCKY (account)
2        if account.balance > 0 then
3            account.balance ← account.balance -1
```

The Goomble software quality control expert, Nellie Nervous, inspects the single-threaded Goomble server code to check for race conditions.

**Q 5.1** Should Nellie find any potential race conditions? Why or why not?

*2007-1-8*

The success of the Goomble site quickly swamps their single-threaded server, limiting Goomble's profits. Goomble hires a server performance expert, Threads Galore, to improve server throughput.

Threads modifies the server as follows: Each **I FEEL LUCKY** click request spawns a new thread, which calls LUCKY (*account*) and then exits. All other requests (e.g., setting up an account, depositing, etc.) are served by a single thread. Threads argues that the bulk of the server traffic consists of player's clicking **I FEEL LUCKY**, so that his solution addresses the main performance problem.

Unfortunately, Nellie doesn't have time to inspect the multithreaded version of the server. She is busy with development of a follow-on product: the Goomba, which simultaneously cleans out your bank account and washes your kitchen floor.

**Q 5.2** Suppose Nellie had inspected Goomble's multithreaded server. Should she have found any potential race conditions? Why or why not?

*2007-1-9*

Willie Windfall, a compulsive Goomble player, has two computers and plays Goomble simultaneously on both (using the same Goomble account). He has mortgaged his house, depleted his retirement fund and the money saved for his kid's education, and his Goomble account is nearly at zero. One morning, clicking furiously on **I FEEL LUCKY** buttons on both screens, he notices that his Goomble balance has jumped to something over four billion dollars.

**Q 5.3.** Explain a possible source of Willie's good fortune. Give a simple scenario involving two threads, T1 and T2, with interleaved execution of lines *2* and *3* in calls to LUCKY (*account*), detailing the timing that could result in a huge *account.balance*. The first step of the scenario is already filled in; fill as many subsequent steps as needed.

1. **T1 evaluates** "**if** *account.balance* > 0", **finds statement is true**
2. 
3. 
4. 

*2007-1-10*

Word of Willie's big win spreads rapidly, and Goomble billionaires proliferate. In a state of panic, the Goomble board calls you in as a consultant to review three possible fixes to the server code to prevent further "gifts" to Goomble customers. Each of the following proposals involves adding a lock (either global or specific to an account) to rule out the unfortunate race:

*Proposal 1*

```
procedure LUCKY (account)
    ACQUIRE (global_lock);
    if account.balance > 0 then
        account.balance ← account.balance - 1;
    RELEASE (global_lock)
```

*Proposal 2*

```
procedure LUCKY (account)
    ACQUIRE (account.lock)
    temp ← account.balance
    RELEASE (account.lock)
    if temp > 0 then
        ACQUIRE (account.lock);
        account.balance ← account.balance - 1;
        RELEASE (account.lock);
```

*Proposal 3*

```
procedure LUCKY (account)
    ACQUIRE (account.lock);
    if account.balance > 0 then
        account.balance ← account.balance - 1
    RELEASE (account.lock);
```

**Q 5.4** Which of the three proposals have race conditions?

*2007-1-11*

**Q 5.5** Which proposal would you recommend deploying, considering both correctness and performance goals?

*2007-1-12*

## 6 Course Swap*

*(Chapter 5 in Chapter 4 setting)*

The Subliminal Sciences Department, in order to reduce the department head's workload, has installed a Web server to help assign lecturers to classes for the

*Credit for developing this problem set goes to Robert T. Morris.

Fall teaching term. There happen to be exactly as many courses as lecturers, and department policy is that every lecturer teach exactly one course and every course have exactly one lecturer. For each lecturer in the department, the server stores the name of the course currently assigned to that lecturer. The server's Web interface supports one request: to swap the courses assigned to a pair of lecturers.

Version One of the server's code looks like this:

```
// CODE VERSION ONE

    assignments[]        // an associative array of course names indexed by lecturer

    procedure SERVER ()
        do forever
            m ← wait for a request message
            value ← m.FUNCTION (m.arguments, ...) // execute function in request message
            send value to m.sender

    procedure EXCHANGE (lecturer1, lecturer2)
        temp ← assignments[lecturer1]
        assignments[lecturer1] ← assignments[lecturer2]
        assignments[lecturer2] ← temp
        return "OK"
```

Because there is only one application thread on the server, the server can handle only one request at a time. Requests comprise a function, and its arguments (in this case EXCHANGE (*lecturer1*, *lecturer2*)), which is executed by the *m*.FUNCTION (*m.arguments*, ...) call in the SERVER ( ) procedure.

For all following questions, assume that there are no lost messages and no crashes. The operating system buffers incoming messages. When the server program asks for a message of a particular type (e.g., a request), the operating system gives it the oldest buffered message of that type.

Assume that network transmission times never exceed a fraction of a second and that computation also takes a fraction of a second. There are no concurrent operations other than those explicitly mentioned or implied by the pseudocode, and no other activity on the server computers.

Suppose the server starts out with the following assignments:

```
assignments["Herodotus"] = "Steganography"
assignments["Augustine"] = "Numerology"
```

**Q 6.1** Lecturers Herodotus and Augustine decide they wish to swap lectures, so that Herodotus teaches Numerology and Augustine teaches Steganography. They each send an EXCHANGE ("Herodotus", "Augustine") request to the server at the same time.

If you look a moment later at the server, which, if any, of the following states are possible?

**A.**

*assignments*["Herodotus"] = "Numerology"
*assignments*["Augustine"] = "Steganography"

**B.**

*assignments*["Herodotus"] = "Steganography"
*assignments*["Augustine"] = "Numerology"

**C.**

*assignments*["Herodotus"] = "Steganography"
*assignments*["Augustine"] = "Steganography"

**D.**

*assignments*["Herodotus"] = "Numerology"
*assignments*["Augustine"] = "Numerology"

The Department of Dialectic decides it wants its own lecturer assignment server. Initially, it installs a completely independent server from that of the Subliminal Sciences Department, with the same rules (an equal number of lecturers and courses, with a one-to-one matching). Later, the two departments decide that they wish to allow their lecturers to teach courses in either department, so they extend the server software in the following way. Lecturers can send either server a CROSSEXCHANGE request, asking to swap courses between a lecturer in that server's department and a lecturer in the other server's department. In order to implement CROSSEXCHANGE, the servers can send each other SET-AND-GET requests, which set a lecturer's course and return the lecturer's previous course. Here's Version Two of the server code, for both departments:

```
// CODE VERSION TWO
procedure SERVER ()          // same as in Version One
procedure EXCHANGE ()        // same as in Version One

procedure CROSSEXCHANGE (local-lecturer, remote-lecturer)
    temp1 ← assignments[local-lecturer]
    send {SET-AND-GET, remote-lecturer, temp1} to the other server
    temp2 ← wait for response to SET-AND-GET
    assignments[local-lecturer] ← temp2
    return "OK"

procedure SET-AND-GET (lecturer, course) {
    old ← assignments[lecturer]
    assignments[lecturer] ← course
    return old
```

Suppose the starting state on the Subliminal Sciences server is:

```
assignments["Herodotus"] = "Steganography"
assignments["Augustine"] = "Numerology"
```

And on the Department of Dialectic server:

```
assignments["Socrates"] = "Epistemology"
assignments["Descartes"] = "Reductionism"
```

**Q 6.2** At the same time, lecturer Herodotus sends a CROSSEXCHANGE ("Herodotus", "Socrates") request to the Subliminal Sciences server, and lecturer Descartes sends a CROSSEXCHANGE ("Descartes", "Augustine") request to the Department of Dialectic server. If you look a minute later at the Subliminal Sciences server, which, if any, of the following states are possible?

**A.**

```
assignments["Herodotus"] = "Steganography"
assignments["Augustine"] = "Numerology"
```

**B.**

```
assignments["Herodotus"] = "Epistemology"
assignments["Augustine"] = "Reductionism"
```

**C.**

```
assignments["Herodotus"] = "Epistemology"
assignments["Augustine"] = "Numerology"
```

In a quest to increase performance, the two departments make their servers multithreaded: each server serves each request in a separate thread. Thus, if multiple requests arrive at roughly the same time, the server may process them in parallel. Each server has multiple processors. Here's the threaded server code, Version Three:

```
// CODE VERSION THREE
procedure EXCHANGE ()                // same as in Version Two
procedure CROSSEXCHANGE ()           // same as in Version Two
procedure SET-AND-GET ()             // same as in Version Two

procedure SERVER ()
    do forever
        m ← wait for a request message
        ALLOCATE_THREAD (DOIT, m)    // create a new thread that runs DOIT (m)

procedure DOIT (m)
    value ← m.FUNCTION(m.arguments, ...)
    send value to m.sender
    EXIT ()                          // terminate this thread
```

**Q 6.3** With the same starting state as the previous question, but with the new version of the code, lecturer Herodotus sends a CROSSEXCHANGE ("Herodotus", "Socrates")

request to the Subliminal Sciences server, and lecturer Descartes sends a CROSSEXCHANGE ("Descartes","Augustine") request to the Department of Dialectic server, at the same time. If you look a minute later at the Subliminal Sciences server, which, if any, of the following states are possible?

**A.**

*assignments*["Herodotus"] = "Steganography"
*assignments*["Augustine"] = "Numerology"

**B.**

*assignments*["Herodotus"] = "Epistemology"
*assignments*["Augustine"] = "Reductionism"

**C.**

*assignments*["Herodotus"] = "Epistemology"
*assignments*["Augustine"] = "Numerology"

An alert student notes that Version Three may be subject to race conditions. He changes the code to have one lock per lecturer, stored in an array called *locks*[]. He changes EXCHANGE CROSSEXCHANGE, and SET-AND-GET to ACQUIRE locks on the lecturer(s) they affect. Here is the result, Version Four:

```
// CODE VERSION FOUR
procedure SERVER ()                // same as in Version Three
procedure DOIT ()                  // same as in Version Three

procedure EXCHANGE (lecturer1, lecturer2)
    ACQUIRE (locks[lecturer1])
    ACQUIRE (locks[lecturer2])
    temp ← assignments[lecturer1]
    assignments[lecturer1] ← assignments[lecturer2]
    assignments[lecturer2] ← temp
    RELEASE (locks[lecturer1])
    RELEASE (locks[lecturer2])
    return "OK"

procedure CROSSEXCHANGE (local-lecturer, remote-lecturer)
    ACQUIRE (locks[local-lecturer])
    temp1 ← assignments[local-lecturer]
    send SET-AND-GET, remote-lecturer, temp1 to other server
    temp2 ← wait for response to SET-AND-GET
    assignments[local-lecturer] ← temp2
    RELEASE (locks[local-lecturer])
    return "OK"

procedure SET-AND-GET (lecturer, course)
    ACQUIRE (locks[lecturer])
    old ← assignments[lecturer]
    assignments[lecturer] ← course
    RELEASE (locks[lecturer])
    return old
```

**Q 6.4** This code is subject to deadlock. Why?

**Q 6.5** For each of the following situations, indicate whether deadlock can occur. In each situation, there is no activity other than that mentioned.

**A.** Client A sends EXCHANGE ("Herodotus", "Augustine") at the same time that client B sends EXCHANGE ("Herodotus", "Augustine"), both to the Subliminal Sciences server.

**B.** Client A sends EXCHANGE ("Herodotus", "Augustine") at the same time that client B sends EXCHANGE ("Augustine", "Herodotus"), both to the Subliminal Sciences server.

**C.** Client A sends CROSSEXCHANGE ("Augustine", "Socrates") to the Subliminal Sciences server at the same time that client B sends CROSSEXCHANGE ("Descartes", "Herodotus") to the Department of Dialectic server.

**D.** Client A sends CROSSEXCHANGE ("Augustine", "Socrates") to the Subliminal Sciences server at the same time that client B sends CROSSEXCHANGE ("Socrates", "Augustine") to the Department of Dialectic server.

**E.** Client A sends CROSSEXCHANGE ("Augustine", "Socrates") to the Subliminal Sciences server at the same time that client B sends CROSSEXCHANGE ("Descartes", "Augustine") to the Department of Dialectic server.

## 7 Banking on Local Remote Procedure Call

*(Chapter 5)*

The bank president has asked Ben Bitdiddle to add enforced modularity to a large banking application. Ben splits the program into two pieces: a client and a service. He wants to use remote procedure calls to communicate between the client and service, which both run on the same physical machine with one processor. Ben explores an implementation, which the literature calls *lightweight remote procedure call* (LRPC). Ben's version of LRPC uses user-level gates. User gates can be bootstrapped using two kernel gates—one gate that registers the name of a user gate and a second gate that performs the actual transfer:

- REGISTER_GATE (*stack*, *address*). It registers address *address* as an entry point, to be executed on the stack *stack*. The kernel stores these addresses in an internal table.
- TRANSFER_TO_GATE (*address*). It transfers control to address *address*. A client uses this call to transfer control to a service. The kernel must first check if *address* is an address that is registered as a gate. If so, the kernel transfers control; otherwise it returns an error to the caller.

We assume that a client and service each run in their own virtual address space. On initialization, the service registers an entry point with REGISTER_GATE and allocates a block, at address *transfer*. Both the client and service map the transfer block in each address space with READ and WRITE permissions. The client and service use this shared transfer page to communicate the arguments to and results of a remote procedure call. The client and service each start with one thread. There are no user programs other than the client and service running on the machine.

The following pseudocode summarizes the initialization:

*Service*

```
procedure INIT_SERVICE ()
    REGISTER_GATE (STACK, receive)
    ALLOCATE_BLOCK (transfer)
    MAP (my_id, transfer, shared_server)
    while TRUE do YIELD ()
```

*Client*

```
procedure INIT_CLIENT ()
    MAP (my_id, transfer, shared_client)
```

When a client performs an LRPC, the client copies the arguments of the LRPC into the transfer page. Then, it calls TRANSFER_TO_GATE to transfer control to the service address space at the registered address *receive*. The client thread, which is now in the service's address space, performs the requested operation (the code for the procedure at the address *receive* is not shown because it is not important for the questions). On returning from the requested operation, the procedure at the address *receive* writes the result parameters in the transfer block and transfers control back to the client's address space to the procedure RETURN_LRPC. Once back in the client address space in RETURN_LRPC, the client copies the results back to the caller. The following pseudocode summarizes the implementation of LRPC:

```
procedure LRPC (id, request)
    COPY (request, shared_client)
    TRANSFER_TO_GATE (receive)
    return

procedure RETURN_LRPC()
    COPY (shared_client, reply)
    return (reply)
```

Now that we know how to use the procedures REGISTER_GATE and TRANSFER_TO_GATE, let's turn our attention to the implementation of TRANSFER_TO_GATE (*entrypoint* is the internal kernel table recording gate information):

```
1   procedure TRANSFER_TO_GATE (address)
2       if id exists such that entrypoint[id].entry = address then
3           R1 ← USER_TO_KERNEL (entrypoint[id].stack)
4           R2 ← address
5           STORE R2, R1          // put address on service's stack
6           SP ← entrypoint[id].stack    // set SP to service stack
7           SUB 4, SP             // adjust stack
8           PMAR ← entrypoint[id].pmar // set page map address
9           USER ← ON             // switch to user mode
10          return                // returns to address
11      else
12          return (ERROR)
```

The procedure checks whether or not the service has registered *address* as an entry point (line *2*). Lines *4–7* push the entry address on the service's stack and set the

register SP to point to the service's stack. To be able to do so, the kernel must translate the address for the stack in the service address space into an address in the kernel address space so that the kernel can write the stack (line *3*). Finally, the procedure stores the page-map address register for the service into PMAR (line *8*), sets the user-mode bit to ON (line *9*), and invokes the gate's procedure by returning from TRANSFER_TO_GATE (line *10*), which loads *address* from the service's stack into PC.

The implementation of this procedure is tricky because its switches address spaces, and thus the implementation must be careful to ensure that it is referring to the appropriate variable in the appropriate address space. For example, after line *8* TRANSFER_TO_GATE runs the next instruction (line *9*) in the service's address space. This works only if the kernel is mapped in both the client and service's address space at the same address.

**Q 7.1** The procedure INIT_SERVICE calls YIELD. In which address space or address spaces is the code that implements the supervisor call YIELD located?

**Q 7.2** For LRPC to work correctly, must the two virtual addresses *transfer* have the same value in the client and service address space?

**Q 7.3** During the execution of the procedure located at address *receive* how many threads are running or are in a call to YIELD in the service address space?

**Q 7.4** How many supervisor calls could the client perform in the procedure LRPC?

**Q 7.5** Ben's goal is to enforce modularity. Which of the following statements are true statements about Ben's LRPC implementation?

- **A.** The client thread cannot transfer control to any address in the server address space.
- **B.** The client thread cannot overwrite any physical memory that is mapped in the server's address space.
- **C.** After the client has invoked TRANSFER_TO_GATE in LRPC, the server is guaranteed to invoke RETURN_LRPC.
- **D.** The procedure LRPC ought to be modified to check the response message and process only valid responses.

**Q 7.6** Assume that REGISTER_GATE and TRANSFER_TO_GATE are also used by other programs. Which of the following statements is true about the implementations of REGISTER_GATE and TRANSFER_TO_GATE?

- **A.** The kernel might use an invalid address when writing the value *address* on the stack passed in by a user program.
- **B.** A user program might use an invalid address when entering the service address space.
- **C.** The kernel transfers control to the server address space with the user-mode bit switched OFF.
- **D.** The kernel enters the server address space only at the registered address entry *address*.

Ben modifies the client to have multiple threads of execution. If one client thread calls the server and the procedure at address *receive* calls YIELD, another client thread can run on the processor.

**Q 7.7** Which of the following statements is true about the implementation of LRPC with multiple threads?

- **A.** On a single-processor machine, there can be race conditions when multiple client threads call LRPC, even if the kernel schedules the threads non-preemptively.
- **B.** On a single-processor machine, there can be race conditions when multiple clients threads call LRPC and the kernel schedules the threads preemptively.
- **C.** On multiprocessor computer, there can be race conditions when multiple client threads call LRPC.
- **D.** It is impossible to have multiple threads if the computer doesn't have multiple physical processors.

*2004-1-4...10*

## 8 The Bitdiddler*

*(Chapter 5)*

Ben Bitdiddle is designing a file system for a new handheld computer, the Bitdiddler, which is designed to be especially simple for, as he likes to say, "people who are just average, like me."

In keeping with his theme of simplicity and ease of use for average people, Ben decides to design 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 following figure illustrates the basic layout of the Bitdiddler file system:

*Credit for developing this problem set goes to Samuel R. Madden.

The file system provides six primary calls: CREATE, OPEN, READ, WRITE, CLOSE, and UNLINK. Ben implements all six correctly and in a straightforward way, as shown below. 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.

```
procedure CREATE (filename)
    scan all non-free inodes to ensure filename is not a duplicate (return ERROR if
    duplicate)
    find a free inode in the inode list
    update the inode with 0 data blocks, mark it as in use, write it to disk
    update the free list to indicate the inode is in use, write free list to disk

procedure OPEN (filename)                // returns a file handle
    scan non-free inodes looking for filename
    if found, allocate and return a file handle fh that refers to that inode

procedure WRITE (fh, buf, len)
    look in file handle fh to determine inode of the file, read inode from disk
    if there is free space in last block of file, write to it
    determine number of new blocks needed, n
    for i ← 1 to n
        use free list to find a free block b
        update free list to show b is in use, write free list to disk
        add b to inode, write inode to disk
        write appropriate data for block b to disk

procedure READ (fh, buf, len)
    look in file handle fh to determine inode of the file, read inode from disk
    read len bytes of data from the current location in file into buf

procedure CLOSE (fh)
    remove fh from the file handle table

procedure UNLINK (filename)
    scan non-free inodes looking for filename, mark that inode as free
    write inode to disk
    mark data blocks used by file as free in free list
    write modified free list blocks to disk
```

Ben writes the following simple application for the Bitdiddler:

```
CREATE (filename)
fh ← OPEN (filename)
WRITE (fh, app_data, LENGTH (app_data))      // app_data is some data to be written
CLOSE (fh)
```

**Q 8.1** Ben notices that if he pulls the batteries out of the Bitdiddler while running his application and then replaces the batteries and reboots the machine, the file his application created exists but contains unexpected data that he didn't write into the

file. Which of the following are possible explanations for this behavior? (Assume that the disk controller never writes partial blocks.)

**A.** The free list entry for a data page allocated by the call to WRITE was written to disk, but neither the inode nor the data page itself was written.
**B.** The inode allocated to Ben's application previously contained a (since deleted) file with the same name. If the system crashed during the call to CREATE, it may cause the old file to reappear with its previous contents.
**C.** The free list entry for a data page allocated by the call to WRITE as well as a new copy of the inode were written to disk, but the data page itself was not.
**D.** The free list entry for a data page allocated by the call to WRITE as well as the data page itself were written to disk, but the new inode was not.

**Q 8.2** Ben decides to fix inconsistencies in the Bitdiddler's file system by scanning its data structures on disk every time the Bitdiddler starts up. Which of the following inconsistencies can be identified using this approach (without modifying the Bitdiddler implementation)?

**A.** In-use blocks that are also on the free list.
**B.** Unused blocks that are not on the free list.
**C.** In-use blocks that contain data from previously unlinked files.
**D.** Blocks used in multiple files.

*2007-3-6 & 7*

## 9 Ben's Kernel

*(Chapter 5)*

Ben develops an operating system for a simple computer. The operating system has a kernel that provides virtual address spaces, threads, and output to a console.

Each application has its own user-level address space and uses one thread. The kernel program runs in the kernel address space but doesn't have its own thread. (The kernel program is described in more detail below.)

The computer has one processor, a memory, a timer chip (which will be introduced later), a console device, and a bus connecting the devices. The processor has a user-mode bit and is a ***multiple register set*** design, which means that it has two sets of program counter (PC), stack pointer (SP), and page-map address registers (PMAR). One set is for user space (the user-mode bit is set to ON): *upc*, *usp*, and *upmar*. The other set is for kernel space (the user-mode bit is set to OFF): *kpc*, *ksp*, and *kpmar*. Only programs in kernel mode are allowed to store to *upmar*, *kpc*, *ksp*, and *kpmar*—storing a value in these registers is an illegal instruction in user mode.

The processor switches from user to kernel mode when one of three events occurs: an application issues an illegal instruction, an application issues a supervisor call instruction (with the SVC instruction), or the processor receives an

interrupt in user mode. The processor switches from user to kernel mode by setting the user-mode bit OFF. When that happens, the processor continues operation but using the current values in the *kpc*, *ksp*, and *kpmar*. The user program counter, stack pointer, and page-map address values remain in *upc*, *usp*, and *upmar*, respectively.

To return from kernel to user space, a kernel program executes the RTI instruction, which sets the user-mode bit to ON, causing the processor to use *upc*, *usp*, and *upmar*. The *kpc*, *ksp*, and *kpmar* values remain unchanged, awaiting the next SVC. In addition to these registers, the processor has four general-purpose registers: *ur0*, *ur1*, *kr0*, and *kr1*. The *ur0* and *ur1* pair are active in user mode. The *kr0* and *kr1* pair are active in kernel mode.

Ben runs two user applications. Each executes the following set of programs:

```
integer t initially 1                    // initial value for shared variable t
procedure MAIN ()
    do forever
        t ← t + t
            PRINT (t)
            YIELD ()
procedure YIELD
        SVC 0
```

PRINT prints the value of *t* on the output console. The output console is an output-only device and generates no interrupts.

The kernel runs each program in its own user-level address space. Each user address space has one thread (with its own stack), which is managed by the kernel:

```
integer currentthread            // index for the current user thread

structure thread[2]              // Storage place for thread state when not running
    integer sp                   // user stack pointer
    integer pc                   // user program counter
    integer pmar                 // user page-map address register
    integer r0                   // user register 0
    integer r1                   // user register 1

procedure DOYIELD ()
    thread[currentthread].sp ← usp                    // save registers
    thread[currentthread].pc ← upc
    thread[currentthread].pmar ← upmar
    thread[currentthread].r0 ← ur0
    thread[currentthread].r1 ← ur1
    currentthread ← (currentthread + 1) modulo 2      // select new thread
    usp ← thread[currentthread].sp                    // restore registers
    upc ← thread[currentthread].pc
    upmar ← thread[currentthread].pmar
    ur0 ← thread[currentthread].r0
    ur1 ← thread[currentthread].r1
```

For simplicity, this non-preemptive thread manager is tailored for just the two user threads that are running on Ben's kernel. The system starts by executing the procedure KERNEL. Here is its code:

```
procedure KERNEL ()
    CREATE_THREAD (MAIN)          // Set up Ben's two threads
    CREATE_THREAD (MAIN)          //
    usp ← thread[1].sp            // initialize user registers for thread 1
    upc ← thread[1].pc
    upmar ← thread[1].pmar
    ur0 ← thread[1].r0
    ur1 ← thread[1].r1
    do forever
        RTI                       // Run a user thread until it issues an SVC
        n ← ???                   // See question Q 9.1
        if n = 0 then DOYIELD()
```

Since the kernel passes control to the user with the RTI instruction, when the user executes an SVC, the processor continues execution in the kernel at the instruction following the RTI.

Ben's operating system sets up three page maps, one for each user program, and one for the kernel program. Ben has carefully set up the page maps so that the three address spaces don't share any physical memory.

**Q 9.1** Describe how the supervisor obtains the value of *n*, which is the identifier for the SVC that the calling program has invoked.

**Q 9.2** How can the current address space be switched?

**A.** By the kernel writing the *kpmar* register.
**B.** By the kernel writing the *upmar* register.
**C.** By the processor changing the user-mode bit.
**D.** By the application writing the *kpmar* or *upmar* registers.
**E.** By DOYIELD saving and restoring *upmar.*

**Q 9.3** Ben runs the system for a while, watching it print several results, and then halts the processor to examine its state. He finds that it is in the kernel, where it is just about to execute the RTI instruction. In which procedure(s) could the user-level thread resume when the kernel executes that RTI instruction?

**A.** In the procedure KERNEL.
**B.** In the procedure MAIN.
**C.** In the procedure YIELD.
**D.** In the procedure DOYIELD.

**Q 9.4** In Ben's design, what mechanisms play a role in enforcing modularity?

**A.** Separate address spaces because wild writes from one application cannot modify the data of the other application.
**B.** User-mode bit because it disallows user programs to write to upmar and kpmar.
**C.** The kernel because it forces threads to give up the processor.
**D.** The application because it has few lines of code.

Ben reads about the timer chip in his hardware manual and decides to modify the kernel to take advantage of it. At initialization time, the kernel starts the timer chip, which will generate an interrupt every 100 milliseconds. (Ben's computer has no other sources of interrupts.) Note that the interrupt-enable bit is OFF when executing in the kernel address space; the processor checks for interrupts only before executing a user-mode instruction. Thus, whenever the timer chip generates an interrupt while the processor is in kernel mode, the interrupt will be delayed until the processor returns to user mode. An interrupt in user mode causes an SVC -1 instruction to be inserted in the instruction stream. Finally, Ben modifies the kernel by replacing the **do forever** loop and adding an interrupt handler, as follows:

```
do forever
    RTI                          // Run a user thread until it issues an SVC
    n ← ???                      // Assume answer to question Q 9.1
    if n = 1 then DOINTERRUPT ()
    if n = 0 then DOYIELD ()

procedure DOINTERRUPT ()
    DOYIELD ()
```

Do not make any assumption about the speed of the processor.

**Q 9.5** Ben again runs the system for a while, watching it print several results, and then he halts the processor to examine its state. Once again, he finds that it is in the kernel, where it is just about to execute the RTI instruction. In which procedure(s) could the user-level thread resume after the kernel executes the RTI instruction?

**A.** In the procedure DOINTERRUPT.
**B.** In the procedure KERNEL.
**C.** In the procedure MAIN.
**D.** In the procedure YIELD.
**E.** In the procedure DOYIELD.

**Q 9.6** In Ben's second design, what mechanisms play a role in enforcing modularity?

**A.** Separate address spaces because wild writes from one application cannot modify the data of the other application.
**B.** User-mode bit because it disallows user programs to write to *upmar* and *kpmar*.
**C.** The timer chip because it, in conjunction with the kernel, forces threads to give up the processor.
**D.** The application because it has few lines of code.

Ben modifies the two user programs to share the variable *t*, by mapping *t* in the virtual address space of both user programs at the same place in physical memory. Now both threads read and write the same *t*.

Note that registers are not shared between threads: the scheduler saves and restores the registers on a thread switch. Ben's simple compiler translates the critical region of code:

$$t \leftarrow t + t$$

into the processor instructions:

```
100  LOAD t, r0        // read t into register 0
104  LOAD t, r1        // read t into register 1
108  ADD r1, r0        // add registers 0 and 1, leave result in register 0
112  STORE r0, t       // store register 0 into t
```

The numbers in the leftmost column in this code are the virtual addresses where the instructions are stored in both virtual address spaces. Ben's processor executes the individual instructions atomically.

**Q 9.7** What values can the applications print (don't worry about overflows)?

**A.** Some odd number.
**B.** Some even number other than a power of two.
**C.** Some power of two.
**D.** 1

In a conference proceedings, Ben reads about an idea called *restartable atomic regions** and implements them. If a thread is interrupted in a critical region, the thread manager restarts the thread at the beginning of the critical region when it resumes the thread. Ben recodes the interrupt handler as follows:

```
procedure DOINTERRUPT ()
    if upc ≥ 100 and upc ≤ 112 then  // Were we in the critical region?
        upc ← 100                    // yes, restart critical region when resumed!
    DOYIELD ()
```

The processor increments the program counter after interpreting an instruction and before processing interrupts.

**Q 9.8** Now, what values can the applications print (don't worry about overflows)?

**A.** Some odd number.
**B.** Some even number other than a power of two.
**C.** Some power of two.
**D.** 1

*Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1992), pages 223–233.

**Q 9.9** Can a second thread enter the region from virtual addresses 100 through 112 while the first thread is in it (i.e., the first thread's *upc* contains a value in the range 100 through 112)?

**A.** Yes, because while the first thread is in the region, an interrupt may cause the processor to switch to the second thread and the second thread might enter the region.
**B.** Yes, because the processor doesn't execute the first three lines of code in DOINTERRUPT atomically.
**C.** Yes, because the processor doesn't execute DOYIELD atomically.
**D.** Yes, because MAIN calls YIELD.

Ben is exploring if he can put just any code in a restartable atomic region. He creates a restartable atomic region that contains three instructions, which swap the content of two variables *a* and *b* using a temporary *x*:

```
100   x ← a
104   a ← b
108   b ← x
```

Ben also modifies DOINTERRUPT, replacing 112 with 108:

```
procedure DOINTERRUPT ()
    if upc ≥ 100 and upc ≤ 108 then      // Were we in the critical region?
        upc ← 100;                       // yes, restart critical region when resumed!
    DOYIELD ()
```

Variables *a* and *b* start out with the values $a = 1$ and $b = 2$, and the timer chip is running.

**Q 9.10** What are some possible outcomes if a thread executes this restartable atomic region and variables *a*, *b*, and *x* are not shared?

**A.** $a = 2$ and $b = 1$
**B.** $a = 1$ and $b = 2$
**C.** $a = 2$ and $b = 2$
**D.** $a = 1$ and $b = 1$

*2003-1-5...13*

## 10 A Picokernel-Based Stock-Ticker System

*(Chapter 5)*

Ben Bitdiddle decides to design a computer system based on a new kernel architecture he calls *picokernels* and on a new hardware platform called *simplePC*. Ben has paid attention to Section 1.1 and is going for extreme simplicity. The simplePC platform contains one simple processor, a page-based virtual memory manager (which translates the virtual addresses issued by the processor), a memory module, and an

input and output device. The processor has two special registers, a program counter (PC) and a stack pointer (SP). The SP points to the value on the top of the stack.

The calling convention for the simplePC processor uses a simple stack model:

- A call to a procedure pushes the address of the instruction after the call onto the stack and then jumps to the procedure.
- Return from a procedure pops the address from the top of the stack and jumps.

Programs on the simplePC don't use local variables. Arguments to procedures are passed in registers, which are *not* saved and restored automatically. Therefore, the only values on the stack are return addresses.

Ben develops a simple stock-ticker system to track the stocks of the start-up he joined. The program reads a message containing a single integer from the input device and prints it on the output device:

```
101.  boolean input_available

1.    procedure READ_INPUT ()
2.        do forever
3.            while input_available = FALSE do nothing // idle loop
4.            PRINT_MSG(quote)
5.            input_available ← FALSE

200.  boolean output_done
201.  structure output_buffer at 71FFF2hex        // hardware address of output buffer
202.      integer quote

12.   procedure PRINT_MSG (m)
13.       output_buffer.quote ← m
14.       while output_done = FALSE do nothing  // idle loop
15.       output_done ← FALSE

17.   procedure MAIN ()
18.       READ_INPUT ()
19.       halt                                  // shutdown computer
```

In addition to the MAIN program, the program contains two procedures: READ_INPUT and PRINT_MSG. The procedure READ_INPUT spin-waits until *input_available* is set to TRUE by the input device (the stock reader). When the input device receives a stock quote, it places the quote value into *msg* and sets *input_available* to TRUE.

The procedure PRINT_MSG prints the message on an output device (a terminal in this case); it writes the value stored in the message to the device and waits until it is printed; the output device sets *output_done* to TRUE when it finishes printing.

The numbers on each line correspond to addresses as issued by the processor to read and write instructions and data. Assume that each line of pseudocode compiles into one machine instruction and that there is an implicit **return** at the end of each procedure.

**Q 10.1** What do these numbers mentioned on each line of the program represent?

**A.** Virtual addresses.
**B.** Physical addresses.
**C.** Page numbers.
**D.** Offsets in a virtual page.

Ben runs the program directly on simplePC, starting in MAIN, and at some point he observes the following values on the stack (remember, only the stock-ticker program is running):

```
stack
19
5      ← stack pointer
```

**Q 10.2** What is the meaning of the value 5 on the stack?

**A.** The return address for the next return instruction.
**B.** The return address for the previous return instruction.
**C.** The current value of PC.
**D.** The current value of SP.

**Q 10.3** Which procedure is being executed by the processor?

**A.** READ_INPUT
**B.** PRINT_MSG
**C.** MAIN

**Q 10.4** PRINT_MSG writes a value to *quote*, which is stored at the address $71FFF2_{hex}$, with the expectation that the value will end up on the terminal. What technique is used to make this work?

**A.** Memory-mapped I/O.
**B.** Sequential I/O.
**C.** Streams.
**D.** Remote procedure call.

Ben wants to run multiple instances of his stock-ticker program on the simplePC platform so that he can obtain more frequent updates to track more accurately his current net worth. Ben buys another input and output device for the system, hooks them up, and he implements a trivial thread manager:

```
300. integer threadtable[2];        // stores stack pointers of threads.
                                    // first slot is threadtable[0]
302. integer current_thread initially 0;

21.   procedure YIELD ()
22.       threadtable[current_thread] ← SP   // move value of SP into table
23.       current_thread ← (current_thread + 1) modulo 2
24.       SP ← threadtable[current_thread]   // load value from table into SP
25.       return
```

Each thread reads from and writes to its own device and has its own stack. Ben also modifies READ_INPUT:

```
100. integer msg[2]                                  // CHANGED to use array
102. boolean input_available[2]                      // CHANGED to use array
30.    procedure READ_INPUT ()
31.        do forever
32.            while input_available[current_thread] = FALSE do        // CHANGED
33.            YIELD ()                              // CHANGED
34.            continue                              // CHANGED
35.        PRINT_MSG (msg[current_thread])           // CHANGED to use array
36.        input_available[current_thread] ← FALSE   // CHANGED to use array
```

Ben powers up the simplePC platform and starts each thread running in MAIN. The two threads switch back and forth correctly. Ben stops the program temporarily and observes the following stacks:

```
stack of thread 0                    stack of thread 1
        19                                   19
        36 ← stack pointer                   34 ← stack pointer
```

**Q 10.5** Thread 0 was running (i.e., *current_thread* = 0). Which instruction will the processor be running after thread 0 executes the **return** instruction in YIELD the next time?

**A.** 34. **continue**
**B.** 19. **halt**
**C.** 35. PRINT_MSG (*msg*[*current_thread*]);
**D.** 36. *input_available*[*current_thread*] ← FALSE;

and which thread will be running?

**Q 10.6** What address values can be on the stack of each thread?

**A.** Addresses of any instruction.
**B.** Addresses to which called procedures return.
**C.** Addresses of any data location.
**D.** Addresses of instructions and data locations.

Ben observes that each thread in the stock-ticker program spends most of its time polling its input variable. He introduces an explicit procedure that the devices can use to notify the threads. He also rearranges the code for modularity:

```
400.  integer state[2];

40.   procedure SCHEDULE_AND_DISPATCH ()
41.       threadtable[current_thread] ← SP
42.       while (what should go here?) do        // See question Q 10.7.
43.           current_thread ← (current_thread + 1) modulo 2
45.       SP ← threadtable[current_thread];
46.       return

50.   procedure YIELD()
51.       state[current_thread] ← WAITING
52.       SCHEDULE_AND_DISPATCH ()
53.       return

60.   procedure NOTIFY (n)
61.       state[n] ← RUNNABLE
62.       return
```

When the input device receives a new stock quote, the device interrupts the processor and saves the PC of the currently running thread on the currently running thread's stack. Then the processor runs the interrupt procedure. When the interrupt handler returns, it pops the return address from the current stack, returning control to a thread. The pseudocode for the interrupt handler is:

```
procedure DEVICE (n)                                  // interrupt for input device n
    push current thread's PC on stack pointed to by SP;
    while input_available[n] = TRUE do nothing;       // wait until read_input is done
                                                      // with the last input
    msg[n] ← stock quote
    input_available[n] ← TRUE
    NOTIFY (n)                                        // notify thread n
    return                                            // i.e., pop PC
```

During the execution of the interrupt handler, interrupts are disabled. Thus, an interrupt handler and the procedures that it calls (e.g., NOTIFY) cannot be interrupted. Interrupts are reenabled when DEVICE returns.

Using the new thread manager, answer the following questions:

**Q 10.7** What expression should be evaluated in the **while** at address 42 to ensure correct operation of the thread package?

**A.** *state*[*current_thread*] = WAITING
**B.** *state*[*current_thread*] = RUNNABLE
**C.** *threadtable*[*current_thread*] = SP
**D.** FALSE

**Q 10.8** Assume thread 0 is running and thread 1 is not running (i.e., it has called YIELD). What event or events need to happen before thread 1 will run?

**A.** Thread 0 calls YIELD.
**B.** The interrupt procedure for input device 1 calls NOTIFY.
**C.** The interrupt procedure for input device 0 calls NOTIFY.
**D.** No events are necessary.

**Q 10.9** What values can be on the stack of each thread?

**A.** Addresses of any instruction except those in the device driver interrupt procedure.
**B.** Addresses of all instructions, including those in the device driver interrupt procedure.
**C.** Addresses to which procedures return.
**D.** Addresses of instructions and data locations.

**Q 10.10** Under which scenario can thread 0 deadlock?

**A.** When device 0 interrupts thread 0 just before the first instruction of YIELD.
**B.** When device 0 interrupts just after thread 0 completed the first instruction of YIELD.
**C.** When device 0 interrupts thread 0 between instructions 35 and 36 in the READ_INPUT procedure on page 454.
**D.** When device 0 interrupts when the processor is executing SCHEDULE_AND_DISPATCH and thread 0 is in the WAITING state.

*2000-1-7…16*

## 11 Ben's Web Service

*(Chapter 5)*

Ben Bitdiddle is so excited about Amazing Computer Company's plans for a new segment-based computer architecture that he takes the job the company offered him.

Amazing Computer Company has observed that using one address space per program puts the text, data, stack, and system libraries in the same address space. For example, a Web server has the program text (i.e., the binary instructions) for the Web server, its internal data structures such as its cache of recently-accessed Web pages, the stack, and a system library for sending and receiving messages all in a single address space. Amazing Computer Company wants to explore how to enforce modularity even further by separating the text, data, stack, and system library using a new memory system.

The Amazing Computer Company has asked every designer in the company to come up with a design to enforce modularity further. In a dusty book about the PDP-11/70, Ben finds a description of a hardware gadget that sits between the processor and the physical memory, translating virtual addresses to physical addresses. The PDP-11/70 used that gadget to allow each program to have its own address space, starting at address 0.

The PDP-11/70 did this through having one segment per program. Conceptually, each segment is a variable-sized, linear array of bytes starting at virtual address 0. Ben bases his memory system on the PDP-11/70's scheme with the intention of implementing hard modularity. Ben defines a segment through a segment descriptor:

```
structure segmentDescriptor
    physicalAddress physAddr
    integer length
```

The *physAddr* field records the address in physical memory where the segment is located. The *length* field records the length of the segment in bytes.

Ben's processor has addresses consisting of 34 bits: 18 bits to identify a segment and 16 bits to identify the byte within the segment:

| *segment_id* | *index* |
|---|---|
| 18 bits | 16 bits |

A virtual address that addresses a byte outside a segment (i.e., an *index* greater than the *length* of the segment) is illegal.

Ben's memory system stores the segment descriptors in a table, *segmentTable*, which has one entry for each segment:

```
structure segmentDescriptor
        segmentTable[NSEGMENT]
```

The segment table is indexed by ***segment_id***. It is shared among all programs and stored at physical address 0.

The processor used by Ben's computer is a simple RISC processor, which reads and writes memory using LOAD and STORE instructions. The LOAD and STORE instructions take a virtual address as their argument. Ben's computer has enough memory that all programs fit in physical memory.

Ben ports a compiler that translates a source program to generate machine instructions for his processor. The compiler translates into a position-independent machine code: JUMP instructions specify an offset relative to the current value of the program counter. To make a call into another segment, it supports the LONGJUMP instruction, which takes a virtual address and jumps to it.

Ben's memory system translates a virtual address to a physical address with TRANSLATE:

```
procedure TRANSLATE (addr)
    segment_id ← addr[0:17]
    segment ← segmentTable[segment_id]
    index ← addr[18:33]
    if index < segment.length then return segment.physAddr + index
    ...             // What should the program do here? (see Q 11.4, below)
```

After successfully computing the physical address, Ben's memory management unit retrieves the addressed data from physical memory and delivers it to the

processor (on a LOAD instruction) or stores the data in physical memory (on a STORE instruction).

**Q 11.1** What is the maximum sensible value of NSEGMENT?

**Q 11.2** Given the structure of a virtual address, what is the maximum size of a segment in bytes?

**Q 11.3** How many bits wide must a physical address be?

**Q 11.4** The missing code on line *6* should

- **A.** signal the processor that the instruction that issued the memory reference has caused an illegal address fault
- **B.** signal the processor that it should change to user mode
- **C.** **return** *index*
- **D.** signal the processor that the instruction that issues the memory reference is an interrupt handler

Ben modifies his Web server to enforce modularity between the different parts of the server. He allocates the text of the program in segment 1, a cache for recently used Web pages in segment 2, the stack in segment 3, and the system library in segment 4. Segment 4 contains the text of the library program but no variables (i.e., the library program doesn't store variables in its own segment).

**Q 11.5** To translate the Web server the compiler has to do which of the following?

- **A.** Compute the physical address for each virtual address.
- **B.** Include the appropriate segment ID in the virtual address used by a LOAD instruction.
- **C.** Generate LONGJUMP instructions for calls to procedures located in different segments.
- **D.** Include the appropriate segment ID in the virtual address used by a STORE instruction.

Ben runs the segment-based implementation of his Web server and to his surprise observes that errors in the Web server program can cause the text of the system library to be overwritten. He studies his design and realizes that the design is bad.

**Q 11.6** What aspect of Ben's design is bad and can cause the observed behavior?

- **A.** A STORE instruction can overwrite the segment ID of an address.
- **B.** A LONGJMP instruction in the Web server program may jump to an address in the library segment that is not the start of a procedure.
- **C.** It doesn't allow for paging of infrequently used memory to a secondary storage device.
- **D.** The web server program may get into an endless loop.

**Q 11.7** Which of the following extensions of Ben's design would address each of the preceding problems?

- **A.** The processor should have a protected user-mode bit, and there should be a separate segment table for kernel and user programs.
- **B.** Each segment descriptor should have a protection bit, which specifies whether the processor can write or only read from this segment.
- **C.** The LONGJMP instruction should be changed so that it can transfer control only to designated entry points of a segment.
- **D.** Segments should all be the same size, just like pages in page-based virtual memory systems.
- **E.** Change the operating system to use a preemptive scheduler.

The system library for Ben's Web server contains code to send and receive messages. A separate program, the network manager, manages the network card that sends and receives messages. The Web server and the network manager each have one thread of execution. Ben wants to understand why he needs eventcounts for sequence coordination of the network manager and the Web server, so he decides to implement the coordination twice, once using eventcounts and the second time using event variables.

Here are Ben's two versions of the Web server:

**Web server using eventcounts**

```
eventcount inCnt
integer doneCnt

procedure SERVE ()
do forever
    AWAIT (inCnt, doneCnt);
    DO_REQUEST ();
    doneCnt ← doneCnt + 1;
```

**Web server using events**

```
event input
integer inCnt
integer doneCnt

procedure SERVE ()
do forever
    while inCnt ≤ doneCnt do      //A
        WAITEVENT (Input);        //B
    DO_REQUEST ();                //C
```

Both versions use a thread manager as described in Chapter 5, except for the changes to support eventcounts or events. The eventcount version is exactly the one described in Chapter 5. The AWAIT procedure has semantics for eventcounts: when the Web server thread calls AWAIT, the thread manager puts the calling thread into the WAITING state unless *inCnt* exceeds *doneCnt*.

The event-based version is almost identical to the eventcount one but has a few changes. An ***event variable*** is a list of threads waiting for the event. The procedure WAITEVENT puts the current executing thread on the list for the event, records that the current thread is in the WAITING state, and releases the processor by calling YIELD.

In both versions, when the Web server has completed processing a packet, it increases *doneCnt*.

The two corresponding versions of the code for handling each packet arrival in the network manager are:

*Network manager using eventcounts*

```
ADVANCE (inCnt)
```

*Network manager using events*

```
inCnt ← inCnt + 1        //D
NOTIFYEVENT (input)      //E
```

The ADVANCE procedure wakes up the Web server thread if it is already asleep. The NOTIFYEVENT procedure removes all threads from the list of the event and puts them into the READY state. The shared variables are stored in a segment shared between the network manager and the Web server.

Ben is a bit worried about writing code that involves coordinating multiple activities, so he decides to test the code carefully. He buys a computer with one processor to run both the Web server and the network manager using a preemptive thread scheduler. Ben ensures that the two threads (the Web server and the network manager) never run inside the thread manager at the same time by turning off interrupts when the processor is running the thread manager's code (which includes ADVANCE, AWAIT, NOTIFYEVENT, and WAITEVENT).

To test the code, Ben changes the thread manager to preempt threads frequently (i.e., each thread runs with a short time slice). Ben runs the old code with eventcounts and the program behaves as expected, but the new code using events has the problem that the Web server sometimes delays processing a packet until the next packet arrives.

**Q 11.8** The program steps that might be causing the problem are marked with letters in the code of the event-based solution above. Using those letters, give a sequence of steps that creates the problem. (Some steps might have to appear more than once, and some might not be necessary to create the problem.)

*2002-1-4...11*

## 12 A Bounded Buffer with Semaphores

*(Chapter 5)*

Using semaphores, DOWN and UP (see Sidebar 5.7), Ben implements an in-kernel bounded buffer as shown in the pseudocode below. The kernel maintains an array of *port_infos*. Each *port_info* contains a bounded buffer. The content of the message structure is not important for this problem, other than that it has a field *dest_port*, which specifies the destination port. When a message arrives from the network, it generates an interrupt, and the network interrupt handler (INTERRUPT) puts the message in the bounded buffer of the port specified in the message. If there is no space in that bounded buffer, the interrupt handler throws the message away. A thread consumes a message by calling RECEIVE_MESSAGE, which removes a message from the bounded buffer of the port it is receiving from.

To coordinate the interrupt handler and a thread calling RECEIVE_MESSAGE, the implementation uses a semaphore. For each port, the kernel keeps a semaphore *n* that counts the number of messages in the port's bounded buffer. If *n* reaches 0, the thread calling DOWN in RECEIVE_MESSAGE will enter the WAITING state. When INTERRUPT adds a message to the buffer, it calls UP on *n*, which will wake up the thread (i.e., set the thread's state to RUNNABLE).

```
structure port_info
    semaphore instance n initially 0
    message instance buffer [NMSG]                    // an array of messages
    long integer in initially 0
    long integer out initially 0

procedure INTERRUPT (message instance m, port_info reference port)
    // an interrupt announcing the arrival of message m
    if port.in − port.out ≥ NMSG then                 // is there space?
        return                                        // No, ignore message
    port.buffer[port.in modulo NMSG] ← m
    port.in ← port.in + 1
    UP (port.in)

procedure RECEIVE_MESSAGE (port_info reference port)
1   ...                                               // another line of code will go here
    DOWN (port.in)
    m ← port.buffer[port.in modulo NMSG]
    port.out ← port.out + 1
    return m
```

The kernel schedules threads preemptively.

**Q 12.1** Assume that there are no concurrent invocations of INTERRUPT and that there are no concurrent invocations of RECEIVE_MESSAGE on the same port. Which of the following statements is true about the implementation of INTERRUPT and RECEIVE_MESSAGE?

**A.** There are no race conditions between two threads that invoke RECEIVE_MESSAGE concurrently on different ports.

**B.** The complete execution of UP in INTERRUPT will not be interleaved between the statements labeled *15* and *16* in DOWN in Sidebar 5.7.

**C.** Because DOWN and UP are atomic, the processor instructions necessary for the subtracting of *sem* in DOWN and adding to *sem* in UP will not be interleaved incorrectly.

**D.** Because *in* and *out* may be shared between the interrupt handler running INTERRUPT and a thread calling RECEIVE_MESSAGE on the same port, it is possible for INTERRUPT to throw away a message, even though there is space in the bounded buffer.

Alyssa claims that semaphores can also be used to make operations atomic. She proposes the following addition to a *port_info* structure:

```
semaphore instance mutex initially ????        // see question below
```

and adds the following line to RECEIVE_MESSAGE, on line 1 in the pseudocode above:

```
DOWN(port.mutex)                                // enter atomic section
```

Alyssa argues that these changes allow threads to concurrently invoke RECEIVE on the same port without race conditions, even if the kernel schedules threads preemptively.

**Q 12.2** To what value can *mutex* be initialized (by replacing ???? with a number in the *semaphore* declaration) to avoid race conditions and deadlocks when multiple threads call RECEIVE_MESSAGE on the same port?

**A.** 0
**B.** 1
**C.** 2
**D.** −1

*2006-1-11&12*

## 13 The Single-Chip NC*

*(Chapter 5)*

Ben Bitdiddle plans to create a revolution in computing with his just-developed $15 single-chip Network Computer, NC. In the NC network system, the network interface thread calls the procedure MESSAGE_ARRIVED when a message arrives. The procedure WAIT_FOR_MESSAGE can be called by a thread to wait for a message. To coordinate the sequences in which threads execute, Ben deploys another commonly used coordination primitive: *condition variables*.

Part of the code in the NC is as follows:

```
lock instance m
boolean message_here
condition instance message_present

procedure MESSAGE_ARRIVED ()
    message_here ← TRUE
    NOTIFY_CONDITION (message_present)  // notify threads waiting on this condition

procedure WAIT_FOR_MESSAGE ()
    ACQUIRE (m)
    while not message_here do
        WAIT_CONDITION (message_present, m);        // release m and wait
    RELEASE (m)
```

The procedures ACQUIRE and RELEASE are the ones described in Chapter 5. NOTIFY_CONDITION (*condition*) atomically wakes up all threads waiting for *condition* to become TRUE. WAIT_CONDITION (*condition, lock*) does several things atomically: it tests *condition*; if TRUE it returns; otherwise it puts the calling thread on the waiting queue for *condition* and releases *lock*. When NOTIFY_CONDITION wakens a thread, that thread becomes

*Credit for developing this problem set goes to David K. Gifford.

runnable, and when the scheduler runs that thread, WAIT_CONDITION reacquires *lock* (waiting, if necessary, until it is available) before returning to its caller.

Assume there are no errors in the implementation of condition variables.

**Q 13.1** It is possible that WAIT_FOR_MESSAGE will wait forever even if a message arrives while it is spinning in the **while** loop. Give an execution ordering of the above statements that would cause this problem. Your answer should be a simple list such as 1, 2, 3, 4.

**Q 13.2** Write new version(s) of MESSAGE_ARRIVED and/or WAIT_FOR_MESSAGE to fix this problem.

*1998-1-3a/b*

## 14 Toastac-25*

*(Chapters 5 and 7 [on-line])*

Louis P. Hacker bought a used Therac-25 (the medical irradiation machine that was involved in several fatal accidents—see Suggestions for Further Reading 1.9.5) for $14.99 at a yard sale. After some slight modifications, he has hooked it up to his home network as a computer-controllable turbo-toaster, which can toast one slice in under 2 milliseconds. He decides to use RPC to control the Toastac-25. Each toasting request starts a new thread on the server, which cooks the toast, returns an acknowledgment (or perhaps a helpful error code, such as "Malfunction 54"), and exits. Each server thread runs the following procedure:

```
procedure SERVER () {
    ACQUIRE (message_buffer_lock)
    DECODE (message)
    ACQUIRE (accelerator_buffer_lock)
    RELEASE (message_buffer_lock)
    COOK_TOAST ()
    ACQUIRE (message_buffer_lock)
    message ← "ack"
    SEND (message)
    RELEASE (accelerator_buffer_lock)
    RELEASE (message_buffer_lock)
```

**Q 14.1** To his surprise, the toaster stops cooking toast the first time it is heavily used! What has gone wrong?

**A.** Two server threads might deadlock because one has *message_buffer_lock* and wants *accelerator_buffer_lock*, while the other has *accelerator_buffer_lock* and wants *message_buffer_lock*.

**B.** Two server threads might deadlock because one has *accelerator_buffer_lock* and *message_buffer_lock*.

**C.** Toastac-25 deadlocks because COOK_TOAST is not an atomic operation.

**D.** Insufficient locking allows inappropriate interleaving of server threads.

*Credit for developing this problem set goes to Eddie Kohler.

Once Louis fixes the multithreaded server, the Toastac gets more use than ever. However, when the Toastac has many simultaneous requests (i.e., there are many threads), he notices that the system performance degrades badly—much more than he expected. Performance analysis shows that competition for locks is not the problem.

**Q 14.2** What is probably going wrong?

**A.** The Toastac system spends all its time context switching between threads.
**B.** The Toastac system spends all its time waiting for requests to arrive.
**C.** The Toastac gets hot, and therefore cooking toast takes longer.
**D.** The Toastac system spends all its time releasing locks.

**Q 14.3** An upgrade to a supercomputer fixes that problem, but it's too late—Louis is obsessed with performance. He switches from RPC to an asynchronous protocol, which groups several requests into a single message if they are made within 2 milliseconds of one another. On his network, which has a very high transit time, he notices that this speeds up some workloads far more than others. Describe a workload that is sped up and a workload that is not sped up. (An example of a possible workload would be one request every 10 milliseconds.)

**Q 14.4** As a design engineering consultant, you are called in to critique Louis's decision to move from RPC to asynchronous client/service. How do you feel about his decision? Remember that the Toastac software sometimes fails with a "Malfunction 54" instead of toasting properly.

*1996-1-5c/d & 1999-1-12/13*

## 15 BOOZE: Ben's Object-Oriented Zoned Environment

*(Chapters 5 and 6)*

Ben Bitdiddle writes a large number of object-oriented programs. Objects come in different sizes, but pages come in a fixed size. Ben is inspired to redesign his page-based virtual memory system (PAGE) into an object memory system. PAGE is a page-based virtual memory system like the one described in Chapter 5 with the extensions for multilevel memory systems from Chapter 6. BOOZE is Ben's *object-based virtual memory* system.* Of course, he can run his programs on either system.

Each BOOZE object has a unique ID called a UID. A UID has three fields: a disk address for the disk block that contains the object; an offset within that disk block where the object starts; and the size of the object.

```
structure uid
    integer blocknr        // disk address for disk block
    integer offset         // offset within block blocknr
    integer size           // size of object
```

*Ben chose this name after reading a paper by Ted Kaehler, "Virtual memory for an object-oriented Language" [Suggestions for Further Reading 6.1.4]. In that paper, Kaehler describes a memory management system called the Object-Oriented Zoned Environment, with the acronym OOZE.

Applications running on BOOZE and PAGE have similar structure. The only difference is that on PAGE, program refer to objects by their virtual address, while on BOOZE programs refer to objects by UIDs.

The two levels of memory in BOOZE and PAGE are main memory and disk. The disk is a linear array of fixed-size blocks of 4 kilobytes. A disk block is addressed by its block number. In *both* systems, the transfer unit between the disk and main memory is a 4-kilobyte block. Objects don't cross disk block boundaries, are smaller than 4 kilobytes, and cannot change size. The page size in PAGE is equal to the disk block size; therefore, when an application refers to an object, PAGE will bring in all objects on the same page.

BOOZE keeps an object map in main memory. The object map contains entries that map a UID to the memory address of the corresponding object.

```
structure mapentry
    uid instance UID
    integer addr
```

On all references to an object, BOOZE translates a UID to an address in main memory. BOOZE uses the following procedure (implemented partly in hardware and partly in software) for translation:

```
procedure ObjectToAddress(UID) returns address
    addr ← isPresent(UID)                  // is UID present in object map?
    if addr ≥ 0 then return addr           // UID is present, return addr
    addr ← findFreeSpace(UID.size)         // allocate space to hold object
    readObject(addr, UID)                  // read object from disk & store at addr
    enterIntoMap(UID, addr)                // enter UID in object map
    return addr                            // return memory address of object
```

isPresent looks up *UID* in the object map; if present, it returns the address of the corresponding object; otherwise, it returns 1. findFreeSpace allocates free space for the object; it might evict another object to make space available for this one. readObject reads the *page* that contains the object, and then copies the *object* to the allocated address.

**Q 15.1** What does *addr* in the *mapentry* data structure denote?

**A.** The memory address at which the object map is located.
**B.** The disk address at which to find a given object.
**C.** The memory address at which to find a given object that is *currently* resident in memory.
**D.** The memory address at which a given non-resident object *would have to be loaded*, when an access is made to it.

**Q 15.2** In what way is BOOZE better than PAGE?

- **A.** Applications running on BOOZE generally use less main memory because BOOZE stores only objects that are in use.
- **B.** Applications running on BOOZE generally run faster because UIDs are smaller than virtual addresses.
- **C.** Applications running on BOOZE generally run faster because BOOZE transfers objects from disk to main memory instead of complete pages.
- **D.** Applications running on BOOZE generally run faster because typical applications will exhibit better locality of reference.

When FINDFREESPACE cannot find enough space to hold the object, it needs to write one or more objects back to the disk to create free space. FINDFREESPACE uses WRITEOBJECT to write an object to the disk.

Ben is figuring out how to implement WRITEOBJECT. He is considering the following options:

```
1.    procedure WRITEOBJECT (addr, UID)
          WRITE(addr, UID.blocknr, 4096)

2.    procedure WRITEOBJECT(addr, UID)
          READ(buffer, UID.blocknr, 4096)
          COPY(addr, buffer + UID.offset, UID.size)
          WRITE(buffer, UID.blocknr, 4096)
```

READ (*mem_addr*, *disk_addr*, 4096) and WRITE (*mem_addr*, *disk_addr*, 4096) read and write a 4-kilobyte page from/to the disk. COPY (*source*, *destination*, *size*) copies *size* bytes from a source address to a destination address in main memory.

**Q 15.3** Which implementation should Ben use?

- **A.** Implementation 2, since implementation 1 is incorrect.
- **B.** Implementation 1, since it is more efficient than implementation 2.
- **C.** Implementation 1, since it is easier to understand.
- **D.** Implementation 2, since it will result in better locality of reference.

Ben now turns his attention to optimizing the performance of BOOZE. In particular, he wants to reduce the number of writes to the disk.

**Q 15.4** Which of the following techniques will reduce the number of writes without losing correctness?

- **A.** Prefetching objects on a read.
- **B.** Delaying writes to disk until the application finishes its computation.
- **C.** Writing to disk only objects that have been modified.
- **D.** Delaying a write of an object to disk until it is accessed again.

Ben decides that he wants even better performance, so he decides to modify FINDFREESPACE. When FINDFREESPACE has to evict an object, it now tries not to write an

object modified in the last 30 seconds (in the belief that it may be used again soon). Ben does this by setting the *dirty* flag when the object is modified. Every 30 seconds, BOOZE calls a procedure WRITE_BEHIND that walks through the object map and writes out all objects that are dirty. After an object has been written, WRITE_BEHIND clears its *dirty* flag. When FINDFREESPACE needs to evict an object to make space for another, clean objects are the *only* candidates for replacement.

When running his applications on the latest version of BOOZE, Ben observes once in a while that BOOZE runs out of physical memory when calling OBJECTTOADDRESS for a new object.

**Q 15.5** Which of these strategies avoids the above problem?

**A.** When FINDFREESPACE cannot find any clean objects, it calls WRITE_BEHIND and then tries to find clean objects again.
**B.** BOOZE could call WRITE_BEHIND every second instead of every 30 seconds.
**C.** When FINDFREESPACE cannot find any clean objects, it picks *one* dirty object, writes the block containing the object to the disk, clears the *dirty* flag, and then uses that address for the new object.
**D.** All of the above strategies.

*1999-1-7…11*

## 16 OutOfMoney.com

*(Chapter 6, with a bit of Chapter 4)*

OutOfMoney.com has decided it needs a real product, so it is laying off most of its Marketing Department. To replace the marketing folks, and on the advice of a senior computer expert, OutOfMoney.com hires a crew of 16-year-olds. The 16-year-olds get together and decide to design and implement a video service that serves MPEG-1 video, so that they can watch Britney Spears on their computers in living color.

Since time to market is crucial, Mark Bitdiddle—Ben's 16-year-old kid brother, who is working for OutOfMoney—surfs the Web to find some code from which they can start. Mark finds some code that looks relevant, and he modifies it for OutOfMoney's video service:

```
procedure SERVICE ()
    do forever
        request ← RECEIVE_MESSAGE ()
        file ← GET_FILE_FROM_DISK (request)
        REPLY (file)
```

The SERVICE procedure waits for a message from a client to arrive on the network. The message contains a *request* for a particular file. The procedure GET_FILE_FROM_DISK reads the file from disk into the memory location *file*. The procedure REPLY sends the file from memory in a message back to the client.

(In the pseudocode, undeclared variables are local variables of the procedure in which they are used, and the variables are thus stored on the stack or in registers.)

Mark and his 16-year-old buddies also write code for a network driver to SEND and RECEIVE network packets, a simple file system to PUT and GET files on a disk, and a loader for booting a machine. They run their code on the bare hardware of an off-the-shelf personal computer with one disk, one processor (a Pentium III), and one network interface card (1 gigabit per second Ethernet). After the machine has booted, it starts one thread running SERVICE.

The disk has an average seek time of 5 milliseconds, a complete rotation takes 6 milliseconds, and its throughput is 10 megabytes per second when no seeks are required.

All files are 1 gigabyte (roughly a half hour of MPEG-1 video). The file system in which the files are stored has no cache, and it allocates data for a file in 8-kilobyte chunks. It pays no attention to file layout when allocating a chunk; as a result, disk blocks of the same file can be all over the disk. A 1-gigabyte file contains 131,072 8-kilobyte blocks.

**Q 16.1** Assuming that the disk is the main bottleneck, how long does the service take to serve a file?

Mark is shocked about the performance. Ben suggests that they should add a cache. Mark, impressed by Ben's knowledge, follows his advice and adds a 1-gigabyte cache, which can hold one file completely:

```
cache [1073741824]                    // 1-gigabyte cache

procedure SERVICE ()
    do forever
        request ← RECEIVE_MESSAGE ()
        file ← LOOK_IN_CACHE (request)
        if file = NULL then
            file ← GET_FILE_FROM_DISK (request)
            ADD_TO_CACHE (request, file)
        REPLY (file)
```

The procedure LOOK_IN_CACHE checks whether the file specified in the request is present in the cache and returns it if present. The procedure ADD_TO_CACHE copies a file to the cache.

**Q 16.2** Mark tests the code by asking once for every video stored. Assuming that the disk is the main bottleneck (serving a file from the cache takes 0 milliseconds), what now is the average time for the service to serve a file?

Mark is happy that the test actually returns every video. He reports back to the only person left in the Marketing Department that the prototype is ready to be evaluated. To keep the investors happy, the marketing person decides to use the prototype to run OutOfMoney's Web site. The one-person Marketing Department loads the machine up

with videos and launches the new Web site with a big PR campaign, blowing their remaining funding.

Seconds after they launch the Web site, OutOfMoney's support organization (also staffed by 16-year-olds) receives e-mail from unhappy users saying that the service is not responding to their requests. The support department measures the load on the service CPU and also the service disk. They observe that the CPU load is low and the disk load is high.

**Q 16.3** What is the most likely reason for this observation?

**A.** The cache is too large.
**B.** The hit ratio for the cache is low.
**C.** The hit ratio for the cache is high.
**D.** The CPU is not fast enough.

The support department beeps Mark, who runs to his brother Ben for help. Ben suggests using the example thread package of Chapter 5. Mark augments the code to use the thread package and after the system boots, it starts 100 threads, each running SERVICE:

```
for i from 1 to 100 do CREATE_THREAD (SERVICE)
```

In addition, mark modifies RECEIVE_MESSAGE and GET_FILE_FROM_DISK to release the processor by calling YIELD when waiting for a new message to arrive or waiting for the disk to complete a disk read. In no other place does his code release the processor. The implementation of the thread package is non-preemptive.

To take advantage of the threaded implementation, Mark modifies the code to read blocks of a file instead of complete files. He also runs to the store and buys some more memory so he can increase the cache size to 4 gigabytes. Here is his latest effort:

```
cache [4 × 1073741824]          // The 4-gigabyte cache, shared by all threads.

procedure SERVICE ()
    do forever
        request ← RECEIVE_MESSAGE ()
        file ← NULL
        for k from 1 to 131072 do
            block ← LOOK_IN_CACHE (request, k)
            if block = NULL then
                block ← GET_BLOCK_FROM_DISK (request, k)
                ADD_TO_CACHE (request, block, k)
            file ← file + block        // + concatenates strings
        REPLY (file)
```

The procedure LOOK_IN_CACHE (*request*, *k*) checks whether block *k* of the file specified in *request* is present; if the block is present, it returns it. The procedure

GET_BLOCK_FROM_DISK reads block *k* of the file specified in *request* from the disk into memory. The procedure ADD_TO_CACHE adds block *k* from the file specified in *request* to the cache.

Mark loads up the service with one video. He retrieves the video successfully. Happy with this result, Mark sends many requests for the single video in parallel to the service. He observes no disk activity.

**Q 16.4** Based on the information so far, what is the most likely explanation why Mark observes no disk activity?

Happy with the progress, Mark makes the service ready for running in production mode. He is worried that he may have to modify the code to deal with concurrency—his past experience has suggested to him that he needs an education, so he is reading Chapter 5. He considers protecting ADD_TO_CACHE with a lock:

```
lock instance cachelock                                  // A lock for the cache

procedure SERVICE ()
    do forever
        request ← RECEIVE_MESSAGE ()
        file ← NULL
        for k from 1 to 131072 do
            block ← LOOK_IN_CACHE (request, k)
            if block = NULL then
                block ← GET_BLOCK_FROM_DISK (request, k)
                ACQUIRE (cachelock)                      // use the lock
                ADD_TO_CACHE (request, block, k)
                RELEASE (cachelock)                      // here, too
            file ← file + block
        REPLY (file)
```

**Q 16.5** Ben argues that these modifications are not useful. Is Ben right?

Mark doesn't like thinking, so he upgrades OutOfMoney's Web site to use the multi-threaded code with locks. When the upgraded Web site goes live, Mark observes that most users watch the same three videos, while a few are watching other videos.

**Q 16.6** Mark observes a hit-ratio of 90% for blocks in the cache. Assuming that the disk is the main bottleneck (serving blocks from the cache takes 0 milliseconds), what is the average time for SERVICE to serve a single movie?

**Q 16.7** Mark loads a new Britney Spears video onto the service and observes operation as the first users start to view it. It is so popular that no users are viewing any other video. Mark sees that the first batch of viewers all start watching the video at about the same time. He observes that the service threads all read block 0 at about the

same time, then all read block 1 at about the same time, and so on. For this workload what is a good cache replacement policy?

**A.** Least-recently used.
**B.** Most-recently used.
**C.** First-in, first-out.
**D.** Last-in, first-out.
**E.** The replacement policy doesn't matter for this workload.

The Marketing Department is extremely happy with the progress. Ben raises another round of money by selling his BMW and launches another PR campaign. The number of users dramatically increases. Unfortunately, under high load the machine stops serving requests and has to be restarted. As a result, some users have to restart their videos from the beginning, and they call up the support department to complain. The problem appears to be some interaction between the network driver and the service threads. The driver and service threads share a fixed-sized input buffer that can hold 1,000 request messages. If the buffer is full and a message arrives, the driver drops the message. When the card receives data from the network, it issues an interrupt to the processor. This interrupt causes the network driver to run immediately on the stack of the currently running thread. The code for the driver and RECEIVE_MESSAGE is as follows:

```
buffer[1000]
lock instance bufferlock

procedure DRIVER ()
    message ← READ_FROM_INTERFACE ()
    ACQUIRE (bufferlock)
    if SPACE_IN_BUFFER () then ADD_TO_BUFFER (message)
    else DISCARD_MESSAGE (message)
    RELEASE (bufferlock)

procedure RECEIVE_MESSAGE ()
    while BUFFER_IS_EMPTY () do YIELD ()
    ACQUIRE (bufferlock)
    message ← REMOVE_FROM_BUFFER ()
    RELEASE (bufferlock)
    return message

procedure INTERRUPT ()
    DRIVER ()
```

**Q 16.8** Which of the following could happen under high load?

**A.** Deadlock when an arriving message interrupts DRIVER.
**B.** Deadlock when an arriving message interrupts a thread that is in RECEIVE_MESSAGE.
**C.** Deadlock when an arriving message interrupts a thread that is in REMOVE_FROM_BUFFER.
**D.** RECEIVE_MESSAGE misses a call to YIELD when the buffer is not empty, because it can be interrupted between the BUFFER_IS_EMPTY test and the call to YIELD.

**Q 16.9** What fixes should Mark implement?

**A.** Delete all the code dealing with locks.
**B.** DRIVER should run as a separate thread, to be awakened by the interrupt.
**C.** INTERRUPT and DRIVER should use an eventcount for sequence coordination.
**D.** DRIVER shouldn't drop packets when the buffer is full.

Mark eliminates the deadlock problems and, to attract more users, announces the availability of a new Britney Spears video. The news spreads rapidly, and an enormous number of requests for this one video start hitting the service. Mark measures the throughput of the service as more and more clients ask for the video. The resulting graph is plotted below. The throughput first increases while the number of clients increases, then reaches a maximum value, and finally drops off.

**Q 16.10** Why does the throughput decrease with a large number of clients?

**A.** The processor spends most of its time taking interrupts.
**B.** The processor spends most of its time updating the cache.
**C.** The processor spends most of its time waiting for the disk accesses to complete.
**D.** The processor spends most of its time removing messages from the buffer.

*2001-1-6...15*

# 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*. [Ch. 9]

**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. [Ch. 2]

**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. [Ch. 1]

**Access control list (ACL)** A list of principals authorized to have access to some object. [Ch. 11]

**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*. [Ch. 2]

**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. [Ch. 9]

**Active fault** A fault that is currently causing an error. Compare with *latent fault*. [Ch. 8]

**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*. [Ch. 7]

**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*. [Ch. 2]

**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. [Ch. 7]

**Address space** The name space of a location-addressed memory, usually a set of contiguous integers (0, 1, 2,...). [Ch. 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. [Ch. 11]

**Advertise** In a network-layer routing protocol, for a participant to tell other participants which network addresses it knows how to reach. [Ch. 7]

**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*.) [Ch. 2]

**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*. [Ch. 9]

**Any-to-any connection** A desirable property of a communication network, that any node be able to communicate with any other. [Ch. 7]

**Archive** A record, usually kept in the form of a log, of old data values, for auditing, recovery from application mistakes, or historical interest. [Ch. 9]

**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. [Ch. 2] 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*. [Ch. 7]

**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. [Ch. 4]

**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. [Ch. 4]

**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 before-or-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-or-nothing, 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*. [Ch. 9]

**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*. [Ch. 9]

**Authentication** Verifying the identity of a principal or the authenticity of a message. [Ch. 11]

**Authentication tag** A cryptographically computed string, associated with a message, that allows a receiver to verify the authenticity of the message. [Ch. 11]

**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. [Ch. 7]

**Authorization** A decision made by an authority to grant a principal permission to perform some operation, such as reading certain information. [Ch. 11]

**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*. [Ch. 8]

**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*. [Ch. 10]

**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*. [Ch. 8]

**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. [Ch. 1]

**Bandwidth** A measure of analog spectrum space for 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. [Ch. 7]

**Batching** A technique to improve performance by combining several operations into a single operation to reduce setup overhead. [Ch. 6]

**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. [Ch. 5] Compare with *all-or-nothing atomicity* and *atomic*. [Ch. 9]

**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. [Ch. 7]

**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. [Ch. 2]

**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 $10^{10}$. [Ch. 7]

**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. [Ch. 7]

**Blind write** An update to a data value $X$ by a transaction that did not previously read $X$. [Ch. 9]

**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. [Ch. 5]

**Bottleneck** The stage in a multistage pipeline that takes longer to perform its task than any of the other stages. [Ch. 6]

**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). [Ch. 7]

**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. [Ch. 7]

**Byzantine fault** A fault that generates inconsistent errors (perhaps maliciously) that can confuse or disrupt fault tolerance or security mechanisms. [Ch. 8]

**Cache** A performance-enhancing module that remembers the result of an expensive computation on the chance that the result may soon be needed again. [Ch. 2]

**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. [Ch. 10]

**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. [Ch. 11]

**Capacity** Any consistent measure of the size or amount of a resource. [Ch. 6]

**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*. [Ch. 9]

**Certificate** A message that attests the binding of a principal identifier to a cryptographic key. [Ch. 11]

**Certificate authority (CA)** A principal that issues and signs certificates. [Ch. 11]

**Certify** To check the accuracy, correctness, and completeness of a security mechanism. [Ch. 11]

**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. [Ch. 9]

**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. [Ch. 7]

**Cipher** Synonym for a *cryptographic transformation*. [Ch. 11]

**Ciphertext** The result of encryption. Compare with *plaintext*. [Ch. 11]

**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. [Ch. 7]

**Cleartext** Synonym for *plaintext*. [Ch. 11]

**Client** A module that initiates actions, such as sending a request to a service. [Ch. 4] At the end-to-end layer of a network, the end that initiates actions. Compare with *service*. [Ch. 7]

**Client/service organization** An organization that enforces modularity among modules of a computer system by limiting the interaction among the modules to messages. [Ch. 4]

**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 will be visible to concurrent threads only after the first thread closes the file. [Ch. 2]

**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. [Ch. 2]

**Coherence** 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. [Ch. 3] 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*. [Ch. 7]

**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-or-nothing 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*. [Ch. 9]

**Communication link** A data communication path between physically separated components. [Ch. 2]

**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. [Ch. 9]

**Complexity** A loosely defined notion that a system has so many components, interconnections, and irregularities that it is difficult to understand, implement, and maintain. [Ch. 1]

**Confidentiality** Limiting information access to authorized principals. *Secrecy* is a synonym. [Ch. 11]

**Confinement** Allowing a potentially untrusted program to have access to data, while ensuring that the program cannot release information. [Ch. 11]

**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.) [Ch. 7]

**Congestion collapse** When an increase in offered load causes a catastrophic decrease in useful work accomplished. [Ch. 7]

**Connection** A communication path that requires maintaining state between successive messages. See *set up* and *tear down.* [Ch. 7]

**Connectionless** Describes a communication path that does not require coordinated state and can be used without set up or tear down. See *connection*. [Ch. 7]

**Consensus** Agreement at separated sites on a data value despite communication failures. [Ch. 10]

**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*. [Ch. 10]

**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. [Ch. 10]

**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. [Ch. 2]

**Context reference** The name of a context. [Ch. 2]

**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. [Ch. 8]

**Control point** An entity that can adjust the capacity of a limited resource or change the load that a source offers. [Ch. 7]

**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. [Ch. 5]

**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. [Ch. 11]

**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. [Ch. 11]

**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. [Ch. 11]

**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. [Ch. 11]

**Cryptography** A discipline of theoretical computer science that specializes in the study of cryptographic transformations and protocols. [Ch. 11]

**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. [Ch. 7]

**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. [Ch. 6]

**Dangling reference** Use of a name that has outlived the binding of that name. [Ch. 3]

**Data integrity** Authenticity of the apparent content of a message or file. [Ch. 11] 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*. [Ch. 7]

**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. [Ch. 7]

**Deadlock** Undesirable interaction among a group of threads in which each thread is waiting for some other thread in the group to make progress. [Ch. 5]

**Decay** Unintended loss of stored state with the passage of time. [Ch. 2]

**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. [Ch. 8]

**Decrypt** To perform a reverse cryptographic transformation on a previously encrypted message to obtain the plaintext. Compare with *encrypt*. [Ch. 11]

**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*. [Ch. 2]

**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*. [Ch. 6]

**Destination** The network attachment point to which the payload of a packet is to be delivered. Sometimes used as shorthand for *destination address*. [Ch. 7]

**Destination address** An identifier of the destination of a packet, usually carried as a field in the header of the packet. [Ch. 7]

**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*. [Ch. 8]

**Digital signature** An authentication tag computed with public-key cryptography. [Ch. 11]

**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. [Ch. 2]

**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*. [Ch. 11]

**Do action** (n.) Term used in some systems for a *redo action*. [Ch. 9]

**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. [Ch. 5]

**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*. [Ch. 8]

**Duplex** Describes a link or connection between two stations that can be used in both directions. Compare with *simplex*, *half-duplex,* and *full-duplex*. [Ch. 7]

**Duplicate suppression** A transport protocol mechanism for achieving at-most-once delivery assurance, by identifying and discarding extra copies of packets or messages. [Ch. 7]

**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 2.1. [Ch. 2]

**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 2.1, but to minimize confusion this text avoids that usage. [Ch. 8]

**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 frame of its caller, then the caller's caller, and so on. Compare with *static scope*. [Ch. 2]

**Earliest deadline first scheduling policy** A scheduling policy for real-time systems that gives priority to the thread with the earliest deadline. [Ch. 6]

**Early drop** A predictive strategy for managing an overloaded resource: the system refuses service to some customers before the queue is full. [Ch. 7]

**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. [Ch. 1]

**Emulation** Faithfully simulating some physical hardware so that the simulated hardware can run any software that the physical hardware can. [Ch. 5]

**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. [Ch. 11]

**End-to-end** Describes communication between network attachment points, as contrasted with communication between points within the network or across a single link. [Ch. 7]

**End-to-end layer** The communication system layer that manages end-to-end communications. [Ch. 7]

**Enforced modularity** Modularity that prevents accidental errors from propagating from one module to another. Compare with *soft modularity*. [Ch. 4]

**Enumerate** To generate a list of all the names that can currently be resolved (that is, that have bindings) in a particular context. [Ch. 2]

**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. [Ch. 1] 2. In an interpreter, the state on which the interpreter should perform the actions directed by program instructions. [Ch. 2]

**Environment reference** The component of an interpreter that tells the interpreter where to find its environment. [Ch. 2]

**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. [Ch. 8]

**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. [Ch. 8]

**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. [Ch. 8]

**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. [Ch. 8]

**Error correction** A scheme to set to the correct value a data value or control signal that is in error. Compare with *error detection*. [Ch. 8]

**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*. [Ch. 7]

**Error detection** A scheme to discover that a data value or control signal is in error. Compare with *error correction*. [Ch. 8]

**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. See also the general definition of *error detection*. Compare with *error-correction code* and *checksum*. [Ch. 7]

**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. [Ch. 7]

**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. [Ch. 5]

**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. [Ch. 10]

**Exactly-once** A protocol assurance that the intended operation or message delivery was performed both at-least-once and at-most-once. [Ch. 4]

**Exception** An interrupt event that pertains to the thread that a processor is currently running. [Ch. 5]

**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.* [Ch. 2]

**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. [Ch. 11]

**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 to obtain a suitable timer value as quickly as possible. See also *exponential random backoff*. [Ch. 7]

**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. [Ch. 9]

**Export** In naming, to provide a name for an object that other objects can use. [Ch. 2]

**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*. [Ch. 8]

**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. [Ch. 8]

**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*. [Ch. 8]

**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. [Ch. 8]

**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. [Ch. 8]

**Fail-vote** Describes an *N*-modular redundancy system with a majority voter. [Ch. 8]

**Failure** The outcome when a component or system does not produce the intended result at its interface. Compare with *fault*. [Ch. 8]

**Failure tolerance** A measure of a system's ability 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. [Ch. 8]

**Fault** A defect in materials, design, or implementation that may (or may not) cause an error and lead to a failure. Compare with *failure*. [Ch. 8]

**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*. [Ch. 8]

**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. [Ch. 8]

**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. [Ch. 2]

**Fingerprint** Another term for a *witness*. [Ch. 10]

**First-come, first-served (FCFS) scheduling policy** A scheduling policy in which requests are processed in the order in which they arrive. [Ch. 6]

**First-in, first-out (FIFO) policy** 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. [Ch. 6]

**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. [Ch. 7] 2. In security, a system that allows untrusted programs to work with sensitive data but confines all program outputs to prevent unauthorized disclosure. [Ch. 11]

**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 write-through usually have a feature that allows the invoker to force some or all of their contents to the secondary storage medium. [Ch. 9]

**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*. [Ch. 8]

**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. [Ch. 11]

**Forwarding table** A table that tells the network layer which link to use to forward a packet, based on its destination address. [Ch. 7]

**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. [Ch. 7]

**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. [Ch. 7]

**Freshness** A property of a message in a security protocol: if the message is fresh, it is assured not to be a replay. [Ch. 11]

**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*. [Ch. 7]

**Gate** A predefined protected entry point into a domain. [Ch. 5]

**Generated name** A name created algorithmically, rather than chosen by a person. [Ch. 3]

**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. [Ch. 2]

**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*. [Ch. 7]

**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. [Ch. 8]

**Hard real-time scheduling policy** A real-time scheduler in which missing a deadline may result in a disaster. [Ch. 6]

**Hash function** A function that algorithmically derives a relatively short, fixed-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*. [Ch. 3]

**Header** Information that a protocol layer adds to the front of a packet. [Ch. 7]

**Hierarchical routing** A routing system that takes advantage of hierarchically assigned network destination addresses to reduce the size of its routing tables. [Ch. 7]

**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. [Ch. 1]

**Hit ratio** In a multilevel memory, the fraction of references satisfied by the primary memory device. [Ch. 6]

**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. [Ch. 7]

**Hot swap** To replace modules in a system while the system continues to provide service. [Ch. 8]

**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. [Ch. 4]

**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. [Ch. 3]

**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. [Ch. 2]

**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. [Ch. 1]

**Incremental backup** A backup copy that contains only data that has changed since making the previous backup copy. [Ch. 10]

**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. [Ch. 2]

**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". [Ch. 1]

**Install** In a system that uses logs to achieve all-or-nothing atomicity, to write data to cell storage. [Ch. 9]

**Instruction reference** A characteristic component of an interpreter: the place from which it will take its next instruction. [Ch. 2]

**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*. [Ch. 7]

**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. [Ch. 6]

**Intermittent fault** A persistent fault that is active only occasionally. Compare with *transient fault*. [Ch. 8]

**International Organization for Standardization (ISO)** An international non-governmental 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. [Ch. 7]

**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. [Ch. 2]

**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. [Ch. 2]

**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. [Ch. 10]

**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*. [Ch. 7]

**Jitter** In real-time applications, variability in the delivery times of successive data elements. [Ch. 7]

**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. [Ch. 6]

**Journal storage** Storage in which a WRITE or PUT appends a new value, rather than overwriting a previously stored value. Compare with *cell storage*. [Ch. 9]

**Kernel** A trusted intermediary that virtualizes resources for mutually distrustful modules running on the same computer. Kernel modules typically run with kernel mode enabled. [Ch. 5]

**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*. [Ch. 5]

**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. [Ch. 11]

**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. [Ch. 11]

**Latency** The delay between a change at the input to a system and the corresponding change at its output. [Ch. 2] 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. [Ch. 8]

**Latent fault** A fault that is not currently causing an error. Compare with *active fault*. [Ch. 8]

**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). [Ch. 1]

**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. [Ch. 6]

**Lexical scope** Another term for *static scope*. [Ch. 2]

**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. [Ch. 3]

**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*. [Ch. 2]. 3. (n.) In data communication, a communication path between two points. [Ch. 7]

**Link layer** The communication system layer that moves data directly from one physical point to another. [Ch. 7]

**List system** A design for an access control system in which each protected object is associated with a list of authorized principals. [Ch. 11]

**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. [Ch. 5]

**Locality of reference** A property of most programs that memory references tend to be clustered in both time and address space. [Ch. 6]

**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. [Ch. 5]

**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. [Ch. 9]

**Lock set** The collection of all locks acquired during the execution of a before-or-after action. [Ch. 9]

**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*. [Ch. 7]

**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. [Ch. 9]

**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*. [Ch. 10]

**Logical locking** Locking of higher-layer data objects such as records or fields of a database. Compare with *physical locking*. [Ch. 9]

**Manchester code** A particular type of phase encoding in which each bit is represented by two bits of opposite value. [Ch. 7]

**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. [Ch. 8]

**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. [Ch. 9]

**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. [Ch. 4]

**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*. [Ch. 8]

**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. [Ch. 8]

**Master** In a multiple-site replication scheme, the site to which updates are directed. Compare with *slave*. [Ch. 10]

**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. [Ch. 7]

**Mean time between failures (MTBF)** The sum of MTTF and MTTR for the same component or system. [Ch. 8]

**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. [Ch. 8]

**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. [Ch. 8]

**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. [Ch. 11]

**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*. [Ch. 2]

**Memoryless** A property of some time-dependent probabilistic processes, that the probability of what happens next does not depend on what has happened before. [Ch. 8]

**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). [Ch. 5]

**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. [Ch. 2]

**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*. [Ch. 7]

**Message authentication** The verification of the integrity of the origin and the data of a message. [Ch. 11]

**Message authentication code (MAC)** An authentication tag computed with shared-secret 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." [Ch. 11]

**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. [Ch. 3]

**Microkernel** A kernel organization in which most operating system components run in separate, user-mode address spaces. [Ch. 5]

**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." [Ch. 8]

**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*. [Ch. 6]

**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. [Ch. 3]

**Module** A system component that can be separately designed, implemented, managed, and replaced. [Ch. 1]

**Monolithic kernel** A kernel organization in which most operating system procedures run in a single, kernel-mode address space. [Ch. 5]

**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. [Ch. 6]

**MTU discovery** A procedure that systematically discovers the smallest maximum transmission unit along the path between two network attachment points. [Ch. 7]

**Multihomed** Describes a single physical interface between the network layer and the end-to-end layer that is associated with more than one network attachment point, each with its own network-layer address. [Ch. 7]

**Multilevel memory** Memory built out of two or more different memory devices that have significantly different latencies and cost per bit. [Ch. 6]

**Multiple lookup** A name-mapping algorithm that tries several contexts in sequence, looking for the first one that can successfully resolve a presented name. [Ch. 2]

**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. [Ch. 7]

**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. [Ch. 7]

***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. [Ch. 8]

**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. [Ch. 8]

**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. [Ch. 8]

**Name** A designator or an identifier of an object or value. A name is an element of a name space. [Ch. 2]

**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*. [Ch. 3]

**Name-mapping algorithm** See *naming scheme.* [Ch. 2]

**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. [Ch. 2]

**Name-to-key binding** A binding between a principal identifier and a cryptographic key. [Ch. 11]

**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. [Ch. 2]

**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. [Ch. 2]

**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. [Ch. 2]

**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*. [Ch. 7]

**Network** A communication system that interconnects more than two things. [Ch. 7]

**Network address** In a network, the identifier of the source or destination of a packet. [Ch. 7]

**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). [Ch. 7]

**Network layer** The communication system layer that forwards data through intermediate links to carry it to its intended destination. [Ch. 7]

**Non-discretionary access control** A property of an access control system. In a non-discretionary 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*. [Ch. 11]

**Non-preemptive scheduling** A scheduling policy in which threads run until they explicitly yield or wait. [Ch. 5]

**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*. [Ch. 2]

**Nonce** A unique identifier that should never be reused. [Ch. 7]

**Object** As used in naming, any software or hardware structure that can have a distinct name. [Ch. 2]

**Offered load** The amount of a shared service that a set of users attempt to utilize. *Presented load* is an occasionally encountered synonym. [Ch. 6]

**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. [Ch. 3]

**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. [Ch. 2]

**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. [Ch. 6]

**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*. [Ch. 9]

**Origin authenticity** Authenticity of the claimed origin of a message. Compare with *data integrity*. [Ch. 11]

**Overload** When offered load exceeds the capacity of a service for a specified period of time. [Ch. 6]

**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*. [Ch. 3]

**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*. [Ch. 7]

**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. [Ch. 7]

**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*. [Ch. 7]

**Page** In a page-based virtual memory system, the unit of translation between virtual addresses and physical addresses. [Ch. 5]

**Page fault** See *missing-page exception*.

**Page map** Data structure employed by the virtual memory manager to map virtual addresses to physical addresses. [Ch. 5]

**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. [Ch. 5]

**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. [Ch. 6]

**Page table** A particular form of a page map, in which the map is organized as an array indexed by page number. [Ch. 5]

**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-and-compare 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.* [Ch. 8]

**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. [Ch. 7]

**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. [Ch. 8]

**Password** A secret character string used to authenticate the claimed identity of an individual. [Ch. 11]

**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*. [Ch. 2]

**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. [Ch. 7]

**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.) [Ch. 7]

**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. [Ch. 9]

**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 2.1. The adjective "persistent" is used in some contexts as a synonym for stable and sometimes also in the sense of immutable. [Ch. 2]

**Persistent fault** A fault that cannot be masked by retry. Compare with *transient fault* and *intermittent fault*. [Ch. 8]

**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. [Ch. 7]

**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*. [Ch. 9]

**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. [Ch. 7]

**Physical address** An address that is translated geometrically to read or write data stored on a device. Compare with *virtual address*. [Ch. 5]

**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*. [Ch. 10]

**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*. [Ch. 9]

**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. [Ch. 7]

**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.* [Ch. 7]

**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*. [Ch. 11]

**Point-to-point** Describes a communication link between two stations, as contrasted with a broadcast or multipoint link. [Ch. 7]

**Polling** A style of interaction between threads or between a processor and a device in which one periodically checks whether the other needs attention. [Ch. 5]

**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. [Ch. 7]

**Preemptive scheduling** A scheduling policy in which a thread manager can interrupt and reschedule a running thread at any time. [Ch. 5]

**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*. [Ch. 6]

**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. [Ch. 9]

**Presentation protocol** A protocol that translates semantics and data of the network to match those of the local programming environment. [Ch. 7]

**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. [Ch. 8]

**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*. [Ch. 10]

**Primary device** In a multilevel memory system, the memory device that is faster and usually more expensive and thus smaller. Compare with *secondary device*. [Ch. 6]

**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. [Ch. 11]

**Priority scheduling policy** A scheduling policy in which some jobs have priority over other jobs. [Ch. 6]

**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. [Ch. 11]

**Private key** In public-key cryptography, the cryptographic key that must be kept secret. Compare with *public key*. [Ch. 11]

**Processing delay** In a communication network, that component of the overall delay contributed by computation that takes place in various protocol layers. [Ch. 7]

**Program counter** A processor register that holds the reference to the current or next instruction that the processor is to execute. [Ch. 2]

**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. [Ch. 9]

**Propagation delay** In a communication network, the component of overall delay contributed by the velocity of propagation of the physical medium used for communication. [Ch. 7]

**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. [Ch. 1]

**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. [Ch. 11]

**Protection group** A principal that is shared by more than one user. [Ch. 11]

**Protocol** An agreement between two communicating parties, for example, on the messages and the format of data that they intend to exchange. [Ch. 7]

**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*. [Ch. 11]

**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. [Ch. 11]

**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. [Ch. 4]

**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*. [Ch. 3]

**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. [Ch. 8]

**Qualified name** A name that includes an explicit context reference. [Ch. 2]

**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. [Ch. 7]

**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. [Ch. 7]

**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. [Ch. 10]

**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). [Ch. 5]

**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. [Ch. 2]

**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. [Ch. 2]

**Random drop** A strategy for managing an overloaded resource: the system refuses service to a queue member chosen at random. [Ch. 7]

**Random early detection (RED)** A combination of random drop and early drop. [Ch. 7]

**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. [Ch. 6]

**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. [Ch. 5]

**Read/write coherence** A property of a memory, that a READ always returns the result of the most recent WRITE. [Ch. 2]

**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. [Ch. 7]

**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. [Ch. 6]

**Real-time scheduling policy** A scheduler that attempts to schedule jobs in such a way that all jobs complete before their deadlines. [Ch. 6]

**Reassembly** Reconstructing a message by arranging, in correct order, the segments it was divided into for transmission. [Ch. 7]

**Reconciliation** A procedure that compares replicas that are intended to be identical and repairs any differences. [Ch. 10]

**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. [Ch. 2]

**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*.) [Ch. 9]

**Redundancy** Extra information added to detect or correct errors in data or control signals. [Ch. 8]

**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*. [Ch. 2]

**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). [Ch. 6]

**Relative path name** A path name that the name resolver resolves in a default context provided by the environment. [Ch. 2]

**Reliability** A statistical measure, the probability that a system is still operating at time $t$, given that it was operating at some earlier time $t_0$. [Ch. 8]

**Reliable delivery** A transport protocol assurance: it provides both at-least-once delivery and data integrity. [Ch. 7]

**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. [Ch. 4]

**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. [Ch. 8]

**Repertoire** The set of operations or actions an interpreter is prepared to perform. The repertoire of a general-purpose processor is its instruction set. [Ch. 2]

**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. [Ch. 8]

**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. [Ch. 10]

**Replication** The technique of using multiple replicas to achieve fault tolerance. [Ch. 8]

**Repudiate** To disown an apparently authenticated message. [Ch. 11]

**Request** The message sent from a client to a service. [Ch. 4]

**Resolve** To perform a name-mapping algorithm from a name to the corresponding value. [Ch. 2]

**Response** The message sent from a service to a client in response to a previous request. [Ch. 4]

**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*. [Ch. 9]

**Rollback recovery** Also known as *undo logging*. 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. Compare with *roll-forward recovery*. [Ch. 9]

**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. [Ch. 2]

**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. [Ch. 6]

**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. [Ch. 7]

**Router** A packet forwarder that also participates in a routing algorithm. [Ch. 7]

**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. [Ch. 7]

**Scheduler** The part of the thread manager that implements the policy for deciding which thread to run. Policies can be preemptive or non-preemptive. [Ch. 5]

**Scope** In a layered naming scheme, the set of contexts in which a particular name is bound to the same value. [Ch. 2]

**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. [Ch. 2]

**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. [Ch. 2]

**Secondary device** In a multilevel memory system, the memory device that is larger but also usually slower. Compare with *primary device*. [Ch. 6]

**Secrecy** Synonym for *confidentiality*. [Ch. 11]

**Secure area** A physical space or a virtual address space in which confidential information can be safely confined. [Ch. 11]

**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. [Ch. 11]

**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. [Ch. 11]

**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. [Ch. 11]

**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. [Ch. 5]
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*. [Ch. 7]

**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. [Ch. 7]

**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. [Ch. 5]

**Sequence coordination** A coordination constraint among threads: for correctness, a certain event in one thread must precede some other certain event in another thread. [Ch. 5]

**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. [Ch. 5]

**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. [Ch. 7]

**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. [Ch. 9]

**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. [Ch. 4]

**Service** A module that responds to actions initiated by clients. [Ch. 4] At the end-to-end layer of a network, the end that responds to actions initiated by the other end. Compare with *client*. [Ch. 7]

**Set up** The steps required to allocate storage space for and initialize the state of a connection. [Ch. 7]

**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. [Ch. 9]

**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. [Ch. 11]

**Shared-secret key** The key used by a shared-secret cryptography system. [Ch. 11]

**Sharing** Allowing an object to be used by more than one other object without requiring multiple copies of the first object. [Ch. 2]

**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". [Ch. 11]

**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*. [Ch. 9]

**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. [Ch. 9]

**Simplex** Describes a link between two stations that can be used in only one direction. Compare with *duplex*, *half-duplex*, and *full-duplex*. [Ch. 7]

**Single-acquire protocol** A simple protocol for locking: a thread can acquire a lock only if some other thread has not already acquired it. [Ch. 5]

**Single-event upset** A synonym for *transient fault*. [Ch. 8]

**Slave** In a multiple-site replication scheme, a site that takes update requests from only the master site. Compare with *master.* [Ch. 10]

**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. [Ch. 7]

**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. [Ch. 10]

**Soft modularity** Modularity defined by convention but not enforced by physical constraints. Compare with *enforced modularity*. [Ch. 4]

**Soft real-time scheduler** A real-time scheduler in which missing a deadline occasionally is acceptable. [Ch. 6]

**Soft state** State of a running program that the program can easily reconstruct if it becomes necessary to abruptly terminate and restart the program. [Ch. 8]

**Source** The network attachment point that originated the payload of a packet. Sometimes used as shorthand for *source address*. [Ch. 7]

**Source address** An identifier of the source of a packet, usually carried as a field in the header of the packet. [Ch. 7]

**Spatial locality** A kind of locality of reference in which the reference string contains clusters of references to adjacent or nearby addresses. [Ch. 6]

**Speaks for** A phrase used to express delegation relationships between principals. "A speaks for B" means that B has delegated some authority to A. [Ch. 11]

**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. [Ch. 6]

**Spin loop** A situation in which a thread waits for an event to happen without releasing the processor. [Ch. 5]

**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 2.1. [Ch. 2]

**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. [Ch. 2]

**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. [Ch. 6]

**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. [Ch. 6]

**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*. [Ch. 7]

**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*. [Ch. 2]

**Station** A device that can send or receive data over a communication link. [Ch. 7]

**Stop and wait** A synonym for *lock step.* [Ch. 7]

**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. [Ch. 2]

**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. [Ch. 7]

**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. [Ch. 7]

**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*. [Ch. 10]

**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. [Ch. 4]

**Supermodule** A set of replicated modules interconnected in such a way that it acts like a single module. [Ch. 8]

**Supervisor call instruction (SVC)** A processor instruction issued by user modules to pass control of the processor to the kernel. [Ch. 5]

**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. [Ch. 6]

**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. [Ch. 2]

**System** A set of interconnected components that has an expected behavior observed at the interface with its environment. Contrast with *environment*. [Ch. 1]

**Tail drop** A strategy for managing an overloaded resource: the system refuses service to the queue entry that arrived most recently. [Ch. 7]

**Tear down** The steps required to reset the state of a connection and deallocate the space that was used for storage of that state. [Ch. 7]

**Temporal locality** A kind of locality of reference in which the reference string contains closely spaced references to the same address. [Ch. 6]

**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. [Ch. 6]

**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. [Ch. 5]

**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. [Ch. 5]

**Threat** A potential security violation from either a planned attack by an adversary or an unintended mistake by a legitimate user. [Ch. 11]

**Throughput** A measure of the rate of useful work done by a service for a given workload. [Ch. 6]

**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. [Ch. 11]

**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.* [Ch. 8]

**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. [Ch. 7]

**Trailer** Information that a protocol layer adds to the end of a packet. [Ch. 7]

**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. [Ch. 9]

**Transactional memory** A memory model in which multiple references to primary memory are both all-or-nothing and before-or-after. [Ch. 9]

**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 *single-event upset*. Compare with *persistent fault* and *intermittent fault*. [Ch. 8]

**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*. [Ch. 7]

**Transmission delay** In a communication network, the component of overall delay contributed by the time spent sending a frame at the available data rate. [Ch. 7]

**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. [Ch. 7]

**Triple-modular redundancy (TMR)** *N*-modular redundancy with $N = 3$. [Ch. 8]

**Trusted computing base (TCB)** That part of a system that must work properly to make the overall system secure. [Ch. 11]

**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. [Ch. 4]

**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. [Ch. 9]

**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*. [Ch. 9]

**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*. [Ch. 9]

**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*. [Ch. 9]

**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. [Ch. 2]

**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. [Ch. 2]

**Universe of values** The set of all possible values that can be named by a particular naming scheme. [Ch. 2]

**Unlimited name space** A name space in which names never have to be reused. [Ch. 3]

**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.* [Ch. 8]

**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. [Ch. 2]

**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*. [Ch. 5]

**Utilization** The percentage of capacity used for a given workload. [Ch. 6]

**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. [Ch. 2]

**Valid construction** The term used by software designers for *fault avoidance*. [Ch. 8]

**Version history** The set of all values for an object or variable that have ever existed, stored in journal storage. [Ch. 9]

**Virtual address** An address that must be translated to a physical address before using it to refer to memory. Compare with *physical address*. [Ch. 5]

**Virtual circuit** A connection intended to carry a stream through a forwarding network, in some ways simulating an electrical circuit. [Ch. 7]

**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. [Ch. 5]

**Virtual machine monitor** The software that implements virtual machines. [Ch. 5]

**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. [Ch. 5]

**Virtual memory manager** A memory manager that implements virtual addresses, resolving them to physical addresses by using, for example, a page map. [Ch. 5]

**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*. [Ch. 2]

**Voter** A device used in some NMR designs to compare the output of several nominally identical replicas that all have the same input. [Ch. 8]

**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. [Ch. 2]

**Window** In flow control, the quantity of data that the receiving side of a transport protocol is prepared to accept from the sending side. [Ch. 7]

**Witness** A (usually cryptographically strong) hash value that attests to the content of a file. Another widely used term for this concept is *fingerprint*. [Ch. 10]

**Working directory** In a file system, a directory used as a default context, for resolution of relative path names. [Ch. 2]

**Working set** The set of all addresses to which a thread refers in the interval $\Delta t$. If the application exhibits locality of reference, this set of addresses will be small compared to the maximum number of possible addresses during $\Delta t$. [Ch. 6]

**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. [Ch. 9]

**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*.) [Ch. 6]

# Index of Concepts

Design principles and hints appear in *underlined italics*. Procedure names appear in SMALL CAPS. Page numbers in **bold face** are in the Glossary. Page numbers that are greyed out are in a section that is [on-line].


## A

abort, **475**, Ch. 9
absolute path name, 68, 72, **475**
abstraction, 22, **475**
  leaky, 30
accelerated aging, Ch. 8
access control list, **475**, Ch. 11
access time, 48
ACK (see acknowledgment)
acknowledgment, **475**, Ch. 7
ACL (see access control list)
ACQUIRE, 225, Ch. 9
action, 53, **475**, Ch. 9
action graph, Probsets
active fault, **475**, Ch. 8
*ad hoc* wireless network, 425, Probsets
adaptive
  routing, **475**, Ch. 7
  timer, Ch. 7
additive increase, Ch. 7
address
  destination, **482**
  in naming, 51, 122, **475**
  in networks, **495**, Ch. 7
  resolution protocol, **475**, Ch. 7
  source, **507**
  space, 51, **475**
  virtual, 206, 243, **511**
*adopt sweeping simplifications*, 40, 149, 160, Ch. 7, Ch. 8, Ch. 9, Ch. 10, Ch. 11
ADVANCE, 276
Advanced Encryption Standard (AES), Ch. 11
adversary, **476**, Ch. 11
advertise, 76, **476**, Ch. 7
alias, 72, **476**
  (see also indirect name)
alibi, 228
all-or-nothing atomicity, 89, **476**, Ch. 9
any-to-any connection, **476**, Ch. 7
application protocol, Ch. 7
arbiter failure, 229
archive, **476**, Ch. 9
  log, Ch. 9
ARP (see address resolution protocol)
assembly, 9
associative memory, 51
asynchronous, 55, 309, **476**, Ch. 7
at-least-once
  protocol assurance, **476**, Ch. 7
  RPC, 170
at-most-once
  protocol assurance, **476**, Ch. 7
  RPC, 170
atomic, **476**
  action, 89, 220, **476**, Ch. 9
  storage, 89, **476**, Ch.9
atomicity, **476**, Ch. 9
  all-or-nothing, 89, **476**, Ch.9
  before-or-after, 46, 89, **477**, Ch. 9
  log, Ch. 9
attachment point (see network attachment point)
authentication, **477**, Ch. 11
  key, Ch. 11
  logic, Ch. 11
  origin, **496**, Ch. 11
  tag, **477**, Ch. 11
authoritative name server, 179
authorization, **477**, Ch. 11
  matrix, Ch. 11
automatic rate adaptation, **477**, Ch. 7
availability, **477**, Ch. 8
*avoid excessive generality*, 16
*avoid rarely used components*, Ch. 8, Ch. 11
AWAIT, 276

## B

backoff
  exponential, **486**, Ch. 7
  exponential random, **486**, Ch. 9
  random, 227

backup copy, **477**, Ch. 10
backward error correction, **477**, Ch. 8
bad-news diode, 38, **477**
bandwidth, **477**, Ch. 7
bang-bang protocol, Ch. 7
base name, 67
batch, 314, **477**
bathtub curve, Ch. 8
*be explicit*, Ch. 8, Ch. 11
before-or-after atomicity, 46, 89, **477**, Ch. 9
Belady's anomaly, 337
best effort, **478**, Ch. 7
  contract, Ch. 7
big-endian numbering, 158
BIND, 63
binding, 27, 61, **478**
  stable, **507**
  user-dependent, 74, **511**
bit error rate, **478**, Ch. 7
bit stuffing, **478**, Ch. 7
blast protocol, Ch. 7
blind write, **478**, Ch. 9
block, 245
  cipher, Ch. 11
  in UNIX, 93
blocking read, Ch. 9
bootstrapping, 223, **478**, Ch. 9
bot, Ch. 11
bottleneck, 300, **478**
  data rate, Ch. 7
bounded buffer, 206
broadcast, 77, **478**, Ch. 7
buffer overrun attack, Ch. 11
burn in, burn out, Ch. 8
burst, **478**, Ch. 7
bus, 80
  address, 81
  arbitration, 81
Byzantine fault, **478**, Ch. 8

## C

CA (see certificate authority)
cache, 51, 332, **478**
  coherence, **479**, Ch. 10
  snoopy, **507**, Ch. 10
capability, **479**, Ch. 11
capacity, 302, 322, **479**
careful storage, Ch. 8
carrier sense multiple access, **485**, Ch. 7
cascading change propagation, Ch. 11
case-
  coercing, 128
  preserving, 128
  sensitive, 128
CBC (see cipher-block chaining)
cell, 46
  storage, **479**, Ch. 9
certificate, **479**, Ch. 11
  authority, **479**, Ch. 11
  self-signed, Ch. 11
certify, **479**, Ch. 11
checkpoint, **479**, Ch. 9
checksum, **479**, Ch. 7
cipher, **479**, Ch. 11
cipher-block chaining, Ch. 11
ciphertext, **479**, Ch. 11
circuit
  switch, **479**, Ch. 7
  virtual, **511**, Ch. 7
cleartext, **479**, Ch. 11
client, 155, **479**, Ch. 7
client/service organization, 159, **479**
clock algorithm, 344
CLOSE, 88
close-to-open consistency, 192, **479**
closure, 68, **480**
coding, Ch. 8
coherence
  cache, **479**, Ch. 10
  read/write, 46, **502**
collision
  Ethernet, **485**, Ch. 7
  hash, Ch. 11
  name, 124, **480**
commit, **480**, Ch. 9
  two-phase, **510**, Ch. 9
communication link, 59, **480**
commutative cryptographic transformation, Ch. 11
COMPARE, 75
compartment, Ch. 11
compensation, **480**, Ch. 10
*complete mediation*, Ch. 11
complexity, 10, **480**
  Kolmogorov, 11
component, 8

computationally secure, Ch. 11
condition variable, 276, 463
conditional failure rate function, Ch. 8
confidentiality, **480**, Ch. 11
confinement, **480**, Ch. 11
conflict, Ch. 10
confusion matrix, 372
congestion, **480**, Ch. 7
  collapse, **480**, Ch. 7
connection, **480**, Ch. 7
connectionless, **481**, Ch. 7
consensus, **481**, Ch. 10
  the consensus problem, Ch. 10
consistency, **481**
  close-to-open, 192, **479**
  eventual, Ch. 10
  external time, Ch. 9
  sequential, Ch. 9
  strict, **508**, Ch. 10
  strong (see consistency, strict)
consistent hashing, Probsets
constituent, 9
constraint, **481**, Ch. 10
context, 62, **481**
context reference, 63, 66, **481**
continuous operation, **481**, Ch. 8
control point, **481**, Ch. 7
convergent encryption, Probsets
cookie, Ch. 11
cooperative multitasking, 269
cooperative scheduling, 269, **481**
copy-on-write, 326
covert channel, **481**, Ch. 11
critical section, 220
cross-layer cooperation, Ch. 7
cryptographic
  hash function, **481**, Ch. 11
  key, **481**, Ch. 11
  transformation, **481**, Ch. 11
  transformation, commutative, Ch. 11
cryptography, **482**, Ch. 11
  public key, **501**, Ch. 11
  shared-secret, **506**, Ch. 11
CSMA/CD (see carrier sense multiple access)
cursor, 88
cursor stability, Ch. 10
cut-through, **482**, Ch. 7

## D

dally, 314
dangling reference, 130, **482**
data integrity
  in communications, **482**, Ch. 7
  in security assurance, **482**, Ch. 11
  in storage, Ch. 10
data rate, **482**, Ch. 7
datagram, Ch. 7
deadlock, 221, **482**, Ch. 9
decay, 46, **482**, Ch. 8
  factor, Ch. 7
  set, **482**, Ch. 8
declassify, Ch. 11
*decouple modules with indirection*, 27, 106, 123, 173, 243, 286, 325, Ch. 7
decrypt, **482**, Ch. 7, Ch. 11
DECRYPT, Ch. 11
default context reference, 66, **482**
defense in depth, Ch. 8, Ch. 11
delay, Ch. 7
  processing, **500**, Ch. 7
  propagation, **500**, Ch. 7
  queuing, **501**, Ch. 7
  transmission, **510**, Ch. 7
delayed authentication, Ch. 11
delegation forwarding, 112
demand
  algorithm, 339
  paging, 346, **482**
dependent outcome record, Ch. 9
*design for iteration*, 37, 228, Ch. 8, Ch. 11
*design principles*, 40, Inside front cover
  *adopt sweeping simplifications*, 40, 149, 160, Ch. 7, Ch. 8, Ch. 9, Ch. 10, Ch. 11
  *avoid excessive generality*, 16
  *avoid rarely used components*, Ch. 8, Ch. 11
  *be explicit*, Ch. 8, Ch. 11
  *complete mediation*, Ch. 11
  *decouple modules with indirection*, 27, 106, 123, 173, 243, 286, 325, Ch. 7
  *design for iteration*, 37, 228, Ch. 8, Ch. 11
  *durability mantra*, Ch. 10
  *economy of mechanism*, Ch. 11
  *end-to-end argument*, Ch. 7, Ch. 8, Ch. 9, Ch. 10, Ch. 11
  *escalating complexity principle*, 14

*design principles*, (*cont'd*)
*fail-safe defaults*, Ch. 11
*golden rule of atomicity*, Ch. 9
*incommensurate scaling rule*, 33, 316, Ch. 7
*keep digging principle*, 37, Ch. 8, Ch. 11
*law of diminishing returns*, 18, 305, Ch. 9
*least privilege principle*, Ch. 11
*minimize common mechanism*, Ch. 11
*minimize secrets*, Ch. 11
*one-writer principle*, 212
*open design principle*, Ch. 11
*principle of least astonishment*, 85, 89, 128, 205, Ch. 11
*robustness principle*, 29, Ch. 8
*safety margin principle*, 24, Ch. 8
*unyielding foundations rule*, 20, 38, 288
destination, **482**, Ch. 7
address, **482**
detectable error, **482**, Ch. 8
dictionary attack, Ch. 11
digital signature, **483**, Ch. 11
dilemma of the two generals, **510**, Ch. 9
*diminishing returns, law of*, 18, 305, Ch. 9
direct
mapping, 346
memory access, 83
directory, 65, **483**
in UNIX, 97
discipline
simple locking, **506**, Ch. 9
system-wide locking, Ch. 9
two-phase locking, **510**, Ch. 9
discovery
of maximum transmission unit, **494**, Ch. 7
of names, 76
discretionary access control, **483**, Ch. 11
dispatcher, 262
distance vector, Ch. 7
divide-by-zero exception, 206
DMA (see direct memory access)
do action (see redo action)
domain
name, 175
virtual memory, 230, **483**
Domain Name System
design of, 175
eventual consistency in, Ch. 10
fault tolerance of, Ch. 8
down time, **483**, Ch. 8
dry run, Ch. 9
duplex, **483**, Ch. 7
duplicate suppression, **483**, Ch. 7
durability, 46, **483**, Ch. 8
log, Ch. 9
*durability mantra*, Ch. 10
durable storage, **483**, Ch. 8
dynamic scope, 68, **483**

## E

earliest deadline first scheduling policy, 360, **484**
early drop, **484**, Ch. 7
echo request, Ch. 7
*economy of mechanism*, Ch. 11
element, 9
elevator algorithm, 361
emergent property, 4, **484**
emulation, 208, **484**
encrypt, **484**, Ch. 7, Ch. 11
ENCRYPT, Ch. 11
encryption key, Ch. 11
end-to-end, **484**
layer, **484**, Ch. 7
*end-to-end argument*, Ch. 7, Ch. 8, Ch. 9, Ch. 10, Ch. 11
enforced modularity, 153, **484**
ENUMERATE, 63
enumerate (in naming), 63, **484**
environment, **484**
of a system, 8
of an interpreter, 53
reference, 53
erasure, **484**, Ch. 8
ergodic, **484**, Ch. 8
error, **484**, Ch. 8
containment, **485**, Ch. 8
correction, **485**, Ch. 7, Ch. 8
detection, **485**, Ch. 7, Ch. 8
*escalating complexity principle*, 14
Ethernet, **485**, Ch. 7
event variable, 460
eventcount, 276, **485**
eventual consistency, **485**, Ch. 10
EWMA (see exponentially weighted moving average)
exactly-once
protocol assurance, **485**, Ch. 7

RPC, 171
exception, 57, 205, 235, **485**
divide-by-zero, 205
illegal instruction, 235
illegal memory reference, 233
indirect, 325
memory reference, 231
missing-page, 328, **494**
permission error, 233
TLB miss, 253
explicit context reference, 66, **485**
explicitness, **485**, Ch. 11
*exploit brute force*, 301
exponential
backoff, **486**, Ch. 7
random backoff, **486**, Ch. 9
exponentially weighted moving average, 355, Ch. 7
export, 60, **486**
external time consistency, Ch. 9


## F


fail-
fast, **486**, Ch. 8
safe, **486**, Ch. 8
secure, **486**, Ch. 8
soft, **486**, Ch. 8
stop, **486**, Ch. 8
vote, **486**, Ch. 8
*fail-safe defaults*, Ch. 11
failure, **486**, Ch. 8
tolerance, **486**, Ch. 8
false positive/negative, 371
fast start, Ch. 7
fate sharing, 153
fault, **486**, Ch. 8
avoidance, **486**, Ch. 8
tolerance, **487**, Ch. 8
tolerance design process, Ch. 8
tolerance model, Ch. 8
FCFS (see first-come, first-served)
FIFO (see first-in, first-out)
file, 87, **487**
in UNIX, 95
memory-mapped, 325
pointer, 88
fingerprint, **487**, Ch. 7
first-come, first-served scheduling policy, 353, **487**
first-in, first-out page removal policy, 336, **487**
fixed
timer, Ch. 7
window, Ch. 7
flooding, 441, Probsets
flow control, **487**, Ch. 7
follow-me forwarding, 112
force, 320, **487**, Ch. 9
forward
error correction, **487**, Ch. 8
secrecy, **487**, Ch. 11
forwarder, Ch. 7
forwarding table, **487**, Ch. 7
fragile name, 121
fragment, **487**
frame, **487**, Ch. 7
freshness, **487**, Ch. 11
full-duplex, **488**, Ch. 7


## G


garbage collection, 131
gate (protected entry), 236, **488**
generality, 15
generated name, 124, **488**
GET, 50
global name, 75, **488**
*golden rule of atomicity*, Ch. 9
granularity, 8, Ch. 9
guaranteed delivery, Ch. 7


## H


half-duplex, **488**, Ch. 7
Hamming distance, **488**, Ch. 8
hard-edged, Ch. 7
hard error, Ch. 8
hard link, 105
hard real-time scheduling policy, 359, **488**
hash function, 125, **488**
hashed MAC, Ch. 11
hazard function, Ch. 8
header, **488**, Ch. 7
heartbeat, Ch. 8
hierarchy, 25, **488**
in naming, 73
in routing, **488**, Ch. 7
high-water mark, Ch. 9

*hints*, 40
*exploit brute force*, 301
*instead of reducing latency, hide it*, 309
*optimize for the common case*, 307, 334, Ch. 9
*separate mechanism from policy*, 331, 349, Ch. 11
hit ratio, 333
HMAC (see hashed MAC)
hop limit, **488**, Ch. 7
hot swap, **488**, Ch. 8
hyperlink, 133

## I

I/O bottleneck, 316
ICMP (see Internet control message protocol)
idempotent, 170, **488**, Ch. 7, Ch. 9
identifier, 127, **489**
illegal instruction, **489**
exception, 235
illegal memory reference exception, 233
IMS (see Information Management System)
in-memory database, Ch. 9
incommensurate scaling, 5, **489**
*incommensurate scaling rule*, 33, 316, Ch. 7
incremental
backup, **489**, Ch. 10
redundancy, Ch. 8
indirect
block, 95
name, 73, 104, **489**
indirection, 27, 61, **489**
exception, 325
infant mortality, Ch. 8
information flow control, Ch. 11
Information Management System, Ch. 9
inode, 95
install, **489**, Ch. 9
*instead of reducing latency, hide it*, 309
instruction
reference, 53, **489**
repertoire, **503**
integrity (see data integrity)
intended load, **489**, Ch. 7
interconnection, 8
interface, 8
interleaving, 310, **489**
intermittent fault, **489**, Ch. 8
International Organization for Standardization, **489**, Ch. 7
Internet, Ch. 7
control message protocol, Ch. 7
protocol, Ch. 7
service provider, 139
interpreter, 53, **490**
interrupt, 53, 235, 283, **490**
invalidate, **490**, Ch. 10
invisible hand, Ch. 7
IP (see Internet protocol)
ISO (see International Organization for Standardization)
isochronous, **490**, Ch. 7
isolation, 220
ISP (see Internet service provider)
iteration, 36

## J

jitter **490**, Ch. 7
job, 352, **490**
journal storage, **490**, Ch. 9

## K

KDC (see key distribution center)
*keep digging principle*, 37, Ch. 8, Ch. 11
kernel, 238, **490**
mode, 234, **490**
key (see cryptographic key)
key distribution center, **490**, Ch. 11
key-based cryptographic transformation, **490**, Ch. 11
Kolmogorov complexity, 11

## L

latency, 49, 302, **490**, Ch. 8
latent fault, **491**, Ch. 8
*law of diminishing returns*, 18, 305, Ch. 9
layer
bypass, 79
end-to-end, **484**, Ch. 7
link, **491**, Ch. 7
network, **496**, Ch. 7
layering, 24, **491**
leaky abstraction, 30
*least astonishment principle*, 85, 89, 128, 205, Ch. 11
*least privilege principle*, Ch. 11

least-recently-used page removal policy, 338, **491**
least significant component, 71
lexical scope (see static scope)
lightweight remote procedure call, 238, 442
limited change propagation, Ch. 11
limited name space, 129, **491**
link
  in communications, 59, **480**
  in naming, 73, **491**
  in UNIX, 99
  layer, **491**, Ch. 7
  soft (see indirect name)
  symbolic (see indirect name)
list system, **491**, Ch. 11
little-endian numbering, 158
livelock, 222, **491**, Ch. 9
locality of reference, 334, **491**
  spatial, 334, **507**
  temporal, 334, **509**
location-addressed memory, 51
lock, 218, **491**, Ch. 9
  compatibility mode, Ch. 9
  manager, Ch. 9
  point, **491**, Ch. 9
  set, **491**, Ch. 9
lock-step protocol, **491**, Ch. 7
locking discipline
  simple, **506**, Ch. 9
  system-wide, Ch. 9
  two-phase, **510**, Ch. 9
log, **491**, Ch. 9
  archive, Ch. 9
  atomicity, Ch. 9
  durability, Ch. 9
  performance, Ch. 9
  record, Ch. 9
  redo, Ch. 9
  sequence number, Ch. 9
  undo, Ch. 9
  write-ahead, **512**, Ch. 9
logical
  copy, **492**, Ch. 10
  locking, **492**, Ch. 9
lost object, 130
LRPC (see lightweight remote procedure call)
LRU (see least-recently used)

## M

MAC
  (see media access control address)
  (see message authentication code)
magnetic disk memory, 49
malware, Ch. 11
Manchester code, **492**, Ch. 7
margin, **492**, Ch. 8
mark point, **492**, Ch. 9
marshal/unmarshal, 157, **492**
maskable error, **492**, Ch. 8
masking, **492**, Ch. 8
massive redundancy, Ch. 8
master, **492**, Ch. 10
maximum transmission unit, **492**, Ch. 7
mean time
  between failures, **492**, Ch. 8
  to failure, **492**, Ch. 8
  to repair, **492**, Ch. 8
media access control address, 126
mediation, **492**, Ch. 11
memory, 45
  associative, 51
  barrier, 47
  location-addressed, 51
  manager, 230, **493**
  manager, multilevel, 325
  manager, virtual, 206, 243, **512**
  -mapped file, 325
  -mapped I/O, 84, **493**
  random access, 50, **501**
  transactional, **510**, Ch. 9
  volatile/non-volatile, 45, **496**, **512**
memory reference exception, 231
memoryless, **493**, Ch. 8
message, 59, **493**, Ch. 7
  authentication, **493**, Ch. 11
  authentication code, **493**, Ch. 11
  representation, 54
message-sending protocol, Ch. 7
message timing diagram, 155
metadata, 91, 120, **493**
microkernel, 240, **493**
*minimize common mechanism*, Ch. 11
*minimize secrets*, Ch. 11
mirror, **493**, Ch. 10
missing-page exception, 328, **494**

mobile host, Ch. 7
modular sharing, 116, **493**
modularity, 19
  enforced, 153, **484**
  soft, 153, **507**
module, 9, **493**, Ch. 8
monolithic kernel, 238, **494**
most-recently-used page removal policy, 340, **494**
most significant component, 72
MRU (see most-recently-used)
MTBF (see mean time between failures)
MTTF (see mean time to failure)
MTTR (see mean time to repair)
MTU (see maximum transmission unit)
MTU discovery, **494**, Ch. 7
multihomed, **494**, Ch. 7
multilevel
  memory, 324, **494**
  memory manager, 325
multiple
  lookup, 73, **494**
  -reader, single-writer protocol, Ch. 9
  register set processor, 447
multiplexing, **494**, Ch. 7
multiplicative decrease, Ch. 7
multipoint, **494**, Ch. 7
multiprogramming, 256
multitasking, 256
Murphy's law, 86
mutual exclusion, 220

## N

N + 1 redundancy, **494**, Ch. 8
N-modular redundancy, **494**, Ch. 8
N-version programming, **494**, Ch. 8
NAK (see negative acknowledgment)
name, 44, **494**
  base, 67
  collision, 124
  conflict, 116, **494**
  discovery, 76
  fragile, 121
  generated, 124, **488**
  global, 75, **488**
  indirect, 73, 104, **489**
  lookup, multiple, 73, **494**
  opaque, 121, **496**
  overloaded, 120, **497**
  path, **498**
  pure, 120, **501**
  qualified, 67, **501**
  resolution, 62
  resolution, recursive, 71, **502**
  well-known, 77, **512**
name-mapping algorithm, 62
name space, 61, **495**
  limited, 129, **491**
  unique identifier, 64, **511**
  universal, 62, **511**
  unlimited, 129, **511**
name-to-key binding, **495**, Ch. 11
namespace (see name space)
naming
  authority, 180
  hierarchy, 73, **495**
  network, 72, **495**
  scheme, 61, **495**
NAT (see network address translation)
negative acknowledgment, **495**, Ch. 7
nested outcome record, Ch. 9
network, **495**, Ch. 7
  address, **495**, Ch. 7
  address translation, Ch. 7
  attachment point, 65, **495**, Ch. 7
  layer, **495**, Ch. 7
  services access point, **496**
Network File System, 184
NFS (see Network File System)
NMR (see N-modular redundancy)
non-blocking read, Ch. 9
non-discretionary access control, **495**, Ch. 11
non-preemptive scheduling, 269, **495**
non-volatile memory, 45, **495**
nonce, **496**, Ch. 7
not-found result, 64
NSAP (see network services access point)

## O

object, 9, 60, **496**
object-based virtual memory, 465
occasionally connected, Ch. 10
offered load, 311, **496**, Ch. 7
on-demand zero-filled page, 326
one-time pad, Ch. 11
*one-writer principle*, 212
opaque name, 121, **496**

OPEN, 88
*open design principle*, Ch. 11
operating system, 78, 79, **496**
OPT (see optimal page-removal policy)
optimal page removal policy, 337, **496**
optimistic concurrency control, **496**, Ch. 9
*optimize for the common case*, 307, 334, Ch. 9
origin authenticity, **496**, Ch. 11
orphan, 130
OSI (see International Organization for Standardization)
outcome record, Ch. 9
overhead, 302
overlay network, 425, Ch. 7, Probsets
overload, 311, **496**
overloaded name, 120, **496**
overprovisioning, Ch. 7

## P

pacing, Ch. 7
packet, **496**, Ch. 7
  forwarding, **496**, Ch. 7
  forwarding network, Ch. 7
  switch, **496**, Ch. 7
page, 245, **496**
  fault (see missing-page exception)
  map, 245, **496**
  on-demand zero-filled, 326
  table, 246, **496**
page-map address register, 247, **496**
page-removal policy, 329, **496**
  clock algorithm, 344
  direct mapping, 346
  first-in, first-out, 336, **487**
  least-recently used, 338, **491**
  most-recently used, 340, **494**
  optimal, 337, **496**
  random, 345
pair-and-compare, **497**, Ch. 8
pair-and-spare, **497**
parallel transmission, **497**, Ch. 7
partition, **497**, Ch. 8, Ch. 10
password, **497**, Ch. 11
patch, 17
path, Ch. 7
  name, 75, **497**
  name, absolute 68, 72, **475**
  name, relative, 72, **503**
  search, 73, 75, **504**
  selection, **497**, Ch. 7
  vector, Ch. 7
payload, **497**, Ch. 7
peer-to-peer
  design, 164
  network, 425
pending, **498**, Ch. 9
performance log, Ch. 9
permission error exception, 233
persistent, 46, **498**
  fault, **498**, Ch. 8
  sender, **498**, Ch. 7
pessimistic concurrency control, **498**, Ch. 9
PGP (see protocol, pretty good privacy)
phase encoding, **498**, Ch. 7
phase-locked loop, Ch. 7
physical
  address, 243, **498**
  copy, **498**, Ch. 10
  locking, **498**, Ch. 9
piggybacking, **498**, Ch. 7
pipeline, **498**, Ch. 7
PKI (see public key infrastructure)
plaintext, **498**, Ch. 11
point-to-point, **499**, Ch. 7
polling, 273, **499**
port, **499**, Ch. 7
precision (in information retrieval), 373
preemptive scheduling, 269, **499**
prepaging, 346
PREPARED
  message, Ch. 9
  state, **499**
presentation
  protocol, **499**, Ch. 7
  service, Ch. 7
presented load (see offered load)
preservation, Ch. 8
presumed commit, Ch. 9
preventive maintenance, **499**, Ch. 8
pricing, Ch. 7
primary
  copy, **499**, Ch. 10
  device, 331, **499**
principal, **499**, Ch. 11
*principle of escalating complexity*, 14

*principle of least astonishment*, 85, 89, 128, 205, Ch. 11
principles (see *design principles*)
priority
  inversion, 358
  scheduling policy, 357, **499**
privacy, **499**, Ch. 11
private key, **499**, Ch. 11
probe, Ch. 7
procedure calling convention, 150
process, 97, 248
processing delay, **500**, Ch. 7
processor multiplexing, 256
producer and consumer problem, 211
program counter, 56, **500**
progress, **500**, Ch. 9
propagation delay, **500**, Ch. 7
propagation of effects, 4, **500**
protection, **500**, Ch. 11
  group, **500**, Ch. 11
protocol, **500**, Ch. 7
  address resolution, **475**, Ch. 7
  application, Ch. 7
  bang-bang, Ch. 7
  blast, Ch. 7
  bus arbitration, 81
  carrier sense multiple access, **485**, Ch. 7
  challenge-response, Ch. 11
  Diffie-Hellman key agreement, Ch. 11
  Internet, Ch. 7
  internet control message, Ch. 7
  Kerberos, Ch. 11
  lock-step, **491**, Ch. 7
  message-sending, Ch. 7
  multiplexing, Ch. 7
  Network File System, 184
  presentation, **499**, Ch. 7
  pretty good privacy, Ch. 11
  ready/acknowledge, **501**, Ch. 7
  real-time transport, Ch. 7
  reliable message stream, Ch. 7
  request/response, Ch. 7
  routing, Ch. 7
  secure shell, Ch. 11
  secure socket layer, Ch. 11
  security, **504**, Ch. 11
  simple network time service, Ch. 7
  stream transport, Ch. 7
  transmission control, Ch. 7
  transport, **509**, Ch. 7
  transport layer security, Ch. 11
  two-phase commit, **510**, Ch. 9
  user datagram, Ch. 7
proxy, 7, 371
pseudocode representation, 54
pseudorandom number generator, Ch. 11
public key, **500**, Ch. 11
  cryptography, **500**, Ch. 11
  infrastructure, Ch. 11
publish/subscribe, 173, **500**
pull, 172
pure name, 120, **500**
purging, **500**, Ch. 8
push, 172
PUT, 50

## Q

quad component, Ch. 8
qualified name, 67, **501**
quantum, 356
quench, **501**, Ch. 7
query, 77
queuing delay, **501**, Ch. 7
quorum, **501**, Ch. 10
quota, 313

## R

race condition, 215, **501**
RAID, 52, **501**
  RAID 1, Ch. 8
  RAID 4, Ch. 8
  RAID 5, Ch. 8
RAM (see random access memory)
random
  access memory, 50, **501**
  backoff, 227
  backoff, exponential, **486**, Ch. 9
  drop **501**, Ch. 7
  early detection, **501**, Ch. 7
  number generator, Ch. 11
  page-removal policy, 345
  pseudorandom number generator, Ch. 11
rate monotonic scheduling policy, 360, **501**
raw storage, Ch. 8
RC4 cipher, Ch. 11

READ, 45
read and set memory, 224, **501**
read-capture, Ch. 9
read/write coherence, 46, **501**
ready/acknowledge protocol, **501**, Ch. 7
real time, 359, **502**, Ch. 7
real-time
  scheduling policy, 359, **502**
  scheduling policy, hard, 359, **488**
  scheduling policy, soft, 359, **506**
  transport protocol, Ch. 7
reassembly, **502**, Ch. 7
recall (in information retrieval), 373
RECEIVE, 59
receive livelock, 350
reconciliation, **502**, Ch. 10
recovery, Ch. 8
recursive
  name resolution, 71, **502**
  replication, Ch. 8
RED (see random early detection)
redo
  action, **502**, Ch. 9
  log, Ch. 9
reduced instruction set computer, 55
redundancy, **502**, Ch. 8
redundant array of independent disks (see RAID)
reference, 60, **502**
  count, 131
  monitor, Ch. 11
  string, 334, **502**
register renaming, Ch. 9
relative path name, 72, **502**
RELEASE, 225, Ch. 9
reliability, **502**, Ch. 8
reliable
  delivery, **502**, Ch. 7
  message stream protocol, Ch. 7
remote procedure call, 167, **502**
reorder buffer, Ch. 9
repair, **503**, Ch. 8
repertoire, 53, **503**
replica, **503**, Ch. 8
replicated state machine, **503**, Ch. 10
replication, **503**
  recursive, Ch. 8
reply, 155
representations
  bit order numbering, 158
  confusion matrix, 372
  message, 54
  pseudocode, 54
  timing diagram, 155
  Venn diagram, 372
  version history, Ch. 9
  wait-for graph, 221
repudiate, **503**
request, 155, **503**
request/response protocol, Ch. 7
resolution, name, 62
resolve, **503**
RESOLVE, 63
response, 155, **503**
restartable atomic region, 451
revectoring, Ch. 8
reverse lookup, 64
revocation, Ch. 11
RISC (see reduced instruction set computer)
Rivest, Shamir, and Adleman cipher, Ch. 11
*robustness principle*, 29, Ch. 8
roll-forward recovery, **503**, Ch. 9
rollback recovery, **503**, Ch. 9
root, 72, **503**
  in UNIX, 102
round-robin scheduling policy, 262, 356, **503**
round-trip time, **503**, Ch. 7
  estimation, Ch. 7
route, Ch. 7
router, **504**, Ch. 7
routing, Ch. 7
  algorithm, **504**, Ch. 7
  protocol, Ch. 7
RPC (see remote procedure call)
RSA (see Rivest, Shamir, and Adleman cipher)
RSM (see read and set memory)
RTP (see real-time transport protocol)

## S

*safety margin principle*, 24, Ch. 8
safety-net approach, Ch. 8, Ch. 11
scheduler, 348, **504**
scheduling policy
  earliest deadline first, 360, **484**
  first-come, first-served, 353, **487**

scheduling policy (*Cont'd*)
hard real-time, 359, **488**
priority, 357, **499**
rate monotonic, 360, **501**
real-time, 359, **502**
round-robin, 262, 356, **503**
shortest-job-first, 354
soft real-time, 359, **506**
scope, 75, **504**
dynamic, 68, **483**
lexical (see scope, static)
static, 68, **507**
search, 73, **504**
in key word query, 75
in name discovery, 76
search path, 73, 75, **504**
second-system effect, 39
secondary device, 331, **504**
secrecy, **504**
secure area, **504**
secure channel, **504**, Ch. 11
Secure Socket Layer, Ch. 11
security, **504**, Ch. 11
protocol, **504**, Ch. 11
seed, Ch. 11
segment
of a message, **504**, Ch. 7
virtual memory, 68, 253, 285, **504**
self-describing storage, 365
self-pacing, **505**, Ch. 7
semaphore, 276, 277, **505**
*separate mechanism from policy*, 331, 349, Ch. 11
sequence coordination, 211, 273, **505**, Ch. 9
sequencer, 276, **505**
sequential consistency, Ch. 9
serial transmission, **505**, Ch. 7
serializability, Probsets
serializable, **505**, Ch. 9
server, 157, **505**
service, 155, **505**, Ch. 7
time, 311, Ch. 7
session service, Ch. 7
set up, **505**, Ch. 7
shadow copy, **505**, Ch. 9
Shannon's capacity theorem, Ch. 7
shared-secret
cryptography, **506**, Ch. 11
key, **506**, Ch. 11
sharing, 60, **506**, Ch. 7
shortcut (see indirect name)
shortest-job-first scheduling policy, 354
sign, **506**, Ch. 7, Ch. 11
simple
locking discipline, **506**, Ch. 9
network time service protocol, Ch. 7
serialization, **506**, Ch. 9
simplex, **506**, Ch. 7
simplicity, 39
single
-acquire protocol, 220, **506**
-event upset, **506**, Ch. 8
point of failure, Ch. 8
state machine, Ch. 10
single-writer, multiple-reader protocol, Ch. 9
Six sigma, Ch. 8
slave, **506**, Ch. 10
sliding window, **506**, Ch. 7
slow start, Ch. 7
snapshot isolation, Ch. 9
snoopy cache, **506**, Ch. 10
SNTP (see protocol, simple network time service)
soft
error, Ch. 8
link (see indirect name)
modularity, 153, **506**
real-time scheduling policy, 359, **507**
state, 189, **506**
source, **507**, Ch. 7
address, **507**
spatial locality, 334, **507**
speaks for, **507**, Ch. 11
speculate, 314, **507**
spin loop, 212, **507**
SSH (see protocol, secure shell)
SSL (see Secure Socket Layer)
stability, 46, **507**
cursor, Ch. 10
stable
binding, 64, **507**
storage, 45
stack
algorithm, 341, **507**
discipline, 150
pointer, 56

starvation, 355, **507**
static
  discipline, 29
  routing, **507**, Ch. 7
  scope, 68, **507**
station, **507**, Ch. 7
  identifier Ch. 7
stop and wait (see lock-step protocol)
storage, 50, **508**
  atomic, **476**
  careful, Ch. 8
  cell, 46, **479**, Ch. 9
  durable, **483**, Ch. 8
  fail-fast, Ch. 8
  journal, **490**, Ch. 9
  leak, 130
  raw, Ch. 8
  stable, 45
store and forward, **508**, Ch. 7
stream, **508**, Ch. 7
  cipher, Ch. 11
  transport protocol, Ch. 7
strict consistency, **508**, Ch. 10
strong consistency (see strict consistency)
stub, 167, **508**
subassembly, 9
submodule, 9
subsystem, 9
supermodule, **508**, Ch. 8
supervisor call instruction, 236, **508**
SVC (see supervisor call instruction)
swapping, 347, **508**
sweeping simplifications
  (see *adopt sweeping simplifications*)
symbolic link (see indirect name)
synonym, 72, **508**
system, 8, **508**
systemwide lock, Ch. 9

## T

Taguchi method, Ch. 8
tail drop, **508**, Ch. 7
TCB (see trusted computing base)
TCP (see transmission control protocol)
TDM (see time-division multiplexing)
tear down, **508**, Ch. 7
temporal
  database, Ch. 10
  locality, 334, **508**
tentatively committed, Ch. 9
test and set memory (see read and set memory)
thrashing, 335, **509**
thread, 204, **509**
  manager, 205, **509**
threat, **509**, Ch. 11
  insider, Ch. 11
throughput, 303, 323, **509**
ticket system, **509**, Ch. 11
tiger team, Ch. 11
time-division multiplexing, Ch. 7
time-domain addressing, Ch. 10
time-sharing, 256
time-to-live, Ch. 10
timed capability, Ch. 11
timer
  adaptive, Ch. 7
  fixed, Ch. 7
timing diagram, 155, 156
TLB (see translation look-aside buffer)
TLB miss exception, 253
TLS (see Transport Layer Security)
TMR (see triple-modular redundancy)
tolerance, 23
tolerated error, **509**, Ch. 8
tombstone, **509**, Ch. 7
tracing garbage collection, 131
trade-off, 6
  binary classification, 7, 371
tragedy of the commons, Ch. 7
trailer, **509**, Ch. 7
transaction, **509**, Ch. 9
transactional memory, **509**, Ch. 9
TRANSFER operation, Ch. 9
transient fault, **509**, Ch. 8
transit time, **509**, Ch. 7
translation look-aside buffer, 253
transmission
  control protocol, Ch. 7
  delay, **509**, Ch. 7
  parallel, **497**, Ch. 7
  serial, **505**, Ch. 7
transport
  protocol, **509**, Ch. 7
  service, Ch. 7
Transport Layer Security, Ch. 11

triple-modular redundancy, **510**, Ch. 8
trusted
  computing base, **510**, Ch. 11
  intermediary, 163, **510**
TTL (see time-to-live)
tunnel (in networks), Ch. 7
two generals dilemma, **510**, Ch. 9
two-phase
  commit, **510**, Ch. 9
  locking discipline, **510**, Ch. 9

## U

UDP (see user datagram protocol)
UNBIND, 63
undo
  action, **510**, Ch. 9
  log, Ch. 9
Uniform Resource Locator, 133
unique identifier name space, 64, **510**
universal name space, 62, **510**
universe of values, 62, **510**
unlimited name space, 129, **510**
untolerated error, **511**, Ch. 8
*unyielding foundations rule*, 20, 38, 288
upcall, Ch. 7
URL (see Uniform Resource Locator)
useful work, 302
user
  datagram protocol, Ch. 7
  -dependent binding, 74, **511**
  mode, 234, **511**
utilization, 302, **511**

## V

valid construction, **511**, Ch. 8
validation (see valid construction)
value, 62, **511**
verify, Ch. 7, Ch. 11
version history, **511**, Ch. 9
virtual
  address, 206, 243, **511**
  address space, 206, 248
  circuit, **511**, Ch. 7
  machine, 208, 290, **511**
  machine monitor, 208, 290, **511**
  memory, 206, 332
  memory manager, 206, 243, **511**
  memory, object-based, 465
  shared memory, 326
virtualization, 201, **511**
virus, Ch. 11
volatile memory, 45, **511**
voter, **511**, Ch. 8

## W

wait-for graph, 221
WAL (see write-ahead log)
watchdog, Ch. 8
waterbed effect, 6
well-known
  name/address, 77, **511**
  port, Ch. 7
window, **512**, Ch. 7
  fixed, Ch. 7
  of validity, Ch. 11
  sliding, **506**, Ch. 7
wired down (page), 331
witness, **512**, Ch. 7, Ch. 10, Ch. 11
work factor, Ch. 11
working
  directory, 67, **512**
  set, 335, **512**
worm, Ch. 11
WRITE, 45
write-ahead log, **512**, Ch. 9
write tearing, 47, **512**
write-through, **512**

## X

X Window System, 162

## Y

yield (in manufacturing), Ch. 8
YIELD (thread primitive), 257