A circular, glowing pattern composed of binary digits (0s and 1s) arranged in concentric rings, radiating from a central bright light source. The colors are primarily shades of green and blue.

Larry L. Peterson and Bruce S. Davie

Fifth Edition

# Computer Networks

a systems approach

**MK**  
MORGAN KAUFMANN

## **In Praise of Computer Networks: A Systems Approach Fifth Edition**

*I have known and used this book for years and I always found it very valuable as a textbook for teaching computer networks as well as a reference book for networking professionals. This Fifth Edition maintains the core value of former editions and brings the clarity of explanation of network protocols in the introduction of the most up-to-date techniques, technologies and requirements of networking. Beyond describing the details of past and current networks, this book successfully motivates the curiosity, and hopefully new research, for the networks of the future.*

**Stefano Basagni**

Northeastern University

*Peterson and Davie have written an outstanding book for the computer networking world. It is a well-organized book that features a very helpful “big picture” systems approach. This book is a must have!*

**Yonshik Choi**

Illinois Institute of Technology

*The Fifth Edition of Computer Networks: A Systems Approach is well-suited for the serious student of computer networks, though it remains accessible to the more casual reader as well. The authors' enthusiasm for their subject is evident throughout; they have a thorough and current grasp of the interesting problems of the field. They explain not only how various protocols work, but also why they work the way they do, and even why certain protocols are the important and interesting ones. The book is also filled with little touches of historical background, from the main text to the “Where Are They Now” sidebars to the papers described in each chapter's “Further Reading” section—these give the reader a perspective on how things came to be the way they are. All in all, this book provides a lucid and literate introduction to networking.*

**Peter Dordal**

Loyola University Chicago

*I have used Computer Networks: A Systems Approach for over five years in an introductory course on communications networks aimed at upper-level undergraduates and first-year Masters students. I have gone through several editions and over the years the book has kept what from the beginning*

*had been its main strength, namely, that it not only describes the 'how,' but also the 'why' and equally important, the 'why not' of things. It is a book that builds engineering intuition, and in this day and age of fast-paced technology changes, this is critical to develop a student's ability to make informed decisions on how to design or select the next generation systems.*

**Roch Guerin**

University of Pennsylvania

*This book is an outstanding introduction to computer networks that is clear, comprehensive, and chock-full of examples. Peterson and Davie have a gift for boiling networking down to simple and manageable concepts without compromising technical rigor. Computer Networks: A Systems Approach strikes an excellent balance between the principles underlying network architecture design and the applications built on top. It should prove invaluable to students and teachers of advanced undergraduate and graduate networking courses.*

**Arvind Krishnamurthy**

University of Washington

*Computer Networks: A Systems Approach has always been one of the best resources available to gain an in-depth understanding of computer networks. The latest edition covers recent developments in the field. Starting with an overview in Chapter 1, the authors systematically explain the basic building blocks of networks. Both hardware and software concepts are presented. The material is capped with a final chapter on applications, which brings all the concepts together. Optional advanced topics are placed in a separate chapter. The textbook also contains a set of exercises of varying difficulty at the end of each chapter which ensure that the students have mastered the material presented.*

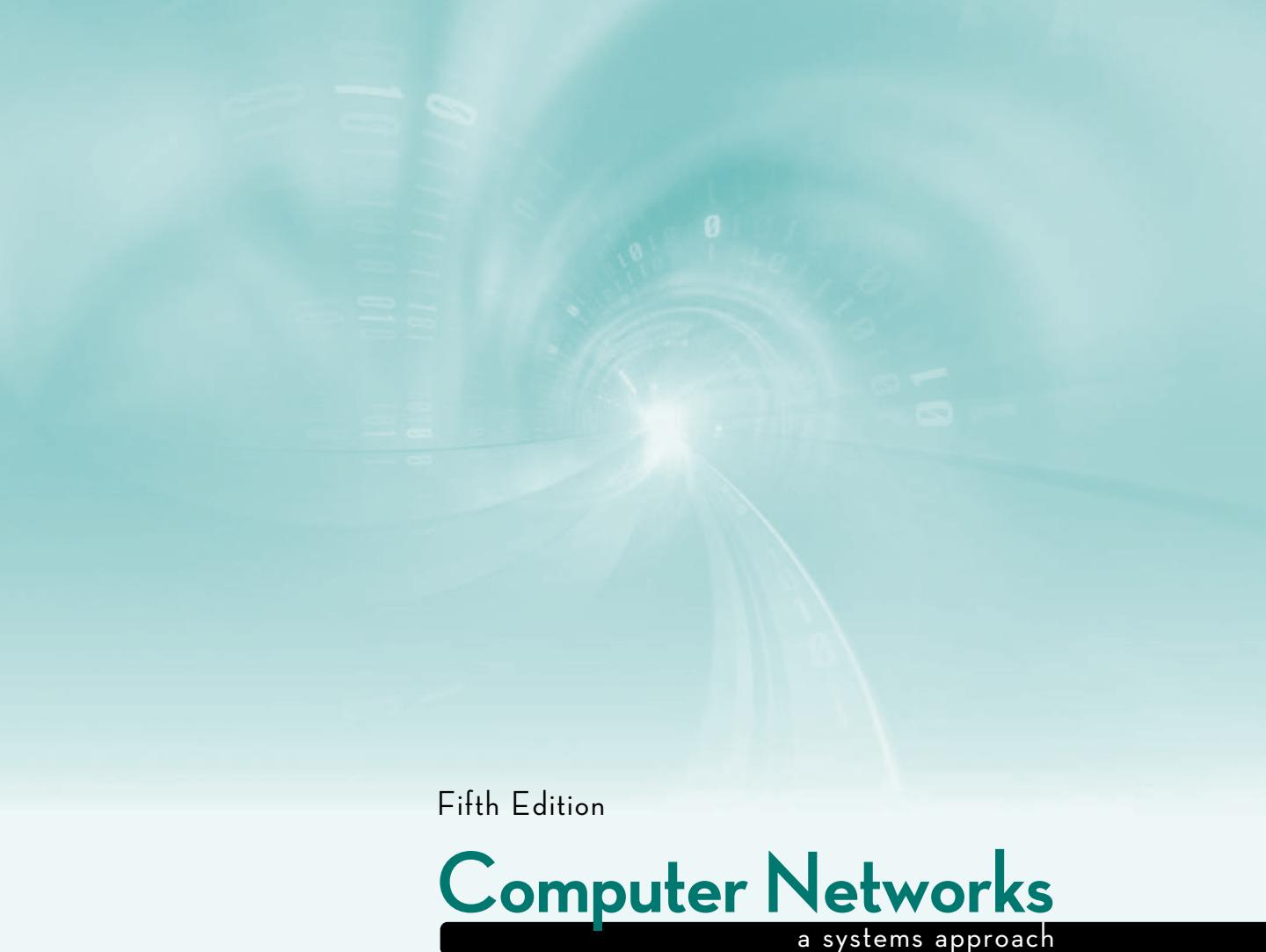
**Karkal Prabhu**

Drexel University

*Peterson and Davie provide a detailed yet clear description of the Internet protocols at all layers. Students will find many study aids that will help them gain a full understanding of the technology that is transforming our society. The book gets better with each edition.*

**Jean Walrand**

University of California at Berkeley



Fifth Edition

# Computer Networks

a systems approach

# Recommended Reading List

For students interested in furthering their understanding of Computer Networking, the content in the following books supplements this textbook:

## ***Network Analysis, Architecture, and Design, 3rd Edition***

By James D. McCabe

ISBN: 9780123704801

## ***The Illustrated Network***

*How TCP/IP Works in a Modern Network*

By Walter Goralski

ISBN: 9780123745415

## ***Interconnecting Smart Objects with IP***

*The Next Internet*

By Jean-Philippe Vasseur and Adam Dunkels

ISBN: 9780123751652

## ***Network Quality of Service Know It All***

Edited by Adrian Farrel

ISBN: 9780123745972

## ***Optical Networks, 3rd Edition***

*A Practical Perspective*

By Rajiv Ramaswami, Kumar Sivarajan and Galen Sasaki

ISBN: 9780123740922

## ***Broadband Cable Access Networks***

*The HFC Plant*

By David Large and James Farmer

ISBN: 9780123744012

## ***Deploying QoS for Cisco IP and Next Generation Networks***

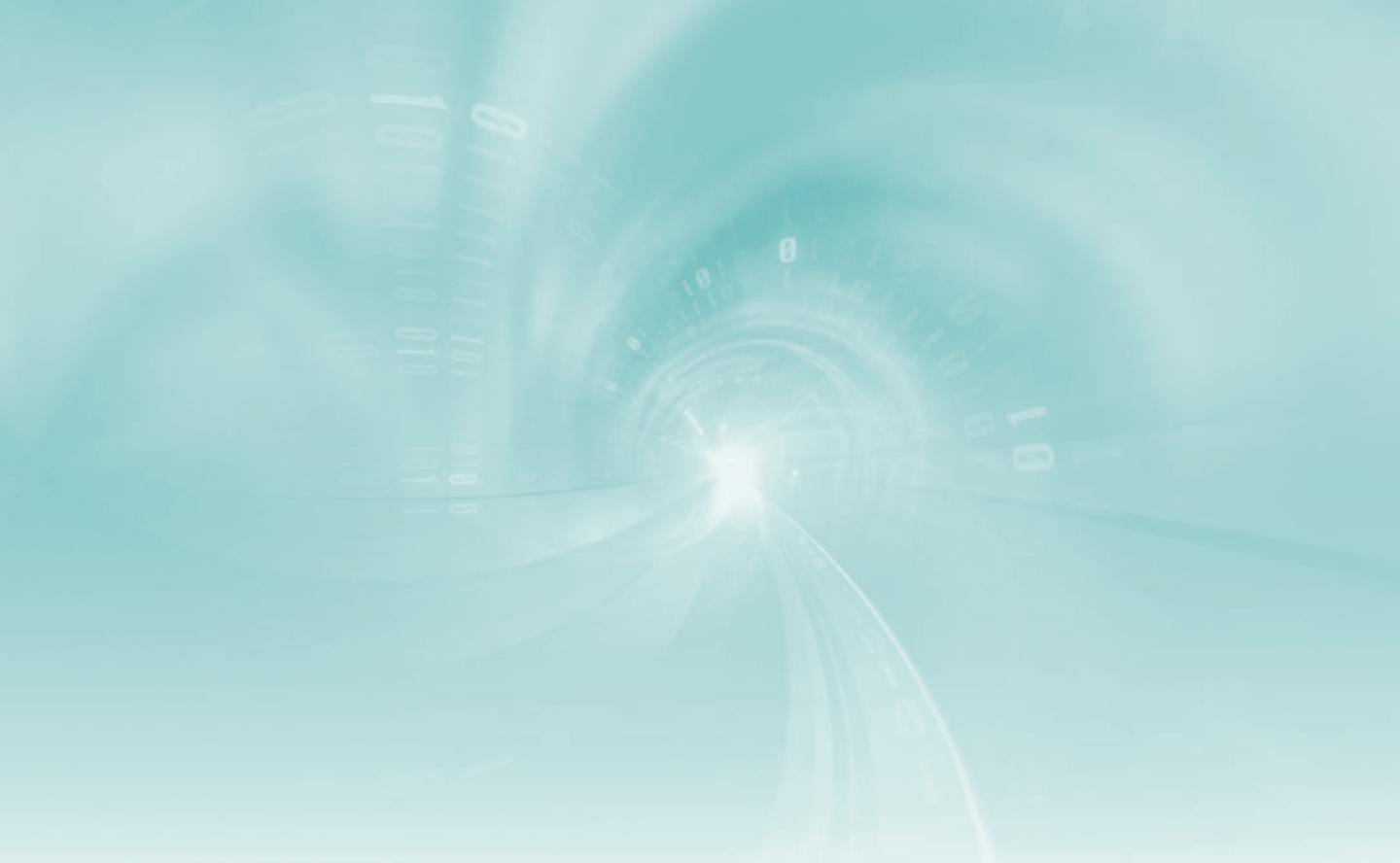
*The Definitive Guide*

By Vinod Joseph and Brett Chapman

ISBN: 9780123744616



[mfp.com](http://mfp.com)



Fifth Edition

# Computer Networks

a systems approach

**Larry L. Peterson and Bruce S. Davie**



ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON  
NEW YORK • OXFORD • PARIS • SAN DIEGO  
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann Publishers is an imprint of Elsevier



**Acquiring Editor:** Rick Adams  
**Development Editor:** Nate McFadden  
**Project Manager:** Paul Gottehrer  
**Designer:** Dennis Schaefer

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

© 2012 Elsevier, Inc. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: [www.elsevier.com/permissions](http://www.elsevier.com/permissions).

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

#### Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods or professional practices, may become necessary. Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information or methods described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

#### Library of Congress Cataloging-in-Publication Data

Peterson, Larry L.

Computer networks : a systems approach / Larry L. Peterson and Bruce S. Davie. – 5th ed.

p. cm. – (The Morgan Kaufmann series in networking)

Includes bibliographical references.

ISBN 978-0-12-385059-1 (hardback)

1. Computer networks. I. Davie, Bruce S. II. Title.

TK5105.5.P479 2011

004.6–dc22

2011000786

#### British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library.

ISBN: 978-0-12-385059-1

For information on all Morgan Kaufmann publications  
visit our website at [www.mkp.com](http://www.mkp.com)

*Typeset by:* diacriTech, India

Printed in the United States of America

11 12 13 14 15 16 10 9 8 7 6 5 4 3 2

Working together to grow  
libraries in developing countries

[www.clscvcr.com](http://www.clscvcr.com) | [www.bookaid.org](http://www.bookaid.org) | [www.sabre.org](http://www.sabre.org)

ELSEVIER BOOK AID International Sabre Foundation

*To Lee Peterson and Robert Davie*

This page intentionally left blank

# Foreword

Once again, this now-classic textbook has been revised to keep it up-to-date with our evolving field. While the Internet and its protocols now dominate networking everywhere, we see continued evolution in the technology used to support the Internet, with switching at “layer 2” providing rich functionality and powerful tools for network management. The previous edition dealt with switching and routing in two chapters, but a presentation based on layers is not always the best way to convey the essentials of the material, since what we call switching and routing actually play similar and complementary roles. This edition of the book looks at these topics in an integrated way, which brings out their functional similarities and differences. More advanced topics in routing have been moved to a second chapter that can be skipped, depending on the emphasis and level of the class.

I have never been a fan of teaching networking based on a purely layered approach, as my foreword to the first edition indicated (we’ve reprinted it in this edition just for fun.) Some key issues in networking, including security and performance, cannot be solved by assigning them to one layer—there cannot be a “performance” layer. These sorts of topics are both critical and cross-cutting, and the organization of this book continues to treat topics, as well as layers. The organization of this book reflects a great deal of experience using it as a classroom textbook, and as well a preference for an approach that brings out fundamentals as well as current practice.

Some moribund technologies are now missing or minimized, including token ring (one of my old favorites, but clearly it was time to go) and ATM. This edition recognizes that we need to pay more attention to application design, and not just packet forwarding. Wireless and mobility gets more attention as well.

The authors, once again, have worked hard to produce a revision that conveys the essentials of the field in a way that is pedagogically effective. I am pleased to say that I think it is better than ever.

David Clark  
November, 2010

This page intentionally left blank

# Foreword to the First Edition

The term *spaghetti code* is universally understood as an insult. All good computer scientists worship the god of modularity, since modularity brings many benefits, including the all-powerful benefit of not having to understand all parts of a problem at the same time in order to solve it. Modularity thus plays a role in presenting ideas in a book, as well as in writing code. If a book's material is organized effectively—modularly—the reader can start at the beginning and actually make it to the end.

The field of network protocols is perhaps unique in that the “proper” modularity has been handed down to us in the form of an international standard: the seven-layer reference model of network protocols from the ISO. This model, which reflects a layered approach to modularity, is almost universally used as a starting point for discussions of protocol organization, whether the design in question conforms to the model or deviates from it.

It seems obvious to organize a networking book around this layered model. However, there is a peril to doing so, because the OSI model is not really successful at organizing the core concepts of networking. Such basic requirements as reliability, flow control, or security can be addressed at most, if not all, of the OSI layers. This fact has led to great confusion in trying to understand the reference model. At times it even requires a suspension of disbelief. Indeed, a book organized strictly according to a layered model has some of the attributes of spaghetti code.

Which brings us to this book. Peterson and Davie follow the traditional layered model, but they do not pretend that this model actually helps in the understanding of the big issues in networking. Instead, the authors organize discussion of fundamental concepts in a way that is independent of layering. Thus, after reading the book, readers will understand flow control, congestion control, reliability enhancement, data representation, and synchronization, and will separately understand the

implications of addressing these issues in one or another of the traditional layers.

This is a timely book. It looks at the important protocols in use today—especially the Internet protocols. Peterson and Davie have a long involvement in and much experience with the Internet. Thus their book reflects not just the theoretical issues in protocol design, but the real factors that matter in practice. The book looks at some of the protocols that are just emerging now, so the reader can be assured of an up-to-date perspective. But most importantly, the discussion of basic issues is presented in a way that derives from the fundamental nature of the problem, not the constraints of the layered reference model or the details of today's protocols. In this regard, what this book presents is both timely and timeless. The combination of real-world relevance, current examples, and careful explanation of fundamentals makes this book unique.

**David D. Clark**  
*Massachusetts Institute of Technology*

# Preface

When the first edition of this book was published in 1996, it was a novelty to be able to order merchandise on the Internet, and a company that advertised its domain name was considered cutting edge. The primary way for a household to connect to the Internet was via a dial-up modem. Today, Internet commerce is a fact of life, and “.com” stocks have gone through an entire boom and bust cycle. Wireless networks are everywhere and new Internet-capable devices such as smartphones and tablets appear on the market at a dizzying pace. It seems the only predictable thing about the Internet is constant change.

Despite these changes, the question we asked in the first edition is just as valid today: What are the underlying concepts and technologies that make the Internet work? The answer is that much of the TCP/IP architecture continues to function just as was envisioned by its creators more than 30 years ago. This isn’t to say that the Internet architecture is uninteresting; quite the contrary. Understanding the design principles that underly an architecture that has not only survived but fostered the kind of growth and change that the Internet has seen over the past 3 decades is precisely the right place to start. Like the previous editions, the Fifth Edition makes the “why” of the Internet architecture its cornerstone.

## Audience

Our intent is that the book should serve as the text for a comprehensive networking class, at either the graduate or upper-division undergraduate level. We also believe that the book’s focus on core concepts should be appealing to industry professionals who are retraining for network-related assignments, as well as current network practitioners who want to understand the “whys” behind the protocols they work with every day and to see the big picture of networking.

It is our experience that both students and professionals learning about networks for the first time often have the impression that network protocols are some sort of edict handed down from on high, and that their job is to learn as many TLAs (Three-Letter Acronyms) as possible. In

fact, protocols are the building blocks of a complex system developed through the application of engineering design principles. Moreover, they are constantly being refined, extended, and replaced based on real-world experience. With this in mind, our goal with this book is to do more than survey the protocols in use today. Instead, we explain the underlying principles of sound network design. We feel that this grasp of underlying principles is the best tool for handling the rate of change in the networking field.

We also recognize that there are many different ways that people approach networks. In contrast to when we wrote our first edition, most people will pick up this book having considerable experience as *users* of networks. Some will be looking to become *designers* of networking products or protocols. Others may be interested in *managing* networks, while an increasingly large number will be current or prospective *application developers* for networked devices. Our focus has traditionally been on the designers of future products and protocols, and that continues to be the case, but in this edition we have tried to address the perspectives of network managers and application developers as well.

### Changes in the Fifth Edition

Even though our focus is on the underlying principles of networking, we illustrate these principles using examples from today's working Internet. Therefore, we added a significant amount of new material to track many of the important recent advances in networking. We also deleted, reorganized, and changed the focus of existing material to reflect changes that have taken place over the past decade.

Perhaps the most significant change we have noticed since writing the first edition is that almost every reader is now familiar with networked applications such as the World Wide Web and email. For this reason, we have increased the focus on applications, starting in the first chapter. We use applications as the motivation for the study of networking, and to derive a set of requirements that a useful network must meet if it is to support both current and future applications on a global scale. However, we retain the problem-solving approach of previous editions that starts with the problem of interconnecting hosts and works its way up the layers to conclude with a detailed examination of application layer issues. We believe it is important to make the topics covered in the book relevant by starting with applications and their needs. At the same time,

we feel that higher layer issues, such as application layer and transport layer protocols, are best understood after the basic problems of connecting hosts and switching packets have been explained. That said, we have made it possible to approach the material in a more *top-down* manner, as described below.

As in prior editions, we have added or increased coverage of important new topics, and brought other topics up to date. Major new or substantially updated topics in this edition are:

- Updated material on wireless technology, particularly the various flavors of 802.11 (Wi-Fi) as well as cellular wireless technologies including the third generation (3G) and emerging 4G standards.
- Updated coverage of congestion control mechanisms, particularly for high bandwidth-delay product networks and wireless networks.
- Updated material on Web Services, including the SOAP and REST (Representational State Transfer) architectures.
- Expanded and updated coverage of interdomain routing and the border gateway protocol (BGP).
- Expanded coverage on protocols for multimedia applications such as voice over IP (VOIP) and video streaming.

We also reduced coverage of some topics that are less relevant today. Protocols moving into the “historic” category for this edition include asynchronous transfer mode (ATM) and token rings.

One of the most significant changes in this edition is the separation of material into “introductory” and “advanced” sections. We wanted to make the book more accessible to people new to networking technologies and protocols, without giving up the advanced material required for upper-level classes. The most apparent effect of this change is that [Chapter 3](#) now covers the basics of switching, routing, and Internetworking, while [Chapter 4](#) covers the more advanced routing topics such as BGP, IP version 6, and multicast. Similarly, transport protocol fundamentals are covered in [Chapter 5](#) with the more advanced material such as TCP congestion control algorithms appearing in [Chapter 6](#). We believe this will make it possible for readers new to the field to grasp important foundational concepts without getting overwhelmed by more complex topics.

As in the last edition, we have included a number of “where are they now?” sidebars. These short discussions, updated for this edition, focus on the success and failure of protocols in the real world. Sometimes they

describe a protocol that most people have written off but which is actually enjoying unheralded success; other times they trace the fate of a protocol that failed to thrive over the long run. The goal of these sidebars is to make the material relevant by showing how technologies have fared in the competitive world of networking.

## Approach

For an area that's as dynamic and changing as computer networks, the most important thing a textbook can offer is perspective—to distinguish between what's important and what's not, and between what's lasting and what's superficial. Based on our experience over the past 25-plus years doing research that has led to new networking technology, teaching undergraduate and graduate students about the latest trends in networking, and delivering advanced networking products to market, we have developed a perspective—which we call the *systems approach*—that forms the soul of this book. The systems approach has several implications:

- *First Principles.* Rather than accept existing artifacts as gospel, we start with first principles and walk you through the thought process that led to today's networks. This allows us to explain *why* networks look like they do. It is our experience that once you understand the underlying concepts, any new protocol that you are confronted with will be relatively easy to digest.
- *Non-layerist.* Although the material is loosely organized around the traditional network layers, starting at the bottom and moving up the protocol stack, we do not adopt a rigidly layerist approach. Many topics—congestion control and security are good examples—have implications up and down the hierarchy, and so we discuss them outside the traditional layered model. Similarly, routers and switches have so much in common (and are often combined as single products) that we discuss them in the same chapter. In short, we believe layering makes a good servant but a poor master; it's more often useful to take an end-to-end perspective.
- *Real-world examples.* Rather than explain how protocols work in the abstract, we use the most important protocols in use today—most of them from the TCP/IP Internet—to illustrate how networks work in practice. This allows us to include real-world experiences in the discussion.

- *Software.* Although at the lowest levels networks are constructed from commodity hardware that can be bought from computer vendors and communication services that can be leased from the phone company, it is the software that allows networks to provide new services and adapt quickly to changing circumstances. It is for this reason that we emphasize how network software is implemented, rather than stopping with a description of the abstract algorithms involved. We also include code segments taken from a working protocol stack to illustrate how you might implement certain protocols and algorithms.
- *End-to-end focus.* Networks are constructed from many building-block pieces, and while it is necessary to be able to abstract away uninteresting elements when solving a particular problem, it is essential to understand how all the pieces fit together to form a functioning network. We therefore spend considerable time explaining the overall end-to-end behavior of networks, not just the individual components, so that it is possible to understand how a complete network operates, all the way from the application to the hardware.
- *Performance.* The systems approach implies doing experimental performance studies, and then using the data you gather both to quantitatively analyze various design options and to guide you in optimizing the implementation. This emphasis on empirical analysis pervades the book.
- *Design Principles.* Networks are like other computer systems—for example, operating systems, processor architectures, distributed and parallel systems, and so on. They are all large and complex. To help manage this complexity, system builders often draw on a collection of design principles. We highlight these design principles as they are introduced throughout the book, illustrated, of course, with examples from computer networks.

## Pedagogy and Features

The Fifth Edition retains the key pedagogical features from prior editions, which we encourage you to take advantage of:

- *Problem statements.* At the start of each chapter, we describe a problem that identifies the next set of issues that must be addressed in the design of a network. This statement introduces and motivates the issues to be explored in the chapter.

- *Shaded sidebars.* Throughout the text, shaded sidebars elaborate on the topic being discussed or introduce a related advanced topic. In many cases, these sidebars relate real-world anecdotes about networking.
- *Where-are-they-now sidebars.* These new elements, a distinctively formatted style of sidebar, trace the success and failure of protocols in real-world deployment.
- *Highlighted paragraphs.* These paragraphs summarize an important nugget of information that we want you to take away from the discussion, such as a widely applicable system design principle.
- *Real protocols.* Even though the book's focus is on core concepts rather than existing protocol specifications, real protocols are used to illustrate most of the important ideas. As a result, the book can be used as a source of reference for many protocols. To help you find the descriptions of the protocols, each applicable section heading parenthetically identifies the protocols described in that section. For example, Section 5.2, which describes the principles of reliable end-to-end protocols, provides a detailed description of TCP, the canonical example of such a protocol.
- *What's Next? discussions.* We conclude the main body of each chapter with an important issue that is currently unfolding in the research community, the commercial world, or society as a whole. We have found that discussing these forward-looking issues helps to make the subject of networking more relevant and exciting.
- *Recommended reading.* These highly selective lists appear at the end of each chapter. Each list generally contains the seminal papers on the topics just discussed. We strongly recommend that advanced readers (e.g., graduate students) study the papers in this reading list to supplement the material covered in the chapter.

## Road Map and Course Use

The book is organized as follows:

- Chapter 1 introduces the set of core ideas that are used throughout the rest of the text. Motivated by wide-spread applications, it discusses what goes into a network architecture, provides an

introduction to protocol implementation issues, and defines the quantitative performance metrics that often drive network design.

- [Chapter 2](#) surveys the many ways that a user can get connected to a larger network such as the Internet, thus introducing the concept of *links*. It also describes many of the issues that all link-level protocols must address, including encoding, framing, and error detection. The most important link technologies today—Ethernet and Wireless—are described here.
- [Chapter 3](#) introduces the basic concepts of switching and routing, starting with the virtual circuit and datagram models. Bridging and LAN switching are covered, followed by an introduction to internetworking, including the Internet Protocol (IP) and routing protocols. The chapter concludes by discussing a range of hardware- and software-based approaches to building routers and switches.
- [Chapter 4](#) covers advanced Internetworking topics. These include multi-area routing protocols, interdomain routing and BGP, IP version 6, multiprotocol label switching (MPLS) and multicast.
- [Chapter 5](#) moves up to the transport level, describing both the Internet’s Transmission Control Protocol (TCP) and Remote Procedure Call (RPC) used to build client-server applications in detail. The Real-time Transport Protocol (RTP), which supports multimedia applications, is also described.
- [Chapter 6](#) discusses congestion control and resource allocation. The issues in this chapter cut across the link level ([Chapter 2](#)), the network level ([Chapters 3 and 4](#)) and the transport level ([Chapter 5](#)). Of particular note, this chapter describes how congestion control works in TCP, and it introduces the mechanisms used to provide quality of service in IP.
- [Chapter 7](#) considers the data sent through a network. This includes both the problems of presentation formatting and data compression. XML is covered here, and the compression section includes explanations of how MPEG video compression and MP3 audio compression work.
- [Chapter 8](#) discusses network security, beginning with an overview of cryptographic tools, the problems of key distribution, and a

discussion of several authentication techniques using both public and private keys. The main focus of this chapter is the building of secure systems, using examples including Pretty Good Privacy (PGP), Secure Shell (SSH), and the IP Security architecture (IPSEC). Firewalls are also covered here.

- [Chapter 9](#) describes a representative sample of network applications, and the protocols they use, including traditional applications like email and the Web, multimedia applications such as IP telephony and video streaming, and overlay networks like peer-to-peer file sharing and content distribution networks. Infrastructure services—the Domain Name System (DNS) and network management—are described. The Web Services architectures for developing new application protocols are also presented here.

For an undergraduate course, extra class time will most likely be needed to help students digest the introductory material in the first chapter, probably at the expense of the more advanced topics covered in [Chapters 4 and 6 through 8](#). [Chapter 9](#) then returns to the popular topic of network applications. An undergraduate class might reasonably skim the more advanced sections (e.g., [Sections 5.3, 9.3.1, 9.3.2](#) and [9.2.2](#).)

In contrast, the instructor for a graduate course should be able to cover the first chapter in only a lecture or two—with students studying the material more carefully on their own—thereby freeing up additional class time to cover [Chapter 4](#) and the later chapters in depth.

For those of you using the book in self-study, we believe that the topics we have selected cover the core of computer networking, and so we recommend that the book be read sequentially, from front to back. In addition, we have included a liberal supply of references to help you locate supplementary material that is relevant to your specific areas of interest, and we have included solutions to select exercises.

The book takes a unique approach to the topic of congestion control by pulling all topics related to congestion control and resource allocation together in a single place—[Chapter 6](#). We do this because the problem of congestion control cannot be solved at any one level, and we want you to consider the various design options at the same time. (This is

consistent with our view that strict layering often obscures important design trade-offs.) A more traditional treatment of congestion control is possible, however, by studying Section 6.2 in the context of Chapter 3 and Section 6.3 in the context of Chapter 5.

### *A Top-Down Pathway*

Because most students today come to a networking class familiar with networked applications, a number of classes take the application as their starting point. While we do cover applications at a high level in Chapter 1, it is not until Chapter 9 that application layer issues are discussed in detail. Recognizing that some professors or readers may wish to follow a more top-down ordering, we suggest the following as a possible way to approach the material in this book.

- **Chapter 1.** This describes applications and their requirements to set the stage for the rest of the material.
- **Chapter 9.** The sections on traditional applications (Section 9.1) and multimedia applications (Section 9.2) will introduce readers to the concepts of network protocols using the examples of applications with which they are already familiar. Section 9.3.1 (DNS) could also be covered.
- **Section 7.2** could be covered next to explain how the data that is generated by multimedia applications is encoded and compressed.
- **Chapter 5.** Transport protocol basics can now be covered, explaining how the data generated by the application layer protocols can be reliably carried across a network.
- **Chapter 3.** Switching, Internetworking, and Routing can be understood as providing the infrastructure over which transport protocols run.
- **Chapter 2.** Finally, the issues of how data is actually encoded and transmitted on physical media such as Ethernets and wireless links can be covered.

Clearly we have skipped quite a few sections in this ordering. For a more advanced course or comprehensive self-study, topics such as resource allocation (Chapter 6), security (Chapter 8), and the advanced topics in Chapter 4 could be added in towards the end. Security could

be covered almost stand-alone, but all these advanced topics will make most sense after IP and TCP have been covered in Chapters 3 and 5 respectively.

Note that the slides made available on our companion site include a set that follows this top-down ordering in addition to the set that follows the order of the book.

## Exercises

Significant effort has gone into improving the exercises with each new edition. In the Second Edition we greatly increased the number of problems and, based on class testing, dramatically improved their quality. In the Third Edition we made two other important changes, which we retained here:

- For those exercises that we felt are particularly challenging or require special knowledge not provided in the book (e.g. probability expertise) we have added an icon  to indicate the extra level of difficulty
- In each chapter we added some extra representative exercises for which worked solutions are provided in the back of the book. These exercises, marked , are intended to provide some help in tackling the other exercises in the book.

In this edition we have added new exercises to reflect the updated content.

The current set of exercises are of several different styles:

- Analytical exercises that ask the student to do simple algebraic calculations that demonstrate their understanding of fundamental relationships
- Design questions that ask the student to propose and evaluate protocols for various circumstances
- Hands-on questions that ask the student to write a few lines of code to test an idea or to experiment with an existing network utility
- Library research questions that ask the student to learn more about a particular topic

Also, as described in more detail below, socket-based programming assignments, as well as simulation labs, are available online.

## Supplemental Materials and Online Resources

To assist instructors, we have prepared an instructor's manual that contains solutions to selected exercises. The manual is available from the publisher.

Additional support materials, including lecture slides, figures from the text, socket-based programming assignments, and sample exams and programming assignments are available through the Morgan Kaufmann Web site at <http://mkp.com/computer-networks>.

And finally, as with the Fourth Edition, a set of laboratory experiments supplement the book. These labs, developed by Professor Emad Aboelela from the University of Massachusetts Dartmouth, use simulation to explore the behavior, scalability, and performance of protocols covered in the book. Sections that discuss material covered by the laboratory exercises are marked with the icon shown in the margin. The simulations use the OPNET simulation toolset, which is available for free to any one using *Computer Networks* in their course.



## Acknowledgments

This book would not have been possible without the help of many people. We would like to thank them for their efforts in improving the end result. Before we do so, however, we should mention that we have done our best to correct the mistakes that the reviewers have pointed out and to accurately describe the protocols and mechanisms that our colleagues have explained to us. We alone are responsible for any remaining errors. If you should find any of these, please send an email to our publisher, Morgan Kaufmann, at [netbugsPD5e@mkp.com](mailto:netbugsPD5e@mkp.com), and we will endeavor to correct them in future printings of this book.

First, we would like to thank the many people who reviewed drafts of all or parts of the manuscript. In addition to those who reviewed prior editions, we wish to thank Peter Dordal, Stefano Basagni, Yonshik Choi, Wenbing Zhao, Sarvesh Kulkarni, James Menth, and John Doyle (and one anonymous reviewer) for their thorough reviews. Thanks also to Dina Katabi and Hari Balakrishnan for their reviews of various sections. We also wish to thank all those who provided feedback and input to help us decide what to do in this edition.

Several members of the Network Systems Group at Princeton contributed ideas, examples, corrections, data, and code to this book. In

particular, we would like to thank Andy Bavier, Tammo Spalink, Mike Wawrzoniak, Stephen Soltesz, and KyoungSoo Park. Thanks also to Shankar M. Banik for developing the two comprehensive sets of slides to accompany the book.

Third, we would like to thank our series editor, David Clark, as well as all the people at Morgan Kaufmann who helped shepherd us through the book-writing process. A special thanks is due to our original sponsoring editor, Jennifer Young; our editor for this edition, Rick Adams; our developmental editor, Nate McFadden; assistant editor David Bevans; and our production editor, Paul Gottehrer. Thanks also to the publisher at MKP, Laura Colantoni, whose leadership inspired us to embark on this revision.

# Contents

Foreword	ix
Foreword to the First Edition	xi
Preface	xiii

## 1 Foundation

Problem: Building a Network	1
1.1 Applications	2
1.1.1 Classes of Applications	3
1.2 Requirements	6
1.2.1 Perspectives	7
1.2.2 Scalable Connectivity	8
1.2.3 Cost-Effective Resource Sharing	13
1.2.4 Support for Common Services	18
1.2.5 Manageability	23
1.3 Network Architecture	24
1.3.1 Layering and Protocols	24
1.3.2 Internet Architecture	33
1.4 Implementing Network Software	36
1.4.1 Application Programming Interface (Sockets)	37
1.4.2 Example Application	40
1.5 Performance	44
1.5.1 Bandwidth and Latency	44
1.5.2 Delay $\times$ Bandwidth Product	48
1.5.3 High-Speed Networks	51
1.5.4 Application Performance Needs	53
1.6 Summary	55
What's Next: Cloud Computing	57
Further Reading	58
Exercises	60

## 2 Getting Connected

Problem: Connecting to a Network	71
2.1 Perspectives on Connecting	72
2.1.1 Classes of Links	75

2.2	Encoding (NRZ, NRZI, Manchester, 4B/5B)	78
2.3	Framing	81
2.3.1	Byte-Oriented Protocols (BISYNC, PPP, DDCMP)	83
2.3.2	Bit-Oriented Protocols (HDLC)	85
2.3.3	Clock-Based Framing (SONET)	88
2.4	Error Detection	91
2.4.1	Two-Dimensional Parity	93
2.4.2	Internet Checksum Algorithm	94
2.4.3	Cyclic Redundancy Check	97
2.5	Reliable Transmission	102
2.5.1	Stop-and-Wait	103
2.5.2	Sliding Window	106
2.5.3	Concurrent Logical Channels	118
2.6	Ethernet and Multiple Access Networks (802.3)	119
2.6.1	Physical Properties	120
2.6.2	Access Protocol	122
2.6.3	Experience with Ethernet	127
2.7	Wireless	128
2.7.1	802.11/Wi-Fi	135
2.7.2	Bluetooth®(802.15.1)	142
2.7.3	Cell Phone Technologies	144
2.8	Summary	148
	What's Next: "The Internet of Things"	150
	Further Reading	151
	Exercises	153

### 3 Internetworking

	Problem: Not All Networks are Directly Connected	169
3.1	Switching and Bridging	170
3.1.1	Datagrams	172
3.1.2	Virtual Circuit Switching	174
3.1.3	Source Routing	186
3.1.4	Bridges and LAN Switches	189
3.2	Basic Internetworking (IP)	203
3.2.1	What Is an Internetwork?	203
3.2.2	Service Model	206
3.2.3	Global Addresses	213
3.2.4	Datagram Forwarding in IP	216
3.2.5	Subnetting and Classless Addressing	220

3.2.6 Address Translation (ARP)	228
3.2.7 Host Configuration (DHCP)	231
3.2.8 Error Reporting (ICMP)	235
3.2.9 Virtual Networks and Tunnels	235
3.3 Routing	240
3.3.1 Network as a Graph	242
3.3.2 Distance Vector (RIP)	243
3.3.3 Link State (OSPF)	252
3.3.4 Metrics	262
3.4 Implementation and Performance	266
3.4.1 Switch Basics	267
3.4.2 Ports	270
3.4.3 Fabrics	273
3.4.4 Router Implementation	277
3.5 Summary	280
What's Next: The Future Internet	281
Further Reading	282
Exercises	284

## 4 Advanced Internetworking

Problem: Scaling to Billions	307
4.1 The Global Internet	308
4.1.1 Routing Areas	310
4.1.2 Interdomain Routing (BGP)	313
4.1.3 IP Version 6 (IPv6)	324
4.2 Multicast	338
4.2.1 Multicast Addresses	340
4.2.2 Multicast Routing (DVMRP, PIM, MSDP)	341
4.3 Multiprotocol Label Switching (MPLS)	354
4.3.1 Destination-Based Forwarding	355
4.3.2 Explicit Routing	362
4.3.3 Virtual Private Networks and Tunnels	364
4.4 Routing among Mobile Devices	369
4.4.1 Challenges for Mobile Networking	369
4.4.2 Routing to Mobile Hosts (Mobile IP)	372
4.5 Summary	379
What's Next: Deployment of IPv6	380
Further Reading	381
Exercises	382

## 5 End-to-End Protocols

Problem: Getting Process to Communicate	391
5.1 Simple Demultiplexer (UDP)	393
5.2 Reliable Byte Stream (TCP)	396
5.2.1 End-to-End Issues	397
5.2.2 Segment Format	400
5.2.3 Connection Establishment and Termination	402
5.2.4 Sliding Window Revisited	407
5.2.5 Triggering Transmission	414
5.2.6 Adaptive Retransmission	418
5.2.7 Record Boundaries	422
5.2.8 TCP Extensions	423
5.2.9 Performance	425
5.2.10 Alternative Design Choices	428
5.3 Remote Procedure Call	431
5.3.1 RPC Fundamentals	431
5.3.2 RPC Implementations (SunRPC, DCE)	440
5.4 Transport for Real-Time Applications (RTP)	447
5.4.1 Requirements	449
5.4.2 RTP Design	452
5.4.3 Control Protocol	456
5.5 Summary	460
What's Next: Transport Protocol Diversity	461
Further Reading	462
Exercises	463

## 6 Congestion Control and Resource Allocation

Problem: Allocating Resources	479
6.1 Issues in Resource Allocation	480
6.1.1 Network Model	481
6.1.2 Taxonomy	485
6.1.3 Evaluation Criteria	488
6.2 Queuing Disciplines	492
6.2.1 FIFO	492
6.2.2 Fair Queuing	494
6.3 TCP Congestion Control	499
6.3.1 Additive Increase/Multiplicative Decrease	500

6.3.2 Slow Start	505
6.3.3 Fast Retransmit and Fast Recovery	510
6.4 Congestion-Avoidance Mechanisms	514
6.4.1 DECbit	515
6.4.2 Random Early Detection (RED)	516
6.4.3 Source-Based Congestion Avoidance	523
6.5 Quality of Service	530
6.5.1 Application Requirements	531
6.5.2 Integrated Services (RSVP)	537
6.5.3 Differentiated Services (EF, AF)	549
6.5.4 Equation-Based Congestion Control	557
6.6 Summary	559
What's Next: Refactoring the Network	560
Further Reading	561
Exercises	563

## 7 End-to-End Data

Problem: What Do We Do with the Data?	579
7.1 Presentation Formatting	581
7.1.1 Taxonomy	583
7.1.2 Examples (XDR, ASN.1, NDR)	587
7.1.3 Markup Languages (XML)	592
7.2 Multimedia Data	596
7.2.1 Lossless Compression Techniques	598
7.2.2 Image Representation and Compression (GIF, JPEG)	601
7.2.3 Video Compression (MPEG)	609
7.2.4 Transmitting MPEG over a Network	614
7.2.5 Audio Compression (MP3)	619
7.3 Summary	621
What's Next: Video Everywhere	622
Further Reading	623
Exercises	624

## 8 Network Security

Problem: Security Attacks	633
8.1 Cryptographic Building Blocks	635
8.1.1 Principles of Ciphers	635
8.1.2 Symmetric-Key Ciphers	638

8.1.3	Public-Key Ciphers	640
8.1.4	Authenticators	643
8.2	Key Predistribution	647
8.2.1	Predistribution of Public Keys	648
8.2.2	Predistribution of Symmetric Keys	653
8.3	Authentication Protocols	654
8.3.1	Originality and Timeliness Techniques	655
8.3.2	Public-Key Authentication Protocols	656
8.3.3	Symmetric-Key Authentication Protocols	658
8.3.4	Diffie-Hellman Key Agreement	662
8.4	Example Systems	664
8.4.1	Pretty Good Privacy (PGP)	665
8.4.2	Secure Shell (SSH)	667
8.4.3	Transport Layer Security (TLS, SSL, HTTPS)	670
8.4.4	IP Security (IPsec)	675
8.4.5	Wireless Security (802.11i)	678
8.5	Firewalls	681
8.5.1	Strengths and Weaknesses of Firewalls	684
8.6	Summary	686
	What's Next: Coming to Grips with Security	688
	Further Reading	689
	Exercises	690

## 9 Applications

	Problem: Applications Need their Own Protocols	697
9.1	Traditional Applications	698
9.1.1	Electronic Mail (SMTP, MIME, IMAP)	700
9.1.2	World Wide Web (HTTP)	708
9.1.3	Web Services	718
9.2	Multimedia Applications	727
9.2.1	Session Control and Call Control (SDP, SIP, H.323)	728
9.2.2	Resource Allocation for Multimedia Applications	739
9.3	Infrastructure Services	744
9.3.1	Name Service (DNS)	745
9.3.2	Network Management (SNMP)	756

9.4 Overlay Networks	759
9.4.1 Routing Overlays	762
9.4.2 Peer-to-Peer Networks	769
9.4.3 Content Distribution Networks	783
9.5 Summary	789
What's Next: New Network Architecture	790
Further Reading	791
Exercises	793
<b>Solutions to Select Exercises</b>	801
<b>Glossary</b>	815
<b>Bibliography</b>	837
<b>Index</b>	851

A SYSTEMS APPROACH

# 1

## Foundation

*I must Create a System, or be enslav'd by another Man's; I will not Reason and Compare: my business is to Create.*

—William Blake

Suppose you want to build a computer network, one that has the potential to grow to global proportions and to support applications as diverse as teleconferencing, video on demand, electronic commerce, distributed computing, and digital libraries. What available technologies would serve as the underlying building blocks, and what kind of software architecture would you design to integrate these building blocks into an effective communication service? Answering this question is the overriding goal of this book—to describe the available building materials and

### PROBLEM: BUILDING A NETWORK

then to show how they can be used to construct a network from the ground up.

Before we can understand how to design a computer network, we should first agree on exactly what a computer network is. At one time, the term *network* meant the set of serial lines used to attach dumb terminals to mainframe computers. Other important networks include the voice telephone network and the cable TV network used to disseminate video signals. The main things these networks have in common are that they are specialized to handle one particular kind of data

(keystrokes, voice, or video) and they typically connect to special-purpose devices (terminals, hand receivers, and television sets).

What distinguishes a computer network from these other types of networks? Probably the most important characteristic of a computer network is its generality. Computer networks are built primarily from general-purpose programmable hardware, and they are not optimized for a particular application like making phone calls or delivering television signals. Instead, they are able to carry many different types of data, and they support a wide, and ever growing, range of applications. Today's computer networks are increasingly taking over the functions previously performed by single-use networks. This chapter looks at some typical applications of computer networks and discusses the requirements that a network designer who wishes to support such applications must be aware of.

Once we understand the requirements, how do we proceed? Fortunately, we will not be building the first network. Others, most notably the community of researchers responsible for the Internet, have gone before us. We will use the wealth of experience generated from the Internet to guide our design. This experience is embodied in a *network architecture* that identifies the available hardware and software components and shows how they can be arranged to form a complete network system.

In addition to understanding how networks are built, it is increasingly important to understand how they are operated or managed and how network applications are developed. Most of us now have computer networks in our homes, offices, and in some cases in our cars, so operating networks is no longer a matter only for a few specialists. And, with the proliferation of programmable, network-attached devices such as smartphones, many more of this generation will develop networked applications than in the past. So we need to consider networks from these multiple perspectives: builders, operators, application developers.

To start us on the road toward understanding how to build, operate, and program a network, this chapter does four things. First, it explores the requirements that different applications and different communities of people place on the network. Second, it introduces the idea of a network architecture, which lays the foundation for the rest of the book. Third, it introduces some of the key elements in the implementation of computer networks. Finally, it identifies the key metrics that are used to evaluate the performance of computer networks.

## 1.1 APPLICATIONS

Most people know the Internet through its applications: the World Wide Web, email, online social networking, streaming audio and video, instant messaging, file-sharing, to name just a few examples. That is to say, we

interact with the Internet as *users* of the network. Internet users represent the largest class of people who interact with the Internet in some way, but there are several other important constituencies. There is the group of people who *create* the applications—a group that has greatly expanded in recent years as powerful programming platforms and new devices such as smartphones have created new opportunities to develop applications quickly and to bring them to a large market. Then there are those who *operate* or *manage* networks—mostly a behind-the-scenes job, but a critical one and often a very complex one. With the prevalence of home networks, more and more people are also becoming, if only in a small way, network operators. Finally, there are those who *design* and *build* the devices and protocols that collectively make up the Internet. That final constituency is the traditional target of networking textbooks such as this one and will continue to be our main focus. However, throughout this book we will also consider the perspectives of application developers and network operators. Considering these perspectives will enable us to better understand the diverse requirements that a network must meet. Application developers will also be able to make applications that work better if they understand how the underlying technology works and interacts with the applications. So, before we start figuring out how to build a network, let's look more closely at the types of applications that today's networks support.

### 1.1.1 Classes of Applications

The World Wide Web is the Internet application that catapulted the Internet from a somewhat obscure tool used mostly by scientists and engineers to the mainstream phenomenon that it is today. The Web itself has become such a powerful platform that many people confuse it with the Internet (as in “the Interwebs”), and it’s a bit of a stretch to say that the Web is a single application.

In its basic form, the Web presents an intuitively simple interface. Users view pages full of textual and graphical objects and click on objects that they want to learn more about, and a corresponding new page appears. Most people are also aware that just under the covers each selectable object on a page is bound to an identifier for the next page or object to be viewed. This identifier, called a Uniform Resource Locator (URL), provides a way of identifying all the possible objects that can be viewed from your web browser. For example,

<http://www.cs.princeton.edu/~ljp/index.html>

is the URL for a page providing information about one of this book’s authors: the string http indicates that the Hypertext Transfer Protocol (HTTP) should be used to download the page, [www.cs.princeton.edu](http://www.cs.princeton.edu) is the name of the machine that serves the page, and

/~llp/index.html

uniquely identifies Larry’s home page at this site.

What most web users are not aware of, however, is that by clicking on just one such URL over a dozen messages may be exchanged over the Internet, and many more than that if the web page is complicated with lots of embedded objects. This message exchange includes up to six messages to translate the server name ([www.cs.princeton.edu](http://www.cs.princeton.edu)) into its Internet Protocol (IP) address (128.112.136.35), three messages to set up a Transmission Control Protocol (TCP) connection between your browser and this server, four messages for your browser to send the HTTP “GET” request and the server to respond with the requested page (and for each side to acknowledge receipt of that message), and four messages to tear down the TCP connection. Of course, this does not include the millions of messages exchanged by Internet nodes throughout the day, just to let each other know that they exist and are ready to serve web pages, translate names to addresses, and forward messages toward their ultimate destination.

Another widespread application class of the Internet is the delivery of “streaming” audio and video. Services such as video on demand and Internet radio use this technology. While we frequently start at a website to initiate a streaming session, the delivery of audio and video has some important differences from fetching a simple web page of text and images. For example, you often don’t want to download an entire video file—a process that might take minutes to hours—before watching the first scene. Streaming audio and video implies a more timely transfer of messages from sender to receiver, and the receiver displays the video or plays the audio pretty much as it arrives.

Note that the difference between streaming applications and the more traditional delivery of a page of text or still images is that humans consume audio and video streams in a continuous manner, and discontinuity—in the form of skipped sounds or stalled video—is not acceptable. By contrast, a page of text can be delivered and read in bits and pieces. This difference affects how the network supports these different classes of applications.

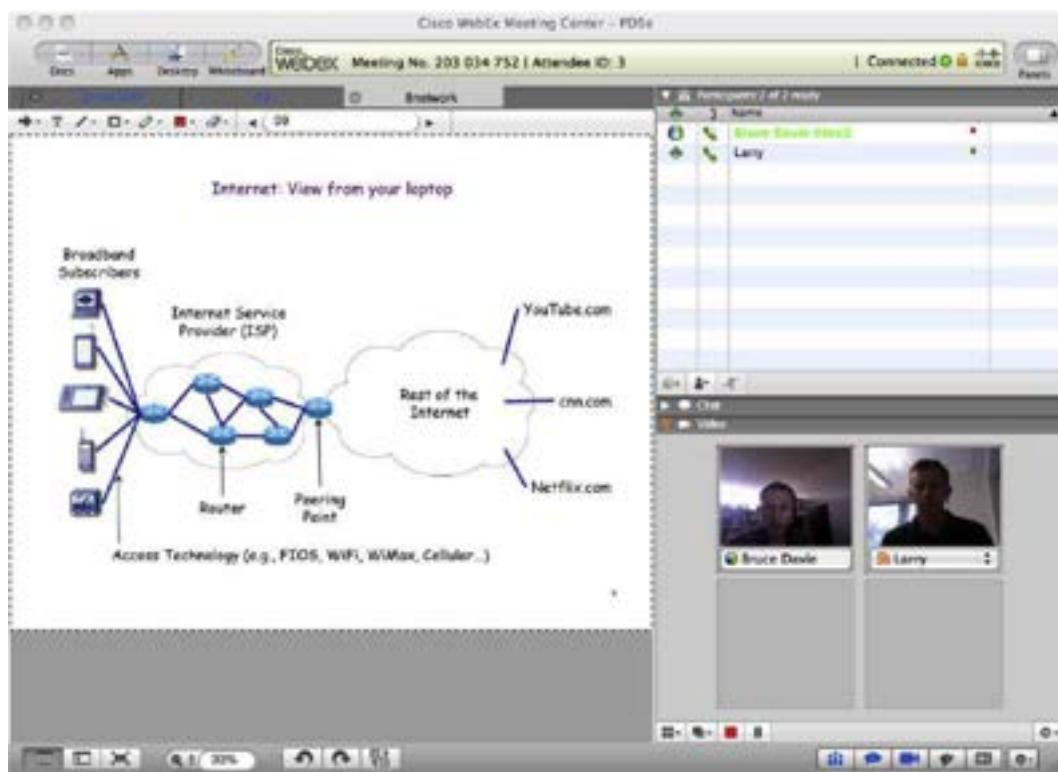
A subtly different application class is **real-time audio and video**. These applications have considerably tighter timing constraints than streaming applications. When using a voice-over-IP application such as Skype™ or a videoconferencing application, the interactions among the participants must be timely. When a person at one end gestures, then that action must be displayed at the other end as quickly as possible. When one person tries to interrupt another, the interrupted person needs to hear that as soon as possible<sup>1</sup> and decide whether to allow the interruption or to keep talking over the interrupter. Too much delay in this sort of environment makes the system unusable. Contrast this with video on demand where, if it takes several seconds from the time the user starts the video until the first image is displayed, the service is still deemed satisfactory. Also, **interactive applications usually entail audio and/or video flows in both directions, while a streaming application is most likely sending video or audio in only one direction.**

Videoconferencing tools that run over the Internet have been around now since the early 1990s but have achieved much more widespread use in the last couple of years, as higher network speeds and more powerful computers have become commonplace. An example of one such system is shown in Figure 1.1. Just as downloading a web page involves a bit more than meets the eye, so too with video applications. Fitting the video content into a relatively low bandwidth network, for example, or making sure that the video and audio remain in sync and arrive in time for a good user experience are all problems that network and protocol designers have to worry about. We'll look at these and many other issues related to multimedia applications later in the book.

Although they are just two examples, downloading pages from the web and participating in a videoconference demonstrate the diversity of applications that can be built on top of the Internet and hint at the complexity of the Internet's design. Later in the book we will develop a more complete taxonomy of application types to help guide our discussion of key design decisions as we seek to build, operate, and use networks that support such a wide range of applications. In Chapter 9, the book concludes by revisiting these two specific applications, as well as several others that illustrate the breadth of what is possible on today's Internet.

---

<sup>1</sup>Not quite "as soon as possible"—human factors research indicates 300 ms is a reasonable upper bound for how much round-trip delay can be tolerated in a telephone call before humans complain, and a 100-ms delay sounds very good.



■ FIGURE 1.1 A multimedia application including videoconferencing.



For now, this quick look at a few typical applications will suffice to enable us to start looking at the problems that must be addressed if we are to build a network that supports such application diversity.

## 1.2 REQUIREMENTS

We have established an ambitious goal for ourselves: to understand how to build a computer network from the ground up. Our approach to accomplishing this goal will be to **start from first principles** and then **ask the kinds of questions we would naturally ask if building an actual network**. At each step, we will use today's protocols to illustrate various design choices available to us, but we will not accept these existing artifacts as gospel. Instead, we will be asking (and answering) the question of *why* networks are designed the way they are. While it is tempting

to settle for just understanding the way it's done today, it is important to recognize the underlying concepts because networks are constantly changing as the technology evolves and new applications are invented. It is our experience that once you understand the fundamental ideas, any new protocol that you are confronted with will be relatively easy to digest.

### 1.2.1 Perspectives

As we noted above, a student of networks can take several perspectives. When we wrote the first edition of this book, the majority of the population had no Internet access at all, and those who did obtained it while at work, at a university, or by a dial-up modem at home. The set of popular applications could be counted on one's fingers. Thus, like most books at the time, ours focused on the perspective of someone who would design networking equipment and protocols. We continue to focus on this perspective, and our hope is that after reading this book you will know how to design the networking equipment and protocols of the future. However, we also want to cover the perspectives of two additional groups that are of increasing importance: those who develop networked applications and those who manage or operate networks. Let's consider how these three groups might list their requirements for a network:

- An *application programmer* would list the services that his or her application needs—for example, a guarantee that each message the application sends will be delivered without error within a certain amount of time or the ability to switch gracefully among different connections to the network as the user moves around.
- A *network operator* would list the characteristics of a system that is easy to administer and manage—for example, in which faults can be easily isolated, new devices can be added to the network and configured correctly, and it is easy to account for usage.
- A *network designer* would list the properties of a cost-effective design—for example, that network resources are efficiently utilized and fairly allocated to different users. Issues of performance are also likely to be important.

This section attempts to distill these different perspectives into a high-level introduction to the major considerations that drive network design and, in doing so, identifies the challenges addressed throughout the rest of this book.

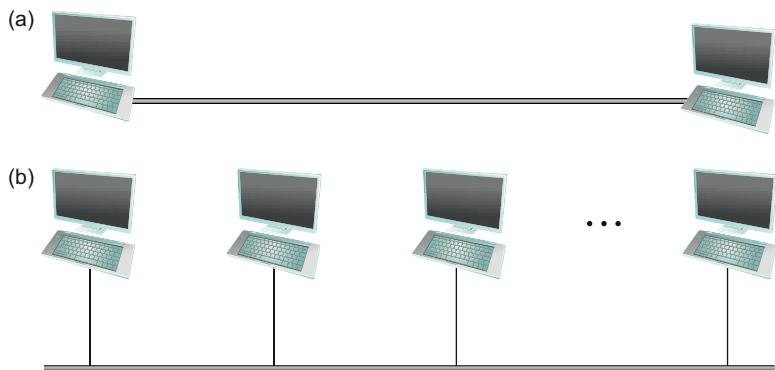
### 1.2.2 Scalable Connectivity

Starting with the obvious, a network must provide connectivity among a set of computers. Sometimes it is enough to build a limited network that connects only a few select machines. In fact, for reasons of privacy and security, many private (corporate) networks have the explicit goal of limiting the set of machines that are connected. In contrast, other networks (of which the Internet is the prime example) are designed to grow in a way that allows them the potential to connect all the computers in the world. A system that is designed to support growth to an arbitrarily large size is said to *scale*. Using the Internet as a model, this book addresses the challenge of scalability.

#### *Links, Nodes, and Clouds*

To understand the requirements of connectivity more fully, we need to take a closer look at how computers are connected in a network. Connectivity occurs at many different levels. At the lowest level, a network can consist of two or more computers directly connected by some physical medium, such as a coaxial cable or an optical fiber. We call such a physical medium a *link*, and we often refer to the computers it connects as *nodes*. (Sometimes a node is a more specialized piece of hardware rather than a computer, but we overlook that distinction for the purposes of this discussion.) As illustrated in Figure 1.2, physical links are sometimes limited to a pair of nodes (such a link is said to be *point-to-point*), while in other cases more than two nodes may share a single physical link (such a link is said to be *multiple-access*). Wireless links, such as those provided by cellular networks and Wi-Fi networks, are an increasingly important class of multiple-access links. It is often the case that multiple-access links are limited in size, in terms of both the geographical distance they can cover and the number of nodes they can connect.

If computer networks were limited to situations in which all nodes are directly connected to each other over a common physical medium, then either networks would be very limited in the number of computers they could connect, or the number of wires coming out of the back of each node would quickly become both unmanageable and very expensive. Fortunately, connectivity between two nodes does not necessarily imply a direct physical connection between them—indirect connectivity may be achieved among a set of cooperating nodes. Consider the

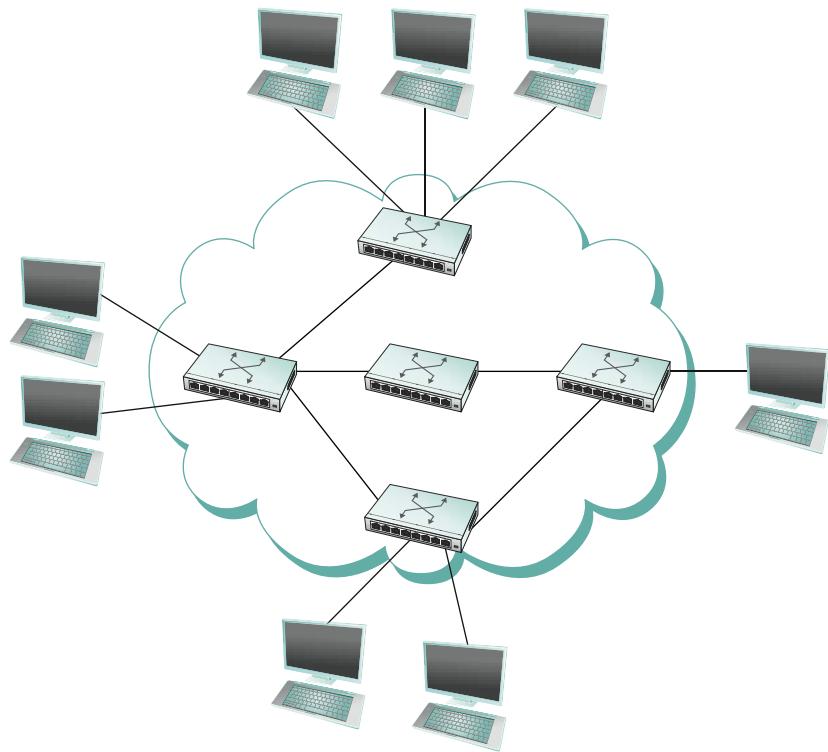


■ FIGURE 1.2 Direct links: (a) point-to-point; (b) multiple-access.

following two examples of how a collection of computers can be indirectly connected.

Figure 1.3 shows a set of nodes, each of which is attached to one or more point-to-point links. Those nodes that are attached to at least two links run software that forwards data received on one link out on another. If organized in a systematic way, these forwarding nodes form a *switched network*. There are numerous types of switched networks, of which the two most common are *circuit switched* and *packet switched*. The former is most notably employed by the telephone system, while the latter is used for the overwhelming majority of computer networks and will be the focus of this book. (Circuit switching is, however, making a bit of a comeback in the optical networking realm, which turns out to be important as demand for network capacity constantly grows.) The important feature of packet-switched networks is that the nodes in such a network send discrete blocks of data to each other. Think of these blocks of data as corresponding to some piece of application data such as a file, a piece of email, or an image. We call each block of data either a *packet* or a *message*, and for now we use these terms interchangeably; we discuss the reason they are not always the same in Section 1.2.3.

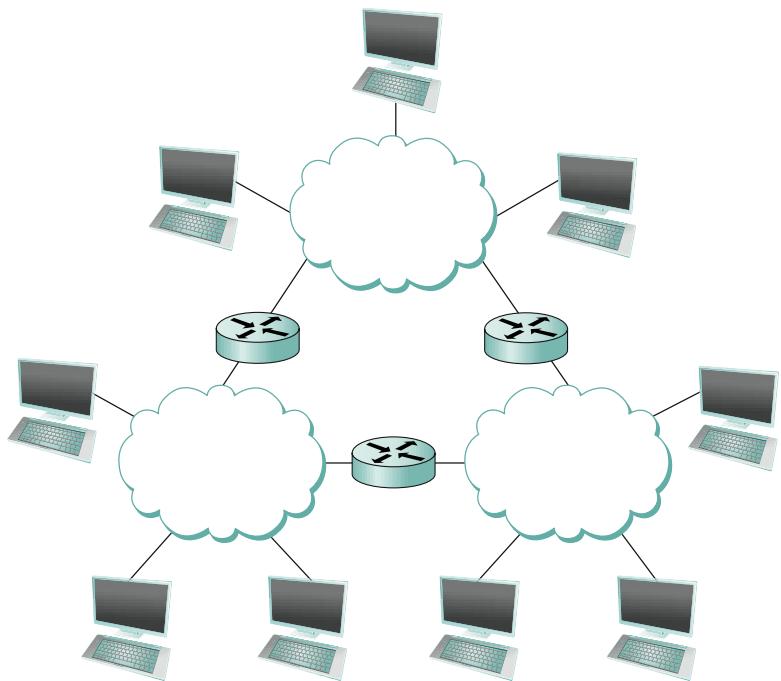
Packet-switched networks typically use a strategy called *store-and-forward*. As the name suggests, each node in a store-and-forward network first receives a complete packet over some link, stores the packet in its internal memory, and then forwards the complete packet to the next



■ FIGURE 1.3 Switched network.

node. In contrast, a circuit-switched network first establishes a dedicated circuit across a sequence of links and then allows the source node to send a stream of bits across this circuit to a destination node. The major reason for using packet switching rather than circuit switching in a computer network is efficiency, discussed in the next subsection.

The cloud in Figure 1.3 distinguishes between the nodes on the inside that *implement* the network (they are commonly called *switches*, and their primary function is to store and forward packets) and the nodes on the outside of the cloud that *use* the network (they are commonly called *hosts*, and they support users and run application programs). Also note that the cloud in Figure 1.3 is one of the most important icons of computer networking. In general, we use a cloud to denote any type of network, whether it is a single point-to-point link, a multiple-access link, or a switched network. Thus, whenever you see a cloud used in a figure,



■ FIGURE 1.4 Interconnection of networks.

you can think of it as a placeholder for any of the networking technologies covered in this book.<sup>2</sup>

A second way in which a set of computers can be indirectly connected is shown in Figure 1.4. In this situation, a set of independent networks (clouds) are interconnected to form an *internetwork*, or *internet* for short. We adopt the Internet's convention of referring to a generic internetwork of networks as a lowercase *i* internet, and the currently operational TCP/IP Internet as the capital *I* Internet. A node that is connected to two or more networks is commonly called a *router* or *gateway*, and it plays much the same role as a switch—it forwards messages from one network to another. Note that an internet can itself be viewed as another kind of network, which means that an internet can be built from an interconnection of internets. Thus, we can recursively build arbitrarily large networks by interconnecting clouds to form larger clouds. It can

<sup>2</sup>Interestingly, the use of clouds in this way predates the term *cloud computing* by at least a couple of decades, but there is a connection between these two usages, which we'll discuss later.

reasonably be argued that this idea of interconnecting widely differing networks was the fundamental innovation of the Internet and that the successful growth of the Internet to global size and billions of nodes was the result of some very good design decisions by the early Internet architects, which we will discuss later.

Just because a set of hosts are directly or indirectly connected to each other does not mean that we have succeeded in providing host-to-host connectivity. The final requirement is that each node must be able to say which of the other nodes on the network it wants to communicate with. This is done by assigning an *address* to each node. An address is a byte string that identifies a node; that is, the network can use a node's address to distinguish it from the other nodes connected to the network. When a source node wants the network to deliver a message to a certain destination node, it specifies the address of the destination node. If the sending and receiving nodes are not directly connected, then the switches and routers of the network use this address to decide how to forward the message toward the destination. The process of determining systematically how to forward messages toward the destination node based on its address is called *routing*.

This brief introduction to addressing and routing has presumed that the source node wants to send a message to a single destination node (*unicast*). While this is the most common scenario, it is also possible that the source node might want to *broadcast* a message to all the nodes on the network. Or, a source node might want to send a message to some subset of the other nodes but not all of them, a situation called *multicast*. Thus, in addition to node-specific addresses, another requirement of a network is that it support multicast and broadcast addresses.



The main idea to take away from this discussion is that we can define a *network* recursively as consisting of two or more nodes connected by a physical link, or as two or more networks connected by a node. In other words, a network can be constructed from a nesting of networks, where at the bottom level, the network is implemented by some physical medium. Among the key challenges in providing network connectivity are the definition of an address for each node that is reachable on the network (including support for broadcast and multicast), and the use of such addresses to forward messages toward the appropriate destination node(s).

### 1.2.3 Cost-Effective Resource Sharing

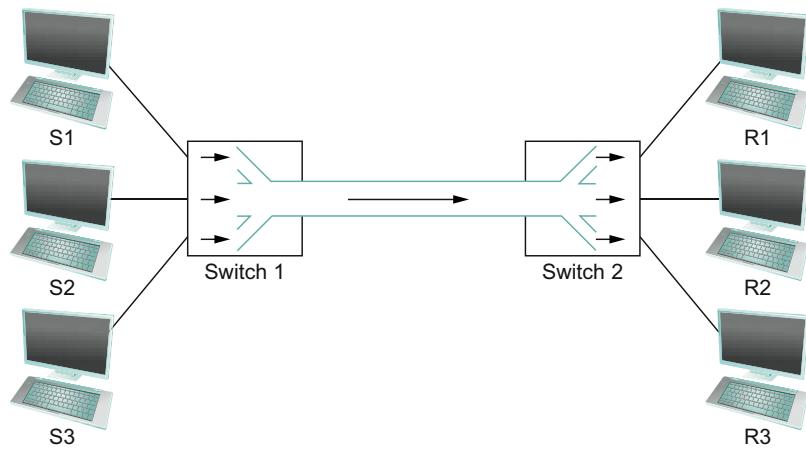
As stated above, this book focuses on packet-switched networks. This section explains the key requirement of computer networks—efficiency—that leads us to packet switching as the strategy of choice.

Given a collection of nodes indirectly connected by a nesting of networks, it is possible for any pair of hosts to send messages to each other across a sequence of links and nodes. Of course, we want to do more than support just one pair of communicating hosts—we want to provide all pairs of hosts with the ability to exchange messages. The question, then, is how do all the hosts that want to communicate share the network, especially if they want to use it at the same time? And, as if that problem isn't hard enough, how do several hosts share the same *link* when they all want to use it at the same time?

To understand how hosts share a network, we need to introduce a fundamental concept, *multiplexing*, which means that a system resource is shared among multiple users. At an intuitive level, multiplexing can be explained by analogy to a timesharing computer system, where a single physical processor is shared (multiplexed) among multiple jobs, each of which believes it has its own private processor. Similarly, data being sent by multiple users can be multiplexed over the physical links that make up a network.

To see how this might work, consider the simple network illustrated in Figure 1.5, where the three hosts on the left side of the network (senders S1–S3) are sending data to the three hosts on the right (receivers R1–R3) by sharing a switched network that contains only one physical link. (For simplicity, assume that host S1 is sending data to host R1, and so on.) In this situation, three flows of data—corresponding to the three pairs of hosts—are multiplexed onto a single physical link by switch 1 and then *demultiplexed* back into separate flows by switch 2. Note that we are being intentionally vague about exactly what a “flow of data” corresponds to. For the purposes of this discussion, assume that each host on the left has a large supply of data that it wants to send to its counterpart on the right.

There are several different methods for multiplexing multiple flows onto one physical link. One common method is *synchronous time-division multiplexing* (STDM). The idea of STDM is to divide time into equal-sized quanta and, in a round-robin fashion, give each flow a chance



■ FIGURE 1.5 Multiplexing multiple logical flows over a single physical link.

to send its data over the physical link. In other words, during time quantum 1, data from S1 to R1 is transmitted; during time quantum 2, data from S2 to R2 is transmitted; in quantum 3, S3 sends data to R3. At this point, the first flow (S1 to R1) gets to go again, and the process repeats. Another method is *frequency-division multiplexing* (FDM). The idea of FDM is to transmit each flow over the physical link at a different frequency, much the same way that the signals for different TV stations are transmitted at a different frequency over the airwaves or on a coaxial cable TV link.

Although simple to understand, both STDM and FDM are limited in two ways. First, if one of the flows (host pairs) does not have any data to send, its share of the physical link—that is, its time quantum or its frequency—remains idle, even if one of the other flows has data to transmit. For example, S3 had to wait its turn behind S1 and S2 in the previous paragraph, even if S1 and S2 had nothing to send. For computer communication, the amount of time that a link is idle can be very large—for example, consider the amount of time you spend reading a web page (leaving the link idle) compared to the time you spend fetching the page. Second, both STDM and FDM are limited to situations in which the maximum number of flows is fixed and known ahead of time. It is not practical

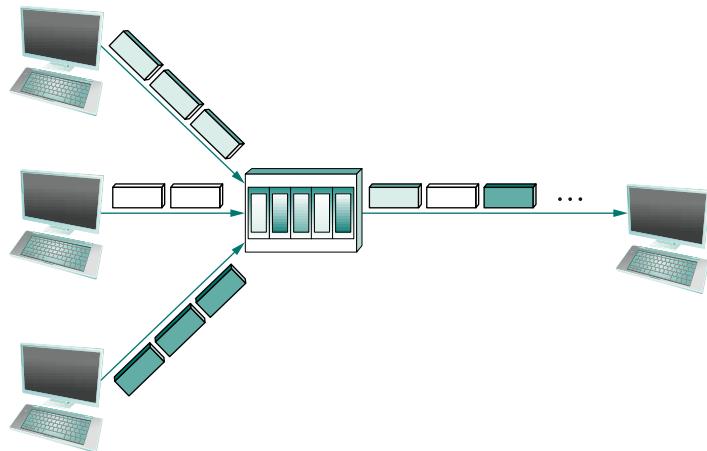
to resize the quantum or to add additional quanta in the case of STDM or to add new frequencies in the case of FDM.

The form of multiplexing that addresses these shortcomings, and of which we make most use in this book, is called *statistical multiplexing*. Although the name is not all that helpful for understanding the concept, statistical multiplexing is really quite simple, with two key ideas. First, it is like STDM in that the physical link is shared over time—first data from one flow is transmitted over the physical link, then data from another flow is transmitted, and so on. Unlike STDM, however, data is transmitted from each flow on demand rather than during a predetermined time slot. Thus, if only one flow has data to send, it gets to transmit that data without waiting for its quantum to come around and thus without having to watch the quanta assigned to the other flows go by unused. It is this avoidance of idle time that gives packet switching its efficiency.

As defined so far, however, statistical multiplexing has no mechanism to ensure that all the flows eventually get their turn to transmit over the physical link. That is, once a flow begins sending data, we need some way to limit the transmission, so that the other flows can have a turn. To account for this need, statistical multiplexing defines an upper bound on the size of the block of data that each flow is permitted to transmit at a given time. This limited-size block of data is typically referred to as a *packet*, to distinguish it from the arbitrarily large *message* that an application program might want to transmit. Because a packet-switched network limits the maximum size of packets, a host may not be able to send a complete message in one packet. The source may need to fragment the message into several packets, with the receiver reassembling the packets back into the original message.

In other words, each flow sends a sequence of packets over the physical link, with a decision made on a packet-by-packet basis as to which flow's packet to send next. Notice that, if only one flow has data to send, then it can send a sequence of packets back-to-back; however, should more than one of the flows have data to send, then their packets are interleaved on the link. [Figure 1.6](#) depicts a switch multiplexing packets from multiple sources onto a single shared link.

The decision as to which packet to send next on a shared link can be made in a number of different ways. For example, in a network consisting of switches interconnected by links such as the one in [Figure 1.5](#), the



■ **FIGURE 1.6** A switch multiplexing packets from multiple sources onto one shared link.

decision would be made by the switch that transmits packets onto the shared link. (As we will see later, not all packet-switched networks actually involve switches, and they may use other mechanisms to determine whose packet goes onto the link next.) Each switch in a packet-switched network makes this decision independently, on a packet-by-packet basis. One of the issues that faces a network designer is how to make this decision in a fair manner. For example, a switch could be designed to service packets on a first-in, first-out (FIFO) basis. Another approach would be to transmit the packets from each of the different flows that are currently sending data through the switch in a round-robin manner. This might be done to ensure that certain flows receive a particular share of the link's bandwidth or that they never have their packets delayed in the switch for more than a certain length of time. A network that attempts to allocate bandwidth to particular flows is sometimes said to support *quality of service* (QoS), a topic that we return to in Chapter 6.

Also, notice in Figure 1.6 that since the switch has to multiplex three incoming packet streams onto one outgoing link, it is possible that the switch will receive packets faster than the shared link can accommodate. In this case, the switch is forced to buffer these packets in its memory. Should a switch receive packets faster than it can send them for an extended period of time, then the switch will eventually run out of

buffer space, and some packets will have to be dropped. When a switch is operating in this state, it is said to be *congested*.

The bottom line is that statistical multiplexing defines a cost-effective way for multiple users (e.g., host-to-host flows of data) to share network resources (links and nodes) in a fine-grained manner. It defines the packet as the granularity with which the links of the network are allocated to different flows, with each switch able to schedule the use of the physical links it is connected to on a per-packet basis. Fairly allocating link capacity to different flows and dealing with congestion when it occurs are the key challenges of statistical multiplexing.

### SANs, LANs, MANs, and WANs

One way to characterize networks is according to their size. Two well-known examples are local area networks (LANs) and wide area networks (WANs); the former typically extend less than 1 km, while the latter can be worldwide. Other networks are classified as metropolitan area networks (MANs), which usually span tens of kilometers. The reason such classifications are interesting is that the size of a network often has implications for the underlying technology that can be used, with a key factor being the amount of time it takes for data to propagate from one end of the network to the other; we discuss this issue more in later chapters.

An interesting historical note is that the term *wide area network* was not applied to the first WANs because there was no other sort of network to differentiate them from. When computers were incredibly rare and expensive, there was no point in thinking about how to connect all the computers in the local area—there was only one computer in that area. Only as computers began to proliferate did LANs become necessary, and the term “WAN” was then introduced to describe the larger networks that interconnected geographically distant computers.

Another kind of network that we need to be aware of is SANs (usually now expanded as *storage area networks*, but formerly also known as *system area networks*). SANs are usually confined to a single room and connect the various components of a large computing system. For example, Fibre Channel is a common SAN technology used to connect high-performance computing systems to storage servers and data vaults. Although this book does not describe such networks in detail, they are worth knowing about because they are often at the leading edge in terms of performance, and because it is increasingly common to connect such networks into LANs and WANs.

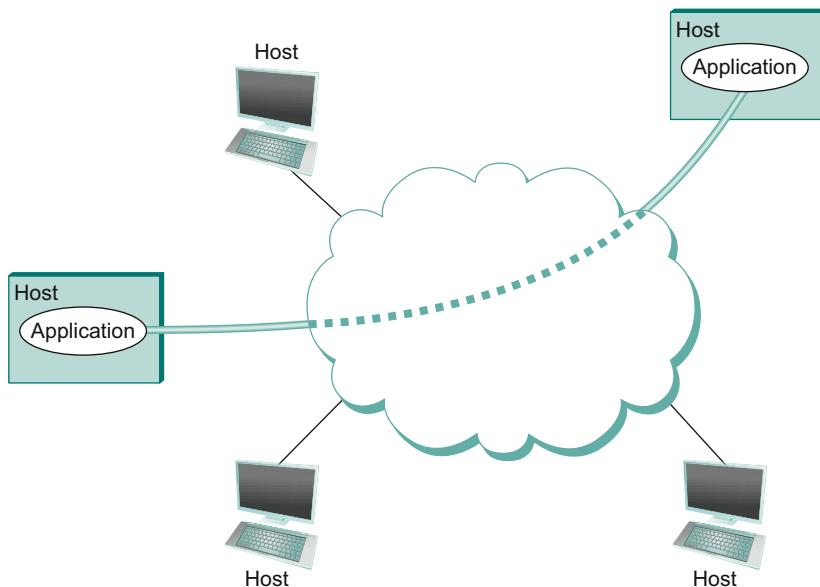
### 1.2.4 Support for Common Services

The previous section outlined the challenges involved in providing cost-effective connectivity among a group of hosts, but it is overly simplistic to view a computer network as simply delivering packets among a collection of computers. It is more accurate to think of a network as providing the means for a set of application processes that are distributed over those computers to communicate. In other words, the next requirement of a computer network is that the application programs running on the hosts connected to the network must be able to communicate in a meaningful way. From the application developer's perspective, the network needs to make his or her life easier.

When two application programs need to communicate with each other, a lot of complicated things must happen beyond simply sending a message from one host to another. One option would be for application designers to build all that complicated functionality into each application program. However, since many applications need common services, it is much more logical to implement those common services once and then to let the application designer build the application using those services. The challenge for a network designer is to identify the right set of common services. The goal is to hide the complexity of the network from the application without overly constraining the application designer.

Intuitively, we view the network as providing logical *channels* over which application-level processes can communicate with each other; each channel provides the set of services required by that application. In other words, just as we use a cloud to abstractly represent connectivity among a set of computers, we now think of a channel as connecting one process to another. Figure 1.7 shows a pair of application-level processes communicating over a logical channel that is, in turn, implemented on top of a cloud that connects a set of hosts. We can think of the channel as being like a pipe connecting two applications, so that a sending application can put data in one end and expect that data to be delivered by the network to the application at the other end of the pipe.

The challenge is to recognize what functionality the channels should provide to application programs. For example, does the application require a guarantee that messages sent over the channel are delivered, or is it acceptable if some messages fail to arrive? Is it necessary that messages arrive at the recipient process in the same order in which they are sent, or does the recipient not care about the order in which messages



■ FIGURE 1.7 Processes communicating over an abstract channel.

arrive? Does the network need to ensure that no third parties are able to eavesdrop on the channel, or is privacy not a concern? In general, a network provides a variety of different types of channels, with each application selecting the type that best meets its needs. The rest of this section illustrates the thinking involved in defining useful channels.

#### *Identifying Common Communication Patterns*

Designing abstract channels involves first understanding the communication needs of a representative collection of applications, then extracting their common communication requirements, and finally incorporating the functionality that meets these requirements in the network.

One of the earliest applications supported on any network is a file access program like the File Transfer Protocol (FTP) or Network File System (NFS). Although many details vary—for example, whether whole files are transferred across the network or only single blocks of the file are read/written at a given time—the communication component of remote file access is characterized by a pair of processes, one that requests that a file be read or written and a second process that honors this request. The

process that requests access to the file is called the *client*, and the process that supports access to the file is called the *server*.

Reading a file involves the client sending a small request message to a server and the server responding with a large message that contains the data in the file. Writing works in the opposite way—the client sends a large message containing the data to be written to the server, and the server responds with a small message confirming that the write to disk has taken place.

A digital library is a more sophisticated application than file transfer, but it requires similar communication services. For example, the *Association for Computing Machinery* (ACM) operates a large digital library of computer science literature at

<http://portal.acm.org/dl.cfm>

This library has a wide range of searching and browsing features to help users find the articles they want, but ultimately much of what it does is respond to user requests for files, such as electronic copies of journal articles, much like an FTP server.

Using file access, a digital library, and the two video applications described in the introduction (videoconferencing and video on demand) as a representative sample, we might decide to provide the following two types of channels: *request/reply* channels and *message stream* channels. The request/reply channel would be used by the file transfer and digital library applications. It would guarantee that every message sent by one side is received by the other side and that only one copy of each message is delivered. The request/reply channel might also protect the privacy and integrity of the data that flows over it, so that unauthorized parties cannot read or modify the data being exchanged between the client and server processes.

The message stream channel could be used by both the video on demand and videoconferencing applications, provided it is parameterized to support both one-way and two-way traffic and to support different delay properties. The message stream channel might not need to guarantee that all messages are delivered, since a video application can operate adequately even if some video frames are not received. It would, however, need to ensure that those messages that are delivered arrive in the same order in which they were sent, to avoid displaying frames out of sequence. Like the request/reply channel, the message stream channel might want

to ensure the privacy and integrity of the video data. Finally, the message stream channel might need to support multicast, so that multiple parties can participate in the teleconference or view the video.

While it is common for a network designer to strive for the smallest number of abstract channel types that can serve the largest number of applications, there is a danger in trying to get away with too few channel abstractions. Simply stated, if you have a hammer, then everything looks like a nail. For example, if all you have are message stream and request/reply channels, then it is tempting to use them for the next application that comes along, even if neither type provides exactly the semantics needed by the application. Thus, network designers will probably be inventing new types of channels—and adding options to existing channels—for as long as application programmers are inventing new applications.

Also note that independent of exactly *what* functionality a given channel provides, there is the question of *where* that functionality is implemented. In many cases, it is easiest to view the host-to-host connectivity of the underlying network as simply providing a *bit pipe*, with any high-level communication semantics provided at the end hosts. The advantage of this approach is that it keeps the switches in the middle of the network as simple as possible—they simply forward packets—but it requires the end hosts to take on much of the burden of supporting semantically rich process-to-process channels. The alternative is to push additional functionality onto the switches, thereby allowing the end hosts to be “dumb” devices (e.g., telephone handsets). We will see this question of how various network services are partitioned between the packet switches and the end hosts (devices) as a recurring issue in network design.

### *Reliability*

As suggested by the examples just considered, reliable message delivery is one of the most important functions that a network can provide. It is difficult to determine how to provide this reliability, however, without first understanding how networks can fail. The first thing to recognize is that computer networks do not exist in a perfect world. Machines crash and later are rebooted, fibers are cut, electrical interference corrupts bits in the data being transmitted, switches run out of buffer space, and, as if these sorts of physical problems aren’t enough to worry about, the software that manages the hardware may contain bugs and sometimes forwards packets into oblivion. Thus, a major requirement of a network

is to recover from certain kinds of failures, so that application programs don't have to deal with them or even be aware of them.

There are three general classes of failure that network designers have to worry about. First, as a packet is transmitted over a physical link, *bit errors* may be introduced into the data; that is, a 1 is turned into a 0 or *vice versa*. Sometimes single bits are corrupted, but more often than not a *burst error* occurs—several consecutive bits are corrupted. Bit errors typically occur because outside forces, such as lightning strikes, power surges, and microwave ovens, interfere with the transmission of data. The good news is that such bit errors are fairly rare, affecting on average only one out of every  $10^6$  to  $10^7$  bits on a typical copper-based cable and one out of every  $10^{12}$  to  $10^{14}$  bits on a typical optical fiber. As we will see, there are techniques that detect these bit errors with high probability. Once detected, it is sometimes possible to correct for such errors—if we know which bit or bits are corrupted, we can simply flip them—while in other cases the damage is so bad that it is necessary to discard the entire packet. In such a case, the sender may be expected to retransmit the packet.

The second class of failure is at the packet, rather than the bit, level; that is, a complete packet is lost by the network. One reason this can happen is that the packet contains an uncorrectable bit error and therefore has to be discarded. A more likely reason, however, is that one of the nodes that has to handle the packet—for example, a switch that is forwarding it from one link to another—is so overloaded that it has no place to store the packet and therefore is forced to drop it. This is the problem of congestion mentioned in Section 1.2.3. Less commonly, the software running on one of the nodes that handles the packet makes a mistake. For example, it might incorrectly forward a packet out on the wrong link, so that the packet never finds its way to the ultimate destination. As we will see, one of the main difficulties in dealing with lost packets is distinguishing between a packet that is indeed lost and one that is merely late in arriving at the destination.

The third class of failure is at the node and link level; that is, a physical link is cut, or the computer it is connected to crashes. This can be caused by software that crashes, a power failure, or a reckless backhoe operator. Failures due to misconfiguration of a network device are also common. While any of these failures can eventually be corrected, they can have a dramatic effect on the network for an extended period of time. However, they need not totally disable the network. In a packet-switched network,

for example, it is sometimes possible to route around a failed node or link. One of the difficulties in dealing with this third class of failure is distinguishing between a failed computer and one that is merely slow or, in the case of a link, between one that has been cut and one that is very flaky and therefore introducing a high number of bit errors.

The key idea to take away from this discussion is that defining useful channels involves both understanding the applications' requirements and recognizing the limitations of the underlying technology. The challenge is to fill in the gap between what the application expects and what the underlying technology can provide. This is sometimes called the *semantic gap*.

### 1.2.5 Manageability

A final requirement, which seems to be neglected or left till last all too often,<sup>3</sup> is that networks need to be managed. Managing a network includes making changes as the network grows to carry more traffic or reach more users, and troubleshooting the network when things go wrong or performance isn't as desired.

This requirement is partly related to the issue of scalability discussed above—as the Internet has scaled up to support billions of users and at least hundreds of millions of hosts, the challenges of keeping the whole thing running correctly and correctly configuring new devices as they are added have become increasingly problematic. Configuring a single router in a network is often a task for a trained expert; configuring thousands of routers and figuring out why a network of such a size is not behaving as expected can become a task beyond any single human. Furthermore, to make the operation of a network scalable and cost-effective, network operators typically require many management tasks to be automated or at least performed by relatively unskilled personnel.

An important development in networking since we wrote the first edition of this book is that networks in the home are now commonplace. This means that network management is no longer the province of experts but needs to be accomplished by consumers with little to no special training. This is sometimes stated as a requirement that networking devices should be “plug-and-play”—a goal that has proven quite elusive. We will discuss

<sup>3</sup>As we have done in this section.

some ways that this requirement has been addressed in part later on, but it is worth noting for now that improving the manageability of networks remains an important area of current research.

### 1.3 NETWORK ARCHITECTURE

In case you hadn't noticed, the previous section established a pretty substantial set of requirements for network design—a computer network must provide general, cost-effective, fair, and robust connectivity among a large number of computers. As if this weren't enough, networks do not remain fixed at any single point in time but must evolve to accommodate changes in both the underlying technologies upon which they are based as well as changes in the demands placed on them by application programs. Furthermore, networks must be manageable by humans of varying levels of skill. Designing a network to meet these requirements is no small task.

To help deal with this complexity, network designers have developed general blueprints—usually called *network architectures*—that guide the design and implementation of networks. This section defines more carefully what we mean by a network architecture by introducing the central ideas that are common to all network architectures. It also introduces two of the most widely referenced architectures—the OSI (or 7-layer) architecture and the Internet architecture.

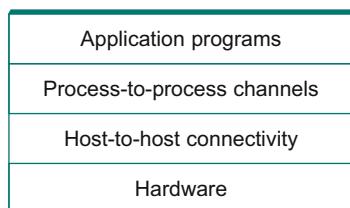
#### 1.3.1 Layering and Protocols

Abstraction—the hiding of details behind a well-defined interface—is the fundamental tool used by system designers to manage complexity. The idea of an abstraction is to define a model that can capture some important aspect of the system, encapsulate this model in an object that provides an interface that can be manipulated by other components of the system, and hide the details of how the object is implemented from the users of the object. The challenge is to identify abstractions that simultaneously provide a service that proves useful in a large number of situations and that can be efficiently implemented in the underlying system. This is exactly what we were doing when we introduced the idea of a channel in the previous section: we were providing an abstraction for applications that hides the complexity of the network from application writers.

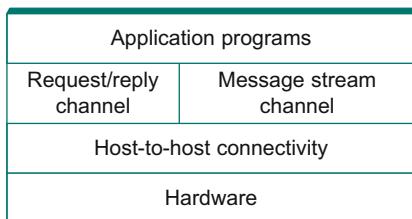
Abstractions naturally lead to layering, especially in network systems. The general idea is that you start with the services offered by the underlying hardware and then add a sequence of layers, each providing a higher (more abstract) level of service. The services provided at the high layers are implemented in terms of the services provided by the low layers. Drawing on the discussion of requirements given in the previous section, for example, we might imagine a simple network as having two layers of abstraction sandwiched between the application program and the underlying hardware, as illustrated in Figure 1.8. The layer immediately above the hardware in this case might provide host-to-host connectivity, abstracting away the fact that there may be an arbitrarily complex network topology between any two hosts. The next layer up builds on the available host-to-host communication service and provides support for process-to-process channels, abstracting away the fact that the network occasionally loses messages, for example.

Layering provides two nice features. First, it decomposes the problem of building a network into more manageable components. Rather than implementing a monolithic piece of software that does everything you will ever want, you can implement several layers, each of which solves one part of the problem. Second, it provides a more modular design. If you decide that you want to add some new service, you may only need to modify the functionality at one layer, reusing the functions provided at all the other layers.

Thinking of a system as a linear sequence of layers is an oversimplification, however. Many times there are multiple abstractions provided at any given level of the system, each providing a different service to the higher layers but building on the same low-level abstractions. To see this, consider the two types of channels discussed in Section 1.2.4: One provides a



■ FIGURE 1.8 Example of a layered network system.



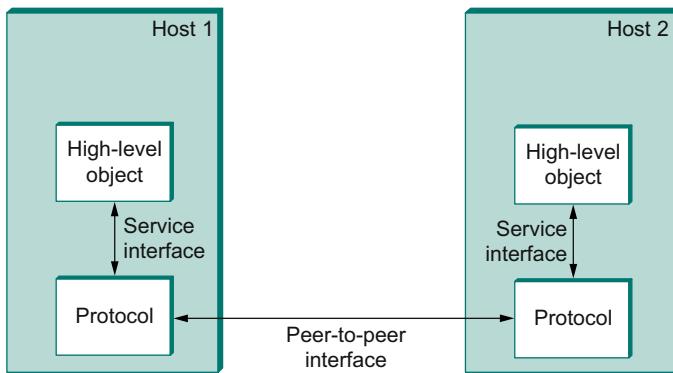
■ **FIGURE 1.9** Layered system with alternative abstractions available at a given layer.

request/reply service and one supports a message stream service. These two channels might be alternative offerings at some level of a multilevel networking system, as illustrated in Figure 1.9.

Using this discussion of layering as a foundation, we are now ready to discuss the architecture of a network more precisely. For starters, the abstract objects that make up the layers of a network system are called *protocols*. That is, a protocol provides a communication service that higher-level objects (such as application processes, or perhaps higher-level protocols) use to exchange messages. For example, we could imagine a network that supports a request/reply protocol and a message stream protocol, corresponding to the request/reply and message stream channels discussed above.

Each protocol defines two different interfaces. First, it defines a *service interface* to the other objects on the same computer that want to use its communication services. This service interface defines the operations that local objects can perform on the protocol. For example, a request/reply protocol would support operations by which an application can send and receive messages. An implementation of the HTTP protocol could support an operation to fetch a page of hypertext from a remote server. An application such as a web browser would invoke such an operation whenever the browser needs to obtain a new page (e.g., when the user clicks on a link in the currently displayed page).

Second, a protocol defines a *peer interface* to its counterpart (peer) on another machine. This second interface defines the form and meaning of messages exchanged between protocol peers to implement the communication service. This would determine the way in which a request/reply protocol on one machine communicates with its peer on another machine. In the case of HTTP, for example, the protocol specification



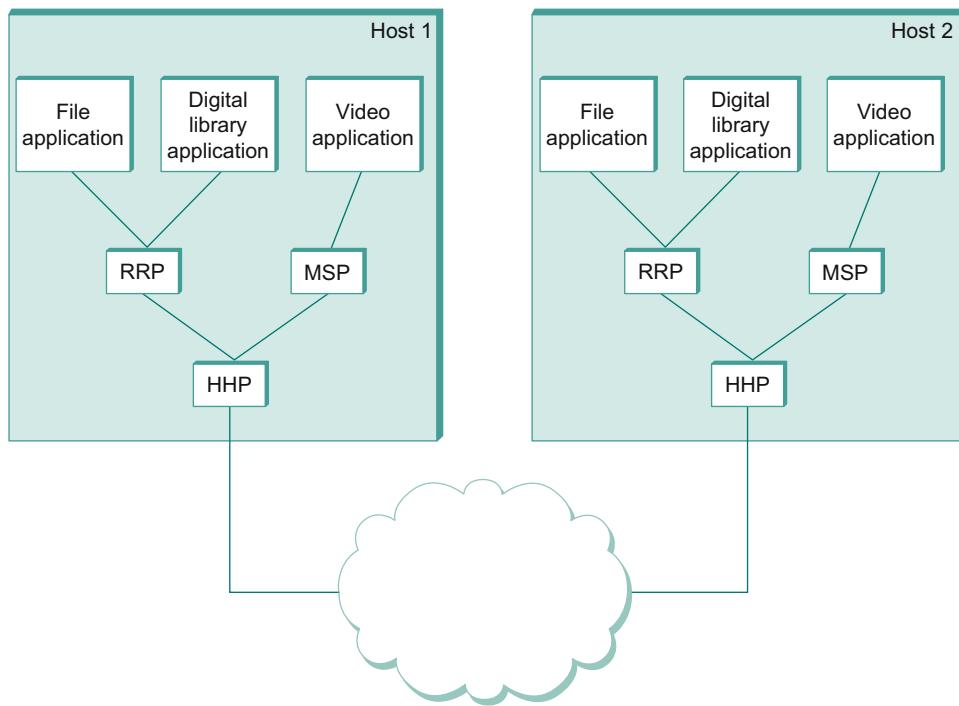
■ FIGURE 1.10 Service interfaces and peer interfaces.

defines in detail how a *GET* command is formatted, what arguments can be used with the command, and how a web server should respond when it receives such a command. (We will look more closely at this particular protocol in Section 9.1.2.)

To summarize, a protocol defines a communication service that it exports locally (the service interface), along with a set of rules governing the messages that the protocol exchanges with its peer(s) to implement this service (the peer interface). This situation is illustrated in Figure 1.10.

Except at the hardware level, where peers directly communicate with each other over a link, peer-to-peer communication is indirect—each protocol communicates with its peer by passing messages to some lower-level protocol, which in turn delivers the message to *its* peer. In addition, there are potentially multiple protocols at any given level, each providing a different communication service. We therefore represent the suite of protocols that make up a network system with a *protocol graph*. The nodes of the graph correspond to protocols, and the edges represent a *depends on* relation. For example, Figure 1.11 illustrates a protocol graph for the hypothetical layered system we have been discussing—protocols RRP (Request/Reply Protocol) and MSP (Message Stream Protocol) implement two different types of process-to-process channels, and both depend on the Host-to-Host Protocol (HHP) which provides a host-to-host connectivity service.

In this example, suppose that the file access program on host 1 wants to send a message to its peer on host 2 using the communication service



■ FIGURE 1.11 Example of a protocol graph.

offered by RRP. In this case, the file application asks RRP to send the message on its behalf. To communicate with its peer, RRP invokes the services of HHP, which in turn transmits the message to its peer on the other machine. Once the message has arrived at the instance of HHP on host 2, HHP passes the message up to RRP, which in turn delivers the message to the file application. In this particular case, the application is said to employ the services of the *protocol stack* RRP/HHP.

Note that the term *protocol* is used in two different ways. Sometimes it refers to the abstract interfaces—that is, the operations defined by the service interface and the form and meaning of messages exchanged between peers, and sometimes it refers to the module that actually implements these two interfaces. To distinguish between the interfaces and the module that implements these interfaces, we generally refer to the former as a *protocol specification*. Specifications are generally expressed using a combination of prose, pseudocode, state transition diagrams, pictures of

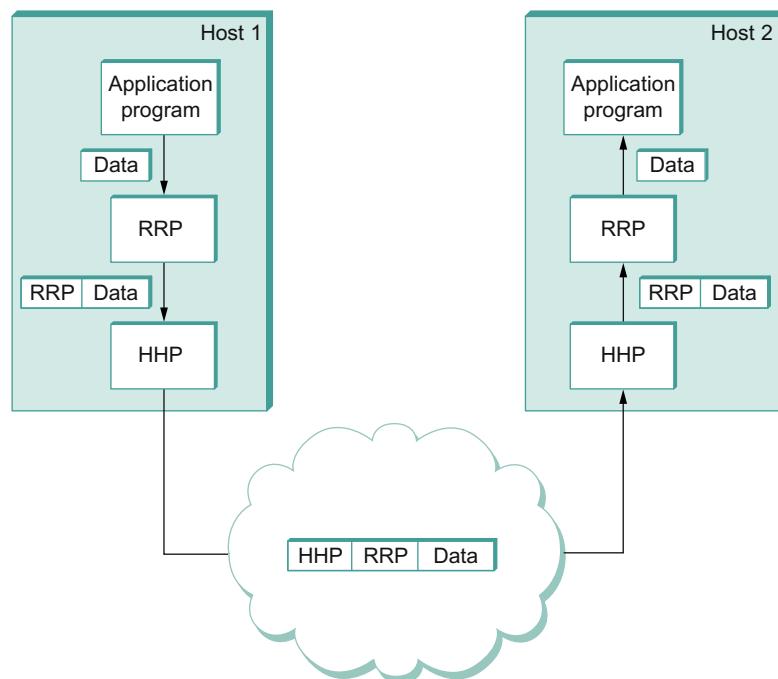
packet formats, and other abstract notations. It should be the case that a given protocol can be implemented in different ways by different programmers, as long as each adheres to the specification. The challenge is ensuring that two different implementations of the same specification can successfully exchange messages. Two or more protocol modules that do accurately implement a protocol specification are said to *interoperate* with each other.

We can imagine many different protocols and protocol graphs that satisfy the communication requirements of a collection of applications. Fortunately, there exist standardization bodies, such as the Internet Engineering Task Force (IETF) and the International Standards Organization (ISO), that establish policies for a particular protocol graph. We call the set of rules governing the form and content of a protocol graph a *network architecture*. Although beyond the scope of this book, standardization bodies have established well-defined procedures for introducing, validating, and finally approving protocols in their respective architectures. We briefly describe the architectures defined by the IETF and ISO shortly, but first there are two additional things we need to explain about the mechanics of protocol layering.

### *Encapsulation*

Consider what happens in Figure 1.11 when one of the application programs sends a message to its peer by passing the message to RRP. From RRP's perspective, the message it is given by the application is an uninterpreted string of bytes. RRP does not care that these bytes represent an array of integers, an email message, a digital image, or whatever; it is simply charged with sending them to its peer. However, RRP must communicate control information to its peer, instructing it how to handle the message when it is received. RRP does this by attaching a *header* to the message. Generally speaking, a header is a small data structure—from a few bytes to a few dozen bytes—that is used among peers to communicate with each other. As the name suggests, headers are usually attached to the front of a message. In some cases, however, this peer-to-peer control information is sent at the end of the message, in which case it is called a *trailer*. The exact format for the header attached by RRP is defined by its protocol specification. The rest of the message—that is, the data being transmitted on behalf of the application—is called the message's *body* or *payload*. We say that the application's data is *encapsulated* in the new message created by RRP.

This process of encapsulation is then repeated at each level of the protocol graph; for example, HHP encapsulates RRP's message by attaching a header of its own. If we now assume that HHP sends the message to its peer over some network, then when the message arrives at the destination host, it is processed in the opposite order: HHP first interprets the HHP header at the front of the message (i.e., takes whatever action is appropriate given the contents of the header) and passes the body of the message (but not the HHP header) up to RRP, which takes whatever action is indicated by the RRP header that its peer attached and passes the body of the message (but not the RRP header) up to the application program. The message passed up from RRP to the application on host 2 is exactly the same message as the application passed down to RRP on host 1; the application does not see any of the headers that have been attached to it to implement the lower-level communication services. This whole process is illustrated in Figure 1.12. Note that in this example, nodes in the



■ FIGURE 1.12 High-level messages are encapsulated inside of low-level messages.

network (e.g., switches and routers) may inspect the HHP header at the front of the message.

Note that when we say a low-level protocol does not interpret the message it is given by some high-level protocol, we mean that it does not know how to extract any meaning from the data contained in the message. It is sometimes the case, however, that the low-level protocol applies some simple transformation to the data it is given, such as to compress or encrypt it. In this case, the protocol is transforming the entire body of the message, including both the original application's data and all the headers attached to that data by higher-level protocols.

### *Multiplexing and Demultiplexing*

Recall from Section 1.2.3 that a fundamental idea of packet switching is to multiplex multiple flows of data over a single physical link. This same idea applies up and down the protocol graph, not just to switching nodes. In Figure 1.11, for example, we can think of RRP as implementing a logical communication channel, with messages from two different applications multiplexed over this channel at the source host and then demultiplexed back to the appropriate application at the destination host.

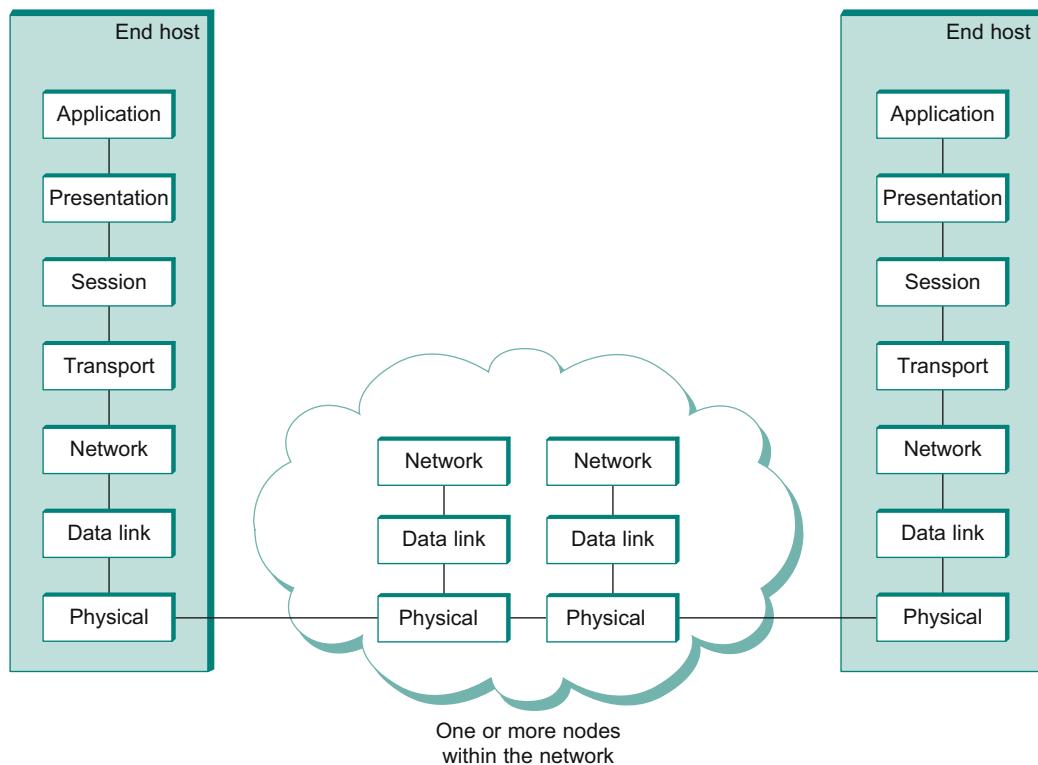
Practically speaking, this simply means that the header that RRP attaches to its messages contains an identifier that records the application to which the message belongs. We call this identifier RRP's *demultiplexing key*, or *demux key* for short. At the source host, RRP includes the appropriate demux key in its header. When the message is delivered to RRP on the destination host, it strips its header, examines the demux key, and demultiplexes the message to the correct application.

RRP is not unique in its support for multiplexing; nearly every protocol implements this mechanism. For example, HHP has its own demux key to determine which messages to pass up to RRP and which to pass up to MSP. However, there is no uniform agreement among protocols—even those within a single network architecture—on exactly what constitutes a demux key. Some protocols use an 8-bit field (meaning they can support only 256 high-level protocols), and others use 16- or 32-bit fields. Also, some protocols have a single demultiplexing field in their header, while others have a pair of demultiplexing fields. In the former case, the same demux key is used on both sides of the communication, while in the latter case each side uses a different key to identify the high-level protocol (or application program) to which the message is to be delivered.

### The 7-Layer Model

The ISO was one of the first organizations to formally define a common way to connect computers. Their architecture, called the *Open Systems Interconnection* (OSI) architecture and illustrated in Figure 1.13, defines a partitioning of network functionality into seven layers, where one or more protocols implement the functionality assigned to a given layer. In this sense, the schematic given in Figure 1.13 is not a protocol graph, *per se*, but rather a *reference model* for a protocol graph. It is often referred to as the 7-layer model.

Starting at the bottom and working up, the *physical* layer handles the transmission of raw bits over a communications link. The *data link* layer then collects a stream of bits into a larger aggregate called a *frame*. Network adaptors, along with device drivers running in the node's operating



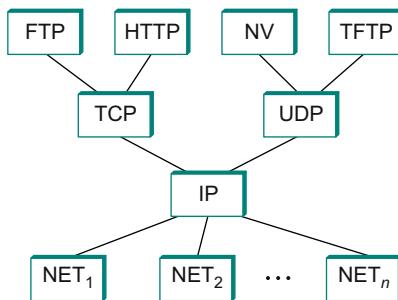
■ FIGURE 1.13 The OSI 7-layer model.

system, typically implement the data link level. This means that frames, not raw bits, are actually delivered to hosts. The *network* layer handles routing among nodes within a packet-switched network. At this layer, the unit of data exchanged among nodes is typically called a *packet* rather than a frame, although they are fundamentally the same thing. The lower three layers are implemented on all network nodes, including switches within the network and hosts connected to the exterior of the network. The *transport* layer then implements what we have up to this point been calling a *process-to-process channel*. Here, the unit of data exchanged is commonly called a *message* rather than a packet or a frame. The transport layer and higher layers typically run only on the end hosts and not on the intermediate switches or routers.

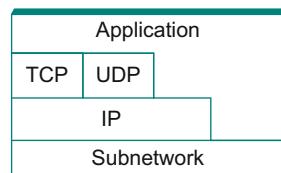
There is less agreement about the definition of the top three layers, in part because they are not always all present, as we will see below. Skipping ahead to the top (seventh) layer, we find the *application* layer. Application layer protocols include things like the Hypertext Transfer Protocol (HTTP), which is the basis of the World Wide Web and is what enables web browsers to request pages from web servers. Below that, the *presentation* layer is concerned with the format of data exchanged between peers—for example, whether an integer is 16, 32, or 64 bits long, whether the most significant byte is transmitted first or last, or how a video stream is formatted. Finally, the *session* layer provides a name space that is used to tie together the potentially different transport streams that are part of a single application. For example, it might manage an audio stream and a video stream that are being combined in a teleconferencing application.

### 1.3.2 Internet Architecture

The Internet architecture, which is also sometimes called the TCP/IP architecture after its two main protocols, is depicted in Figure 1.14. An alternative representation is given in Figure 1.15. The Internet architecture evolved out of experiences with an earlier packet-switched network called the ARPANET. Both the Internet and the ARPANET were funded by the Advanced Research Projects Agency (ARPA), one of the research and development funding agencies of the U.S. Department of Defense. The Internet and ARPANET were around before the OSI architecture, and the experience gained from building them was a major influence on the OSI reference model.



■ FIGURE 1.14 Internet protocol graph.



■ FIGURE 1.15 Alternative view of the Internet architecture. The *subnetwork* layer was historically referred to as the *network* layer and is now often referred to as the layer or simply *layer 2*.

While the 7-layer OSI model can, with some imagination, be applied to the Internet, a 4-layer model is often used instead. At the lowest level is a wide variety of network protocols, denoted NET<sub>1</sub>, NET<sub>2</sub>, and so on. In practice, these protocols are implemented by a combination of hardware (e.g., a network adaptor) and software (e.g., a network device driver). For example, you might find Ethernet or wireless protocols (such as the 802.11 Wi-Fi standards) at this layer. (These protocols in turn may actually involve several sublayers, but the Internet architecture does not presume anything about them.) The second layer consists of a single protocol—the *Internet Protocol* (IP). This is the protocol that supports the interconnection of multiple networking technologies into a single, logical internetwork. The third layer contains two main protocols—the *Transmission Control Protocol* (TCP) and the *User Datagram Protocol* (UDP). TCP and UDP provide alternative logical channels to application programs: TCP provides a reliable byte-stream channel, and UDP provides an unreliable datagram delivery channel (*datagram* may be thought of as a synonym for message). In the language of the Internet, TCP and UDP

are sometimes called *end-to-end* protocols, although it is equally correct to refer to them as *transport* protocols.

Running above the transport layer is a range of application protocols, such as HTTP, FTP, Telnet (remote login), and the Simple Mail Transfer Protocol (SMTP), that enable the interoperation of popular applications. To understand the difference between an application layer protocol and an application, think of all the different World Wide Web browsers that are or have been available (e.g., Firefox, Safari, Netscape, Mosaic, Internet Explorer). There is a similarly large number of different implementations of web servers. The reason that you can use any one of these application programs to access a particular site on the Web is that they all conform to the same application layer protocol: HTTP. Confusingly, the same term sometimes applies to both an application and the application layer protocol that it uses (e.g., FTP is often used as the name of an application that implements the FTP protocol).

Most people who work actively in the networking field are familiar with both the Internet architecture and the 7-layer OSI architecture, and there is general agreement on how the layers map between architectures. The Internet's application layer is considered to be at layer 7, its transport layer is layer 4, the IP (internetworking or just network) layer is layer 3, and the link or subnet layer below IP is layer 2.

The Internet architecture has three features that are worth highlighting. First, as best illustrated by Figure 1.15, the Internet architecture does not imply strict layering. The application is free to bypass the defined transport layers and to directly use IP or one of the underlying networks. In fact, programmers are free to define new channel abstractions or applications that run on top of any of the existing protocols.

Second, if you look closely at the protocol graph in Figure 1.14, you will notice an hourglass shape—wide at the top, narrow in the middle, and wide at the bottom. This shape actually reflects the central philosophy of the architecture. That is, IP serves as the focal point for the architecture—it defines a common method for exchanging packets among a wide collection of networks. Above IP there can be arbitrarily many transport protocols, each offering a different channel abstraction to application programs. Thus, the issue of delivering messages from host to host is completely separated from the issue of providing a useful process-to-process communication service. Below IP, the architecture allows for arbitrarily

many different network technologies, ranging from Ethernet to wireless to single point-to-point links.

A final attribute of the Internet architecture (or more accurately, of the IETF culture) is that in order for a new protocol to be officially included in the architecture, there must be both a protocol specification and at least one (and preferably two) representative implementations of the specification. The existence of working implementations is required for standards to be adopted by the IETF. This cultural assumption of the design community helps to ensure that the architecture's protocols can be efficiently implemented. Perhaps the value the Internet culture places on working software is best exemplified by a quote on T-shirts commonly worn at IETF meetings:

*We reject kings, presidents, and voting. We believe in rough consensus and running code.*

(David Clark)



Of these three attributes of the Internet architecture, the hourglass design philosophy is important enough to bear repeating. The hourglass's narrow waist represents a minimal and carefully chosen set of global capabilities that allows both higher-level applications and lower-level communication technologies to coexist, share capabilities, and evolve rapidly. The narrow-waisted model is critical to the Internet's ability to adapt rapidly to new user demands and changing technologies.

#### 1.4 IMPLEMENTING NETWORK SOFTWARE

Network architectures and protocol specifications are essential things, but a good blueprint is not enough to explain the phenomenal success of the Internet: The number of computers connected to the Internet has grown exponentially for almost 3 decades (although precise numbers are now hard to come by). The number of users of the Internet was estimated to be around 1.8 billion in 2009—an impressive percentage of the world's population.

What explains the success of the Internet? There are certainly many contributing factors (including a good architecture), but one thing that has made the Internet such a runaway success is the fact that so much of its functionality is provided by software running in general-purpose computers. The significance of this is that new functionality can be added readily with "just a small matter of programming." As a result, new

applications and services—electronic commerce, videoconferencing, and IP telephony, to name a few—have been showing up at an incredible pace.

A related factor is the massive increase in computing power available in commodity machines. Although computer networks have always been capable in principle of transporting any kind of information, such as digital voice samples, digitized images, and so on, this potential was not particularly interesting if the computers sending and receiving that data were too slow to do anything useful with the information. Virtually all of today's computers are capable of playing back digitized voice at full speed and can display video at a speed and resolution that are useful for some (but by no means all) applications. Thus, today's networks are increasingly used to carry multimedia, and their support for it will only improve as computing hardware becomes faster.

In the years since the first edition of this book appeared, the writing of networked applications has become a much more mainstream activity and less a job just for a few specialists. Many factors have played into this, including better tools to make the job easier for nonspecialists and the opening up of new markets such as applications for smartphones.

The point to note is that knowing how to implement network software is an essential part of understanding computer networks, and while the odds are you will not be tasked to implement a low-level protocol like IP, there is a good chance you will find reason to implement an application-level protocol—the elusive “killer app” that will lead to unimaginable fame and fortune. To get you started, this section introduces some of the issues involved in implementing a network application on top of the Internet. Typically, such programs are simultaneously an application (i.e., designed to interact with users) and a protocol (i.e., communicates with peers across the network). Chapter 9 concludes the book by returning to the topic of network applications (application-level protocols) by exploring several popular examples.

### 1.4.1 Application Programming Interface (Sockets)

The place to start when implementing a network application is the interface exported by the network. Since most network protocols are implemented in software (especially those high in the protocol stack), and nearly all computer systems implement their network protocols as part of the operating system, when we refer to the interface “exported by the network,” we are generally referring to the interface that the OS provides

to its networking subsystem. This interface is often called the network *application programming interface* (API).

Although each operating system is free to define its own network API (and most have), over time certain of these APIs have become widely supported; that is, they have been ported to operating systems other than their native system. This is what has happened with the *socket interface* originally provided by the Berkeley distribution of Unix, which is now supported in virtually all popular operating systems, and is the foundation of language-specific interfaces, such as the Java socket library. The advantages of industry-wide support for a single API are that applications can be easily ported from one OS to another and developers can easily write applications for multiple operating systems.

Before describing the socket interface, it is important to keep two concerns separate in your mind. Each protocol provides a certain set of *services*, and the API provides a *syntax* by which those services can be invoked on a particular computer system. The implementation is then responsible for mapping the tangible set of operations and objects defined by the API onto the abstract set of services defined by the protocol. If you have done a good job of defining the interface, then it will be possible to use the syntax of the interface to invoke the services of many different protocols. Such generality was certainly a goal of the socket interface, although it's far from perfect.

The main abstraction of the socket interface, not surprisingly, is the *socket*. A good way to think of a socket is as the point where a local application process attaches to the network. The interface defines operations for creating a socket, attaching the socket to the network, sending/receiving messages through the socket, and closing the socket. To simplify the discussion, we will limit ourselves to showing how sockets are used with TCP.

The first step is to create a socket, which is done with the following operation:

```
int socket(int domain, int type, int protocol)
```

The reason that this operation takes three arguments is that the socket interface was designed to be general enough to support any underlying protocol suite. Specifically, the domain argument specifies the protocol *family* that is going to be used: PF\_INET denotes the Internet family, PF\_UNIX denotes the Unix pipe facility, and PF\_PACKET denotes direct

access to the network interface (i.e., it bypasses the TCP/IP protocol stack). The type argument indicates the semantics of the communication. `SOCK_STREAM` is used to denote a byte stream. `SOCK_DGRAM` is an alternative that denotes a message-oriented service, such as that provided by UDP. The protocol argument identifies the specific protocol that is going to be used. In our case, this argument is `UNSPEC` because the combination of `PF_INET` and `SOCK_STREAM` implies TCP. Finally, the return value from `socket` is a *handle* for the newly created socket—that is, an identifier by which we can refer to the socket in the future. It is given as an argument to subsequent operations on this socket.

The next step depends on whether you are a client or a server. On a server machine, the application process performs a *passive* open—the server says that it is prepared to accept connections, but it does not actually establish a connection. The server does this by invoking the following three operations:

```
int bind(int socket, struct sockaddr *address, int addr_len)
int listen(int socket, int backlog)
int accept(int socket, struct sockaddr *address, int *addr_len)
```

The `bind` operation, as its name suggests, binds the newly created socket to the specified address. This is the network address of the *local* participant—the server. Note that, when used with the Internet protocols, `address` is a data structure that includes both the IP address of the server and a TCP port number. (As we will see in [Chapter 5](#), ports are used to indirectly identify processes. They are a form of *demux keys* as defined in [Section 1.3.1](#).) The port number is usually some well-known number specific to the service being offered; for example, web servers commonly accept connections on port 80.

The `listen` operation then defines how many connections can be pending on the specified socket. Finally, the `accept` operation carries out the passive open. It is a blocking operation that does not return until a remote participant has established a connection, and when it does complete it returns a *new* socket that corresponds to this just-established connection, and the `address` argument contains the *remote* participant's address. Note that when `accept` returns, the original socket that was given as an argument still exists and still corresponds to the passive open; it is used in future invocations of `accept`.

On the client machine, the application process performs an *active* open; that is, it says who it wants to communicate with by invoking the following single operation:

```
int connect(int socket, struct sockaddr *address, int addr_len)
```

This operation does not return until TCP has successfully established a connection, at which time the application is free to begin sending data. In this case, `address` contains the remote participant's address. In practice, the client usually specifies only the remote participant's address and lets the system fill in the local information. Whereas a server usually listens for messages on a well-known port, a client typically does not care which port it uses for itself; the OS simply selects an unused one.

Once a connection is established, the application processes invoke the following two operations to send and receive data:

```
int send(int socket, char *message, int msg_len, int flags)  
int recv(int socket, char *buffer, int buf_len, int flags)
```

The first operation sends the given `message` over the specified `socket`, while the second operation receives a message from the specified `socket` into the given `buffer`. Both operations take a set of `flags` that control certain details of the operation.

### 1.4.2 Example Application

We now show the implementation of a simple client/server program that uses the socket interface to send messages over a TCP connection. The program also uses other Unix networking utilities, which we introduce as we go. Our application allows a user on one machine to type in and send text to a user on another machine. It is a simplified version of the Unix `talk` program, which is similar to the program at the core of an instant messaging application.

#### *Client*

We start with the client side, which takes the name of the remote machine as an argument. It calls the Unix utility `gethostbyname` to translate this name into the remote host's IP address. The next step is to construct the address data structure (`sin`) expected by the socket interface. Notice that this data structure specifies that we'll be using the socket to connect to the Internet (`AF_INET`). In our example, we use TCP port 5432 as the well-known server port; this happens to be a port that has not been

assigned to any other Internet service. The final step in setting up the connection is to call `socket` and `connect`. Once the `connect` operation returns, the connection is established and the client program enters its main loop, which reads text from standard input and sends it over the socket.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 5432
#define MAX_LINE 256

int
main(int argc, char * argv[])
{
    FILE *fp;
    struct hostent *hp;
    struct sockaddr_in sin;
    char *host;
    char buf[MAX_LINE];
    int s;
    int len;

    if (argc==2) {
        host = argv[1];
    }
    else {
        fprintf(stderr, "usage: simplex-talk host\n");
        exit(1);
    }

    /* translate host name into peer's IP address */
    hp = gethostbyname(host);
    if (!hp) {
        fprintf(stderr, "simplex-talk: unknown host: %s\n", host);
        exit(1);
    }
```

```
/* build address data structure */
bzero((char *)&sin, sizeof(sin));
sin.sin_family = AF_INET;
bcopy(hp->h_addr, (char *)&sin.sin_addr, hp->h_length);
sin.sin_port = htons(SERVER_PORT);

/* active open */
if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("simplex-talk: socket");
    exit(1);
}
if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
{
    perror("simplex-talk: connect");
    close(s);
    exit(1);
}
/* main loop: get and send lines of text */
while (fgets(buf, sizeof(buf), stdin)) {
    buf[MAX_LINE-1] = '\0';
    len = strlen(buf) + 1;
    send(s, buf, len, 0);
}
```

### *Server*

The server is equally simple. It first constructs the address data structure by filling in its own port number (**SERVER.PORT**). By not specifying an IP address, the application program is willing to accept connections on any of the local host's IP addresses. Next, the server performs the preliminary steps involved in a passive open; it creates the socket, binds it to the local address, and sets the maximum number of pending connections to be allowed. Finally, the main loop waits for a remote host to try to connect, and when one does, it receives and prints out the characters that arrive on the connection.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 5432
#define MAX_PENDING 5
#define MAX_LINE    256

int
main()
{
    struct sockaddr_in sin;
    char buf[MAX_LINE];
    int len;
    int s, new_s;

    /* build address data structure */
    bzero((char *)&sin, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;
    sin.sin_port = htons(SERVER_PORT);

    /* setup passive open */
    if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("simplex-talk: socket");
        exit(1);
    }
    if ((bind(s, (struct sockaddr *)&sin, sizeof(sin))) < 0) {
        perror("simplex-talk: bind");
        exit(1);
    }
    listen(s, MAX_PENDING);

    /* wait for connection, then receive and print text */
    while(1) {
        if ((new_s = accept(s, (struct sockaddr *)&sin, &len)) < 0) {
            perror("simplex-talk: accept");
            exit(1);
        }
        while (len = recv(new_s, buf, sizeof(buf), 0))
```

```
        fputs(buf, stdout);
        close(new_s);
    }
}
```

## 1.5 PERFORMANCE

Up to this point, we have focused primarily on the functional aspects of network. Like any computer system, however, computer networks are also expected to perform well. This is because the effectiveness of computations distributed over the network often depends directly on the efficiency with which the network delivers the computation's data. While the old programming adage “first get it right and then make it fast” is valid in many settings, in networking it is usually necessary to “design for performance.” It is therefore important to understand the various factors that impact network performance.

### 1.5.1 Bandwidth and Latency

Network performance is measured in two fundamental ways: *bandwidth* (also called *throughput*) and *latency* (also called *delay*). The bandwidth of a network is given by the number of bits that can be transmitted over the network in a certain period of time. For example, a network might have a bandwidth of 10 million bits/second (Mbps), meaning that it is able to deliver 10 million bits every second. It is sometimes useful to think of bandwidth in terms of how long it takes to transmit each bit of data. On a 10-Mbps network, for example, it takes 0.1 microsecond ( $\mu\text{s}$ ) to transmit each bit.

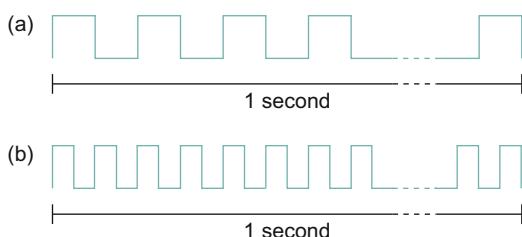
While you can talk about the bandwidth of the network as a whole, sometimes you want to be more precise, focusing, for example, on the bandwidth of a single physical link or of a logical process-to-process channel. At the physical level, bandwidth is constantly improving, with no end in sight. Intuitively, if you think of a second of time as a distance you could measure with a ruler and bandwidth as how many bits fit in that distance, then you can think of each bit as a pulse of some width. For example, each bit on a 1-Mbps link is  $1 \mu\text{s}$  wide, while each bit on a 2-Mbps link is  $0.5 \mu\text{s}$  wide, as illustrated in [Figure 1.16](#). The more sophisticated the transmitting and receiving technology, the narrower each bit can become and, thus, the higher the bandwidth. For logical process-to-process channels, bandwidth is also influenced by other factors, including how many times the software that implements the channel has to handle, and possibly transform, each bit of data.

### Bandwidth and Throughput

*Bandwidth* and *throughput* are two of the most confusing terms used in networking. While we could try to give you a precise definition of each term, it is important that you know how other people might use them and for you to be aware that they are often used interchangeably. First of all, bandwidth is literally a measure of the width of a frequency band. For example, a voice-grade telephone line supports a frequency band ranging from 300 to 3300 Hz; it is said to have a bandwidth of  $3300 \text{ Hz} - 300 \text{ Hz} = 3000 \text{ Hz}$ . If you see the word *bandwidth* used in a situation in which it is being measured in hertz, then it probably refers to the range of signals that can be accommodated.

When we talk about the bandwidth of a communication link, we normally refer to the number of bits per second that can be transmitted on the link. This is also sometimes called the *data rate*. We might say that the bandwidth of an Ethernet link is 10 Mbps. A useful distinction can also be made, however, between the maximum data rate that is available on the link and the number of bits per second that we can actually transmit over the link in practice. We tend to use the word *throughput* to refer to the *measured performance* of a system. Thus, because of various inefficiencies of implementation, a pair of nodes connected by a link with a bandwidth of 10 Mbps might achieve a throughput of only 2 Mbps. This would mean that an application on one host could send data to the other host at 2 Mbps.

Finally, we often talk about the bandwidth *requirements* of an application. This is the number of bits per second that it needs to transmit over the network to perform acceptably. For some applications, this might be “whatever I can get”; for others, it might be some fixed number (preferably no more than the available link bandwidth); and for others, it might be a number that varies with time. We will provide more on this topic later in this section.



■ **FIGURE 1.16** Bits transmitted at a particular bandwidth can be regarded as having some width: (a) bits transmitted at 1 Mbps (each bit is 1  $\mu\text{s}$  wide); (b) bits transmitted at 2 Mbps (each bit is 0.5  $\mu\text{s}$  wide).

The second performance metric, latency, corresponds to how long it takes a message to travel from one end of a network to the other. (As with bandwidth, we could be focused on the latency of a single link or an end-to-end channel.) Latency is measured strictly in terms of time. For example, a transcontinental network might have a latency of 24 milliseconds (ms); that is, it takes a message 24 ms to travel from one coast of North America to the other. There are many situations in which it is more important to know how long it takes to send a message from one end of a network to the other and back, rather than the one-way latency. We call this the *round-trip time* (RTT) of the network.

We often think of latency as having three components. First, there is the speed-of-light propagation delay. This delay occurs because nothing, including a bit on a wire, can travel faster than the speed of light. If you know the distance between two points, you can calculate the speed-of-light latency, although you have to be careful because light travels across different media at different speeds: It travels at  $3.0 \times 10^8$  m/s in a vacuum,  $2.3 \times 10^8$  m/s in a copper cable, and  $2.0 \times 10^8$  m/s in an optical fiber. Second, there is the amount of time it takes to transmit a unit of data. This is a function of the network bandwidth and the size of the packet in which the data is carried. Third, there may be queuing delays inside the network, since packet switches generally need to store packets for some time before forwarding them on an outbound link, as discussed in Section 1.2.3. So, we could define the total latency as

$$\text{Latency} = \text{Propagation} + \text{Transmit} + \text{Queue}$$

$$\text{Propagation} = \text{Distance}/\text{SpeedOfLight}$$

$$\text{Transmit} = \text{Size}/\text{Bandwidth}$$

where Distance is the length of the wire over which the data will travel, SpeedOfLight is the effective speed of light over that wire, Size is the size of the packet, and Bandwidth is the bandwidth at which the packet is transmitted. Note that if the message contains only one bit and we are talking about a single link (as opposed to a whole network), then the Transmit and Queue terms are not relevant, and latency corresponds to the propagation delay only.

Bandwidth and latency combine to define the performance characteristics of a given link or channel. Their relative importance, however, depends on the application. For some applications, latency dominates

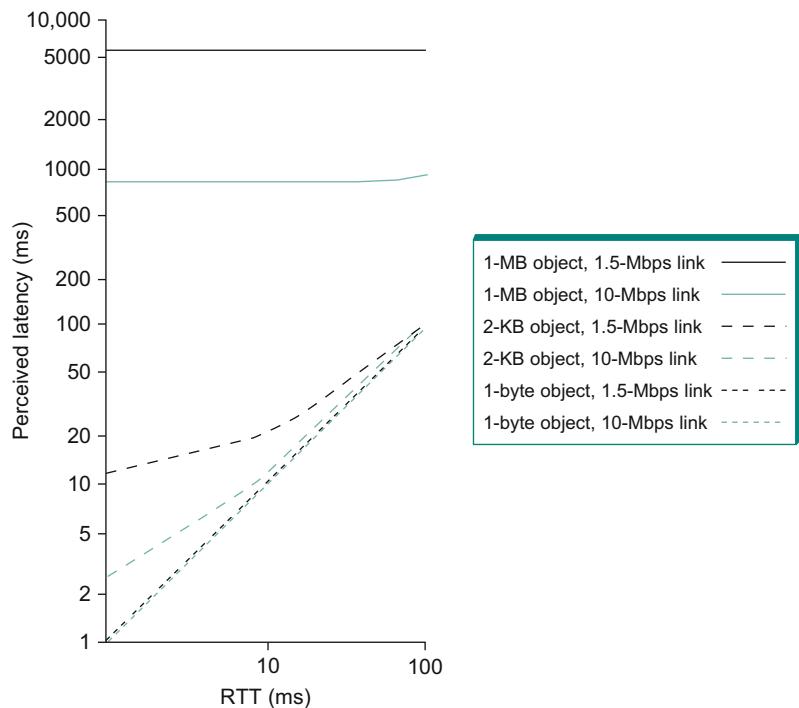
bandwidth. For example, a client that sends a 1-byte message to a server and receives a 1-byte message in return is latency bound. Assuming that no serious computation is involved in preparing the response, the application will perform much differently on a transcontinental channel with a 100-ms RTT than it will on an across-the-room channel with a 1-ms RTT. Whether the channel is 1 Mbps or 100 Mbps is relatively insignificant, however, since the former implies that the time to transmit a byte (Transmit) is 8  $\mu$ s and the latter implies Transmit = 0.08  $\mu$ s.

In contrast, consider a digital library program that is being asked to fetch a 25-megabyte (MB) image—the more bandwidth that is available, the faster it will be able to return the image to the user. Here, the bandwidth of the channel dominates performance. To see this, suppose that the channel has a bandwidth of 10 Mbps. It will take 20 seconds to transmit the image ( $25 \times 10^6 \times 8 \text{ bits} \div 10 \times 10^6 \text{ Mbps} = 20 \text{ seconds}$ ), making it relatively unimportant if the image is on the other side of a 1-ms channel or a 100-ms channel; the difference between a 20.001-second response time and a 20.1-second response time is negligible.

Figure 1.17 gives you a sense of how latency or bandwidth can dominate performance in different circumstances. The graph shows how long it takes to move objects of various sizes (1 byte, 2 KB, 1 MB) across networks with RTTs ranging from 1 to 100 ms and link speeds of either 1.5 or 10 Mbps. We use logarithmic scales to show relative performance. For a 1-byte object (say, a keystroke), latency remains almost exactly equal to the RTT, so that you cannot distinguish between a 1.5-Mbps network and a 10-Mbps network. For a 2-KB object (say, an email message), the link speed makes quite a difference on a 1-ms RTT network but a negligible difference on a 100-ms RTT network. And for a 1-MB object (say, a digital image), the RTT makes no difference—it is the link speed that dominates performance across the full range of RTT.

Note that throughout this book we use the terms *latency* and *delay* in a generic way to denote how long it takes to perform a particular function, such as delivering a message or moving an object. When we are referring to the specific amount of time it takes a signal to propagate from one end of a link to another, we use the term *propagation delay*. Also, we make it clear in the context of the discussion whether we are referring to the one-way latency or the round-trip time.

As an aside, computers are becoming so fast that when we connect them to networks, it is sometimes useful to think, at least figuratively, in

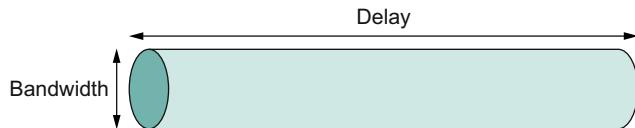


■ FIGURE 1.17 Perceived latency (response time) versus round-trip time for various object sizes and link speeds.

terms of *instructions per mile*. Consider what happens when a computer that is able to execute 1 billion instructions per second sends a message out on a channel with a 100-ms RTT. (To make the math easier, assume that the message covers a distance of 5000 miles.) If that computer sits idle the full 100 ms waiting for a reply message, then it has forfeited the ability to execute 100 million instructions, or 20,000 instructions per mile. It had better have been worth going over the network to justify this waste.

### 1.5.2 Delay $\times$ Bandwidth Product

It is also useful to talk about the product of these two metrics, often called the *delay  $\times$  bandwidth product*. Intuitively, if we think of a channel between a pair of processes as a hollow pipe (see Figure 1.18), where the latency corresponds to the length of the pipe and the bandwidth gives the



■ FIGURE 1.18 Network as a pipe.

diameter of the pipe, then the  $\text{delay} \times \text{bandwidth}$  product gives the volume of the pipe—the maximum number of bits that could be in transit through the pipe at any given instant. Said another way, if latency (measured in time) corresponds to the length of the pipe, then given the width of each bit (also measured in time) you can calculate how many bits fit in the pipe. For example, a transcontinental channel with a one-way latency of 50 ms and a bandwidth of 45 Mbps is able to hold

$$\begin{aligned} & 50 \times 10^{-3} \text{ s} \times 45 \times 10^6 \text{ bits/s} \\ & = 2.25 \times 10^6 \text{ bits} \end{aligned}$$

or approximately 280 KB of data. In other words, this example channel (pipe) holds as many bytes as the memory of a personal computer from the early 1980s could hold.

The  $\text{delay} \times \text{bandwidth}$  product is important to know when constructing high-performance networks because it corresponds to how many bits the sender must transmit before the first bit arrives at the receiver. If the sender is expecting the receiver to somehow signal that bits are starting to arrive, and it takes another channel latency for this signal to propagate back to the sender, then the sender can send up one  $\text{RTT} \times \text{bandwidth}$  worth of data before hearing from the receiver that all is well. The bits in the pipe are said to be “in flight,” which means that if the receiver tells the sender to stop transmitting it might receive up to one  $\text{RTT} \times \text{bandwidth}$ ’s worth of data before the sender manages to respond. In our example above, that amount corresponds to  $5.5 \times 10^6$  bits (671 KB) of data. On the other hand, if the sender does not fill the pipe—send a whole  $\text{RTT} \times \text{bandwidth}$  product’s worth of data before it stops to wait for a signal—the sender will not fully utilize the network.

Note that most of the time we are interested in the RTT scenario, which we simply refer to as the *delay × bandwidth product*, without explicitly saying that “delay” is the RTT (i.e., the one-way delay multiplied

by two). Usually, whether the “delay” in “delay  $\times$  bandwidth” means one-way latency or RTT is made clear by the context. Table 1.1 shows some examples of RTT  $\times$  bandwidth products for some typical network links.

### How Big Is a Mega?

There are several pitfalls you need to be aware of when working with the common units of networking—MB, Mbps, KB, and kbps. The first is to distinguish carefully between bits and bytes. Throughout this book, we always use a lowercase *b* for bits and a capital *B* for bytes. The second is to be sure you are using the appropriate definition of mega (M) and kilo (K). *Mega*, for example, can mean either  $2^{20}$  or  $10^6$ . Similarly, *kilo* can be either  $2^{10}$  or  $10^3$ . What is worse, in networking we typically use both definitions. Here’s why:

Network bandwidth, which is often specified in terms of Mbps, is typically governed by the speed of the clock that paces the transmission of the bits. A clock that is running at 10 MHz is used to transmit bits at 10 Mbps. Because the *mega* in MHz means  $10^6$  hertz, Mbps is usually also defined as  $10^6$  bits per second. (Similarly, kbps is  $10^3$  bits per second.) On the other hand, when we talk about a message that we want to transmit, we often give its size in kilobytes. Because messages are stored in the computer’s memory, and memory is typically measured in powers of two, the *K* in KB is usually taken to mean  $2^{10}$ . (Similarly, MB usually means  $2^{20}$ .) When you put the two together, it is not uncommon to talk about sending a 32-KB message over a 10-Mbps channel, which should be interpreted to mean  $32 \times 2^{10} \times 8$  bits are being transmitted at a rate of  $10 \times 10^6$  bits per second. This is the interpretation we use throughout the book, unless explicitly stated otherwise.

The good news is that many times we are satisfied with a back-of-the-envelope calculation, in which case it is perfectly reasonable to make the approximation that  $10^6$  is really equal to  $2^{20}$  (making it easy to convert between the two definitions of mega). This approximation introduces only a 5% error. We can even make the approximation in some cases that a byte has 10 bits, a 20% error but good enough for order-of-magnitude estimates.

To help you in your quick-and-dirty calculations, 100 ms is a reasonable number to use for a cross-country round-trip time—at least when the country in question is the United States—and 1 ms is a good approximation of an RTT across a local area network. In the case of the former, we increase the 48-ms round-trip time implied by the speed of light over a fiber to 100 ms because there are, as we have said, other sources of delay, such as the processing time in the switches inside the network. You can also be sure that the path taken by the fiber between two points will not be a straight line.

**Table 1.1 Sample Delay × Bandwidth Products**

<b>Link type</b>	<b>Bandwidth (typical)</b>	<b>One-way distance (typical)</b>	<b>Round-trip delay</b>	<b>RTT × Bandwidth</b>
Dial-up	56 kbps	10 km	87 $\mu$ s	5 bits
Wireless LAN	54 Mbps	50 m	0.33 $\mu$ s	18 bits
Satellite	45 Mbps	35,000 km	230 ms	10 Mb
Cross-country fiber	10 Gbps	4,000 km	40 ms	400 Mb

### 1.5.3 High-Speed Networks

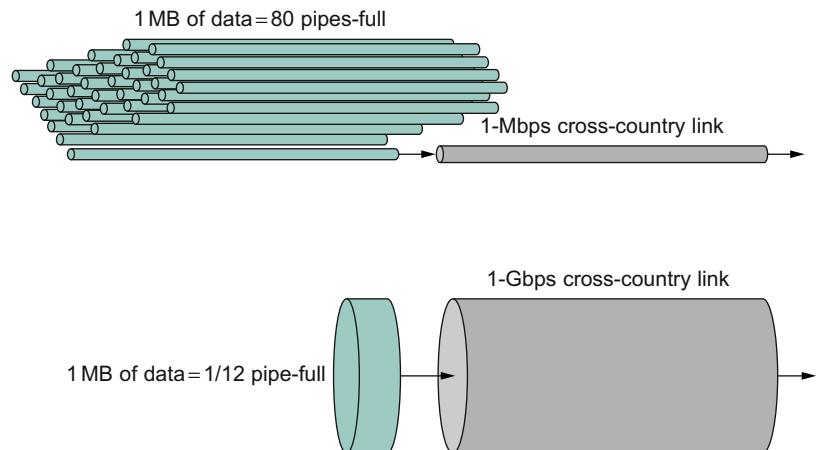
The bandwidths available on today's networks are increasing at a dramatic rate, and there is eternal optimism that network bandwidth will continue to improve. This causes network designers to start thinking about what happens in the limit or, stated another way, what is the impact on network design of having infinite bandwidth available.

Although high-speed networks bring a dramatic change in the bandwidth available to applications, in many respects their impact on how we think about networking comes in what does *not* change as bandwidth increases: the speed of light. To quote Scotty from *Star Trek*, “Ye cannae change the laws of physics.”<sup>4</sup> In other words, “high speed” does not mean that latency improves at the same rate as bandwidth; the transcontinental RTT of a 1-Gbps link is the same 100 ms as it is for a 1-Mbps link.

To appreciate the significance of ever-increasing bandwidth in the face of fixed latency, consider what is required to transmit a 1-MB file over a 1-Mbps network versus over a 1-Gbps network, both of which have an RTT of 100 ms. In the case of the 1-Mbps network, it takes 80 round-trip times to transmit the file; during each RTT, 1.25% of the file is sent. In contrast, the same 1-MB file doesn't even come close to filling 1 RTT's worth of the 1-Gbps link, which has a delay × bandwidth product of 12.5 MB.

Figure 1.19 illustrates the difference between the two networks. In effect, the 1-MB file looks like a stream of data that needs to be transmitted across a 1-Mbps network, while it looks like a single packet on a 1-Gbps network. To help drive this point home, consider that a 1-MB file is to a 1-Gbps network what a 1-KB *packet* is to a 1-Mbps network.

<sup>4</sup>Scots dialect for “You cannot change the laws of physics.”



**FIGURE 1.19** Relationship between bandwidth and latency. A 1-MB file would fill the 1-Mbps link 80 times but only fill the 1-Gbps link 1/12 of one time.

Another way to think about the situation is that more data can be transmitted during each RTT on a high-speed network, so much so that a single RTT becomes a significant amount of time. Thus, while you wouldn't think twice about the difference between a file transfer taking 101 RTTs rather than 100 RTTs (a relative difference of only 1%), suddenly the difference between 1 RTT and 2 RTTs is significant—a 100% increase. In other words, latency, rather than throughput, starts to dominate our thinking about network design.

Perhaps the best way to understand the relationship between throughput and latency is to return to basics. The effective end-to-end throughput that can be achieved over a network is given by the simple relationship

$$\text{Throughput} = \frac{\text{TransferSize}}{\text{TransferTime}}$$

where TransferTime includes not only the elements of one-way Latency identified earlier in this section, but also any additional time spent requesting or setting up the transfer. Generally, we represent this relationship as

$$\text{TransferTime} = \text{RTT} + \frac{1}{\text{Bandwidth}} \times \text{TransferSize}$$

We use RTT in this calculation to account for a request message being sent across the network and the data being sent back. For example, consider a

situation where a user wants to fetch a 1-MB file across a 1-Gbps network with a round-trip time of 100 ms. The TransferTime includes both the transmit time for 1 MB ( $1/1 \text{ Gbps} \times 1 \text{ MB} = 8 \text{ ms}$ ) and the 100-ms RTT, for a total transfer time of 108 ms. This means that the effective throughput will be

$$1 \text{ MB}/108 \text{ ms} = 74.1 \text{ Mbps}$$

not 1 Gbps. Clearly, transferring a larger amount of data will help improve the effective throughput, where in the limit an infinitely large transfer size will cause the effective throughput to approach the network bandwidth. On the other hand, having to endure more than 1 RTT—for example, to retransmit missing packets—will hurt the effective throughput for any transfer of finite size and will be most noticeable for small transfers.

#### 1.5.4 Application Performance Needs

The discussion in this section has taken a network-centric view of performance; that is, we have talked in terms of what a given link or channel will support. The unstated assumption has been that application programs have simple needs—they want as much bandwidth as the network can provide. This is certainly true of the aforementioned digital library program that is retrieving a 25-MB image; the more bandwidth that is available, the faster the program will be able to return the image to the user.

However, some applications are able to state an upper limit on how much bandwidth they need. Video applications are a prime example. Suppose one wants to stream a video that is one quarter the size of a standard TV screen; that is, it has a resolution of 352 by 240 pixels. If each pixel is represented by 24 bits of information, as would be the case for 24-bit color, then the size of each frame would be

$$(352 \times 240 \times 24)/8 = 247.5 \text{ KB}$$

If the application needs to support a frame rate of 30 frames per second, then it might request a throughput rate of 75 Mbps. The ability of the network to provide more bandwidth is of no interest to such an application because it has only so much data to transmit in a given period of time.

Unfortunately, the situation is not as simple as this example suggests. Because the difference between any two adjacent frames in a video

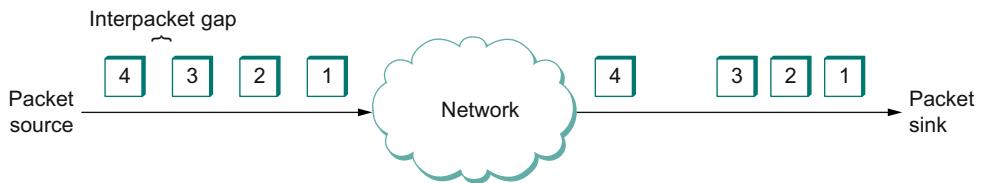
stream is often small, it is possible to compress the video by transmitting only the differences between adjacent frames. Each frame can also be compressed because not all the detail in a picture is readily perceived by a human eye. The compressed video does not flow at a constant rate, but varies with time according to factors such as the amount of action and detail in the picture and the compression algorithm being used. Therefore, it is possible to say what the average bandwidth requirement will be, but the instantaneous rate may be more or less.

The key issue is the time interval over which the average is computed. Suppose that this example video application can be compressed down to the point that it needs only 2 Mbps, on average. If it transmits 1 megabit in a 1-second interval and 3 megabits in the following 1-second interval, then over the 2-second interval it is transmitting at an average rate of 2 Mbps; however, this will be of little consolation to a channel that was engineered to support no more than 2 megabits in any one second. Clearly, just knowing the average bandwidth needs of an application will not always suffice.

Generally, however, it is possible to put an upper bound on how large a burst an application like this is likely to transmit. A burst might be described by some peak rate that is maintained for some period of time. Alternatively, it could be described as the number of bytes that can be sent at the peak rate before reverting to the average rate or some lower rate. If this peak rate is higher than the available channel capacity, then the excess data will have to be buffered somewhere, to be transmitted later. Knowing how big of a burst might be sent allows the network designer to allocate sufficient buffer capacity to hold the burst. We will return to the subject of describing bursty traffic accurately in [Chapter 6](#).

Analogous to the way an application's bandwidth needs can be something other than "all it can get," an application's delay requirements may be more complex than simply "as little delay as possible." In the case of delay, it sometimes doesn't matter so much whether the one-way latency of the network is 100 ms or 500 ms as how much the latency varies from packet to packet. The variation in latency is called *jitter*.

Consider the situation in which the source sends a packet once every 33 ms, as would be the case for a video application transmitting frames 30 times a second. If the packets arrive at the destination spaced out exactly 33 ms apart, then we can deduce that the delay experienced by each packet in the network was exactly the same. If the spacing



■ FIGURE 1.20 Network-induced jitter.

between when packets arrive at the destination—sometimes called the *inter-packet gap*—is variable, however, then the delay experienced by the sequence of packets must have also been variable, and the network is said to have introduced jitter into the packet stream, as shown in Figure 1.20. Such variation is generally not introduced in a single physical link, but it can happen when packets experience different queuing delays in a multihop packet-switched network. This queuing delay corresponds to the Queue component of latency defined earlier in this section, which varies with time.

To understand the relevance of jitter, suppose that the packets being transmitted over the network contain video frames, and in order to display these frames on the screen the receiver needs to receive a new one every 33 ms. If a frame arrives early, then it can simply be saved by the receiver until it is time to display it. Unfortunately, if a frame arrives late, then the receiver will not have the frame it needs in time to update the screen, and the video quality will suffer; it will not be smooth. Note that it is not necessary to eliminate jitter, only to know how bad it is. The reason for this is that if the receiver knows the upper and lower bounds on the latency that a packet can experience, it can delay the time at which it starts playing back the video (i.e., displays the first frame) long enough to ensure that in the future it will always have a frame to display when it needs it. The receiver delays the frame, effectively smoothing out the jitter, by storing it in a buffer. We return to the topic of jitter in Chapter 9.

## 1.6 SUMMARY

Computer networks, and in particular the Internet, have experienced enormous growth over the past 30 years and are now able to provide a

wide range of services, from conducting business to providing access to entertainment to enabling social networks. Much of this growth can be attributed to the general-purpose nature of computer networks, and in particular to the ability to add new functionality to the network by writing software that runs on affordable, high-performance computers. With this in mind, the overriding goal of this book is to describe computer networks in such a way that when you finish reading it you should feel that, if you had an army of programmers at your disposal, you could actually build a fully functional computer network from the ground up. This chapter lays the foundation for realizing this goal.

The first step we have taken toward this goal is to carefully identify exactly what we expect from a network. For example, a network must first provide cost-effective and scalable connectivity among a set of computers. This is accomplished through a nested interconnection of nodes and links and by sharing this hardware base through the use of statistical multiplexing. This results in a packet-switched network, on top of which we then define a collection of process-to-process communication services.

The second step is to define a layered architecture that will serve as a blueprint for our design. The central objects of this architecture are network protocols. Protocols both provide a communication service to higher-level protocols and define the form and meaning of messages exchanged with their peers running on other machines. We have briefly surveyed two of the most widely used architectures: the 7-layer OSI architecture and the Internet architecture. This book most closely follows the Internet architecture, both in its organization and as a source of examples.

The third step is to implement the network's protocols and application programs, usually in software. Both protocols and applications need an interface by which they invoke the services of other protocols in the network subsystem. The socket interface is the most widely used interface between application programs and the network subsystem, but a slightly different interface is typically used within the network subsystem.

Finally, the network as a whole must offer high performance, where the two performance metrics we are most interested in are latency and throughput. As we will see in later chapters, it is the product of these two metrics—the so-called delay  $\times$  bandwidth product—that often plays a critical role in protocol design.

It's apparent that computer networks have become an integral part of the everyday lives of vast numbers of people. What began over 40 years ago as experimental systems like the ARPANET—connecting mainframe computers over long-distance telephone lines—has turned into a pervasive part of our lives. It has also become big business, and where there is big business there are lots of players. In this case, we have the computing industry, which has become increasingly involved in integrating computation and communication; the telephone and cable operators, which recognize the market for carrying all sorts of data, not just voice and television; and, perhaps most importantly, the many entrepreneurs creating new Internet-based applications and services such as voice over IP (VOIP), online games, virtual worlds,

### WHAT'S NEXT: CLOUD COMPUTING

search services, content hosting, electronic commerce, and so on. It's noteworthy that one of today's biggest names in "cloud computing," Amazon.com, achieved that position by first adopting Internet technologies to sell consumer products such as books and then making their computing infrastructure available to others as a service over the network.

A few years ago, a reasonable goal for networking might have been to provide network access to every home, but in developed countries at least that process is now far along. Ubiquitous networking now includes getting access from anywhere, including on planes and trains, and on an increasingly wide range of devices. Whereas the Internet largely evolved in an era of fixed mainframe and then personal computers, today the set of devices to be connected together includes mobile phones and even smaller devices such as sensors (which might also be mobile). Thus, it seems clear that the Internet will have to continue to scale to support several orders of magnitude more devices than today and that many of these devices will be mobile, perhaps intermittently connected over wireless links of highly variable quality. At the same time, these devices will be connected to large data centers—filled with tens of thousands of processors and many petabytes of storage—that will store and analyze the data being generated, all with the hope of enabling even more powerful applications that help us navigate our daily lives. And, the devices that we carry are often just a means of accessing "the cloud"—the amorphous set of machines that store and process our documents, photos, data, social networks, etc., which we expect to be able to access from anywhere.

Predictions about the future of networking have a tendency to look silly a few years down the road (many high-profile predictions about an imminent meltdown of the Internet, for example, have failed to come true). What we can say with confidence is that there remain plenty of technical challenges—issues of connectivity, manageability, scalability, usability, performance, reliability, security, fairness, cost-effectiveness, etc.—that stand between the current state of the art and the sort of global, ubiquitous, heterogeneous network that many believe is imminent. In other words, networking as a field is very much alive with interesting problems still to be solved, and it is these problems and the tools for solving them that are the focus of this book.

---

### ■ FURTHER READING

Computer networks are not the first communication-oriented technology to have found their way into the everyday fabric of our society. For example, the early part of this century saw the introduction of the telephone, and then during the 1950s television became widespread. When considering the future of networking—how widely it will spread and how we will use it—it is instructive to study this history. Our first reference is a good starting point for doing this (the entire issue is devoted to the first 100 years of telecommunications).

The second reference is considered one of the seminal papers on the Internet architecture. The final two papers are not specific to networking but present viewpoints that capture the “systems approach” of this book. The Saltzer et al. paper motivates and describes one of the most widely applied rules of network architecture—the *end-to-end argument*—which continues to be highly cited today. The paper by Mashey describes the thinking behind RISC (Reduced Instruction Set Computer) architectures; as we will soon discover, making good judgments about where to place functionality in a complex system is what system design is all about.

- Pierce, J. Telephony—A personal view. *IEEE Communications* 22(5):116–120, May 1984.
- Clark, D. The design philosophy of the DARPA Internet protocols. *Proceedings of the SIGCOMM '88 Symposium*, pages 106–114, August 1988.
- Saltzer, J., D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems* 2(4):277–288, November 1984.

- Mashey, J. RISC, MIPS, and the motion of complexity. *UniForum 1986 Conference Proceedings*, pages 116–124, February 1986.

Several texts offer an introduction to computer networking: Stallings gives an encyclopedic treatment of the subject, with an emphasis on the lower levels of the OSI hierarchy [Sta07]; Comer gives a good introduction to the Internet architecture [Com05].

To put computer networking into a larger context, two books—one dealing with the past and the other looking toward the future—are must reading. The first is Holzmann and Pehrson's *The Early History of Data Networks* [HP95]. Surprisingly, many of the ideas covered in the book you are now reading were invented during the 1700s. The second is *Realizing the Information Future: The Internet and Beyond*, a book prepared by the Computer Science and Telecommunications Board of the National Research Council [NRC94].

In this book we try to bring a systems approach to the field of computer networking. We recommend Saltzer and Kaashoek's general treatment of computer systems [SK09], which teaches many important principles that apply to networking as well as other systems. Operating systems in particular are important to many aspects of networking; Tanenbaum [Tan07] provides an introduction to OS concepts.

To follow the history of the Internet from its beginning, the reader is encouraged to peruse the Internet's *Request for Comments* (RFC) series of documents. These documents, which include everything from the TCP specification to April Fools' jokes, are retrievable at <http://www.ietf.org/rfc.html>. For example, the protocol specifications for TCP, UDP, and IP are available in RFC 793, 768, and 791, respectively.

To gain a better appreciation for the Internet philosophy and culture, two references are recommended; both are also quite entertaining. Padlipsky gives a good description of the early days, including a pointed comparison of the Internet and OSI architectures [Pad85]. For an account of what really happens behind the scenes at the Internet Engineering Task Force, we recommend Boorsook's article [Boo95].

There is a wealth of articles discussing various aspects of protocol implementations. A good starting point is to understand two complete protocol implementation environments: the Stream mechanism from System V Unix [Rit84] and the *x*-kernel [HP91]. In addition, [LMQ89] and [SW95] describe the widely used Berkeley Unix implementation of TCP/IP.

More generally, a large body of work addresses the issue of structuring and optimizing protocol implementations. Clark was one of the first to discuss the relationship between modular design and protocol performance [Cla82]. Later papers then introduce the use of upcalls in structuring protocol code [Cla85] and study the processing overheads in TCP [CJRS89]. Finally, [WM87] describes how to gain efficiency through appropriate design and implementation choices.

Several papers have introduced specific techniques and mechanisms that can be used to improve protocol performance. For example, [HMPT89] describes some of the mechanisms used in the *x*-kernel environment, while [MD93], [VL87], and [DP93] present a variety of techniques for improving protocol performance. Also, the performance of protocols running on parallel processors—locking is a key issue in such environments—is discussed in [BG93] and [NYKT94].

Finally, we conclude the Further Reading section of each chapter with a set of live references, URLs for locations on the World Wide Web where you can learn more about the topics discussed in that chapter. Since these references are live, it is possible that they will not remain active for an indefinite period of time. For this reason, we limit the set of live references at the end of each chapter to sites that export software, provide a service, or report on the activities of an ongoing working group or standardization body. In other words, we only give URLs for the kinds of material that cannot easily be referenced using standard citations. For this chapter, we include three live references:

- <http://mkp.com/computer-networks>: information about this book, including supplements, addenda, and so on
- <http://www.ietf.org/>: information about the IETF, its working groups, standards, etc.
- <http://dblp.uni-trier.de/db/index.html>: a searchable bibliography of computer science research papers

## EXERCISES

1. Use anonymous FTP to connect to [ftp.rfc-editor.org](ftp://rfc-editor.org) (directory `in-notes`), and retrieve the RFC index. Also, retrieve the protocol specifications for TCP, IP, and UDP.

2. The Unix utility `whois` can be used to find the domain name corresponding to an organization, or *vice versa*. Read the man page documentation for `whois` and experiment with it. Try `whois princeton.edu` and `whois princeton`, for starters. As an alternative, explore the `whois` interface at <http://www.internic.net/whois.html>.
3. Calculate the total time required to transfer a 1000-KB file in the following cases, assuming an RTT of 50 ms, a packet size of 1 KB data, and an initial  $2 \times$  RTT of “handshaking” before data is sent:
- (a) The bandwidth is 1.5 Mbps, and data packets can be sent continuously.
  - (b) The bandwidth is 1.5 Mbps, but after we finish sending each data packet we must wait one RTT before sending the next.
  - (c) The bandwidth is “infinite,” meaning that we take transmit time to be zero, and up to 20 packets can be sent per RTT.
  - (d) The bandwidth is infinite, and during the first RTT we can send one packet ( $2^{1-1}$ ), during the second RTT we can send two packets ( $2^{2-1}$ ), during the third we can send four ( $2^{3-1}$ ), and so on. (A justification for such an exponential increase will be given in Chapter 6.)
- ✓ 4. Calculate the total time required to transfer a 1.5-MB file in the following cases, assuming an RTT of 80 ms, a packet size of 1 KB data, and an initial  $2 \times$  RTT of “handshaking” before data is sent:
- (a) The bandwidth is 10 Mbps, and data packets can be sent continuously.
  - (b) The bandwidth is 10 Mbps, but after we finish sending each data packet we must wait one RTT before sending the next.
  - (c) The link allows infinitely fast transmit, but limits bandwidth such that only 20 packets can be sent per RTT.
  - (d) Zero transmit time as in (c), but during the first RTT we can send one packet, during the second RTT we can send two packets, during the third we can send four ( $2^{3-1}$ ), etc. (A justification for such an exponential increase will be given in Chapter 6.)
5. Consider a point-to-point link 4 km in length. At what bandwidth would propagation delay (at a speed of  $2 \times 10^8$ m/s) equal

transmit delay for 100-byte packets? What about 512-byte packets?

- ✓ 6. Consider a point-to-point link 50 km in length. At what bandwidth would propagation delay (at a speed of  $2 \times 10^8$  m/s) equal transmit delay for 100-byte packets? What about 512-byte packets?
- 7. What properties of postal addresses would be likely to be shared by a network addressing scheme? What differences might you expect to find? What properties of telephone numbering might be shared by a network addressing scheme?
- 8. One property of addresses is that they are unique; if two nodes had the same address, it would be impossible to distinguish between them. What other properties might be useful for network addresses to have? Can you think of any situations in which network (or postal or telephone) addresses might *not* be unique?
- 9. Give an example of a situation in which multicast addresses might be beneficial.
- 10. What differences in traffic patterns account for the fact that STDM is a cost-effective form of multiplexing for a voice telephone network and FDM is a cost-effective form of multiplexing for television and radio networks, yet we reject both as not being cost effective for a general-purpose computer network?
- 11. How “wide” is a bit on a 10-Gbps link? How long is a bit in copper wire, where the speed of propagation is  $2.3 \times 10^8$  m/s?
- 12. How long does it take to transmit  $x$  KB over a  $y$ -Mbps link? Give your answer as a ratio of  $x$  and  $y$ .
- 13. Suppose a 1-Gbps point-to-point link is being set up between the Earth and a new lunar colony. The distance from the moon to the Earth is approximately 385,000 km, and data travels over the link at the speed of light— $3 \times 10^8$  m/s.
  - (a) Calculate the minimum RTT for the link.
  - (b) Using the RTT as the delay, calculate the delay  $\times$  bandwidth product for the link.

- (c) What is the significance of the delay  $\times$  bandwidth product computed in (b)?
- (d) A camera on the lunar base takes pictures of the Earth and saves them in digital format to disk. Suppose Mission Control on Earth wishes to download the most current image, which is 25 MB. What is the minimum amount of time that will elapse between when the request for the data goes out and the transfer is finished?
- ✓ 14. Suppose a 128-kbps point-to-point link is set up between the Earth and a rover on Mars. The distance from the Earth to Mars (when they are closest together) is approximately 55 Gm, and data travels over the link at the speed of light— $3 \times 10^8$  m/s.
- (a) Calculate the minimum RTT for the link.
  - (b) Calculate the delay  $\times$  bandwidth product for the link.
  - (c) A camera on the rover takes pictures of its surroundings and sends these to Earth. How quickly after a picture is taken can it reach Mission Control on Earth? Assume that each image is 5 Mb in size.
15. For each of the following operations on a remote file server, discuss whether they are more likely to be delay sensitive or bandwidth sensitive:
- (a) Open a file.
  - (b) Read the contents of a file.
  - (c) List the contents of a directory.
  - (d) Display the attributes of a file.
16. Calculate the latency (from first bit sent to last bit received) for the following:
- (a) 100-Mbps Ethernet with a single store-and-forward switch in the path and a packet size of 12,000 bits. Assume that each link introduces a propagation delay of  $10 \mu\text{s}$  and that the switch begins retransmitting immediately after it has finished receiving the packet.
  - (b) Same as (a) but with three switches.
  - (c) Same as (a), but assume the switch implements “cut-through” switching; it is able to begin retransmitting the packet after the first 200 bits have been received.

- ✓ 17. Calculate the latency (from first bit sent to last bit received) for:
- (a) 1-Gbps Ethernet with a single store-and-forward switch in the path and a packet size of 5000 bits. Assume that each link introduces a propagation delay of  $10 \mu\text{s}$  and that the switch begins retransmitting immediately after it has finished receiving the packet.
  - (b) Same as (a) but with three switches.
  - (c) Same as (b), but assume the switch implements “cut-through” switching; it is able to begin retransmitting the packet after the first 128 bits have been received.
18. Calculate the effective bandwidth for the following cases. For (a) and (b) assume there is a steady supply of data to send; for (c) simply calculate the average over 12 hours.
- (a) 100-Mbps Ethernet through three store-and-forward switches as in Exercise 16(b). Switches can send on one link while receiving on the other.
  - (b) Same as (a) but with the sender having to wait for a 50-byte acknowledgment packet after sending each 12,000-bit data packet.
  - (c) Overnight (12-hour) shipment of 100 DVDs that hold 4.7 GB each.
19. Calculate the  $\text{delay} \times \text{bandwidth}$  product for the following links. Use one-way delay, measured from first bit sent to first bit received.
- (a) 100-Mbps Ethernet with a delay of  $10 \mu\text{s}$ .
  - (b) 100-Mbps Ethernet with a single store-and-forward switch like that of Exercise 16(b), packet size of 12,000 bits, and  $10 \mu\text{s}$  per link propagation delay.
  - (c) 1.5-Mbps T1 link, with a transcontinental one-way delay of 50 ms.
  - (d) 1.5-Mbps T1 link between two groundstations communicating via a satellite in geosynchronous orbit, 35,900 km high. The only delay is speed-of-light propagation delay from Earth to the satellite *and back*.
20. Hosts A and B are each connected to a switch S via 100-Mbps links as in Figure 1.21. The propagation delay on each link is



■ FIGURE 1.21 Diagram for Exercise 20.

20  $\mu$ s. S is a store-and-forward device; it begins retransmitting a received packet 35  $\mu$ s after it has finished receiving it. Calculate the total time required to transmit 10,000 bits from A to B

- (a) As a single packet.
  - (b) As two 5000-bit packets sent one right after the other.
21. Suppose a host has a 1-MB file that is to be sent to another host. The file takes 1 second of CPU time to compress 50% or 2 seconds to compress 60%.
- (a) Calculate the bandwidth at which each compression option takes the same total compression + transmission time.
  - (a) Explain why latency does not affect your answer.
22. Suppose that a certain communications protocol involves a per-packet overhead of 50 bytes for headers and framing. We send 1 million bytes of data using this protocol; however, one data byte is corrupted and the entire packet containing it is thus lost. Give the total number of overhead + loss bytes for packet data sizes of 1000, 10,000, and 20,000 bytes. Which size is optimal?
23. Assume you wish to transfer an  $n$  B file along a path composed of the source, destination, 7 point-to-point links, and 5 switches. Suppose each link has a propagation delay of 2 ms and a bandwidth of 4 Mbps, and that the switches support both circuit and packet switching. Thus, you can either break the file up into 1-KB packets or set up a circuit through the switches and send the file as one contiguous bitstream. Suppose that packets have 24 B of packet header information and 1000 B of payload, store-and-forward packet processing at each switch incurs a 1-ms delay after the packet had been completely received, packets may be sent continuously without waiting for acknowledgments, and circuit setup requires a 1-KB message to make one round trip on the path, incurring a 1-ms delay at each switch after the message has been completely received. Assume

switches introduce no delay to data traversing a circuit. You may also assume that filesize is a multiple of 1000 B.

- (a) For what filesize  $n$  B is the total number of bytes sent across the network less for circuits than for packets?
  - (b) For what filesize  $n$  B is the total latency incurred before the entire file arrives at the destination less for circuits than for packets?
  - (c) How sensitive are these results to the number of switches along the path? To the bandwidth of the links? To the ratio of packet size to packet header size?
  - (d) How accurate do you think this model of the relative merits of circuits and packets is? Does it ignore important considerations that discredit one or the other approach? If so, what are they?
24. Consider a network with a ring topology, link bandwidths of 100 Mbps, and propagation speed  $2 \times 10^8$  m/s. What would the circumference of the loop be to exactly contain one 1500-byte packet, assuming nodes do not introduce delay? What would the circumference be if there was a node every 100 m, and each node introduced 10 bits of delay?
25. Compare the channel requirements for voice traffic with the requirements for the real-time transmission of music, in terms of bandwidth, delay, and jitter. What would have to improve? By approximately how much? Could any channel requirements be relaxed?
26. For the following, assume that no data compression is done, although in practice this would almost never be the case. For (a) to (c), calculate the bandwidth necessary for transmitting in real time:
- (a) Video at a resolution of  $640 \times 480$ , 3 bytes/pixel, 30 frames/second.
  - (b) Video at a resolution of  $160 \times 120$ , 1 byte/pixel, 5 frames/second.
  - (c) CD-ROM music, assuming one CD holds 75 minutes' worth and takes 650 MB.

- (d) Assume a fax transmits an  $8 \times 10$ -inch black-and-white image at a resolution of 72 pixels per inch. How long would this take over a 14.4-kbps modem?
- ✓ 27. For the following, as in the previous problem, assume that no data compression is done. Calculate the bandwidth necessary for transmitting in real time:
- (a) High-definition video at a resolution of  $1920 \times 1080$ , 24 bits/pixel, 30 frames/second.
  - (b) POTS (plain old telephone service) voice audio of 8-bit samples at 8 KHz.
  - (c) GSM mobile voice audio of 260-bit samples at 50 Hz.
  - (d) HCDH high-definition audio of 24-bit samples at 88.2 kHz.
28. Discuss the relative performance needs of the following applications in terms of average bandwidth, peak bandwidth, latency, jitter, and loss tolerance:
- (a) File server.
  - (b) Print server.
  - (c) Digital library.
  - (d) Routine monitoring of remote weather instruments.
  - (e) Voice.
  - (f) Video monitoring of a waiting room.
  - (g) Television broadcasting.
29. Suppose a shared medium M offers to hosts  $A_1, A_2, \dots, A_N$  in round-robin fashion an opportunity to transmit one packet; hosts that have nothing to send immediately relinquish M. How does this differ from STDM? How does network utilization of this scheme compare with STDM?
- ★ 30. Consider a simple protocol for transferring files over a link. After some initial negotiation, A sends data packets of size 1 KB to B; B then replies with an acknowledgment. A always waits for each ACK before sending the next data packet; this is known as *stop-and-wait*. Packets that are overdue are presumed lost and are retransmitted.

- (a) In the absence of any packet losses or duplications, explain why it is not necessary to include any “sequence number” data in the packet headers.
- (b) Suppose that the link can lose occasional packets, but that packets that do arrive always arrive in the order sent. Is a 2-bit sequence number (that is,  $N \bmod 4$ ) enough for A and B to detect and resend any lost packets? Is a 1-bit sequence number enough?
- (c) Now suppose that the link can deliver out of order and that sometimes a packet can be delivered as much as 1 minute after subsequent packets. How does this change the sequence number requirements?

- ★ 31. Suppose hosts A and B are connected by a link. Host A continuously transmits the current time from a high-precision clock, at a regular rate, fast enough to consume all the available bandwidth. Host B reads these time values and writes them each paired with its own time from a local clock synchronized with A's. Give qualitative examples of B's output assuming the link has
- (a) High bandwidth, high latency, low jitter.
  - (b) Low bandwidth, high latency, high jitter.
  - (c) High bandwidth, low latency, low jitter, occasional lost data.
- For example, a link with zero jitter, a bandwidth high enough to write on every other clock tick, and a latency of 1 tick might yield something like (0000,0001), (0002,0003), (0004,0005).
32. Obtain and build the simplex-talk sample socket program shown in the text. Start one server and one client, in separate windows. While the first client is running, start 10 other clients that connect to the same server; these other clients should most likely be started in the background with their input redirected from a file. What happens to these 10 clients? Do their `connect()`s fail, or time out, or succeed? Do any other calls block? Now let the first client exit. What happens? Try this with the server value `MAX_PENDING` set to 1 as well.
33. Modify the simplex-talk socket program so that each time the client sends a line to the server, the server sends the line back to the client. The client (and server) will now have to make alternating calls to `recv()` and `send()`.

34. Modify the `simplex-talk` socket program so that it uses UDP as the transport protocol, rather than TCP. You will have to change `SOCK_STREAM` to `SOCK_DGRAM` in both the client and the server. Then, in the server, remove the calls to `listen()` and `accept()`, and replace the two nested loops at the end with a single loop that calls `recv()` with socket `s`. Finally, see what happens when two such UDP clients simultaneously connect to the same UDP server, and compare this to the TCP behavior.
35. Investigate the different options and parameters one can set for a TCP connection. (Do “`man tcp`” on Unix.) Experiment with various parameter settings to see how they affect TCP performance.
36. The Unix utility `ping` can be used to find the RTT to various Internet hosts. Read the man page for `ping`, and use it to find the RTT to `www.cs.princeton.edu` in New Jersey and `www.cisco.com` in California. Measure the RTT values at different times of day, and compare the results. What do you think accounts for the differences?
37. The Unix utility `traceroute`, or its Windows equivalent `tracert`, can be used to find the sequence of routers through which a message is routed. Use this to find the path from your site to some others. How well does the number of hops correlate with the RTT times from `ping`? How well does the number of hops correlate with geographical distance?
38. Use `traceroute`, above, to map out some of the routers within your organization (or to verify none is used).

A SYSTEMS APPROACH

EMS APPROACH

A SYSTEMS APPROACH

A SYSTEMS APPROACH

A SYSTEMS APPROACH

# Getting Connected



*It is a mistake to look too far ahead. Only one link in the chain of destiny can be handled at a time.*

—Winston Churchill

In Chapter 1 we saw that networks consist of links interconnecting nodes. One of the fundamental problems we face is how to connect two nodes together. We also introduced the “cloud” abstraction to represent a network without revealing all of its internal complexities. So we also need to address the similar problem of connecting a host to a cloud. This, in effect, is the problem every Internet Service Provider faces when it wants to connect a new customer to the network: how to connect one more nodes to the ISP’s cloud?

## PROBLEM: CONNECTING TO A NETWORK

Whether we want to construct a trivial two-node network with one link or connect the one-billionth host to an existing network like the Internet, we need to address a common set of issues. First, we need some physical medium over which to make the connection. The medium may be a length of wire, a piece of optical fiber, or some less tangible medium (such as air) through which electromagnetic radiation (e.g., radio waves) can be transmitted. It may cover a small area (e.g., an office

building) or a wide area (e.g., transcontinental). Connecting two nodes with a suitable medium is only the first step, however. Five additional problems must be addressed before the nodes can successfully exchange packets.

The first is *encoding* bits onto the transmission medium so that they can be understood by a receiving node. Second is the matter of delineating the sequence of bits transmitted over the link into complete messages that can be delivered to the end node. This is the *framing* problem, and the messages delivered to the end hosts are often called *frames* (or sometimes *packets*). Third, because frames are sometimes corrupted during transmission, it is necessary to detect these errors and take the appropriate action; this is the *error detection* problem. The fourth issue is making a link appear reliable in spite of the fact that it corrupts frames from time to time. Finally, in those cases where the link is shared by multiple hosts—as is often the case with wireless links, for example—it is necessary to mediate access to this link. This is the *media access control* problem.

Although these five issues—encoding, framing, error detection, reliable delivery, and access mediation—can be discussed in the abstract, they are very real problems that are addressed in different ways by different networking technologies. This chapter considers these issues in the context of three specific network technologies: point-to-point links, Carrier Sense Multiple Access (CSMA) networks (of which Ethernet is the most famous example), and wireless networks (for which 802.11 is the most widespread standard<sup>1</sup>). The goal of this chapter is simultaneously to survey the available network technology and to explore these five fundamental issues. We will examine what it takes to make a wide variety of different physical media and link technologies useful as building blocks for the construction of robust, scalable networks.

## 2.1 PERSPECTIVES ON CONNECTING

As we saw in Chapter 1, networks are constructed from two classes of hardware building blocks: *nodes* and *links*. In this chapter, we focus on what it takes to make a useful link, so that large, reliable networks containing millions of links can be built.

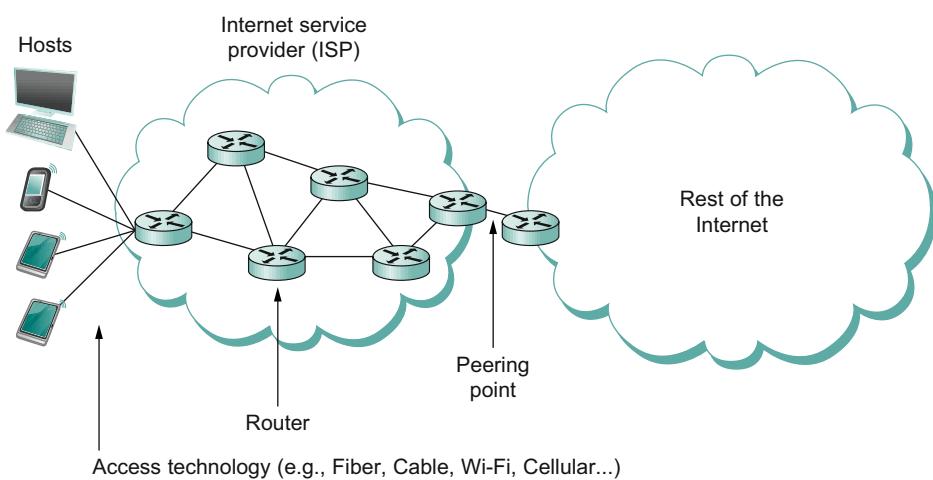
While the operators of large networks deal with links that span hundreds or thousands of kilometers connecting refrigerator-sized routers, the typical user of a network encounters links mostly as a way to connect a computer to the global Internet. Sometimes this link will be a wireless

---

<sup>1</sup>Strictly speaking, 802.11 is a set of standards.

(Wi-Fi) link in a coffee shop; sometimes it is an Ethernet link in a office building or university; for an increasingly large (and fortunate) slice of the population, it is a fiber optic link provided by a telecommunications company or ISP; and many others use some sort of copper wire or cable to connect. Fortunately, there are many common strategies used on these seemingly disparate types of links so that they can all be made reliable and useful to higher layers in the protocol stack. This chapter will examine those strategies.

Figure 2.1 illustrates various types of links as seen by a typical end-user of today's Internet. On the left, we see a variety of end-user devices ranging from mobile phones to PDAs to full-fledged computers connected by various means to an Internet Service Provider. While those links might be of any type mentioned above, or some other type, they all look the same in this picture—a straight line connecting a device to a router. Also, there are some links that connect routers together inside the ISP and a link that connects the ISP to the “rest of the Internet,” which consists of lots of other ISPs and the hosts to which they connect. These links all look alike not just because we're not very good artists but because part of the role of a network architecture (as discussed in Section 1.3) is to provide a common abstraction of something as complex and diverse as a link. The idea is that your laptop or smartphone doesn't have to care what sort of link it is connected to—the only thing that matters is that it has a link



■ FIGURE 2.1 An end-user's view of the Internet.

to the Internet. Similarly, a router doesn't have to care what sort of link connects it to other routers—it can send a packet on the link with a pretty good expectation that the packet will reach the other end of the link.

How do we make all these different types of link look sufficiently alike to end users and routers? Essentially, we have to deal with all the physical limitations and shortcomings of links that exist in the real world. We sketched out some of these issues in the opening problem statement for this chapter. The first issue is that links are made of some physical material that can propagate signals (such as radio waves or other sorts of electromagnetic radiation), but what we really want to do is send *bits*. In later sections of this chapter, we'll look at how to encode bits for transmission on a physical medium, followed by the other issues mentioned above. By the end of this chapter, we'll understand how to send complete packets over just about any sort of link, no matter what physical medium is involved.

### Link Capacity and the Shannon-Hartley Theorem

There has been an enormous body of work done in the related areas of signal processing and information theory, studying everything from how signals degrade over distance to how much data a given signal can effectively carry. The most notable piece of work in this area is a formula known as the *Shannon-Hartley theorem*.<sup>2</sup> Simply stated, this theorem gives an upper bound to the capacity of a link, in terms of bits per second (bps); as a function of the signal-to-noise ratio of the link, measured in decibels (dB); and the bandwidth of the channel, measured in Hertz (Hz). (As noted previously, *bandwidth* is a bit of an overloaded term in communications; here we use it to refer to the range of frequencies available for communication.)

As an example, we can apply the Shannon-Hartley theorem to determine the rate at which a dial-up modem can be expected to transmit binary data over a voice-grade phone line without suffering from too high an error rate. A standard voice-grade phone line typically supports a frequency range of 300 Hz to 3300 Hz, a channel bandwidth of 3 kHz.

The theorem is typically given by the following formula:

$$C = B \log_2(1 + S/N)$$

where  $C$  is the achievable channel capacity measured in bits per second,  $B$  is the bandwidth of the channel in Hz (3300 Hz – 300 Hz = 3000 Hz),  $S$  is the

<sup>2</sup>Sometimes called simply *Shannon's theorem*, but Shannon actually had quite a few theorems.

average signal power, and  $N$  is the average noise power. The signal-to-noise ratio ( $S/N$ , or SNR) is usually expressed in decibels, related as follows:

$$\text{SNR} = 10 \times \log_{10}(S/N)$$

Thus, a typical signal-to-noise ratio of 30 dB would imply that  $S/N = 1000$ . Thus, we have

$$C = 3000 \times \log_2(1001)$$

which equals approximately 30 kbps.

When dial-up modems were the main way to connect to the Internet in the 1990s, 56 kbps was a common advertised capacity for a modem (and continues to be about the upper limit for dial-up). However, the modems often achieved lower speeds in practice, because they didn't always encounter a signal-to-noise ratio high enough to achieve 56 kbps.

The Shannon-Hartley theorem is equally applicable to all sorts of links ranging from wireless to coaxial cable to optical fiber. It should be apparent that there are really only two ways to build a high-capacity link: start with a high-bandwidth channel or achieve a high signal-to-noise ratio, or, preferably, both. Also, even those conditions won't guarantee a high-capacity link—it often takes quite a bit of ingenuity on the part of people who design channel coding schemes to achieve the theoretical limits of a channel. This ingenuity is particularly apparent today in wireless links, where there is a great incentive to get the most bits per second from a given amount of wireless spectrum (the channel bandwidth) and signal power level (and hence SNR).

### 2.1.1 Classes of Links

While most readers of this book have probably encountered at least a few different types of links, it will help to understand some of the broad classes of links that exist and their general properties. For a start, all practical links rely on some sort of electromagnetic radiation propagating through a medium or, in some cases, through free space. One way to characterize links, then, is by the medium they use—typically copper wire in some form, as in Digital Subscriber Line (DSL) and coaxial cable; optical fiber, as in both commercial fiber-to-the-home services and many long-distance links in the Internet's backbone; or air/free space for wireless links.

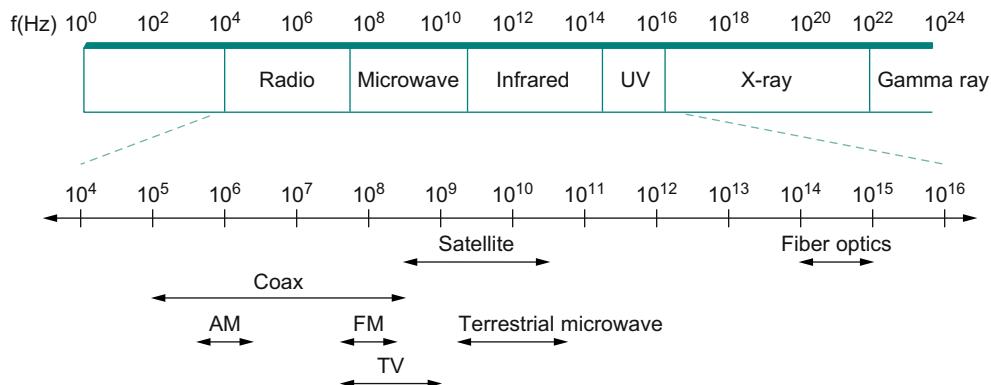
Another important link characteristic is the *frequency*, measured in hertz, with which the electromagnetic waves oscillate. The distance between a pair of adjacent maxima or minima of a wave, typically

measured in meters, is called the wave's *wavelength*. Since all electromagnetic waves travel at the speed of light (which in turn depends on the medium), that speed divided by the wave's frequency is equal to its wavelength. We have already seen the example of a voice-grade telephone line, which carries continuous electromagnetic signals ranging between 300 Hz and 3300 Hz; a 300-Hz wave traveling through copper would have a wavelength of

$$\begin{aligned}\text{SpeedOfLightInCopper} \div \text{Frequency} \\ = 2/3 \times 3 \times 10^8 \div 300 \\ = 667 \times 10^3 \text{ meters}\end{aligned}$$

Generally, electromagnetic waves span a much wider range of frequencies, ranging from radio waves, to infrared light, to visible light, to x-rays and gamma rays. Figure 2.2 depicts the electromagnetic spectrum and shows which media are commonly used to carry which frequency bands.

So far we understand a link to be a physical medium carrying signals in the form of electromagnetic waves. Such links provide the foundation for transmitting all sorts of information, including the kind of data we are interested in transmitting—binary data (1s and 0s). We say that the binary data is *encoded* in the signal. The problem of encoding binary data onto electromagnetic signals is a complex topic. To help make the topic more manageable, we can think of it as being divided into two layers. The lower



■ FIGURE 2.2 Electromagnetic spectrum.

layer is concerned with *modulation*—varying the frequency, amplitude, or phase of the signal to effect the transmission of information. A simple example of modulation is to vary the power (amplitude) of a single wavelength. Intuitively, this is equivalent to turning a light on and off. Because the issue of modulation is secondary to our discussion of links as a building block for computer networks, we simply assume that it is possible to transmit a pair of distinguishable signals—think of them as a “high” signal and a “low” signal—and we consider only the upper layer, which is concerned with the much simpler problem of encoding binary data onto these two signals. Section 2.2 discusses such encodings.

Another way to classify links is in terms of how they are used. Various economic and deployment issues tend to influence where different link types are found. Most consumers interact with the Internet either through wireless networks (which they encounter in coffee shops, airports, universities, etc.) or through so-called “last mile” links provided by Internet Service Providers, as illustrated in Figure 2.1. These link types are summarized in Table 2.1. They typically are chosen because they are cost-effective ways of reaching millions of consumers; DSL, for example, was deployed over the existing twisted pair copper wires that already existed for plain old telephone services. Most of these technologies are not sufficient for building a complete network from scratch—for example, you’ll likely need some long-distance, very high-speed links to interconnect cities in a large network.

Modern long-distance links are almost exclusively fiber today, with coaxial cables having been largely replaced over the last couple of decades. These links typically use a technology called SONET (Synchronous Optical Network), which was developed to meet the demanding

**Table 2.1 Common Services Available to Connect Your Home**

Service	Bandwidth (typical)
Dial-up	28–56 kbps
ISDN	64–128 kbps
DSL	128 kbps–100 Mbps
CATV (cable TV)	1–40 Mbps
FTTH (fibre to the home)	50 Mbps–1 Gbps

management requirements of telephone carriers. We'll take a closer look at SONET in Section 2.3.3.

Finally, in addition to last-mile and backbone links, there are the links that you find inside a building or a campus—generally referred to as *local area networks* (LANs). Ethernet, described in Section 2.6, has for some time been the dominant technology in this space, having displaced token ring technologies after many years. While Ethernet continues to be popular, it is now mostly seen alongside wireless technologies based around the 802.11 standards, which we will discuss in Section 2.7.

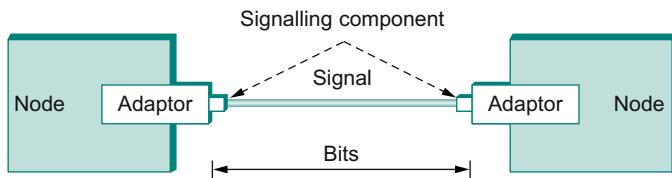
This survey of link types is by no means exhaustive but should have given you a taste of the diversity of link types that exist and some of the reasons for that diversity. In the coming sections, we will see how networking protocols can take advantage of that diversity and present a consistent view of the network to higher layers in spite of all the low-level complexity.

## 2.2 ENCODING (NRZ, NRZI, MANCHESTER, 4B/5B)

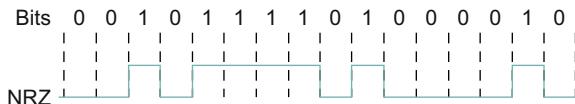
The first step in turning nodes and links into usable building blocks is to understand how to connect them in such a way that bits can be transmitted from one node to the other. As mentioned in the preceding section, signals propagate over physical links. The task, therefore, is to encode the binary data that the source node wants to send into the signals that the links are able to carry and then to decode the signal back into the corresponding binary data at the receiving node. We ignore the details of modulation and assume we are working with two discrete signals: high and low. In practice, these signals might correspond to two different voltages on a copper-based link or two different power levels on an optical link.

Most of the functions discussed in this chapter are performed by a *network adaptor*—a piece of hardware that connects a node to a link. The network adaptor contains a signalling component that actually encodes bits into signals at the sending node and decodes signals into bits at the receiving node. Thus, as illustrated in Figure 2.3, signals travel over a link between two signalling components, and bits flow between network adaptors.

Let's return to the problem of encoding bits onto signals. The obvious thing to do is to map the data value 1 onto the high signal and the data value 0 onto the low signal. This is exactly the mapping used by an encoding scheme called, cryptically enough, *non-return to zero* (NRZ).



■ FIGURE 2.3 Signals travel between signalling components; bits flow between adaptors.



■ FIGURE 2.4 NRZ encoding of a bit stream.

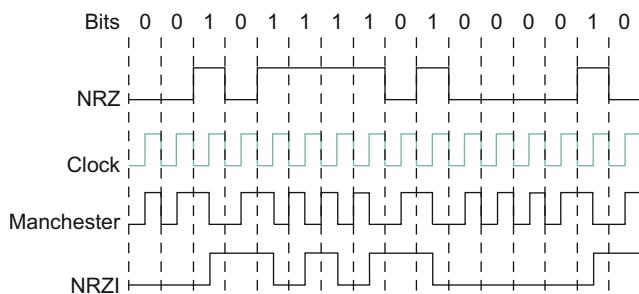
For example, Figure 2.4 schematically depicts the NRZ-encoded signal (bottom) that corresponds to the transmission of a particular sequence of bits (top).

The problem with NRZ is that a sequence of several consecutive 1s means that the signal stays high on the link for an extended period of time; similarly, several consecutive 0s means that the signal stays low for a long time. There are two fundamental problems caused by long strings of 1s or 0s. The first is that it leads to a situation known as *baseline wander*. Specifically, the receiver keeps an average of the signal it has seen so far and then uses this average to distinguish between low and high signals. Whenever the signal is significantly lower than this average, the receiver concludes that it has just seen a 0; likewise, a signal that is significantly higher than the average is interpreted to be a 1. The problem, of course, is that too many consecutive 1s or 0s cause this average to change, making it more difficult to detect a significant change in the signal.

The second problem is that frequent transitions from high to low and *vice versa* are necessary to enable *clock recovery*. Intuitively, the clock recovery problem is that both the encoding and the decoding processes are driven by a clock—every clock cycle the sender transmits a bit and the receiver recovers a bit. The sender's and the receiver's clocks have to be precisely synchronized in order for the receiver to recover the same bits the sender transmits. If the receiver's clock is even slightly faster or slower than the sender's clock, then it does not correctly decode the signal. You could imagine sending the clock to the receiver over a separate wire, but

this is typically avoided because it makes the cost of cabling twice as high. So, instead, the receiver derives the clock from the received signal—the clock recovery process. Whenever the signal changes, such as on a transition from 1 to 0 or from 0 to 1, then the receiver knows it is at a clock cycle boundary, and it can resynchronize itself. However, a long period of time without such a transition leads to clock drift. Thus, clock recovery depends on having lots of transitions in the signal, no matter what data is being sent.

One approach that addresses this problem, called *non-return to zero inverted* (NRZI), has the sender make a transition from the current signal to encode a 1 and stay at the current signal to encode a 0. This solves the problem of consecutive 1s, but obviously does nothing for consecutive 0s. NRZI is illustrated in Figure 2.5. An alternative, called *Manchester encoding*, does a more explicit job of merging the clock with the signal by transmitting the exclusive OR of the NRZ-encoded data and the clock. (Think of the local clock as an internal signal that alternates from low to high; a low/high pair is considered one clock cycle.) The Manchester encoding is also illustrated in Figure 2.5. Observe that the Manchester encoding results in 0 being encoded as a low-to-high transition and 1 being encoded as a high-to-low transition. Because both 0s and 1s result in a transition to the signal, the clock can be effectively recovered at the receiver. (There is also a variant of the Manchester encoding, called *Differential Manchester*, in which a 1 is encoded with the first half of the signal equal to the last half of the previous bit's signal and a 0 is encoded with the first half of the signal opposite to the last half of the previous bit's signal.)



■ FIGURE 2.5 Different encoding strategies.

The problem with the Manchester encoding scheme is that it doubles the rate at which signal transitions are made on the link, which means that the receiver has half the time to detect each pulse of the signal. The rate at which the signal changes is called the link's *baud rate*. In the case of the Manchester encoding, the bit rate is half the baud rate, so the encoding is considered only 50% efficient. Keep in mind that if the receiver had been able to keep up with the faster baud rate required by the Manchester encoding in Figure 2.5, then both NRZ and NRZI could have been able to transmit twice as many bits in the same time period.

A final encoding that we consider, called *4B/5B*, attempts to address the inefficiency of the Manchester encoding without suffering from the problem of having extended durations of high or low signals. The idea of 4B/5B is to insert extra bits into the bit stream so as to break up long sequences of 0s or 1s. Specifically, every 4 bits of actual data are encoded in a 5-bit code that is then transmitted to the receiver; hence, the name 4B/5B. The 5-bit codes are selected in such a way that each one has no more than one leading 0 and no more than two trailing 0s. Thus, when sent back-to-back, no pair of 5-bit codes results in more than three consecutive 0s being transmitted. The resulting 5-bit codes are then transmitted using the NRZI encoding, which explains why the code is only concerned about consecutive 0s—NRZI already solves the problem of consecutive 1s. Note that the 4B/5B encoding results in 80% efficiency.

Table 2.2 gives the 5-bit codes that correspond to each of the 16 possible 4-bit data symbols. Notice that since 5 bits are enough to encode 32 different codes, and we are using only 16 of these for data, there are 16 codes left over that we can use for other purposes. Of these, code 11111 is used when the line is idle, code 00000 corresponds to when the line is dead, and 00100 is interpreted to mean halt. Of the remaining 13 codes, 7 of them are not valid because they violate the “one leading 0, two trailing 0s,” rule, and the other 6 represent various control symbols. As we will see later in this chapter, some framing protocols make use of these control symbols.

## 2.3 FRAMING

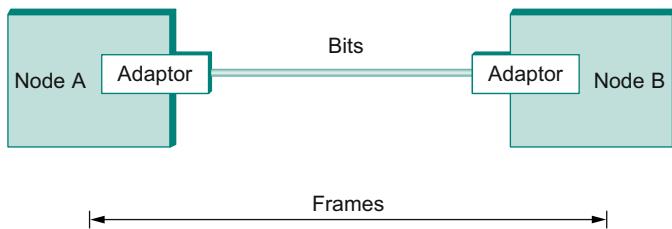
Now that we have seen how to transmit a sequence of bits over a point-to-point link—from adaptor to adaptor—let's consider the scenario illustrated in Figure 2.6. Recall from Chapter 1 that we are focusing

**Table 2.2 4B/5B Encoding**

4-Bit Data Symbol	5-Bit Code
0000	11110
0001	01001
0010	10100
0011	10101
0100	01010
0101	01011
0110	01110
0111	01111
1000	10010
1001	10011
1010	10110
1011	10111
1100	11010
1101	11011
1110	11100
1111	11101

on packet-switched networks, which means that blocks of data (called *frames* at this level), not bit streams, are exchanged between nodes. It is the network adaptor that enables the nodes to exchange frames. When node A wishes to transmit a frame to node B, it tells its adaptor to transmit a frame from the node's memory. This results in a sequence of bits being sent over the link. The adaptor on node B then collects together the sequence of bits arriving on the link and deposits the corresponding frame in B's memory. Recognizing exactly what set of bits constitutes a frame—that is, determining where the frame begins and ends—is the central challenge faced by the adaptor.

There are several ways to address the framing problem. This section uses several different protocols to illustrate the various points in the design space. Note that while we discuss framing in the context of point-to-point links, the problem is a fundamental one that must also be addressed in multiple-access networks like Ethernet and token rings.



■ FIGURE 2.6 Bits flow between adaptors, frames between hosts.

### 2.3.1 Byte-Oriented Protocols (BISYNC, PPP, DDCMP)

One of the oldest approaches to framing—it has its roots in connecting terminals to mainframes—is to view each frame as a collection of bytes (characters) rather than a collection of bits. Such a *byte-oriented* approach is exemplified by older protocols such as the Binary Synchronous Communication (BISYNC) protocol developed by IBM in the late 1960s, and the Digital Data Communication Message Protocol (DDCMP) used in Digital Equipment Corporation's DECNET. The more recent and widely used Point-to-Point Protocol (PPP) provides another example of this approach.

#### *Sentinel-Based Approaches*

Figure 2.7 illustrates the BISYNC protocol's frame format. This figure is the first of many that you will see in this book that are used to illustrate frame or packet formats, so a few words of explanation are in order. We show a packet as a sequence of labeled fields. Above each field is a number indicating the length of that field in bits. Note that the packets are transmitted beginning with the leftmost field.

BISYNC uses special characters known as *sentinel characters* to indicate where frames start and end. The beginning of a frame is denoted by sending a special SYN (synchronization) character. The data portion of the frame is then contained between two more special characters: STX (start of text) and ETX (end of text). The SOH (start of header) field serves much the same purpose as the STX field. The problem with the sentinel approach, of course, is that the ETX character might appear in the data portion of the frame. BISYNC overcomes this problem by “escaping” the ETX character by preceding it with a DLE (data-link-escape) character whenever it appears in the body of a frame; the DLE character is also



■ FIGURE 2.7 BISYNC frame format.

escaped (by preceding it with an extra DLE) in the frame body. (C programmers may notice that this is analogous to the way a quotation mark is escaped by the backslash when it occurs inside a string.) This approach is often called *character stuffing* because extra characters are inserted in the data portion of the frame.

The frame format also includes a field labeled CRC (cyclic redundancy check), which is used to detect transmission errors; various algorithms for error detection are presented in Section 2.4. Finally, the frame contains additional header fields that are used for, among other things, the link-level reliable delivery algorithm. Examples of these algorithms are given in Section 2.5.

The more recent Point-to-Point Protocol (PPP), which is commonly used to carry Internet Protocol packets over various sorts of point-to-point links, is similar to BISYNC in that it also uses sentinels and character stuffing. The format for a PPP frame is given in Figure 2.8. The special start-of-text character, denoted as the Flag field in Figure 2.8, is 01111110. The Address and Control fields usually contain default values and so are uninteresting. The Protocol field is used for demultiplexing; it identifies the high-level protocol such as IP or IPX (an IP-like protocol developed by Novell). The frame payload size can be negotiated, but it is 1500 bytes by default. The Checksum field is either 2 (by default) or 4 bytes long.

The PPP frame format is unusual in that several of the field sizes are negotiated rather than fixed. This negotiation is conducted by a protocol called the Link Control Protocol (LCP). PPP and LCP work in tandem: LCP sends control messages encapsulated in PPP frames—such messages are denoted by an LCP identifier in the PPP Protocol field—and then turns around and changes PPP's frame format based on the information contained in those control messages. LCP is also involved in establishing a link between two peers when both sides detect that communication over the link is possible (e.g., when each optical receiver detects an incoming signal from the fiber to which it connects).



■ FIGURE 2.8 PPP frame format.



■ FIGURE 2.9 DDCMP frame format.

### Byte-Counting Approach

As every Computer Science 101 student knows, the alternative to detecting the end of a file with a sentinel value is to include the number of items in the file at the beginning of the file. The same is true in framing—the number of bytes contained in a frame can be included as a field in the frame header. The DECNET’s DDCMP uses this approach, as illustrated in Figure 2.9. In this example, the COUNT field specifies how many bytes are contained in the frame’s body.

One danger with this approach is that a transmission error could corrupt the count field, in which case the end of the frame would not be correctly detected. (A similar problem exists with the sentinel-based approach if the ETX field becomes corrupted.) Should this happen, the receiver will accumulate as many bytes as the bad COUNT field indicates and then use the error detection field to determine that the frame is bad. This is sometimes called a *framing error*. The receiver will then wait until it sees the next SYN character to start collecting the bytes that make up the next frame. It is therefore possible that a framing error will cause back-to-back frames to be incorrectly received.

### 2.3.2 Bit-Oriented Protocols (HDLC)

Unlike these byte-oriented protocols, a bit-oriented protocol is not concerned with byte boundaries—it simply views the frame as a collection of bits. These bits might come from some character set, such as ASCII; they might be pixel values in an image; or they could be instructions and operands from an executable file. The Synchronous Data Link Control

(SDLC) protocol developed by IBM is an example of a bit-oriented protocol; SDLC was later standardized by the ISO as the High-Level Data Link Control (HDLC) protocol. In the following discussion, we use HDLC as an example; its frame format is given in Figure 2.10.

HDLC denotes both the beginning and the end of a frame with the distinguished bit sequence 01111110. This sequence is also transmitted during any times that the link is idle so that the sender and receiver can keep their clocks synchronized. In this way, both protocols essentially use the sentinel approach. Because this sequence might appear anywhere in the body of the frame—in fact, the bits 01111110 might cross byte boundaries—bit-oriented protocols use the analog of the DLE character, a technique known as *bit stuffing*.

Bit stuffing in the HDLC protocol works as follows. On the sending side, any time five consecutive 1s have been transmitted from the body of the message (i.e., excluding when the sender is trying to transmit the distinguished 01111110 sequence), the sender inserts a 0 before transmitting the next bit. On the receiving side, should five consecutive 1s arrive, the receiver makes its decision based on the next bit it sees (i.e., the bit following the five 1s). If the next bit is a 0, it must have been stuffed, and so the receiver removes it. If the next bit is a 1, then one of two things is true: Either this is the end-of-frame marker or an error has been introduced into the bit stream. By looking at the *next* bit, the receiver can distinguish between these two cases. If it sees a 0 (i.e., the last 8 bits it has looked at are 01111110), then it is the end-of-frame marker; if it sees a 1 (i.e., the last 8 bits it has looked at are 01111111), then there must have been an error and the whole frame is discarded. In the latter case, the receiver has to wait for the next 01111110 before it can start receiving again, and, as a consequence, there is the potential that the receiver will fail to receive two consecutive frames. Obviously, there are still ways that framing errors can go undetected, such as when an entire spurious end-of-frame pattern is generated by errors, but these failures are relatively unlikely. Robust ways of detecting errors are discussed in Section 2.4.



■ FIGURE 2.10 HDLC frame format.

An interesting characteristic of bit stuffing, as well as character stuffing, is that the size of a frame is dependent on the data that is being sent in the payload of the frame. It is in fact not possible to make all frames exactly the same size, given that the data that might be carried in any frame is arbitrary. (To convince yourself of this, consider what happens if the last byte of a frame's body is the ETX character.) A form of framing that ensures that all frames are the same size is described in the next subsection.

### What's in a Layer?

One of the important contributions of the OSI reference model presented in Chapter 1 was providing some vocabulary for talking about protocols and, in particular, protocol layers. This vocabulary has provided fuel for plenty of arguments along the lines of “Your protocol does function X at layer Y, and the OSI reference model says it should be done at layer Z—that’s a layer violation.” In fact, figuring out the right layer at which to perform a given function can be very difficult, and the reasoning is usually a lot more subtle than “What does the OSI model say?” It is partly for this reason that this book avoids a rigidly layerist approach. Instead, it shows you a lot of functions that need to be performed by protocols and looks at some ways that they have been successfully implemented.

In spite of our nonlayerist approach, sometimes we need convenient ways to talk about classes of protocols, and the name of the layer at which they operate is often the best choice. Thus, for example, this chapter focuses primarily on link-layer protocols. (Bit encoding, described in Section 2.2, is the exception, being considered a physical-layer function.) Link-layer protocols can be identified by the fact that they run over single links—the type of network discussed in this chapter. Network-layer protocols, by contrast, run over switched networks that contain lots of links interconnected by switches or routers. Topics related to network-layer protocols are discussed in Chapters 3 and 4.

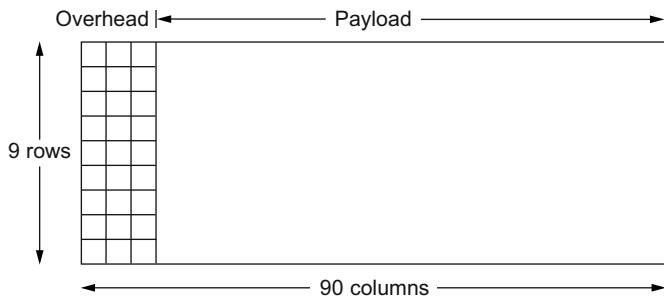
Note that protocol layers are supposed to be helpful—they provide helpful ways to talk about classes of protocols, and they help us divide the problem of building networks into manageable subtasks. However, they are not meant to be overly restrictive—the mere fact that something is a layer violation does not end the argument about whether it is a worthwhile thing to do. In other words, layering makes a good slave, but a poor master. A particularly interesting argument about the best layer in which to place a certain function comes up when we look at congestion control in Chapter 6.

### 2.3.3 Clock-Based Framing (SONET)

A third approach to framing is exemplified by the Synchronous Optical Network (SONET) standard. For lack of a widely accepted generic term, we refer to this approach simply as *clock-based framing*. SONET was first proposed by Bell Communications Research (Bellcore), and then developed under the American National Standards Institute (ANSI) for digital transmission over optical fiber; it has since been adopted by the ITU-T. SONET has been for many years the dominant standard for long-distance transmission of data over optical networks.

An important point to make about SONET before we go any further is that the full specification is substantially larger than this book. Thus, the following discussion will necessarily cover only the high points of the standard. Also, SONET addresses both the framing problem and the encoding problem. It also addresses a problem that is very important for phone companies—the multiplexing of several low-speed links onto one high-speed link. (In fact, much of SONET's design reflects the fact that phone companies have to be concerned with multiplexing large numbers of the 64-kbps channels that traditionally are used for telephone calls.) We begin with SONET's approach to framing and discuss the other issues following.

As with the previously discussed framing schemes, a SONET frame has some special information that tells the receiver where the frame starts and ends; however, that is about as far as the similarities go. Notably, no bit stuffing is used, so that a frame's length does not depend on the data being sent. So the question to ask is “How does the receiver know where each frame starts and ends?” We consider this question for the lowest-speed SONET link, which is known as STS-1 and runs at 51.84 Mbps. An STS-1 frame is shown in [Figure 2.11](#). It is arranged as 9 rows of 90 bytes each, and the first 3 bytes of each row are overhead, with the rest being available for data that is being transmitted over the link. The first 2 bytes of the frame contain a special bit pattern, and it is these bytes that enable the receiver to determine where the frame starts. However, since bit stuffing is not used, there is no reason why this pattern will not occasionally turn up in the payload portion of the frame. To guard against this, the receiver looks for the special bit pattern consistently, hoping to see it appearing once every 810 bytes, since each frame is  $9 \times 90 = 810$  bytes long. When



■ FIGURE 2.11 A SONET STS-1 frame.

the special pattern turns up in the right place enough times, the receiver concludes that it is in sync and can then interpret the frame correctly.

One of the things we are not describing due to the complexity of SONET is the detailed use of all the other overhead bytes. Part of this complexity can be attributed to the fact that SONET runs across the carrier's optical network, not just over a single link. (Recall that we are glossing over the fact that the carriers implement a network, and we are instead focusing on the fact that we can lease a SONET link from them and then use this link to build our own packet-switched network.) Additional complexity comes from the fact that SONET provides a considerably richer set of services than just data transfer. For example, 64 kbps of a SONET link's capacity is set aside for a voice channel that is used for maintenance.

The overhead bytes of a SONET frame are encoded using NRZ, the simple encoding described in the previous section where 1s are high and 0s are low. However, to ensure that there are plenty of transitions to allow the receiver to recover the sender's clock, the payload bytes are *scrambled*. This is done by calculating the exclusive OR (XOR) of the data to be transmitted and by the use of a well-known bit pattern. The bit pattern, which is 127 bits long, has plenty of transitions from 1 to 0, so that XORing it with the transmitted data is likely to yield a signal with enough transitions to enable clock recovery.

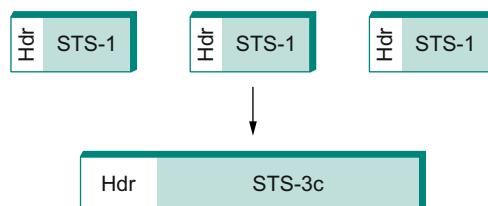
SONET supports the multiplexing of multiple low-speed links in the following way. A given SONET link runs at one of a finite set of possible rates, ranging from 51.84 Mbps (STS-1) to 2488.32 Mbps (STS-48), and beyond. Note that all of these rates are integer multiples of STS-1. The

significance for framing is that a single SONET frame can contain sub-frames for multiple lower-rate channels. A second related feature is that each frame is 125  $\mu$ s long. This means that at STS-1 rates, a SONET frame is 810 bytes long, while at STS-3 rates, each SONET frame is 2430 bytes long. Notice the synergy between these two features:  $3 \times 810 = 2430$ , meaning that three STS-1 frames fit exactly in a single STS-3 frame.

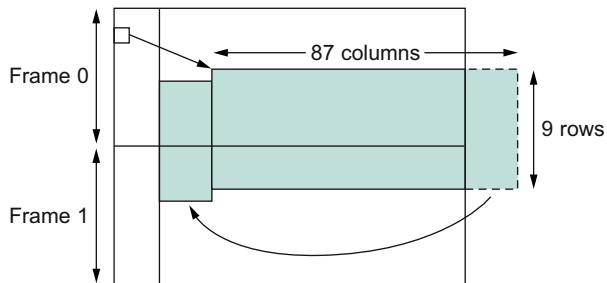
Intuitively, the STS- $N$  frame can be thought of as consisting of  $N$  STS-1 frames, where the bytes from these frames are interleaved; that is, a byte from the first frame is transmitted, then a byte from the second frame is transmitted, and so on. The reason for interleaving the bytes from each STS- $N$  frame is to ensure that the bytes in each STS-1 frame are evenly paced; that is, bytes show up at the receiver at a smooth 51 Mbps, rather than all bunched up during one particular  $1/N$ th of the 125- $\mu$ s interval.

Although it is accurate to view an STS- $N$  signal as being used to multiplex  $N$  STS-1 frames, the payload from these STS-1 frames can be linked together to form a larger STS- $N$  payload; such a link is denoted STS- $Nc$  (for *concatenated*). One of the fields in the overhead is used for this purpose. Figure 2.12 schematically depicts concatenation in the case of three STS-1 frames being concatenated into a single STS-3c frame. The significance of a SONET link being designated as STS-3c rather than STS-3 is that, in the former case, the user of the link can view it as a single 155.25-Mbps pipe, whereas an STS-3 should really be viewed as three 51.84-Mbps links that happen to share a fiber.

Finally, the preceding description of SONET is overly simplistic in that it assumes that the payload for each frame is completely contained within the frame. (Why wouldn't it be?) In fact, we should view the STS-1 frame just described as simply a placeholder for the frame, where the actual payload may *float* across frame boundaries. This situation is illustrated



■ FIGURE 2.12 Three STS-1 frames multiplexed onto one STS-3c frame.



■ FIGURE 2.13 SONET frames out of phase.

in Figure 2.13. Here we see both the STS-1 payload floating across two STS-1 frames and the payload shifted some number of bytes to the right and, therefore, wrapped around. One of the fields in the frame overhead points to the beginning of the payload. The value of this capability is that it simplifies the task of synchronizing the clocks used throughout the carriers' networks, which is something that carriers spend a lot of their time worrying about.

## 2.4 ERROR DETECTION

As discussed in Chapter 1, bit errors are sometimes introduced into frames. This happens, for example, because of electrical interference or thermal noise. Although errors are rare, especially on optical links, some mechanism is needed to detect these errors so that corrective action can be taken. Otherwise, the end user is left wondering why the C program that successfully compiled just a moment ago now suddenly has a syntax error in it, when all that happened in the interim is that it was copied across a network file system.

There is a long history of techniques for dealing with bit errors in computer systems, dating back to at least the 1940s. Hamming and Reed-Solomon codes are two notable examples that were developed for use in punch card readers, when storing data on magnetic disks, and in early core memories. This section describes some of the error detection techniques most commonly used in networking.

Detecting errors is only one part of the problem. The other part is correcting errors once detected. Two basic approaches can be taken when the recipient of a message detects an error. One is to notify the sender

that the message was corrupted so that the sender can retransmit a copy of the message. If bit errors are rare, then in all probability the retransmitted copy will be error free. Alternatively, some types of error detection algorithms allow the recipient to reconstruct the correct message even after it has been corrupted; such algorithms rely on *error-correcting codes*, discussed below.

One of the most common techniques for detecting transmission errors is a technique known as the *cyclic redundancy check* (CRC). It is used in nearly all the link-level protocols discussed in the previous section (e.g., HDLC, DDCMP), as well as in the CSMA and wireless protocols described later in this chapter. Section 2.4.3 outlines the basic CRC algorithm. Before discussing that approach, we consider two simpler schemes: *two-dimensional parity* and *checksums*. The former is used by the BISYNC protocol when it is transmitting ASCII characters (CRC is used as the error-detecting code when BISYNC is used to transmit EBCDIC<sup>3</sup>), and the latter is used by several Internet protocols.

The basic idea behind any error detection scheme is to add redundant information to a frame that can be used to determine if errors have been introduced. In the extreme, we could imagine transmitting two complete copies of the data. If the two copies are identical at the receiver, then it is probably the case that both are correct. If they differ, then an error was introduced into one (or both) of them, and they must be discarded. This is a rather poor error detection scheme for two reasons. First, it sends  $n$  redundant bits for an  $n$ -bit message. Second, many errors will go undetected—any error that happens to corrupt the same bit positions in the first and second copies of the message. In general, the goal of error detecting codes is to provide a high probability of detecting errors combined with a relatively low number of redundant bits.

Fortunately, we can do a lot better than this simple scheme. In general, we can provide quite strong error detection capability while sending only  $k$  redundant bits for an  $n$ -bit message, where  $k \ll n$ . On an Ethernet, for example, a frame carrying up to 12,000 bits (1500 bytes) of data requires only a 32-bit CRC code, or as it is commonly expressed, uses CRC-32. Such a code will catch the overwhelming majority of errors, as we will see below.

---

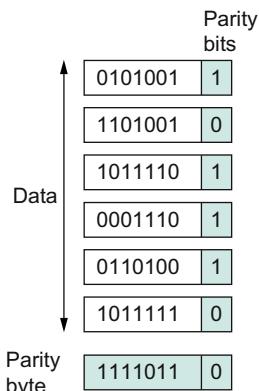
<sup>3</sup>An alternative character encoding scheme used in the 1960s.

We say that the extra bits we send are redundant because they add no new information to the message. Instead, they are derived directly from the original message using some well-defined algorithm. Both the sender and the receiver know exactly what that algorithm is. The sender applies the algorithm to the message to generate the redundant bits. It then transmits both the message and those few extra bits. When the receiver applies the same algorithm to the received message, it should (in the absence of errors) come up with the same result as the sender. It compares the result with the one sent to it by the sender. If they match, it can conclude (with high likelihood) that no errors were introduced in the message during transmission. If they do not match, it can be sure that either the message or the redundant bits were corrupted, and it must take appropriate action—that is, discarding the message or correcting it if that is possible.

One note on the terminology for these extra bits. In general, they are referred to as *error-detecting codes*. In specific cases, when the algorithm to create the code is based on addition, they may be called a *checksum*. We will see that the Internet checksum is appropriately named: It is an error check that uses a summing algorithm. Unfortunately, the word *checksum* is often used imprecisely to mean any form of error-detecting code, including CRCs. This can be confusing, so we urge you to use the word *checksum* only to apply to codes that actually do use addition and to use *error-detecting code* to refer to the general class of codes described in this section.

### 2.4.1 Two-Dimensional Parity

Two-dimensional parity is exactly what the name suggests. It is based on “simple” (one-dimensional) parity, which usually involves adding one extra bit to a 7-bit code to balance the number of 1s in the byte. For example, odd parity sets the eighth bit to 1 if needed to give an odd number of 1s in the byte, and even parity sets the eighth bit to 1 if needed to give an even number of 1s in the byte. Two-dimensional parity does a similar calculation for each bit position across each of the bytes contained in the frame. This results in an extra parity byte for the entire frame, in addition to a parity bit for each byte. Figure 2.14 illustrates how two-dimensional even parity works for an example frame containing 6 bytes of data. Notice that the third bit of the parity byte is 1 since there is an odd number of 1s in the third bit across the 6 bytes in the frame. It can be shown that two-dimensional parity catches all 1-, 2-, and 3-bit errors, and most 4-bit



■ FIGURE 2.14 Two-dimensional parity.

errors. In this case, we have added 14 bits of redundant information to a 42-bit message, and yet we have stronger protection against common errors than the “repetition code” described above.

### 2.4.2 Internet Checksum Algorithm

A second approach to error detection is exemplified by the Internet checksum. Although it is not used at the link level, it nevertheless provides the same sort of functionality as CRCs and parity, so we discuss it here. We will see examples of its use in Sections 3.2, 5.1, and 5.2.

The idea behind the Internet checksum is very simple—you add up all the words that are transmitted and then transmit the result of that sum. The result is the checksum. The receiver performs the same calculation on the received data and compares the result with the received checksum. If any transmitted data, including the checksum itself, is corrupted, then the results will not match, so the receiver knows that an error occurred.

You can imagine many different variations on the basic idea of a checksum. The exact scheme used by the Internet protocols works as follows. Consider the data being checksummed as a sequence of 16-bit integers. Add them together using 16-bit ones complement arithmetic (explained below) and then take the ones complement of the result. That 16-bit number is the checksum.

In ones complement arithmetic, a negative integer ( $-x$ ) is represented as the complement of  $x$ ; that is, each bit of  $x$  is inverted. When adding

numbers in ones complement arithmetic, a carryout from the most significant bit needs to be added to the result. Consider, for example, the addition of  $-5$  and  $-3$  in ones complement arithmetic on 4-bit integers:  $+5$  is 0101, so  $-5$  is 1010;  $+3$  is 0011, so  $-3$  is 1100. If we add 1010 and 1100, ignoring the carry, we get 0110. In ones complement arithmetic, the fact that this operation caused a carry from the most significant bit causes us to increment the result, giving 0111, which is the ones complement representation of  $-8$  (obtained by inverting the bits in 1000), as we would expect.

The following routine gives a straightforward implementation of the Internet's checksum algorithm. The `count` argument gives the length of `buf` measured in 16-bit units. The routine assumes that `buf` has already been padded with 0s to a 16-bit boundary.

```
u_short
cksum(u_short *buf, int count)
{
    register u_long sum = 0;

    while (count--)
    {
        sum += *buf++;
        if (sum & 0xFFFF0000)
        {
            /* carry occurred,
               so wrap around */
            sum &= 0xFFFF;
            sum++;
        }
    }
    return ~(sum & 0xFFFF);
}
```

This code ensures that the calculation uses ones complement arithmetic rather than the twos complement that is used in most machines. Note the `if` statement inside the `while` loop. If there is a carry into the top 16 bits of `sum`, then we increment `sum` just as in the previous example.

Compared to our repetition code, this algorithm scores well for using a small number of redundant bits—only 16 for a message of any length—but it does not score extremely well for strength of error detection. For example, a pair of single-bit errors, one of which increments a word and one of which decrements another word by the same amount, will go undetected. The reason for using an algorithm like this in spite of its relatively weak protection against errors (compared to a CRC, for example) is simple: This algorithm is much easier to implement in software. Experience in the ARPANET suggested that a checksum of this form was adequate. One reason it is adequate is that this checksum is the last line of defense in an end-to-end protocol; the majority of errors are picked up by stronger error detection algorithms, such as CRCs, at the link level.

### Simple Probability Calculations

When dealing with network errors and other unlikely (we hope) events, we often have use for simple back-of-the-envelope probability estimates. A useful approximation here is that if two independent events have *small* probabilities  $p$  and  $q$ , then the probability of either event is  $p + q$ ; the exact answer is  $1 - (1 - p)(1 - q) = p + q - pq$ . For  $p = q = .01$ , this estimate is .02, while the exact value is .0199.

For a simple application of this, suppose that the per-bit error rate on a link is 1 in  $10^7$ . Now suppose we are interested in estimating the probability of at least one bit in a 10,000-bit packet being errored. Using the above approximation repeatedly over all the bits, we can say that we are interested in the probability of the first bit being errored, or the second bit, or the third, etc. Assuming bit errors are all independent (which they aren't), we can therefore estimate that the probability of at least one error in a 10,000-bit ( $10^4$  bit) packet is  $10^4 \times 10^{-7} = 10^{-3}$ . The exact answer, computed as  $1 - P(\text{no errors})$ , would be  $1 - (1 - 10^{-7})^{10,000} = .00099950$ .

For a slightly more complex application, we compute the probability of exactly two errors in such a packet; this is the probability of an error that would sneak past a 1-parity-bit checksum. If we consider two particular bits in the packet, say bit  $i$  and bit  $j$ , the probability of those exact bits being errored is  $10^{-7} \times 10^{-7}$ . Now the total number of possible bit pairs in the packet is  $\binom{10^4}{2} = 10^4 \times (10^4 - 1)/2 \approx 5 \times 10^7$ . So again using the approximation of repeatedly adding the probabilities of many rare events (in this case, of any possible bit pair being errored), our total probability of at least two errored bits is  $5 \times 10^7 \times 10^{-14} = 5 \times 10^{-7}$ .

### 2.4.3 Cyclic Redundancy Check

It should be clear by now that a major goal in designing error detection algorithms is to maximize the probability of detecting errors using only a small number of redundant bits. Cyclic redundancy checks use some fairly powerful mathematics to achieve this goal. For example, a 32-bit CRC gives strong protection against common bit errors in messages that are thousands of bytes long. The theoretical foundation of the cyclic redundancy check is rooted in a branch of mathematics called *finite fields*. While this may sound daunting, the basic ideas can be easily understood.

To start, think of an  $(n + 1)$ -bit message as being represented by an  $n$  degree polynomial, that is, a polynomial whose highest-order term is  $x^n$ . The message is represented by a polynomial by using the value of each bit in the message as the coefficient for each term in the polynomial, starting with the most significant bit to represent the highest-order term. For example, an 8-bit message consisting of the bits 10011010 corresponds to the polynomial

$$\begin{aligned}M(x) &= 1 \times x^7 + 0 \times x^6 + 0 \times x^5 + 1 \times x^4 \\&\quad + 1 \times x^3 + 0 \times x^2 + 1 \times x^1 \\&\quad + 0 \times x^0 \\&= x^7 + x^4 + x^3 + x^1\end{aligned}$$

We can thus think of a sender and a receiver as exchanging polynomials with each other.

For the purposes of calculating a CRC, a sender and receiver have to agree on a *divisor* polynomial,  $C(x)$ .  $C(x)$  is a polynomial of degree  $k$ . For example, suppose  $C(x) = x^3 + x^2 + 1$ . In this case,  $k = 3$ . The answer to the question “Where did  $C(x)$  come from?” is, in most practical cases, “You look it up in a book.” In fact, the choice of  $C(x)$  has a significant impact on what types of errors can be reliably detected, as we discuss below. There are a handful of divisor polynomials that are very good choices for various environments, and the exact choice is normally made as part of the protocol design. For example, the Ethernet standard uses a well-known polynomial of degree 32.

When a sender wishes to transmit a message  $M(x)$  that is  $n + 1$  bits long, what is actually sent is the  $(n + 1)$ -bit message plus  $k$  bits. We call the complete transmitted message, including the redundant bits,  $P(x)$ . What

we are going to do is contrive to make the polynomial representing  $P(x)$  exactly divisible by  $C(x)$ ; we explain how this is achieved below. If  $P(x)$  is transmitted over a link and there are no errors introduced during transmission, then the receiver should be able to divide  $P(x)$  by  $C(x)$  exactly, leaving a remainder of zero. On the other hand, if some error is introduced into  $P(x)$  during transmission, then in all likelihood the received polynomial will no longer be exactly divisible by  $C(x)$ , and thus the receiver will obtain a nonzero remainder implying that an error has occurred.

It will help to understand the following if you know a little about polynomial arithmetic; it is just slightly different from normal integer arithmetic. We are dealing with a special class of polynomial arithmetic here, where coefficients may be only one or zero, and operations on the coefficients are performed using modulo 2 arithmetic. This is referred to as “polynomial arithmetic modulo 2.” Since this is a networking book, not a mathematics text, let’s focus on the key properties of this type of arithmetic for our purposes (which we ask you to accept on faith):

- Any polynomial  $B(x)$  can be divided by a divisor polynomial  $C(x)$  if  $B(x)$  is of higher degree than  $C(x)$ .
- Any polynomial  $B(x)$  can be divided once by a divisor polynomial  $C(x)$  if  $B(x)$  is of the same degree as  $C(x)$ .
- The remainder obtained when  $B(x)$  is divided by  $C(x)$  is obtained by performing the exclusive OR (XOR) operation on each pair of matching coefficients.

For example, the polynomial  $x^3 + 1$  can be divided by  $x^3 + x^2 + 1$  (because they are both of degree 3) and the remainder would be  $0 \times x^3 + 1 \times x^2 + 0 \times x^1 + 0 \times x^0 = x^2$  (obtained by XORing the coefficients of each term). In terms of messages, we could say that 1001 can be divided by 1101 and leaves a remainder of 0100. You should be able to see that the remainder is just the bitwise exclusive OR of the two messages.

Now that we know the basic rules for dividing polynomials, we are able to do long division, which is necessary to deal with longer messages. An example appears below.

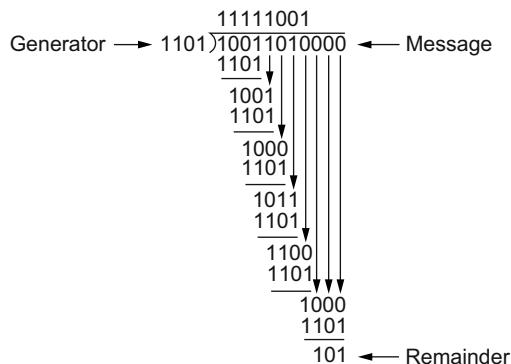
Recall that we wanted to create a polynomial for transmission that is derived from the original message  $M(x)$ , is  $k$  bits longer than  $M(x)$ , and is exactly divisible by  $C(x)$ . We can do this in the following way:

1. Multiply  $M(x)$  by  $x^k$ ; that is, add  $k$  zeros at the end of the message.  
Call this zero-extended message  $T(x)$ .
2. Divide  $T(x)$  by  $C(x)$  and find the remainder.
3. Subtract the remainder from  $T(x)$ .

It should be obvious that what is left at this point is a message that is exactly divisible by  $C(x)$ . We may also note that the resulting message consists of  $M(x)$  followed by the remainder obtained in step 2, because when we subtracted the remainder (which can be no more than  $k$  bits long), we were just XORing it with the  $k$  zeros added in step 1. This part will become clearer with an example.

Consider the message  $x^7 + x^4 + x^3 + x^1$ , or 10011010. We begin by multiplying by  $x^3$ , since our divisor polynomial is of degree 3. This gives 10011010000. We divide this by  $C(x)$ , which corresponds to 1101 in this case. Figure 2.15 shows the polynomial long-division operation. Given the rules of polynomial arithmetic described above, the long-division operation proceeds much as it would if we were dividing integers. Thus, in the first step of our example, we see that the divisor 1101 divides once into the first four bits of the message (1001), since they are of the same degree, and leaves a remainder of 100 (1101 XOR 1001). The next step is to bring down a digit from the message polynomial until we get another polynomial with the same degree as  $C(x)$ , in this case 1001. We calculate the remainder again (100) and continue until the calculation is complete. Note that the “result” of the long division, which appears at the top of the calculation, is not really of much interest—it is the remainder at the end that matters.

You can see from the very bottom of Figure 2.15 that the remainder of the example calculation is 101. So we know that 10011010000 minus 101 would be exactly divisible by  $C(x)$ , and this is what we send. The minus operation in polynomial arithmetic is the logical XOR operation, so we actually send 10011010101. As noted above, this turns out to be just the original message with the remainder from the long division calculation appended to it. The recipient divides the received polynomial by  $C(x)$  and, if the result is 0, concludes that there were no errors. If the result is nonzero, it may be necessary to discard the corrupted message; with some codes, it may be possible to *correct* a small error (e.g., if the error affected only one bit). A code that enables error correction is called an *error-correcting code* (ECC).



■ FIGURE 2.15 CRC calculation using polynomial long division.

Now we will consider the question of where the polynomial  $C(x)$  comes from. Intuitively, the idea is to select this polynomial so that it is very unlikely to divide evenly into a message that has errors introduced into it. If the transmitted message is  $P(x)$ , we may think of the introduction of errors as the addition of another polynomial  $E(x)$ , so the recipient sees  $P(x) + E(x)$ . The only way that an error could slip by undetected would be if the received message could be evenly divided by  $C(x)$ , and since we know that  $P(x)$  can be evenly divided by  $C(x)$ , this could only happen if  $E(x)$  can be divided evenly by  $C(x)$ . The trick is to pick  $C(x)$  so that this is very unlikely for common types of errors.

One common type of error is a single-bit error, which can be expressed as  $E(x) = x^i$  when it affects bit position  $i$ . If we select  $C(x)$  such that the first and the last term (that is, the  $x^k$  and  $x^0$  terms) are nonzero, then we already have a two-term polynomial that cannot divide evenly into the one term  $E(x)$ . Such a  $C(x)$  can, therefore, detect all single-bit errors. In general, it is possible to prove that the following types of errors can be detected by a  $C(x)$  with the stated properties:

- All single-bit errors, as long as the  $x^k$  and  $x^0$  terms have nonzero coefficients
- All double-bit errors, as long as  $C(x)$  has a factor with at least three terms
- Any odd number of errors, as long as  $C(x)$  contains the factor  $(x + 1)$

### Error Detection or Error Correction?

We have mentioned that it is possible to use codes that not only detect the presence of errors but also enable errors to be corrected. Since the details of such codes require yet more complex mathematics than that required to understand CRCs, we will not dwell on them here. However, it is worth considering the merits of correction versus detection.

At first glance, it would seem that correction is always better, since with detection we are forced to throw away the message and, in general, ask for another copy to be transmitted. This uses up bandwidth and may introduce latency while waiting for the retransmission. However, there is a downside to correction, as it generally requires a greater number of redundant bits to send an error-correcting code that is as strong (that is, able to cope with the same range of errors) as a code that only detects errors. Thus, while error detection requires more bits to be sent when errors occur, error correction requires more bits to be sent *all the time*. As a result, error correction tends to be most useful when (1) errors are quite probable, as they may be, for example, in a wireless environment, or (2) the cost of retransmission is too high, for example, because of the latency involved retransmitting a packet over a satellite link.

The use of error-correcting codes in networking is sometimes referred to as *forward error correction* (FEC) because the correction of errors is handled “in advance” by sending extra information, rather than waiting for errors to happen and dealing with them later by retransmission. FEC is commonly used in wireless networks such as 802.11.

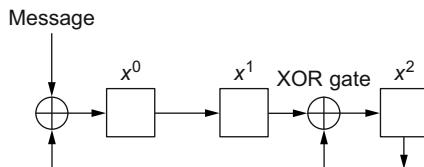
- Any “burst” error (i.e., sequence of consecutive errored bits) for which the length of the burst is less than  $k$  bits (Most burst errors of length greater than  $k$  bits can also be detected.)

Six versions of  $C(x)$  are widely used in link-level protocols (shown in Table 2.3). For example, Ethernet uses CRC-32, while HDLC uses CRC-CCITT. ATM, as described in Chapter 3, uses CRC-8, CRC-10, and CRC-32.

Finally, we note that the CRC algorithm, while seemingly complex, is easily implemented in hardware using a  $k$ -bit shift register and XOR gates. The number of bits in the shift register equals the degree of the generator polynomial ( $k$ ). Figure 2.16 shows the hardware that would be used for the generator  $x^3 + x^2 + 1$  from our previous example. The message is shifted

**Table 2.3 Common CRC Polynomials**

CRC	$C(x)$
CRC-8	$x^8 + x^2 + x^1 + 1$
CRC-10	$x^{10} + x^9 + x^5 + x^4 + x^1 + 1$
CRC-12	$x^{12} + x^{11} + x^3 + x^2 + x + 1$
CRC-16	$x^{16} + x^{15} + x^2 + 1$
CRC-CCITT	$x^{16} + x^{12} + x^5 + 1$
CRC-32	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11}$ $+ x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$

**FIGURE 2.16** CRC calculation using shift register.

in from the left, beginning with the most significant bit and ending with the string of  $k$  zeros that is attached to the message, just as in the long division example. When all the bits have been shifted in and appropriately XORed, the register contains the remainder—that is, the CRC (most significant bit on the right). The position of the XOR gates is determined as follows: If the bits in the shift register are labeled 0 through  $k - 1$ , left to right, then put an XOR gate in front of bit  $n$  if there is a term  $x^n$  in the generator polynomial. Thus, we see an XOR gate in front of positions 0 and 2 for the generator  $x^3 + x^2 + x^0$ .

## 2.5 RELIABLE TRANSMISSION

As we saw in the previous section, frames are sometimes corrupted while in transit, with an error code like CRC used to detect such errors. While some error codes are strong enough also to correct errors, in practice the overhead is typically too large to handle the range of bit and burst errors

that can be introduced on a network link. Even when error-correcting codes are used (e.g., on wireless links) some errors will be too severe to be corrected. As a result, some corrupt frames must be discarded. A link-level protocol that wants to deliver frames reliably must somehow recover from these discarded (lost) frames.

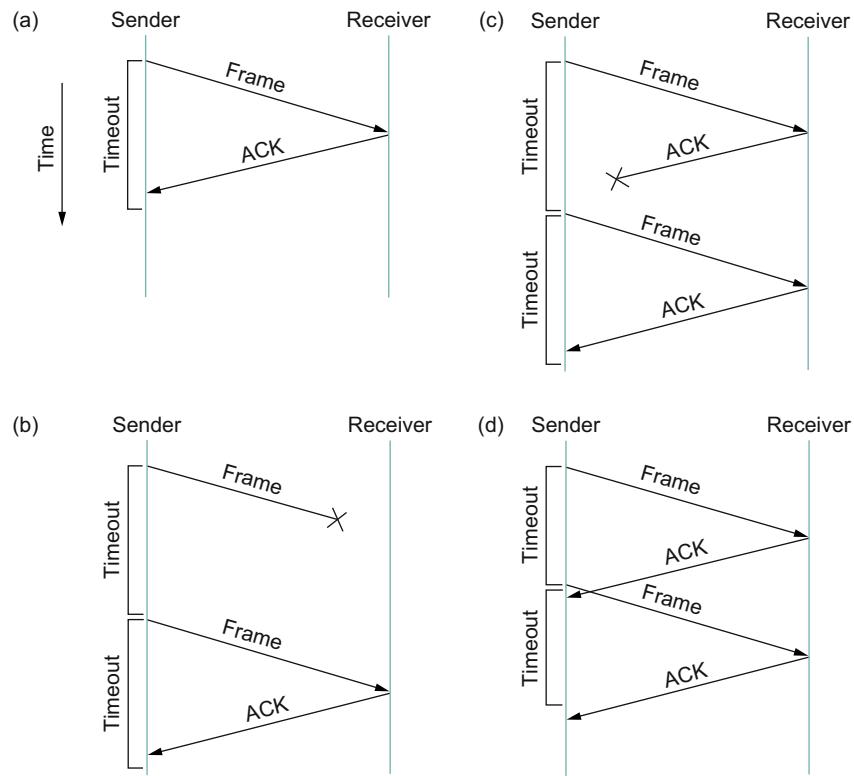
It's worth noting that reliability is a function that *may* be provided at the link level, but many modern link technologies omit this function. Furthermore, reliable delivery is frequently provided at higher levels, including both transport (as described in Section 5.2) and, sometimes, the application layer (Chapter 9). Exactly where it should be provided is a matter of some debate and depends on many factors. We describe the basics of reliable delivery here, since the principles are common across layers, but you should be aware that we're not just talking about a link-layer function (see the “What's in a Layer?” sidebar above for more on this).

This is usually accomplished using a combination of two fundamental mechanisms—*acknowledgments* and *timeouts*. An acknowledgment (ACK for short) is a small control frame that a protocol sends back to its peer saying that it has received an earlier frame. By control frame we mean a header without any data, although a protocol can *piggyback* an ACK on a data frame it just happens to be sending in the opposite direction. The receipt of an acknowledgment indicates to the sender of the original frame that its frame was successfully delivered. If the sender does not receive an acknowledgment after a reasonable amount of time, then it *retransmits* the original frame. This action of waiting a reasonable amount of time is called a *timeout*.

The general strategy of using acknowledgments and timeouts to implement reliable delivery is sometimes called *automatic repeat request* (normally abbreviated ARQ). This section describes three different ARQ algorithms using generic language; that is, we do not give detailed information about a particular protocol's header fields.

### 2.5.1 Stop-and-Wait

The simplest ARQ scheme is the *stop-and-wait* algorithm. The idea of stop-and-wait is straightforward: After transmitting one frame, the sender waits for an acknowledgment before transmitting the next frame. If the acknowledgment does not arrive after a certain period of time, the sender times out and retransmits the original frame.



■ **FIGURE 2.17** Timeline showing four different scenarios for the stop-and-wait algorithm. (a) The ACK is received before the timer expires; (b) and (c) show the situation in which the original frame and the ACK, respectively, are lost; (d) shows the situation in which the timeout fires too soon.

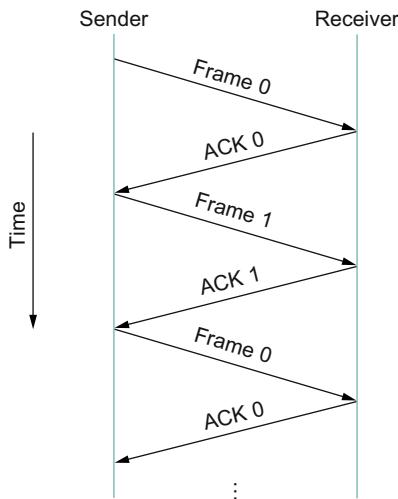
Figure 2.17 illustrates four different scenarios that result from this basic algorithm. This figure is a timeline, a common way to depict a protocol's behavior (see also the sidebar on this sort of diagram). The sending side is represented on the left, the receiving side is depicted on the right, and time flows from top to bottom. Figure 2.17(a) shows the situation in which the ACK is received before the timer expires; (b) and (c) show the situation in which the original frame and the ACK, respectively, are lost; and (d) shows the situation in which the timeout fires too soon. Recall that by "lost" we mean that the frame was corrupted while in transit, that this corruption was detected by an error code on the receiver, and that the frame was subsequently discarded.

### Timelines and Packet Exchange Diagrams

Figures 2.17 and 2.18 are two examples of a frequently used tool in teaching, explaining, and designing protocols: the timeline or packet exchange diagram. You are going to see many more of them in this book—see Figure 9.9 for a more complex example. They are very useful because they capture visually the behavior over time of a distributed system—something that can be quite hard to analyze. When designing a protocol, you often have to be prepared for the unexpected—a system crashes, a message gets lost, or something that you expected to happen quickly turns out to take a long time. These sorts of diagrams can often help us understand what might go wrong in such cases and thus help a protocol designer be prepared for every eventuality.

There is one important subtlety in the stop-and-wait algorithm. Suppose the sender sends a frame and the receiver acknowledges it, but the acknowledgment is either lost or delayed in arriving. This situation is illustrated in timelines (c) and (d) of Figure 2.17. In both cases, the sender times out and retransmits the original frame, but the receiver will think that it is the next frame, since it correctly received and acknowledged the first frame. This has the potential to cause duplicate copies of a frame to be delivered. To address this problem, the header for a stop-and-wait protocol usually includes a 1-bit sequence number—that is, the sequence number can take on the values 0 and 1—and the sequence numbers used for each frame alternate, as illustrated in Figure 2.18. Thus, when the sender retransmits frame 0, the receiver can determine that it is seeing a second copy of frame 0 rather than the first copy of frame 1 and therefore can ignore it (the receiver still acknowledges it, in case the first ACK was lost).

The main shortcoming of the stop-and-wait algorithm is that it allows the sender to have only one outstanding frame on the link at a time, and this may be far below the link's capacity. Consider, for example, a 1.5-Mbps link with a 45-ms round-trip time. This link has a delay  $\times$  bandwidth product of 67.5 Kb, or approximately 8 KB. Since the sender can send only one frame per RTT, and assuming a frame size of 1 KB, this



■ FIGURE 2.18 Timeline for stop-and-wait with 1-bit sequence number.

implies a maximum sending rate of

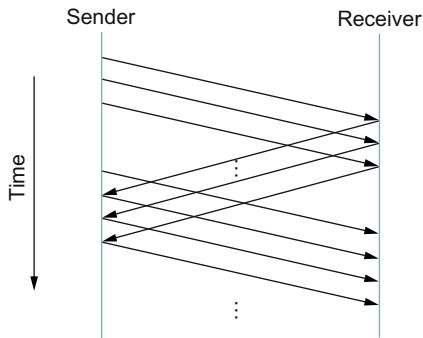
$$\begin{aligned}
 & \text{Bits Per Frame} \div \text{Time Per Frame} \\
 &= 1024 \times 8 \div 0.045 \\
 &= 182 \text{ kbps}
 \end{aligned}$$

or about one-eighth of the link's capacity. To use the link fully, then, we'd like the sender to be able to transmit up to eight frames before having to wait for an acknowledgment.

The significance of the  $\text{delay} \times \text{bandwidth}$  product is that it represents the amount of data that could be in transit. We would like to be able to send this much data without waiting for the first acknowledgment. The principle at work here is often referred to as *keeping the pipe full*. The algorithms presented in the following two subsections do exactly this.

## 2.5.2 Sliding Window

Consider again the scenario in which the link has a  $\text{delay} \times \text{bandwidth}$  product of 8 KB and frames are 1 KB in size. We would like the sender to be ready to transmit the ninth frame at pretty much the same moment that the ACK for the first frame arrives. The algorithm that allows us to



■ FIGURE 2.19 Timeline for the sliding window algorithm.

do this is called *sliding window*, and an illustrative timeline is given in Figure 2.19.

#### *The Sliding Window Algorithm*

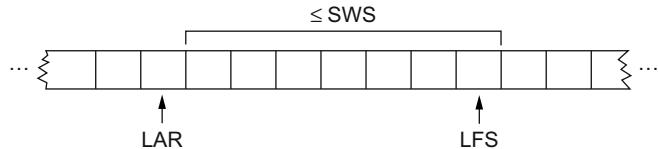
The sliding window algorithm works as follows. First, the sender assigns a *sequence number*, denoted SeqNum, to each frame. For now, let's ignore the fact that SeqNum is implemented by a finite-size header field and instead assume that it can grow infinitely large. The sender maintains three variables: The *send window size*, denoted SWS, gives the upper bound on the number of outstanding (unacknowledged) frames that the sender can transmit; LAR denotes the sequence number of the *last acknowledgment received*; and LFS denotes the sequence number of the *last frame sent*. The sender also maintains the following invariant:

$$\text{LFS} - \text{LAR} \leq \text{SWS}$$

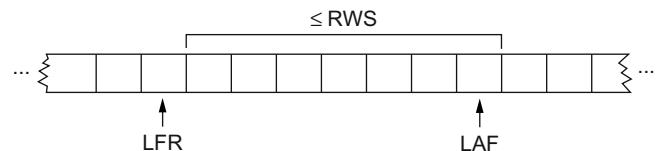
This situation is illustrated in Figure 2.20.

When an acknowledgment arrives, the sender moves LAR to the right, thereby allowing the sender to transmit another frame. Also, the sender associates a timer with each frame it transmits, and it retransmits the frame should the timer expire before an ACK is received. Notice that the sender has to be willing to buffer up to SWS frames since it must be prepared to retransmit them until they are acknowledged.

The receiver maintains the following three variables: The *receive window size*, denoted RWS, gives the upper bound on the number of out-of-order frames that the receiver is willing to accept; LAF denotes the sequence number of the *largest acceptable frame*; and LFR denotes the



■ FIGURE 2.20 Sliding window on sender.



■ FIGURE 2.21 Sliding window on receiver.

sequence number of the *last frame received*. The receiver also maintains the following invariant:

$$\text{LAF} - \text{LFR} \leq \text{RWS}$$

This situation is illustrated in Figure 2.21.

When a frame with sequence number SeqNum arrives, the receiver takes the following action. If  $\text{SeqNum} \leq \text{LFR}$  or  $\text{SeqNum} > \text{LAF}$ , then the frame is outside the receiver's window and it is discarded. If  $\text{LFR} < \text{SeqNum} \leq \text{LAF}$ , then the frame is within the receiver's window and it is accepted. Now the receiver needs to decide whether or not to send an ACK. Let SeqNumToAck denote the largest sequence number not yet acknowledged, such that all frames with sequence numbers less than or equal to SeqNumToAck have been received. The receiver acknowledges the receipt of SeqNumToAck, even if higher numbered packets have been received. This acknowledgment is said to be cumulative. It then sets  $\text{LFR} = \text{SeqNumToAck}$  and adjusts  $\text{LAF} = \text{LFR} + \text{RWS}$ .

For example, suppose  $\text{LFR} = 5$  (i.e., the last ACK the receiver sent was for sequence number 5), and  $\text{RWS} = 4$ . This implies that  $\text{LAF} = 9$ . Should frames 7 and 8 arrive, they will be buffered because they are within the receiver's window. However, no ACK needs to be sent since frame 6 has yet to arrive. Frames 7 and 8 are said to have arrived out of order. (Technically, the receiver could resend an ACK for frame 5 when frames 7 and 8 arrive.)

Should frame 6 then arrive—perhaps it is late because it was lost the first time and had to be retransmitted, or perhaps it was simply delayed<sup>4</sup>—the receiver acknowledges frame 8, bumps LFR to 8, and sets LAF to 12. If frame 6 was in fact lost, then a timeout will have occurred at the sender, causing it to retransmit frame 6.

We observe that when a timeout occurs, the amount of data in transit decreases, since the sender is unable to advance its window until frame 6 is acknowledged. This means that when packet losses occur, this scheme is no longer keeping the pipe full. The longer it takes to notice that a packet loss has occurred, the more severe this problem becomes.

Notice that, in this example, the receiver could have sent a *negative acknowledgment* (NAK) for frame 6 as soon as frame 7 arrived. However, this is unnecessary since the sender's timeout mechanism is sufficient to catch this situation, and sending NAKs adds additional complexity to the receiver. Also, as we mentioned, it would have been legitimate to send additional acknowledgments of frame 5 when frames 7 and 8 arrived; in some cases, a sender can use duplicate ACKs as a clue that a frame was lost. Both approaches help to improve performance by allowing early detection of packet losses.

Yet another variation on this scheme would be to use *selective acknowledgments*. That is, the receiver could acknowledge exactly those frames it has received rather than just the highest numbered frame received in order. So, in the above example, the receiver could acknowledge the receipt of frames 7 and 8. Giving more information to the sender makes it potentially easier for the sender to keep the pipe full but adds complexity to the implementation.

The sending window size is selected according to how many frames we want to have outstanding on the link at a given time; SWS is easy to compute for a given delay  $\times$  bandwidth product.<sup>5</sup> On the other hand, the receiver can set RWS to whatever it wants. Two common settings are RWS = 1, which implies that the receiver will not buffer any frames that arrive out of order, and RWS = SWS, which implies that the receiver can

---

<sup>4</sup>It's unlikely that a packet could be delayed in this way on a point-to-point link, but later on we will see this same algorithm used on more complex networks where such delays are possible.

<sup>5</sup>Easy, that is, if we know the delay and the bandwidth. Sometimes we do not, and estimating them well is a challenge to protocol designers. We discuss this further in Chapter 5.

buffer any of the frames the sender transmits. It makes no sense to set  $\text{RWS} > \text{SWS}$  since it's impossible for more than  $\text{SWS}$  frames to arrive out of order.

#### *Finite Sequence Numbers and Sliding Window*

We now return to the one simplification we introduced into the algorithm—our assumption that sequence numbers can grow infinitely large. In practice, of course, a frame's sequence number is specified in a header field of some finite size. For example, a 3-bit field means that there are eight possible sequence numbers,  $0 \dots 7$ . This makes it necessary to reuse sequence numbers or, stated another way, sequence numbers wrap around. This introduces the problem of being able to distinguish between different incarnations of the same sequence numbers, which implies that the number of possible sequence numbers must be larger than the number of outstanding frames allowed. For example, stop-and-wait allowed one outstanding frame at a time and had two distinct sequence numbers.

Suppose we have one more number in our space of sequence numbers than we have potentially outstanding frames; that is,  $\text{SWS} \leq \text{MaxSeqNum} - 1$ , where  $\text{MaxSeqNum}$  is the number of available sequence numbers. Is this sufficient? The answer depends on  $\text{RWS}$ . If  $\text{RWS} = 1$ , then  $\text{MaxSeqNum} \geq \text{SWS} + 1$  is sufficient. If  $\text{RWS}$  is equal to  $\text{SWS}$ , then having a  $\text{MaxSeqNum}$  just one greater than the sending window size is not good enough. To see this, consider the situation in which we have the eight sequence numbers 0 through 7, and  $\text{SWS} = \text{RWS} = 7$ . Suppose the sender transmits frames  $0 \dots 6$ , they are successfully received, but the ACKs are lost. The receiver is now expecting frames  $7, 0 \dots 5$ , but the sender times out and sends frames  $0 \dots 6$ . Unfortunately, the receiver is expecting the second incarnation of frames  $0 \dots 5$  but gets the first incarnation of these frames. This is exactly the situation we wanted to avoid.

It turns out that the sending window size can be no more than half as big as the number of available sequence numbers when  $\text{RWS} = \text{SWS}$ , or stated more precisely,

$$\text{SWS} < (\text{MaxSeqNum} + 1)/2$$

Intuitively, what this is saying is that the sliding window protocol alternates between the two halves of the sequence number space, just as stop-and-wait alternates between sequence numbers 0 and 1. The only

difference is that it continually slides between the two halves rather than discretely alternating between them.

Note that this rule is specific to the situation where  $RWS = SWS$ . We leave it as an exercise to determine the more general rule that works for arbitrary values of  $RWS$  and  $SWS$ . Also note that the relationship between the window size and the sequence number space depends on an assumption that is so obvious that it is easy to overlook, namely that frames are not reordered in transit. This cannot happen on a direct point-to-point link since there is no way for one frame to overtake another during transmission. However, we will see the sliding window algorithm used in a different environment in Chapter 5, and we will need to devise another rule.

### *Implementation of Sliding Window*

The following routines illustrate how we might implement the sending and receiving sides of the sliding window algorithm. The routines are taken from a working protocol named, appropriately enough, Sliding Window Protocol (SWP). So as not to concern ourselves with the adjacent protocols in the protocol graph, we denote the protocol sitting above SWP as the high-level protocol (HLP) and the protocol sitting below SWP as the link-level protocol (LLP).

We start by defining a pair of data structures. First, the frame header is very simple: It contains a sequence number (`SeqNum`) and an acknowledgment number (`AckNum`). It also contains a `Flags` field that indicates whether the frame is an ACK or carries data.

```
typedef u_char SwpSeqno;

typedef struct {
    SwpSeqno SeqNum;      /* sequence number of this frame */
    SwpSeqno AckNum;     /* ack of received frame */
    u_char    Flags;      /* up to 8 bits worth of flags */
} SwpHdr;
```

Next, the state of the sliding window algorithm has the following structure. For the sending side of the protocol, this state includes variables `LAR` and `LFS`, as described earlier in this section, as well as a queue that holds frames that have been transmitted but not yet acknowledged (`sendQ`). The sending state also includes a *counting semaphore* called `sendWindowNotFull`. We will see how this is used below, but generally

a semaphore is a synchronization primitive that supports `semWait` and `semSignal` operations. Every invocation of `semSignal` increments the semaphore by 1, and every invocation of `semWait` decrements `s` by 1, with the calling process blocked (suspended) should decrementing the semaphore cause its value to become less than 0. A process that is blocked during its call to `semWait` will be allowed to resume as soon as enough `semSignal` operations have been performed to raise the value of the semaphore above 0.

For the receiving side of the protocol, the state includes the variable `NFE`. This is the *next frame expected*, the frame with a sequence number one more than the last frame received (LFR), described earlier in this section. There is also a queue that holds frames that have been received out of order (`recvQ`). Finally, although not shown, the sender and receiver sliding window sizes are defined by constants `SWS` and `RWS`, respectively.

```

typedef struct {
    /* sender side state: */
    SwpSeqno     LAR;           /* seqno of last ACK
                                   received */
    SwpSeqno     LFS;           /* last frame sent */
    Semaphore    sendWindowNotFull;
    SwpHdr       hdr;           /* pre-initialized header */
    struct sendQ_slot {
        Event      timeout;    /* event associated with send
                               -timeout */
        Msg       msg;
    } sendQ[SWS];
}

/* receiver side state: */
SwpSeqno     NFE;           /* seqno of next frame
                           expected */
struct recvQ_slot {
    int       received; /* is msg valid? */
    Msg       msg;
} recvQ[RWS];
} SwpState;

```

The sending side of SWP is implemented by procedure `sendSWP`. This routine is rather simple. First, `semWait` causes this process to block on a

semaphore until it is OK to send another frame. Once allowed to proceed, `sendSWP` sets the sequence number in the frame's header, saves a copy of the frame in the transmit queue (`sendQ`), schedules a timeout event to handle the case in which the frame is not acknowledged, and sends the frame to the next-lower-level protocol, which we denote as `LINK`.

One detail worth noting is the call to `store_swp_hdr` just before the call to `msgAddHdr`. This routine translates the C structure that holds the SWP header (`state->hdr`) into a byte string that can be safely attached to the front of the message (`hbuf`). This routine (not shown) must translate each integer field in the header into network byte order and remove any padding that the compiler has added to the C structure. The issue of byte order is discussed more fully in Section 7.1, but for now it is enough to assume that this routine places the most significant bit of a multiword integer in the byte with the highest address.

Another piece of complexity in this routine is the use of `semWait` and the `sendWindowNotFull` semaphore. `sendWindowNotFull` is initialized to the size of the sender's sliding window, `SWS` (this initialization is not shown). Each time the sender transmits a frame, the `semWait` operation decrements this count and blocks the sender should the count go to 0. Each time an ACK is received, the `semSignal` operation invoked in `deliverSWP` (see below) increments this count, thus unblocking any waiting sender.

```
static int
sendSWP(SwpState *state, Msg *frame)
{
    struct sendQ_slot *slot;
    hbuf[HLEN];

    /* wait for send window to open */
    semWait(&state->sendWindowNotFull);
    state->hdr.SeqNum = ++state->LFS;
    slot = &state->sendQ[state->hdr.SeqNum % SWS];
    store_swp_hdr(state->hdr, hbuf);
    msgAddHdr(frame, hbuf, HLEN);
    msgSaveCopy(&slot->msg, frame);
    slot->timeout = evSchedule(swpTimeout, slot,
        SWP_SEND_TIMEOUT);
    return send(LINK, frame);
}
```

Before continuing to the receive side of SWP, we need to reconcile a seeming inconsistency. On the one hand, we have been saying that a high-level protocol invokes the services of a low-level protocol by calling the `send` operation, so we would expect that a protocol that wants to send a message via SWP would call `send(SWP, packet)`. On the other hand, the procedure that implements SWP's `send` operation is called `sendSWP`, and its first argument is a state variable (`SwpState`). What gives? The answer is that the operating system provides glue code that translates the generic call to `send` into a protocol-specific call to `sendSWP`. This glue code maps the first argument to `send` (the magic protocol variable `SWP`) into both a function pointer to `sendSWP` and a pointer to the protocol state that SWP needs to do its job. The reason we have the high-level protocol indirectly invoke the protocol-specific function through the generic function call is that we want to limit how much information the high-level protocol has coded in it about the low-level protocol. This makes it easier to change the protocol graph configuration at some time in the future.

Now we move on to SWP's protocol-specific implementation of the `deliver` operation, which is given in procedure `deliverSWP`. This routine actually handles two different kinds of incoming messages: ACKs for frames sent earlier from this node and data frames arriving at this node. In a sense, the ACK half of this routine is the counterpart to the sender side of the algorithm given in `sendSWP`. A decision as to whether the incoming message is an ACK or a data frame is made by checking the `Flags` field in the header. Note that this particular implementation does not support piggybacking ACKs on data frames.

When the incoming frame is an ACK, `deliverSWP` simply finds the slot in the transmit queue (`sendQ`) that corresponds to the ACK, cancels the timeout event, and frees the frame saved in that slot. This work is actually done in a loop since the ACK may be cumulative. The only other thing to notice about this case is the call to subroutine `swpInWindow`. This subroutine, which is given below, ensures that the sequence number for the frame being acknowledged is within the range of ACKs that the sender currently expects to receive.

When the incoming frame contains data, `deliverSWP` first calls `msgStripHdr` and `load_swp_hdr` to extract the header from the frame. Routine `load_swp_hdr` is the counterpart to `store_swp_hdr` discussed earlier; it translates a byte string into the C data structure that holds the SWP header. `deliverSWP` then calls `swpInWindow` to make sure the sequence

number of the frame is within the range of sequence numbers that it expects. If it is, the routine loops over the set of consecutive frames it has received and passes them up to the higher-level protocol by invoking the `deliverHLP` routine. It also sends a cumulative ACK back to the sender, but does so by looping over the receive queue (it does not use the `SqNumToAck` variable used in the prose description given earlier in this section).

```
static int
deliverSWP(SwpState state, Msg *frame)
{
    SwpHdr    hdr;
    char      *hbuf;

    hbuf = msgStripHdr(frame, HLEN);
    load_swp_hdr(&hdr, hbuf)
    if (hdr->Flags & FLAG_ACK_VALID)
    {
        /* received an acknowledgment---do SENDER side */
        if (swpInWindow(hdr.AckNum, state->LAR + 1,
                        state->LFS))
        {
            do
            {
                struct sendQ_slot *slot;

                slot = &state->sendQ[++state->LAR % SWS];
                evCancel(slot->timeout);
                msgDestroy(&slot->msg);
                semSignal(&state->sendWindowNotFull);
            } while (state->LAR != hdr.AckNum);
        }
    }

    if (hdr.Flags & FLAG_HAS_DATA)
    {
        struct recvQ_slot *slot;

        /* received data packet---do RECEIVER side */
    }
}
```

```

slot = &state->recvQ[hdr.SeqNum % RWS];
if (!swpInWindow(hdr.SeqNum, state->NFE,
                  state->NFE + RWS - 1))
{
    /* drop the message */
    return SUCCESS;
}
msgSaveCopy(&slot->msg, frame);
slot->received = TRUE;
if (hdr.SeqNum == state->NFE)
{
    Msg m;

    while (slot->received)
    {
        deliver(HLP, &slot->msg);
        msgDestroy(&slot->msg);
        slot->received = FALSE;
        slot = &state->recvQ[++state->NFE % RWS];
    }
    /* send ACK: */
    prepare_ack(&m, state->NFE - 1);
    send(LINK, &m);
    msgDestroy(&m);
}
return SUCCESS;
}

```

Finally, `swpInWindow` is a simple subroutine that checks to see if a given sequence number falls between some minimum and maximum sequence number.

```

static bool
swpInWindow(SwpSeqno seqno, SwpSeqno min, SwpSeqno max)
{
    SwpSeqno pos, maxpos;

```

```
    pos      = seqno - min;          /* pos *should* be in range [0..MAX) */
    maxpos = max - min + 1;         /* maxpos is in range [0..MAX] */
    return pos < maxpos;
}
```

### Frame Order and Flow Control

The sliding window protocol is perhaps the best known algorithm in computer networking. What is easily confusing about the algorithm, however, is that it can be used to serve three different roles. The first role is the one we have been concentrating on in this section—to reliably deliver frames across an unreliable link. (In general, the algorithm can be used to reliably deliver messages across an unreliable network.) This is the core function of the algorithm.

The second role that the sliding window algorithm can serve is to preserve the order in which frames are transmitted. This is easy to do at the receiver—since each frame has a sequence number, the receiver just makes sure that it does not pass a frame up to the next-higher-level protocol until it has already passed up all frames with a smaller sequence number. That is, the receiver buffers (i.e., does not pass along) out-of-order frames. The version of the sliding window algorithm described in this section does preserve frame order, although we could imagine a variation in which the receiver passes frames to the next protocol without waiting for all earlier frames to be delivered. A question we should ask ourselves is whether we really need the sliding window protocol to keep the frames in order, or whether, instead, this is unnecessary functionality at the link level. Unfortunately, we have not yet seen enough of the network architecture to answer this question; we first need to understand how a sequence of point-to-point links is connected by switches to form an end-to-end path.

The third role that the sliding window algorithm sometimes plays is to support *flow control*—a feedback mechanism by which the receiver is able to throttle the sender. Such a mechanism is used to keep the sender from over-running the receiver—that is, from transmitting more data than the receiver is able to process. This is usually accomplished by augmenting the sliding window protocol so that the receiver not only acknowledges frames it has received but also informs the sender of how many frames it has room to receive. The number of frames that the receiver is capable of receiving corresponds to how much free buffer

space it has. As in the case of ordered delivery, we need to make sure that flow control is necessary at the link level before incorporating it into the sliding window protocol.



One important concept to take away from this discussion is the system design principle we call *separation of concerns*. That is, you must be careful to distinguish between different functions that are sometimes rolled together in one mechanism, and you must make sure that each function is necessary and being supported in the most effective way. In this particular case, reliable delivery, ordered delivery, and flow control are sometimes combined in a single sliding window protocol, and we should ask ourselves if this is the right thing to do at the link level. With this question in mind, we revisit the sliding window algorithm in Chapter 3 (we show how X.25 networks use it to implement hop-by-hop flow control) and in Chapter 5 (we describe how TCP uses it to implement a reliable byte-stream channel).

### 2.5.3 Concurrent Logical Channels

The data link protocol used in the ARPANET provides an interesting alternative to the sliding window protocol, in that it is able to keep the pipe full while still using the simple stop-and-wait algorithm. One important consequence of this approach is that the frames sent over a given link are not kept in any particular order. The protocol also implies nothing about flow control.

The idea underlying the ARPANET protocol, which we refer to as *concurrent logical channels*, is to multiplex several logical channels onto a single point-to-point link and to run the stop-and-wait algorithm on each of these logical channels. There is no relationship maintained among the frames sent on any of the logical channels, yet because a different frame can be outstanding on each of the several logical channels the sender can keep the link full.

More precisely, the sender keeps 3 bits of state for each channel: a boolean, saying whether the channel is currently busy; the 1-bit sequence number to use the next time a frame is sent on this logical channel; and the next sequence number to expect on a frame that arrives on this channel. When the node has a frame to send, it uses the lowest idle channel, and otherwise it behaves just like stop-and-wait.

In practice, the ARPANET supported 8 logical channels over each ground link and 16 over each satellite link. In the ground-link case, the

header for each frame included a 3-bit channel number and a 1-bit sequence number, for a total of 4 bits. This is exactly the number of bits the sliding window protocol requires to support up to 8 outstanding frames on the link when RWS = SWS.

## 2.6 ETHERNET AND MULTIPLE ACCESS NETWORKS (802.3)



Developed in the mid-1970s by researchers at the Xerox Palo Alto Research Center (PARC), the Ethernet eventually became the dominant local area networking technology, emerging from a pack of competing technologies. Today, it competes mainly with 802.11 wireless networks but remains extremely popular in campus networks and data centers. The more general name for the technology behind the Ethernet is Carrier Sense, Multiple Access with Collision Detect (CSMA/CD).

As indicated by the CSMA name, the Ethernet is a multiple-access network, meaning that a set of nodes sends and receives frames over a shared link. You can, therefore, think of an Ethernet as being like a bus that has multiple stations plugged into it. The “carrier sense” in CSMA/CD means that all the nodes can distinguish between an idle and a busy link, and “collision detect” means that a node listens as it transmits and can therefore detect when a frame it is transmitting has interfered (collided) with a frame transmitted by another node.

The Ethernet has its roots in an early packet radio network, called Aloha, developed at the University of Hawaii to support computer communication across the Hawaiian Islands. Like the Aloha network, the fundamental problem faced by the Ethernet is how to mediate access to a shared medium fairly and efficiently (in Aloha, the medium was the atmosphere, while in the Ethernet the medium is a coax cable). The core idea in both Aloha and the Ethernet is an algorithm that controls when each node can transmit.

Interestingly, modern Ethernet links are now largely point to point; that is, they connect one host to an Ethernet *switch*, or they interconnect switches. Hence, “multiple access” techniques are not used much in today’s Ethernets. At the same time, wireless networks have become enormously popular, so the multiple access technologies that started in Aloha are today again mostly used in wireless networks such as 802.11 (Wi-Fi) networks. These networks will be discussed in Section 2.7.

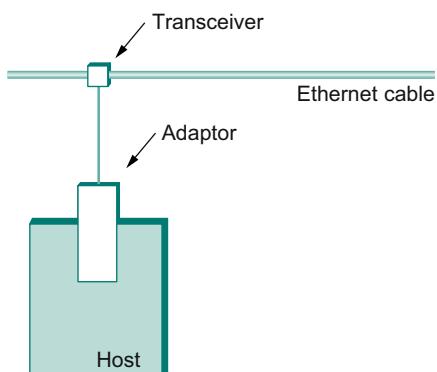
We will discuss Ethernet switches in the next chapter. For now, we'll focus on how a single Ethernet link works. And even though multi-access Ethernet is becoming a bit of a historical curiosity, the principles of multi-access networks continue to be important enough to warrant some further discussion, which we provide below.

Digital Equipment Corporation and Intel Corporation joined Xerox to define a 10-Mbps Ethernet standard in 1978. This standard then formed the basis for IEEE standard 802.3, which additionally defines a much wider collection of physical media over which an Ethernet can operate, including 100-Mbps, 1-Gbps, and 10-Gbps versions.

### 2.6.1 Physical Properties

Ethernet segments were originally implemented using coaxial cable of length up to 500 m. (Modern Ethernets use twisted copper pairs, usually a particular type known as "Category 5," or optical fibers, and in some cases can be quite a lot longer than 500 m.) This cable was similar to the type used for cable TV. Hosts connected to an Ethernet segment by tapping into it. A *transceiver*, a small device directly attached to the tap, detected when the line was idle and drove the signal when the host was transmitting. It also received incoming signals. The transceiver, in turn, connected to an Ethernet adaptor, which was plugged into the host. This configuration is shown in Figure 2.22.

Multiple Ethernet segments can be joined together by *repeaters*. A repeater is a device that forwards digital signals, much like an amplifier

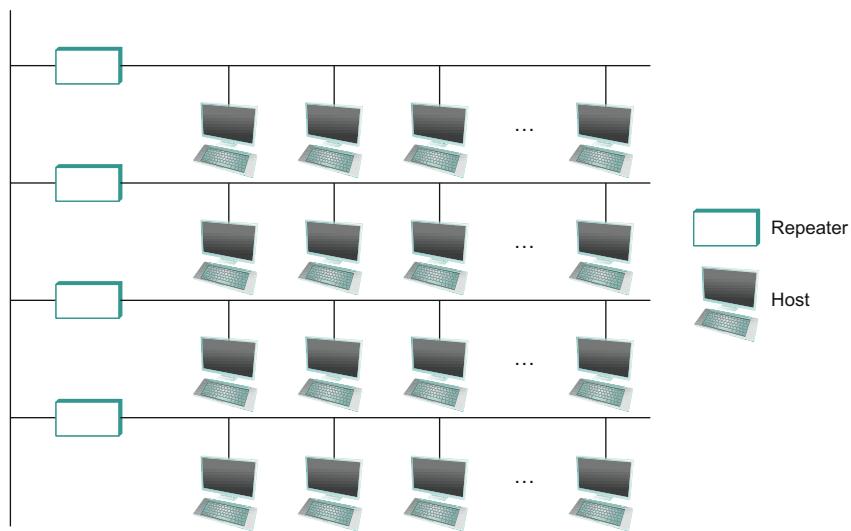


■ FIGURE 2.22 Ethernet transceiver and adaptor.

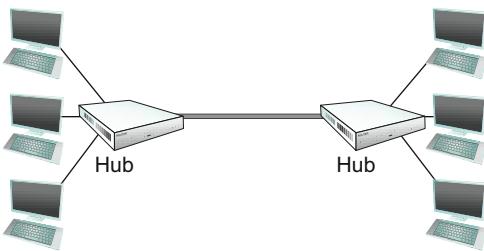
forwards analog signals. Repeaters understand only bits, not frames; however, no more than four repeaters could be positioned between any pair of hosts, meaning that a classical Ethernet had a total reach of only 2500 m. For example, using just two repeaters between any pair of hosts supports a configuration similar to the one illustrated in Figure 2.23—that is, a segment running down the spine of a building with a segment on each floor.

It's also possible to create a multiway repeater, sometimes called a *hub*, as illustrated in Figure 2.24. A hub just repeats whatever it hears on one port out all its other ports.

Any signal placed on the Ethernet by a host is broadcast over the entire network; that is, the signal is propagated in both directions, and repeaters and hubs forward the signal on all outgoing segments. Terminators attached to the end of each segment absorb the signal and keep it from bouncing back and interfering with trailing signals. The original Ethernet specifications used the Manchester encoding scheme described in Section 2.2, while 4B/5B encoding or the similar 8B/10B scheme is used today on higher speed Ethernets.



■ FIGURE 2.23 Ethernet repeater.



■ FIGURE 2.24 Ethernet hub.

It is important to understand that whether a given Ethernet spans a single segment, a linear sequence of segments connected by repeaters, or multiple segments connected in a star configuration by a hub, data transmitted by any one host on that Ethernet reaches all the other hosts. This is the good news. The bad news is that all these hosts are competing for access to the same link, and, as a consequence, they are said to be in the same *collision domain*. The multi-access part of the Ethernet is all about dealing with the competition for the link that arises in a collision domain.

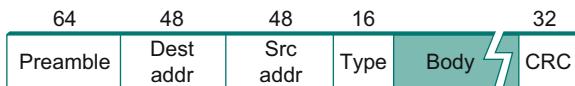
### 2.6.2 Access Protocol

We now turn our attention to the algorithm that controls access to a shared Ethernet link. This algorithm is commonly called the Ethernet's *media access control* (MAC). It is typically implemented in hardware on the network adaptor. We will not describe the hardware *per se*, but instead focus on the algorithm it implements. First, however, we describe the Ethernet's frame format and addresses.

#### Frame Format

Each Ethernet frame is defined by the format given in Figure 2.25.<sup>6</sup> The 64-bit preamble allows the receiver to synchronize with the signal; it is a sequence of alternating 0s and 1s. Both the source and destination hosts are identified with a 48-bit address. The packet type field serves as the demultiplexing key; it identifies to which of possibly many higher-level protocols this frame should be delivered. Each frame contains up to 1500 bytes of data. Minimally, a frame must contain at least 46 bytes of data, even if this means the host has to pad the frame before transmitting it. The reason for this minimum frame size is that the frame must

<sup>6</sup>This frame format is from the Digital–Intel–Xerox standard; the 802.3 version differs slightly.



■ FIGURE 2.25 Ethernet frame format.

be long enough to detect a collision; we discuss this more below. Finally, each frame includes a 32-bit CRC. Like the HDLC protocol described in Section 2.3.2, the Ethernet is a bit-oriented framing protocol. Note that from the host’s perspective, an Ethernet frame has a 14-byte header: two 6-byte addresses and a 2-byte type field. The sending adaptor attaches the preamble and CRC before transmitting, and the receiving adaptor removes them.

#### Addresses

Each host on an Ethernet—in fact, every Ethernet host in the world—has a unique Ethernet address. Technically, the address belongs to the adaptor, not the host; it is usually burned into ROM. Ethernet addresses are typically printed in a form humans can read as a sequence of six numbers separated by colons. Each number corresponds to 1 byte of the 6-byte address and is given by a pair of hexadecimal digits, one for each of the 4-bit nibbles in the byte; leading 0s are dropped. For example, 8:0:2b:e4:b1:2 is the human-readable representation of Ethernet address

00001000 00000000 00101011 11100100 10110001 00000010

To ensure that every adaptor gets a unique address, each manufacturer of Ethernet devices is allocated a different prefix that must be prepended to the address on every adaptor they build. For example, Advanced Micro Devices has been assigned the 24-bit prefix x080020 (or 8:0:20). A given manufacturer then makes sure the address suffixes it produces are unique.

Each frame transmitted on an Ethernet is received by every adaptor connected to that Ethernet. Each adaptor recognizes those frames addressed to its address and passes only those frames on to the host. (An adaptor can also be programmed to run in *promiscuous* mode, in which case it delivers all received frames to the host, but this is not the normal mode.) In addition to these *unicast* addresses, an Ethernet address consisting of all 1s is treated as a *broadcast* address; all adaptors pass frames addressed to the broadcast address up to the host. Similarly, an address

that has the first bit set to 1 but is not the broadcast address is called a *multicast* address. A given host can program its adaptor to accept some set of multicast addresses. Multicast addresses are used to send messages to some subset of the hosts on an Ethernet (e.g., all file servers). To summarize, an Ethernet adaptor receives all frames and accepts

- Frames addressed to its own address
- Frames addressed to the broadcast address
- Frames addressed to a multicast address, if it has been instructed to listen to that address
- All frames, if it has been placed in promiscuous mode

It passes to the host only the frames that it accepts.

#### *Transmitter Algorithm*

As we have just seen, the receiver side of the Ethernet protocol is simple; the real smarts are implemented at the sender's side. The transmitter algorithm is defined as follows.

When the adaptor has a frame to send and the line is idle, it transmits the frame immediately; there is no negotiation with the other adaptors. The upper bound of 1500 bytes in the message means that the adaptor can occupy the line for only a fixed length of time.

When an adaptor has a frame to send and the line is busy, it waits for the line to go idle and then transmits immediately.<sup>7</sup> The Ethernet is said to be a *1-persistent* protocol because an adaptor with a frame to send transmits with probability 1 whenever a busy line goes idle. In general, a *p-persistent* algorithm transmits with probability  $0 \leq p \leq 1$  after a line becomes idle and defers with probability  $q = 1 - p$ . The reasoning behind choosing a  $p < 1$  is that there might be multiple adaptors waiting for the busy line to become idle, and we don't want all of them to begin transmitting at the same time. If each adaptor transmits immediately with a probability of, say, 33%, then up to three adaptors can be waiting to transmit and the odds are that only one will begin transmitting when the line becomes idle. Despite this reasoning, an Ethernet adaptor always transmits immediately after noticing that the network has become idle and has been very effective in doing so.

---

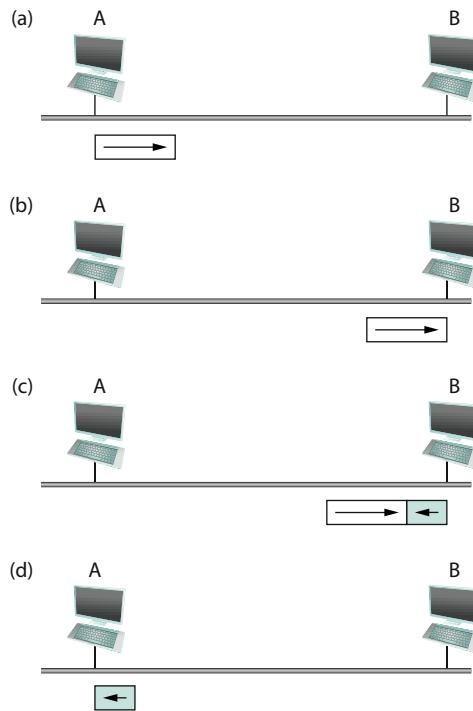
<sup>7</sup>To be more precise, all adaptors wait 9.6  $\mu$ s after the end of one frame before beginning to transmit the next frame. This is true for both the sender of the first frame as well as those nodes listening for the line to become idle.

To complete the story about  $p$ -persistent protocols for the case when  $p < 1$ , you might wonder how long a sender that loses the coin flip (i.e., decides to defer) has to wait before it can transmit. The answer for the Aloha network, which originally developed this style of protocol, was to divide time into discrete slots, with each slot corresponding to the length of time it takes to transmit a full frame. Whenever a node has a frame to send and it senses an empty (idle) slot, it transmits with probability  $p$  and defers until the next slot with probability  $q = 1 - p$ . If that next slot is also empty, the node again decides to transmit or defer, with probabilities  $p$  and  $q$ , respectively. If that next slot is not empty—that is, some other station has decided to transmit—then the node simply waits for the next idle slot and the algorithm repeats.

Returning to our discussion of the Ethernet, because there is no centralized control it is possible for two (or more) adaptors to begin transmitting at the same time, either because both found the line to be idle or because both had been waiting for a busy line to become idle. When this happens, the two (or more) frames are said to *collide* on the network. Each sender, because the Ethernet supports collision detection, is able to determine that a collision is in progress. At the moment an adaptor detects that its frame is colliding with another, it first makes sure to transmit a 32-bit jamming sequence and then stops the transmission. Thus, a transmitter will minimally send 96 bits in the case of a collision: 64-bit preamble plus 32-bit jamming sequence.

One way that an adaptor will send only 96 bits—which is sometimes called a *runt frame*—is if the two hosts are close to each other. Had the two hosts been farther apart, they would have had to transmit longer, and thus send more bits, before detecting the collision. In fact, the worst-case scenario happens when the two hosts are at opposite ends of the Ethernet. To know for sure that the frame it just sent did not collide with another frame, the transmitter may need to send as many as 512 bits. Not coincidentally, every Ethernet frame must be at least 512 bits (64 bytes) long: 14 bytes of header plus 46 bytes of data plus 4 bytes of CRC.

Why 512 bits? The answer is related to another question you might ask about an Ethernet: Why is its length limited to only 2500 m? Why not 10 or 1000 km? The answer to both questions has to do with the fact that the farther apart two nodes are, the longer it takes for a frame sent by one to reach the other, and the network is vulnerable to a collision during this time.



■ **FIGURE 2.26** Worst-case scenario: (a) A sends a frame at time  $t$ ; (b) A's frame arrives at B at time  $t + d$ ; (c) B begins transmitting at time  $t + d$  and collides with A's frame; (d) B's runt (32-bit) frame arrives at A at time  $t + 2d$ .

Figure 2.26 illustrates the worst-case scenario, where hosts A and B are at opposite ends of the network. Suppose host A begins transmitting a frame at time  $t$ , as shown in (a). It takes it one link latency (let's denote the latency as  $d$ ) for the frame to reach host B. Thus, the first bit of A's frame arrives at B at time  $t + d$ , as shown in (b). Suppose an instant before host A's frame arrives (i.e., B still sees an idle line), host B begins to transmit its own frame. B's frame will immediately collide with A's frame, and this collision will be detected by host B (c). Host B will send the 32-bit jamming sequence, as described above. (B's frame will be a runt.) Unfortunately, host A will not know that the collision occurred until B's frame reaches it, which will happen one link latency later, at time  $t + 2 \times d$ , as shown in (d). Host A must continue to transmit until this time in order to detect the collision. In other words, host A must transmit for  $2 \times d$

to be sure that it detects all possible collisions. Considering that a maximally configured Ethernet is 2500 m long, and that there may be up to four repeaters between any two hosts, the round-trip delay has been determined to be 51.2  $\mu$ s, which on a 10-Mbps Ethernet corresponds to 512 bits. The other way to look at this situation is that we need to limit the Ethernet's maximum latency to a fairly small value (e.g., 51.2  $\mu$ s) for the access algorithm to work; hence, an Ethernet's maximum length must be something on the order of 2500 m.

Once an adaptor has detected a collision and stopped its transmission, it waits a certain amount of time and tries again. Each time it tries to transmit but fails, the adaptor doubles the amount of time it waits before trying again. This strategy of doubling the delay interval between each retransmission attempt is a general technique known as *exponential backoff*. More precisely, the adaptor first delays either 0 or 51.2  $\mu$ s, selected at random. If this effort fails, it then waits 0, 51.2, 102.4, or 153.6  $\mu$ s (selected randomly) before trying again; this is  $k \times 51.2$  for  $k = 0 \dots 3$ . After the third collision, it waits  $k \times 51.2$  for  $k = 0 \dots 2^3 - 1$ , again selected at random. In general, the algorithm randomly selects a  $k$  between 0 and  $2^n - 1$  and waits  $k \times 51.2$   $\mu$ s, where  $n$  is the number of collisions experienced so far. The adaptor gives up after a given number of tries and reports a transmit error to the host. Adaptors typically retry up to 16 times, although the backoff algorithm caps  $n$  in the above formula at 10.

### 2.6.3 Experience with Ethernet

Because Ethernets have been around for so many years and are so popular, we have a great deal of experience in using them. One of the most important observations people have made about multi-access Ethernets is that they work best under lightly loaded conditions. This is because under heavy loads (typically, a utilization of over 30% is considered heavy on an Ethernet) too much of the network's capacity is wasted by collisions.

To mitigate these concerns, multi-access Ethernets were typically used in a far more conservative way than the standard allows. For example, most Ethernets had fewer than 200 hosts connected to them, which is far fewer than the maximum of 1024. Similarly, most Ethernets were far shorter than 2500 m, with a round-trip delay of closer to 5  $\mu$ s than 51.2  $\mu$ s. Another factor that made Ethernets practical is that, even though Ethernet adaptors do not implement link-level flow control, the hosts typically

provide an end-to-end flow-control mechanism, as we will see later. As a result, it is rare to find situations in which any one host is continuously pumping frames onto the network.

Finally, it is worth saying a few words about why Ethernets have been so successful, so that we can understand the properties we should emulate with any LAN technology that tries to replace it. First, an Ethernet is extremely easy to administer and maintain: There were no switches in the original Ethernets, no routing or configuration tables to be kept up-to-date, and it is easy to add a new host to the network. It is hard to imagine a simpler network to administer. Second, it is inexpensive: Cable is cheap, and the only other cost is the network adaptor on each host. Ethernet became deeply entrenched for these reasons, and any switch-based approach that aspired to displace it required additional investment in infrastructure (the switches), on top of the cost of each adaptor. As we will see in the next chapter, a switch-based technology did eventually succeed in replacing multi-access Ethernet: switched Ethernet. Retaining the simplicity of administration (and familiarity) was a key reason for this success.



## 2.7 WIRELESS

Wireless technologies differ from wired links in some important ways, while at the same time sharing many common properties. Like wired links, issues of bit errors are of great concern—typically even more so due to the unpredictable noise environment of most wireless links. Framing and reliability also have to be addressed. Unlike wired links, power is a big issue for wireless, especially because wireless links are often used by small mobile devices (like phones and sensors) that have limited access to power (e.g., a small battery). Furthermore, you can't go blasting away at arbitrarily high power with a radio transmitter—there are concerns about interference with other devices and usually regulations about how much power a device may emit at any given frequency.

Wireless media are also inherently multi-access; it's difficult to direct your radio transmission to just a single receiver or to avoid receiving radio signals from any transmitter with enough power in your neighborhood. Hence, media access control is a central issue for wireless links. And, because it's hard to control who receives your signal when you transmit over the air, issues of eavesdropping may also have to be addressed.

## Where Are They Now?

### TOKEN RINGS

For many years, there were two main ways to build a LAN: Ethernet or token ring. The most prevalent form of token ring was invented by IBM, and standardized as IEEE 802.5. Token rings have a number of things in common with Ethernet: The ring behaves like a single shared medium and employs a distributed algorithm to decide which station can transmit onto that medium at any given time, and every node on a given ring can see all the packets transmitted by other nodes.

The most obvious difference between token ring and Ethernet is the topology; whereas an Ethernet is a bus, the nodes in a token ring form a loop. That is, each node is connected to a pair of neighbors, one upstream and one downstream. The “token” is just a special sequence of bits that circulates around the ring; each node receives and then forwards the token. When a node that has a frame to transmit sees the token, it takes the token off the ring (i.e., it does not forward the special bit pattern) and instead inserts its frame into the ring. Each node along the way simply forwards the frame, with the destination node saving a copy and forwarding the message onto the next node on the ring. When the frame makes its way back around to the sender, this node strips its frame off the ring (rather than continuing to forward it) and reinserts the token. In this way, some node downstream will have the opportunity to transmit a frame. The media access algorithm is fair in the sense that as the token circulates around the ring, each node gets a chance to transmit. Nodes are serviced in a round-robin fashion.

Many different variants of token rings appeared over the decades, with the Fiber Distributed Data Interface (FDDI) being one of the last to see significant deployment. In the end, token rings lost out to the Ethernet, especially with the advent of Ethernet switching and high-speed Ethernet variants (100-Mbit and gigabit Ethernet).



### LAB APPENDIX A: Token Ring

There is a baffling assortment of different wireless technologies, each of which makes different tradeoffs in various dimensions. One simple way to categorize the different technologies is by the data rates they provide and how far apart communicating nodes can be. Other important differences include which part of the electromagnetic spectrum they use (including whether it requires a license) and how much power they consume. In this section, we discuss three prominent wireless technologies:

**Table 2.4 Overview of Leading Wireless Technologies**

	<b>Bluetooth (802.15.1)</b>	<b>Wi-Fi (802.11)</b>	<b>3G Cellular</b>
Typical link length	10 m	100 m	Tens of kilometers
Typical data rate	2 Mbps (shared)	54 Mbps (shared)	Hundreds of kbps (per connection)
Typical use	Link a peripheral to a computer	Link a computer to a wired base	Link a mobile phone to a wired tower
Wired technology analogy	USB	Ethernet	DSL

Wi-Fi (more formally known as 802.11), Bluetooth®, and the third-generation or “3G” family of cellular wireless standards. Table 2.4 gives an overview of these technologies and how they compare to each other.

You may recall from Section 1.5 that bandwidth sometimes means the width of a frequency band in hertz and sometimes the data rate of a link. Because both these concepts come up in discussions of wireless networks, we’re going to use *bandwidth* here in its stricter sense—width of a frequency band—and use the term *data rate* to describe the number of bits per second that can be sent over the link, as in Table 2.4.

Because wireless links all share the same medium, the challenge is to share that medium efficiently, without unduly interfering with each other. Most of this sharing is accomplished by dividing it up along the dimensions of frequency and space. Exclusive use of a particular frequency in a particular geographic area may be allocated to an individual entity such as a corporation. It is feasible to limit the area covered by an electromagnetic signal because such signals weaken, or *attenuate*, with the distance from their origin. To reduce the area covered by your signal, reduce the power of your transmitter.

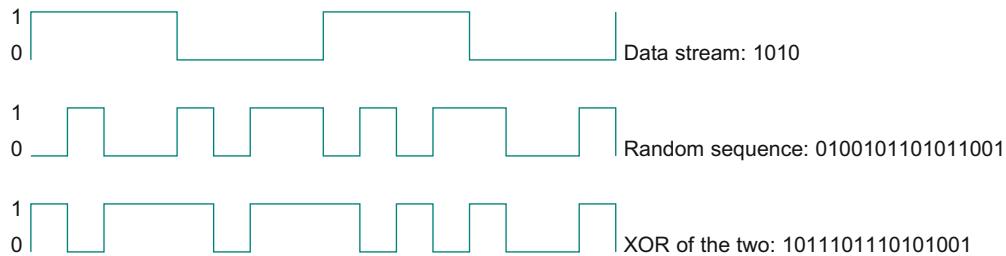
These allocations are typically determined by government agencies, such as the Federal Communications Commission (FCC) in the United States. Specific bands (frequency ranges) are allocated to certain uses. Some bands are reserved for government use. Other bands are reserved for uses such as AM radio, FM radio, television, satellite communication, and cellular phones. Specific frequencies within these bands are then

licensed to individual organizations for use within certain geographical areas. Finally, several frequency bands are set aside for license-exempt usage—bands in which a license is not needed.

Devices that use license-exempt frequencies are still subject to certain restrictions to make that otherwise unconstrained sharing work. Most important of these is a limit on transmission power. This limits the range of a signal, making it less likely to interfere with another signal. For example, a cordless phone (a common unlicensed device) might have a range of about 100 feet.

One idea that shows up a lot when spectrum is shared among many devices and applications is *spread spectrum*. The idea behind spread spectrum is to spread the signal over a wider frequency band, so as to minimize the impact of interference from other devices. (Spread spectrum was originally designed for military use, so these “other devices” were often attempting to jam the signal.) For example, *frequency hopping* is a spread spectrum technique that involves transmitting the signal over a random sequence of frequencies; that is, first transmitting at one frequency, then a second, then a third, and so on. The sequence of frequencies is not truly random but is instead computed algorithmically by a pseudorandom number generator. The receiver uses the same algorithm as the sender and initializes it with the same seed; hence, it is able to hop frequencies in sync with the transmitter to correctly receive the frame. This scheme reduces interference by making it unlikely that two signals would be using the same frequency for more than the infrequent isolated bit.

A second spread spectrum technique, called *direct sequence*, adds redundancy for greater tolerance of interference. Each bit of data is represented by multiple bits in the transmitted signal so that, if some of the transmitted bits are damaged by interference, there is usually enough redundancy to recover the original bit. For each bit the sender wants to transmit, it actually sends the exclusive-OR of that bit and  $n$  random bits. As with frequency hopping, the sequence of random bits is generated by a pseudorandom number generator known to both the sender and the receiver. The transmitted values, known as an  $n$ -bit *chipping code*, spread the signal across a frequency band that is  $n$  times wider than the frame would have otherwise required. Figure 2.27 gives an example of a 4-bit chipping sequence.



■ FIGURE 2.27 Example 4-bit chipping sequence.

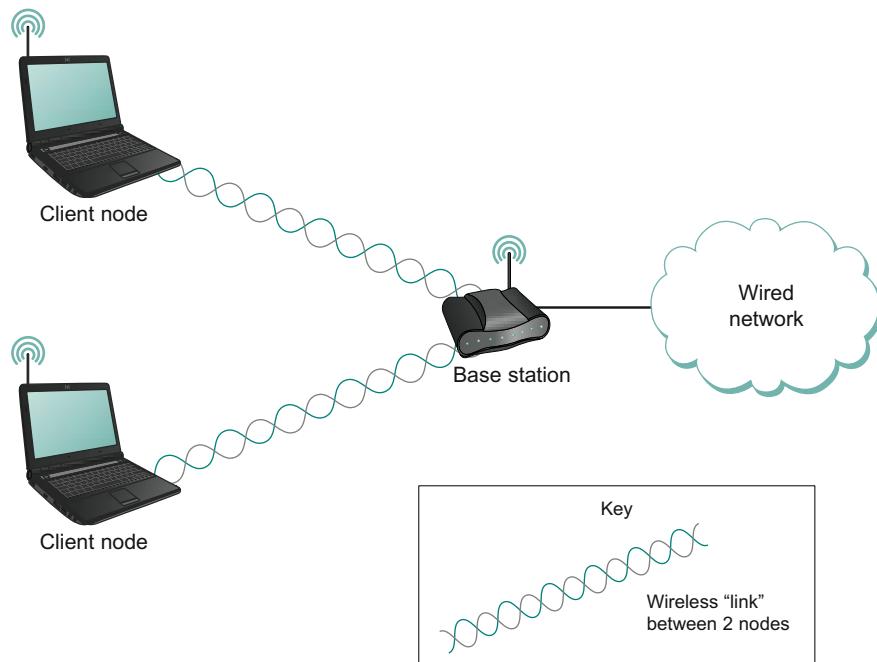
Different parts of the electromagnetic spectrum have different properties, making some better suited to communication, and some less so. For example, some can penetrate buildings and some cannot. Governments regulate only the prime communication portion: the radio and microwave ranges. As demand for prime spectrum increases, there is great interest in the spectrum that is becoming available as analog television is phased out in favor of digital.<sup>8</sup>

In many wireless networks today we observe that there are two different classes of endpoints. One endpoint, sometimes described as the *base station*, usually has no mobility but has a wired (or at least high-bandwidth) connection to the Internet or other networks, as shown in Figure 2.28. The node at the other end of the link—shown here as a client node—is often mobile and relies on its link to the base station for all of its communication with other nodes.

Observe that in Figure 2.28 we have used a wavy pair of lines to represent the wireless “link” abstraction provided between two devices (e.g., between a base station and one of its client nodes). One of the interesting aspects of wireless communication is that it naturally supports point-to-multipoint communication, because radio waves sent by one device can be simultaneously received by many devices. However, it is often useful to create a point-to-point link abstraction for higher layer protocols, and we will see examples of how this works later in this section.

Note that in Figure 2.28, communication between non-base (client) nodes is routed via the base station. This is in spite of the fact that radio waves emitted by one client node may well be received by other

<sup>8</sup>Thanks to advances in video coding and modulation, digital video broadcasts require less spectrum to be allocated for each TV channel.

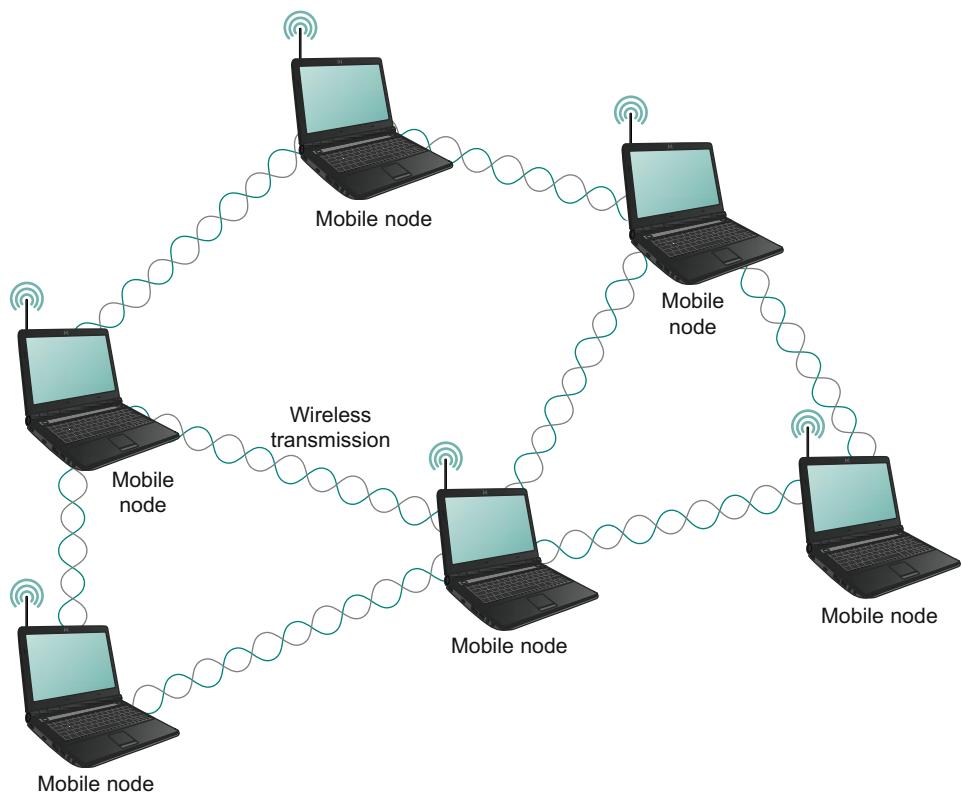


■ FIGURE 2.28 A wireless network using a base station.

client nodes—the common base station model does not permit direct communication between the client nodes.

This topology implies three qualitatively different levels of mobility. The first level is no mobility, such as when a receiver must be in a fixed location to receive a directional transmission from the base station. The second level is mobility within the range of a base, as is the case with Bluetooth. The third level is mobility between bases, as is the case with cell phones and Wi-Fi.

An alternative topology that is seeing increasing interest is the *mesh* or *ad hoc* network. In a wireless mesh, nodes are peers; that is, there is no special base station node. Messages may be forwarded via a chain of peer nodes as long as each node is within range of the preceding node. This is illustrated in Figure 2.29. This allows the wireless portion of a network to extend beyond the limited range of a single radio. From the point of view of competition between technologies, this allows a shorter-range technology to extend its range and potentially compete with a longer-range technology. Meshes also offer fault tolerance by providing



■ FIGURE 2.29 A wireless *ad hoc* or mesh network.

multiple routes for a message to get from point A to point B. A mesh network can be extended incrementally, with incremental costs. On the other hand, a mesh network requires non-base nodes to have a certain level of sophistication in their hardware and software, potentially increasing per-unit costs and power consumption, a critical consideration for battery-powered devices. Wireless mesh networks are of considerable research interest (see the further reading section for some references), but they are still in their relative infancy compared to networks with base stations. Wireless sensor networks, another hot emerging technology, often form wireless meshes.

Now that we have covered some of the common wireless issues, let's take a look at the details of a few common wireless technologies.

### 2.7.1 802.11/Wi-Fi

Most readers will have used a wireless network based on the IEEE 802.11 standards, often referred to as *Wi-Fi*.<sup>9</sup> Wi-Fi is technically a trademark, owned by a trade group called the Wi-Fi Alliance, which certifies product compliance with 802.11. Like Ethernet, 802.11 is designed for use in a limited geographical area (homes, office buildings, campuses), and its primary challenge is to mediate access to a shared communication medium—in this case, signals propagating through space.

#### *Physical Properties*

802.11 defines a number of different physical layers that operate in various frequency bands and provide a range of different data rates. At the time of writing, 802.11n provides the highest maximum data rate, topping out at 600 Mbps.

The original 802.11 standard defined two radio-based physical layers standards, one using frequency hopping (over 79 1-MHz-wide frequency bandwidths) and the other using direct sequence spread spectrum (with an 11-bit chipping sequence). Both provided data rates in the 2 Mbps range. The physical layer standard 802.11b was added subsequently. Using a variant of direct sequence, 802.11b provides up to 11 Mbps. These three standards all operated in the license-exempt 2.4-GHz frequency band of the electromagnetic spectrum. Then came 802.11a, which delivers up to 54 Mbps using a variant of FDM called *orthogonal frequency division multiplexing (OFDM)*; 802.11a runs in the license-exempt 5-GHz band. On one hand, this band is less used, so there is less interference. On the other hand, there is more absorption of the signal and it is limited to almost line of sight. 802.11g followed; 802.11g also uses OFDM, delivers up to 54 Mbps, and is backward compatible with 802.11b (and returns to the 2.4-GHz band).

Most recently 802.11n has appeared on the scene, with a standard that was approved in 2009 (although pre-standard products also existed). 802.11n achieves considerable advances in maximum possible data rate using multiple antennas and allowing greater wireless channel bandwidths. The use of multiple antennas is often called *MIMO* for multiple-input, multiple-output.

---

<sup>9</sup>There is some debate over whether Wi-Fi stands for “wireless fidelity,” by analogy to Hi-Fi, or whether it is just a catchy name that doesn’t stand for anything other than 802.11.

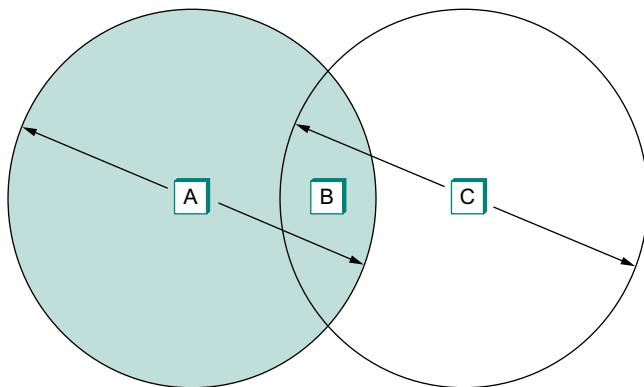
It is common for commercial products to support more than one flavor of 802.11; some base stations support all four variants (a,b, g, and n). This not only ensures compatibility with any device that supports any one of the standards but also makes it possible for two such products to choose the highest bandwidth option for a particular environment.

It is worth noting that while all the 802.11 standards define a *maximum* bit rate that can be supported, they mostly support lower bit rates as well; for example, 802.11a allows for bit rates of 6, 9, 12, 18, 24, 36, 48, and 54 Mbps. At lower bit rates, it is easier to decode transmitted signals in the presence of noise. Different modulation schemes are used to achieve the various bit rates; in addition, the amount of redundant information in the form of error-correcting codes is varied. (See [Section 2.4](#) for an introduction to error-detecting codes.) More redundant information means higher resilience to bit errors at the cost of lowering the effective data rate (since more of the transmitted bits are redundant).

The systems try to pick an optimal bit rate based on the noise environment in which they find themselves; the algorithms for bit rate selection can be quite complex (see the Further Reading section for an example). Interestingly, the 802.11 standards do not specify a particular approach but leave the algorithms to the various vendors. The basic approach to picking a bit rate is to estimate the bit error rate either by directly measuring the signal-to-noise ratio (SNR) at the physical layer or by estimating the SNR by measuring how often packets are successfully transmitted and acknowledged. In some approaches, a sender will occasionally probe a higher bit rate by sending one or more packets at that rate to see if it succeeds.

#### *Collision Avoidance*

At first glance, it might seem that a wireless protocol would follow the same algorithm as the Ethernet—wait until the link becomes idle before transmitting and back off should a collision occur—and, to a first approximation, this is what 802.11 does. The additional complication for wireless is that, while a node on an Ethernet receives every other node's transmissions and can transmit and receive at the same time, neither of these conditions holds for wireless nodes. This makes detection of collisions rather more complex. The reason why wireless nodes cannot usually transmit and receive at the same time (on the same frequency) is that the power generated by the transmitter is much higher than any received

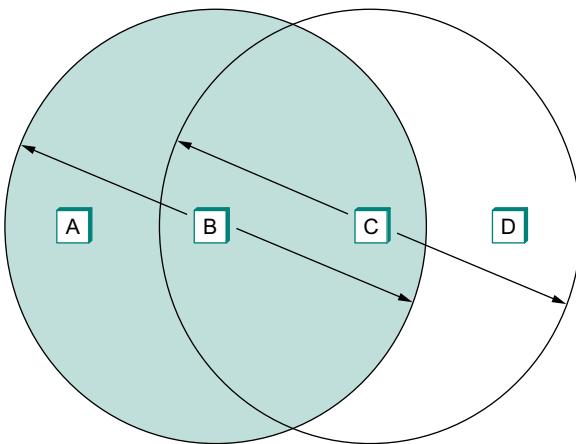


■ **FIGURE 2.30** The hidden node problem. Although A and C are hidden from each other, their signals can collide at B. (B's reach is not shown.)

signal is likely to be and so swamps the receiving circuitry. The reason why a node may not receive transmissions from another node is because that node may be too far away or blocked by an obstacle. This situation is a bit more complex than it first appears, as the following discussion will illustrate.

Consider the situation depicted in Figure 2.30, where A and C are both within range of B but not each other. Suppose both A and C want to communicate with B and so they each send it a frame. A and C are unaware of each other since their signals do not carry that far. These two frames collide with each other at B, but unlike an Ethernet, neither A nor C is aware of this collision. A and C are said to be *hidden nodes* with respect to each other.

A related problem, called the *exposed node problem*, occurs under the circumstances illustrated in Figure 2.31, where each of the four nodes is able to send and receive signals that reach just the nodes to its immediate left and right. For example, B can exchange frames with A and C but it cannot reach D, while C can reach B and D but not A. Suppose B is sending to A. Node C is aware of this communication because it hears B's transmission. It would be a mistake, however, for C to conclude that it cannot transmit to anyone just because it can hear B's transmission. For example, suppose C wants to transmit to node D. This is not a problem since C's transmission to D will not interfere with A's ability to receive from B. (It would interfere with A sending to B, but B is transmitting in our example.)



■ **FIGURE 2.31** The exposed node problem. Although B and C are exposed to each other’s signals, there is no interference if B transmits to A while C transmits to D. (A and D’s reaches are not shown.)

802.11 addresses these problems by using CSMA/CA, where the CA stands for collision *avoidance*, in contrast to the collision *detection* of CSMA/CD used on Ethernets. There are a few pieces to make this work.

The Carrier Sense part seems simple enough: Before sending a packet, the transmitter checks if it can hear any other transmissions; if not, it sends. However, because of the hidden terminal problem, just waiting for the absence of signals from other transmitters does not guarantee that a collision will not occur from the perspective of the receiver. For this reason, one part of CSMA/CA is an explicit ACK from the receiver to the sender. If the packet was successfully decoded and passed its CRC at the receiver, the receiver sends an ACK back to the sender.

Note that if a collision does occur, it will render the entire packet useless.<sup>10</sup> For this reason, 802.11 adds an optional mechanism called RTS-CTS (Ready to Send-Clear to Send). This goes some way toward addressing the hidden terminal problem. The sender sends an RTS—a short packet—to the intended receiver, and if that packet is received successfully the receiver responds with another short packet, the CTS. Even though the RTS may not have been heard by a hidden terminal, the CTS probably will be. This effectively tells the nodes within range of the receiver that they should not send anything for a while—the amount of

<sup>10</sup>Current research tries to recover partial packets, but that is not yet part of 802.11.

time of the intended transmission is included in the RTS and CTS packets. After that time plus a small interval has passed, the carrier can be assumed to be available again, and another node is free to try to send.

Of course, two nodes might detect an idle link and try to transmit an RTS frame at the same time, causing their RTS frames to collide with each other. The senders realize the collision has happened when they do not receive the CTS frame after a period of time, in which case they each wait a random amount of time before trying again. The amount of time a given node delays is defined by an exponential backoff algorithm very much like that used on the Ethernet (see [Section 2.6.2](#)).

After a successful RTS-CTS exchange, the sender sends its data packet and, if all goes well, receives an ACK for that packet. In the absence of a timely ACK, the sender will try again to request usage of the channel again, using the same process described above. By this time, of course, other nodes may again be trying to get access to the channel as well.

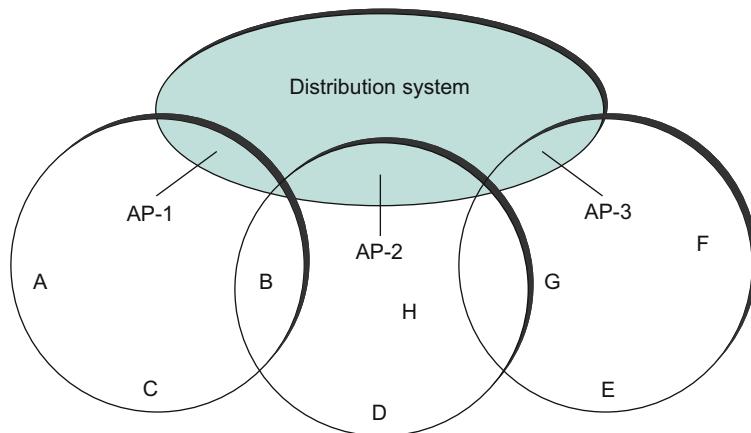
### *Distribution System*

As described so far, 802.11 would be suitable for a network with a mesh (*ad hoc*) topology, and development of an 802.11s standard for mesh networks is nearing completion. At the current time, however, nearly all 802.11 networks use a base-station-oriented topology.

Instead of all nodes being created equal, some nodes are allowed to roam (e.g., your laptop) and some are connected to a wired network infrastructure. 802.11 calls these base stations *access points* (APs), and they are connected to each other by a so-called *distribution system*. [Figure 2.32](#) illustrates a distribution system that connects three access points, each of which services the nodes in some region. Each access point operates on some channel in the appropriate frequency range, and each AP will typically be on a different channel than its neighbors.

The details of the distribution system are not important to this discussion—it could be an Ethernet, for example. The only important point is that the distribution network operates at the link layer, the same protocol layer as the wireless links. In other words, it does not depend on any higher-level protocols (such as the network layer).

Although two nodes can communicate directly with each other if they are within reach of each other, the idea behind this configuration is that each node associates itself with one access point. For node A to communicate with node E, for example, A first sends a frame to its access



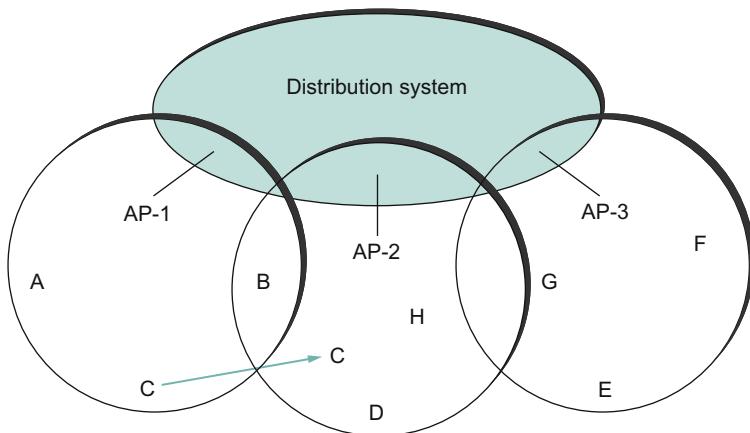
■ FIGURE 2.32 Access points connected to a distribution system.

point (AP-1), which forwards the frame across the distribution system to AP-3, which finally transmits the frame to E. How AP-1 knew to forward the message to AP-3 is beyond the scope of 802.11; it may have used the bridging protocol described in the next chapter (Section 3.1.4). What 802.11 does specify is how nodes select their access points and, more interestingly, how this algorithm works in light of nodes moving from one cell to another.

The technique for selecting an AP is called *scanning* and involves the following four steps:

1. The node sends a Probe frame.
2. All APs within reach reply with a Probe Response frame.
3. The node selects one of the access points and sends that AP an Association Request frame.
4. The AP replies with an Association Response frame.

A node engages this protocol whenever it joins the network, as well as when it becomes unhappy with its current AP. This might happen, for example, because the signal from its current AP has weakened due to the node moving away from it. Whenever a node acquires a new AP, the new AP notifies the old AP of the change (this happens in step 4) via the distribution system.



■ FIGURE 2.33 Node mobility.

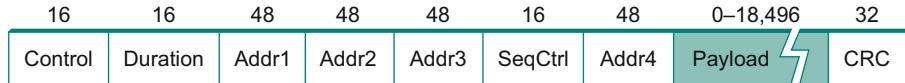
Consider the situation shown in Figure 2.33, where node C moves from the cell serviced by AP-1 to the cell serviced by AP-2. As it moves, it sends Probe frames, which eventually result in Probe Response frames from AP-2. At some point, C prefers AP-2 over AP-1, and so it associates itself with that access point.

The mechanism just described is called *active scanning* since the node is actively searching for an access point. APs also periodically send a Beacon frame that advertises the capabilities of the access point; these include the transmission rates supported by the AP. This is called *passive scanning*, and a node can change to this AP based on the Beacon frame simply by sending an Association Request frame back to the access point.

#### Frame Format

Most of the 802.11 frame format, which is depicted in Figure 2.34, is exactly what we would expect. The frame contains the source and destination node addresses, each of which is 48 bits long; up to 2312 bytes of data; and a 32-bit CRC. The Control field contains three subfields of interest (not shown): a 6-bit Type field that indicates whether the frame carries data, is an RTS or CTS frame, or is being used by the scanning algorithm, and a pair of 1-bit fields—called ToDS and FromDS—that are described below.

The peculiar thing about the 802.11 frame format is that it contains four, rather than two, addresses. How these addresses are interpreted



■ FIGURE 2.34 802.11 frame format.

depends on the settings of the ToDS and FromDS bits in the frame's Control field. This is to account for the possibility that the frame had to be forwarded across the distribution system, which would mean that the original sender is not necessarily the same as the most recent transmitting node. Similar reasoning applies to the destination address. In the simplest case, when one node is sending directly to another, both the DS bits are 0, Addr1 identifies the target node, and Addr2 identifies the source node. In the most complex case, both DS bits are set to 1, indicating that the message went from a wireless node onto the distribution system, and then from the distribution system to another wireless node. With both bits set, Addr1 identifies the ultimate destination, Addr2 identifies the immediate sender (the one that forwarded the frame from the distribution system to the ultimate destination), Addr3 identifies the intermediate destination (the one that accepted the frame from a wireless node and forwarded it across the distribution system), and Addr4 identifies the original source. In terms of the example given in Figure 2.32, Addr1 corresponds to E, Addr2 identifies AP-3, Addr3 corresponds to AP-1, and Addr4 identifies A.

## 2.7.2 Bluetooth® (802.15.1)

Bluetooth fills the niche of very short range communication between mobile phones, PDAs, notebook computers, and other personal or peripheral devices. For example, Bluetooth can be used to connect a mobile phone to a headset or a notebook computer to a keyboard. Roughly speaking, Bluetooth is a more convenient alternative to connecting two devices with a wire. In such applications, it is not necessary to provide much range or bandwidth. This means that Bluetooth radios can use quite low power transmission, since transmission power is one of the main factors affecting bandwidth and range of wireless links. This matches the target applications for Bluetooth-enabled devices—most of them are battery powered (such as the ubiquitous phone headset) and hence it is important that they not consume much power.<sup>11</sup>

<sup>11</sup>And who really wants a high-power radio transmitter in their ear?

Bluetooth operates in the license-exempt band at 2.45 GHz. Bluetooth links have typical bandwidths around 1 to 3 Mbps and a range of about 10 m. For this reason, and because the communicating devices typically belong to one individual or group, Bluetooth is sometimes categorized as a Personal Area Network (PAN).

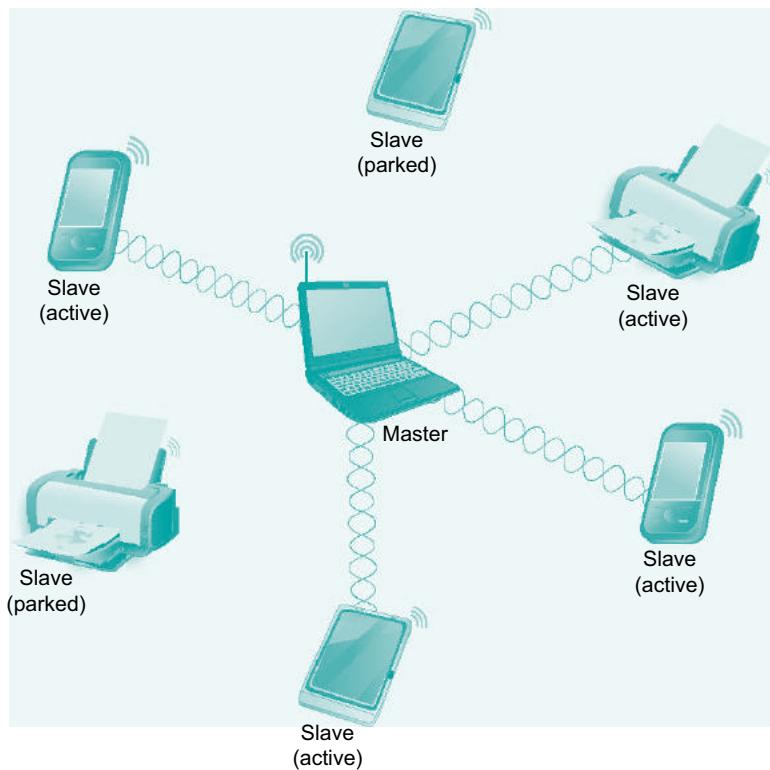
Bluetooth is specified by an industry consortium called the *Bluetooth Special Interest Group*. It specifies an entire suite of protocols, going beyond the link layer to define application protocols, which it calls *profiles*, for a range of applications. For example, there is a profile for synchronizing a PDA with a personal computer. Another profile gives a mobile computer access to a wired LAN in the manner of 802.11, although this was not Bluetooth's original goal. The IEEE 802.15.1 standard is based on Bluetooth but excludes the application protocols.

The basic Bluetooth network configuration, called a *piconet*, consists of a master device and up to seven slave devices, as shown in Figure 2.35. Any communication is between the master and a slave; the slaves do not communicate directly with each other. Because slaves have a simpler role, their Bluetooth hardware and software can be simpler and cheaper.

Since Bluetooth operates in an license-exempt band, it is required to use a spread spectrum technique (as discussed at the start of this section) to deal with possible interference in the band. It uses frequency-hopping with 79 *channels* (frequencies), using each for 625  $\mu$ s at a time. This provides a natural time slot for Bluetooth to use for synchronous time division multiplexing. A frame takes up 1, 3, or 5 consecutive time slots. Only the master can start to transmit in odd-numbered slots. A slave can start to transmit in an even-numbered slot—but only in response to a request from the master during the previous slot, thereby preventing any contention between the slave devices.

A slave device can be *parked*; that is, it is set to an inactive, low-power state. A parked device cannot communicate on the piconet; it can only be reactivated by the master. A piconet can have up to 255 parked devices in addition to its active slave devices.

In the realm of very low-power, short-range communication there are a few other technologies besides Bluetooth. One of these is ZigBee<sup>®</sup>, devised by the ZigBee alliance and standardized as IEEE 802.15.4. It is designed for situations where the bandwidth requirements are low and power consumption must be very low to give very long battery life. It is also intended to be simpler and cheaper than Bluetooth, making it



■ FIGURE 2.35 A Bluetooth piconet.

financially feasible to incorporate in cheaper devices such as *sensors*. Sensors are becoming an increasingly important class of networked device, as technology advances to the point where very cheap small devices can be deployed in large quantities to monitor things like temperature, humidity, and energy consumption in a building.

### 2.7.3 Cell Phone Technologies

While cellular telephone technology had its beginnings around voice communication, data services based on cellular standards have become increasingly popular (thanks in part to the increasing capabilities of mobile phones or *smartphones*). One drawback compared to the

technologies just described has tended to be the cost to users, due in part to cellular's use of licensed spectrum (which has historically been sold off to cellular phone operators for astronomical sums). The frequency bands that are used for cellular telephones (and now for cellular data) vary around the world. In Europe, for example, the main bands for cellular phones are at 900 MHz and 1800 MHz. In North America, 850-MHz and 1900-MHz bands are used. This global variation in spectrum usage creates problems for users who want to travel from one part of the world to another and has created a market for phones that can operate at multiple frequencies (e.g., a tri-band phone can operate at three of the four frequency bands mentioned above). That problem, however, pales in comparison to the proliferation of incompatible standards that have plagued the cellular communication business. Only recently have some signs of convergence on a small set of standards appeared. And, finally, there is the problem that, because most cellular technology was designed for voice communication, high-bandwidth data communication has been a relatively recent addition to the standards.

Like 802.11 and WiMAX, cellular technology relies on the use of base stations that are part of a wired network. The geographic area served by a base station's antenna is called a *cell*. A base station could serve a single cell or use multiple directional antennas to serve multiple cells. Cells don't have crisp boundaries, and they overlap. Where they overlap, a mobile phone could potentially communicate with multiple base stations. This is somewhat similar to the 802.11 picture shown in Figure 2.32. At any time, however, the phone is in communication with, and under the control of, just one base station. As the phone begins to leave a cell, it moves into an area of overlap with one or more other cells. The current base station senses the weakening signal from the phone and gives control of the phone to whichever base station is receiving the strongest signal from it. If the phone is involved in a call at the time, the call must be transferred to the new base station in what is called a *handoff*.

As we noted above, there is not one unique standard for cellular, but rather a collection of competing technologies that support data traffic in different ways and deliver different speeds. These technologies are loosely categorized by *generation*. The first generation (1G) was analog, and thus of limited interest from a data communications perspective. Second-generation standards moved to digital and introduced wireless

data services, while third generation (3G) allowed greater bandwidths and simultaneous voice and data transmission. Most of the widely deployed mobile phone networks today support some sort of 3G, with 4G starting to appear. Because each of the generations encompasses a family of standards and technologies, it's often a matter of some debate (and marketing interest) as to whether a particular network is 3G or some other generation.

The concept of a third generation was established before there was any implementation of 3G technologies, with the aim of shaping a single international standard that would provide much higher data bandwidth than 2G. Unfortunately, a single standard did not emerge, and this trend seems likely to continue with 4G. Interestingly, however, most of the 3G standards are based on variants of CDMA (Code Division Multiple Access).

CDMA uses a form of spread spectrum to multiplex the traffic from multiple devices into a common wireless channel. Each transmitter uses a pseudorandom chipping code at a frequency that is high relative to the data rate and sends the exclusive OR of the data with the chipping code. Each transmitter's code follows a sequence that is known to the intended receiver—for example, a base station in a cellular network assigns a unique code sequence to each mobile device with which it is currently associated. When a large number of devices broadcast their signals in the same cell and frequency band, the sum of all the transmissions looks like random noise. However, a receiver who knows the code being used by a given transmitter can extract that transmitter's data from the apparent noise.

Compared to other multiplexing techniques, CDMA has some good properties for bursty data. There is no hard limit on how many users can share a piece of spectrum—you just need to make sure they all have unique chipping codes. The bit error rate does however go up with increasing numbers of concurrent transmitters. This makes it very well suited for applications where many users exist but at any given instant many of them are not transmitting—which pretty well describes many data applications such as web surfing. And, in practical systems when it is hard to achieve very tight synchronization among all the mobile handsets, CDMA achieves better spectral efficiency (i.e., it gets closer to the theoretical limits of the Shannon–Hartley theorem) than other multiplexing schemes like TDMA.

### Cross-Layer Issues in Wireless

One interesting aspect of wireless networks that has received a great deal of attention in the research community in recent years is the way they challenge the conventions of layered protocol architectures. For example, the 802.11 standards enable you to create a link abstraction that connects one node to another in what appears to be a point-to-point manner. Having done that, any higher layer protocol can just treat the link like any other point-to-point link. But is that the right approach?

Consider, for example, three nodes A, B, and C in a row such as shown in Figure 2.30. If we want to get a packet from A to C, a conventional approach would be for A to send a packet to B and B to send the packet to C. But, in reality, the range over which a given node can send packets isn't a nice crisply defined circle as shown here, but rather it drops off slowly with increasing distance. So it may well be that A can send a packet to C with, say, 30% likelihood of success while it can send a packet to B with 80% likelihood of success. So sometimes (30% of the time) there would be no need for B to forward the packet to C, as C would already have it. Hence, it might be nice for C to tell B "Don't bother to forward that packet—I've got it already." Such an approach was actually tested on a wireless testbed called Roofnet near the Massachusetts Institute of Technology and was shown to increase throughput substantially over conventional approaches. But this approach also means that A, B, and C can no longer just act like they are connected by simple links; we have passed information that is specifically related to wireless links up to a higher layer. Some people shout "layer violation" when they see such a thing, while others (like the authors of this book) admire the ingenuity of those who improved performance by thinking beyond traditional layering.

There are countless other examples where passing some information up from the wireless link layer to higher layers can provide benefits; also, it can help to pass information up from the physical layer to the link layer. There is a fine balancing act here. Layering is a great tool—without it, networks would be impossibly difficult to reason about and to construct on a large scale. But we need to be aware that whenever we hide information—which layering does—we might lose something we really would have been better not hiding. We should think of layering (or any other form of abstraction) as a tool rather than an inviolable rule.

### Security of Wireless Links

One of the fairly obvious problems of wireless links compared to wires or fibers is that you can't be too sure where your data has gone. You can

probably figure out if it was received by the intended receiver, but there is no telling how many other receivers might have also picked up your transmission. So, if you are concerned about the privacy of your data, wireless networks present a challenge.

Even if you are not concerned about data privacy—or perhaps have taken care of it in some other way (see [Chapter 8](#) for discussion of this topic)—you may be concerned about an unauthorized user injecting data into your network. If nothing else, such a user might be able to consume resources that you would prefer to consume yourself, such as the finite bandwidth between your house and your ISP.

For these reasons, wireless networks typically come with some sort of mechanism to control access to both the link itself and the transmitted data. These mechanisms are often categorized as *wireless security*. Security is a large topic in its own right, to which we devote [Chapter 8](#), and we'll look at the details of wireless security in that context in [Section 8.4.5](#).

### Satellite Communications

One other form of wireless communication that sees application in certain scenarios is based around the use of satellites. Satellite phones (satphones) use communication satellites as base stations, communicating on frequency bands that have been reserved internationally for satellite use. Consequently, service is available even where there are no cellular base stations. Satellite phones are rarely used where cellular is available, since service is typically much more expensive. (Someone has to pay for putting the satellites into orbit.) Satphones are also larger and heavier than modern cell phones, because of the need to transmit and receive over much longer distances to reach satellites rather than cell phone towers. Satellite communication is more extensively used in television and radio broadcasting, taking advantage of the fact that the signal is broadcast, not point-to-point. High-bandwidth data communication via satellite is commercially available, but its relatively high price (for both equipment and service) limits its use to regions where no alternative is available.

## 2.8 SUMMARY

This chapter introduced the many and varied types of links that are used to connect users to existing networks and to construct large networks from scratch. While links vary enormously in their detailed

characteristics, there are many problems and techniques for solving them that are common. We looked at the five key problems that must be solved so that two or more nodes connected by some medium can exchange messages with each other.

The first problem is to encode the bits that make up a binary message into the signal at the source node and then to recover the bits from the signal at the receiving node. This is the encoding problem, and it is made challenging by the need to keep the sender's and receiver's clocks synchronized. We discussed four different encoding techniques—NRZ, NRZI, Manchester, and 4B/5B—which differ largely in how they encode clock information along with the data being transmitted. One of the key attributes of an encoding scheme is its efficiency, the ratio of signal pulses to encoded bits.

Once it is possible to transmit bits between nodes, the next step is to figure out how to package these bits into frames. This is the framing problem, and it boils down to being able to recognize the beginning and end of each frame. Again, we looked at several different techniques, including byte-oriented protocols, bit-oriented protocols, and clock-based protocols.

Assuming that each node is able to recognize the collection of bits that make up a frame, the third problem is to determine if those bits are in fact correct or if they have possibly been corrupted in transit. This is the error detection problem, and we looked at three different approaches: cyclic redundancy check, two-dimensional parity, and checksums. Of these, the CRC approach gives the strongest guarantees and is the most widely used at the link level.

Given that some frames will arrive at the destination node containing errors and thus will have to be discarded, the next problem is how to recover from such losses. The goal is to make the link appear reliable. The general approach to this problem is called *ARQ* and involves using a combination of acknowledgments and timeouts. We looked at three specific ARQ algorithms: stop-and-wait, sliding window, and concurrent channels. What makes these algorithms interesting is how effectively they use the link, with the goal being to keep the pipe full.

The final problem is not relevant to point-to-point links, but it is the central issue in multiple-access links: how to mediate access to a shared link so that all nodes eventually have a chance to transmit their data. In this case, we looked at a variety of media access protocols—Ethernet and several wireless protocols—that have been put to practical use in building

local area networks. Media access in wireless networks is made more complicated by the fact that some nodes may be hidden from each other due to range limitations of radio transmission. Most of the common wireless protocols today designate some nodes as wired or base-station nodes, while the other mobile nodes communicate with a base station. Wireless standards and technologies are rapidly evolving, with mesh networks, in which all nodes communicate as peers, now beginning to emerge.

**A**s the processing power and memory capacity of small, inexpensive, low-power devices have continued to increase, the very concept of an Internet “host” has undergone a significant shift. Whereas the Internet of the 1970s and 1980s was mostly used to connect fixed computers, and today’s Internet hosts are often laptops or mobile phones, it is becoming feasible to think of much smaller objects, such as sensors and actuators, as legitimate Internet hosts. These devices are so small and potentially numerous that they have led to the concept of an “Internet of Things”—an Internet in which the majority of objects, ranging from light switches to boxes of inventory in a factory, might be addressable Internet “hosts.”

#### WHAT’S NEXT: “THE INTERNET OF THINGS”

While the concept of networking vast numbers of tiny objects might sound like science fiction (and perhaps dystopian fiction at that), there are many concrete and practical applications of this idea. One of the most popular is the idea of controlling energy consumption through the application of networking to everyday appliances. Light switches, power outlets, and appliances could all be fitted with sensors (to measure electrical load, ambient temperature, etc.) and actuators (e.g., to control when devices are active, such as postponing the use of a washing machine until an off-peak period when electricity is cheaper). This concept often appears under the title of “smart grids” and is actively being pursued by energy companies and equipment vendors today.

Pushing networking out to trillions of small, lower-power, inexpensive, and intermittently connected devices raises a host of technical challenges. As a simple example, the design of IP version 6, which we'll discuss in Chapter 4, was somewhat influenced by the realization that the number of addresses needed may be much larger than the number of conventional computers in the world. Similarly, new routing protocols are being developed to move data efficiently among devices that may have very low energy budgets and unreliable wireless connections to the rest of the world. There are even new operating systems developed specifically to run on tiny devices with limited power, CPU, and memory resources.

Exactly how this "Internet of Things" vision will play out remains to be seen, but at this point it seems clear that the Internet is moving beyond the original vision of just interconnecting computers. The applications that are enabled by interconnecting trillions of smart objects are just beginning to be realized.

---

## FURTHER READING

One of the most important contributions in computer networking over the last 20 years is the original paper by Metcalf and Boggs (1976) introducing the Ethernet. Many years later, Boggs, Mogul, and Kent (1988) reported their practical experiences with Ethernet, debunking many of the myths that had found their way into the literature over the years. Both papers are must reading. The third paper laid much of the groundwork for the development of wireless networks including 802.11.

- Metcalf, R., and D. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM* 19(7):395–403, July 1976.
- Boggs, D., J. Mogul, and C. Kent. Measured capacity of an Ethernet. *Proceedings of the SIGCOMM '88 Symposium*, pages 222–234, August 1988.
- Bharghavan, V., A. Demers, S. Shenker, and L. Zhang. MACAW: A media access protocol for wireless LANs. *Proceedings of the SIGCOMM '94 Symposium*, pages 212–225, August 1994.

There are countless textbooks with a heavy emphasis on the lower levels of the network hierarchy, with a particular focus on *telecommunications*—networking from the phone company's perspective. Books

by Spragins et al. [SHP91] and Minoli [Min93] are two good examples. Several other books concentrate on various local area network technologies. Of these, Stallings's book is the most comprehensive [Sta00], while Jain [Jai94] gives a good introduction to the low-level details of optical communication.

Wireless networking is a very active area of research, with many novel and interesting papers appearing each year. Gupta and Kumar's paper [GK00] establishes the theory behind capacity of wireless networks. Basagni et al. [BCGS04] provide a good introduction to *ad hoc* wireless networks. Bicket et al. [BABM05] describe the Roofnet wireless mesh network experiment, and Biswas and Morris [BM05] present ExOR, which ran on Roofnet. The latter paper was an early example of using cross-layer information to improve performance of a wireless network. A different use of cross-layer techniques to improve throughput in the face of bit errors is described by Jamieson et al. [JB07]. Wong et al. [WYLB06] look at the problem of how to pick the correct rate of data transmission given all the tradeoffs around error rate and bandwidth in a wireless channel. Katti et al. [KRH<sup>+</sup>06] established the viability of using network coding to improve the performance of wireless networks.

A recent book by Xiao et al. [XCL10] surveys many aspects of sensor networking. Vasseur and Dunkels [VD10] provide a forward-looking view of how the “Internet of Things” might play out with the adoption of Internet protocols to interconnect sensors and other smart objects.

For an introduction to information theory, Blahut's book is a good place to start [Bla87], along with Shannon's seminal paper on link capacity [Sha48].

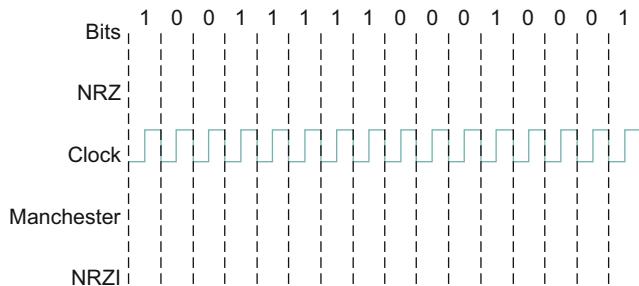
For a general introduction to the mathematics behind error codes, Rao and Fujiwara [RF89] is recommended. For a detailed discussion of the mathematics of CRCs in particular, along with some more information about the hardware used to calculate them, see Peterson and Brown [PB61].

Finally, we recommend the following live reference:

- <http://standards.ieee.org/>: status of various IEEE network-related standards, including Ethernet and 802.11

**EXERCISES**

1. Show the NRZ, Manchester, and NRZI encodings for the bit pattern shown in Figure 2.36. Assume that the NRZI signal starts out low.



■ FIGURE 2.36 Diagram for Exercise 1.

2. Show the 4B/5B encoding, and the resulting NRZI signal, for the following bit sequence:

1110 0101 0000 0011



3. Show the 4B/5B encoding, and the resulting NRZI signal, for the following bit sequence:

1101 1110 1010 1101 1011 1110 1110 1111

4. In the 4B/5B encoding (Table 2.2), only two of the 5-bit codes used end in two 0s. How many possible 5-bit sequences are there (used by the existing code or not) that meet the stronger restriction of having at most one leading and at most one trailing 0? Could all 4-bit sequences be mapped to such 5-bit sequences?

5. Assuming a framing protocol that uses bit stuffing, show the bit sequence transmitted over the link when the frame contains the following bit sequence:

110101111010111110101111110

Mark the stuffed bits.

6. Suppose the following sequence of bits arrives over a link:

1101011110101111001011110110

Show the resulting frame after any stuffed bits have been removed. Indicate any errors that might have been introduced into the frame.



7. Suppose the following sequence of bits arrives over a link:

011010111101010011111101100111110

Show the resulting frame after any stuffed bits have been removed. Indicate any errors that might have been introduced into the frame.

8. Suppose you want to send some data using the BISYNC framing protocol and the last 2 bytes of your data are DLE and ETX. What sequence of bytes would be transmitted immediately prior to the CRC?

9. For each of the following framing protocols, give an example of a byte/bit sequence that should never appear in a transmission:

- (a) BISYNC  
(b) HDLC



10. Assume that a SONET receiver resynchronizes its clock whenever a 1 bit appears; otherwise, the receiver samples the signal in the middle of what it believes is the bit's time slot.

- (a) What relative accuracy of the sender's and receiver's clocks is required in order to receive correctly 48 zero bytes (one ATM cell's worth) in a row?  
(b) Consider a forwarding station A on a SONET STS-1 line, receiving frames from the downstream end B and retransmitting them upstream. What relative accuracy of A's and B's clocks is required to keep A from accumulating more than one extra frame per minute?

11. Show that two-dimensional parity allows detection of all 3-bit errors.

12. Give an example of a 4-bit error that would not be detected by two-dimensional parity, as illustrated in Figure 2.14. What is the general set of circumstances under which 4-bit errors will be undetected?

13. Show that two-dimensional parity provides the receiver enough information to correct any 1-bit error (assuming the receiver knows only 1 bit is bad), but not any 2-bit error.
14. Show that the Internet checksum will never be 0xFFFF (that is, the final value of sum will not be 0x0000) unless every byte in the buffer is 0. (Internet specifications in fact require that a checksum of 0x0000 be transmitted as 0xFFFF; the value 0x0000 is then reserved for an omitted checksum. Note that, in ones complement arithmetic, 0x0000 and 0xFFFF are both representations of the number 0.)
15. Prove that the Internet checksum computation shown in the text is independent of byte order (host order or network order) except that the bytes in the final checksum should be swapped later to be in the correct order. Specifically, show that the sum of 16-bit words can be computed in either byte order. For example, if the one's complement sum (denoted by  $+$ ) of 16-bit words is represented as follows,

$$[A,B] +' [C,D] +' \dots +' [Y,Z]$$

the following swapped sum is the same as the original sum above:

$$[B,A] +' [D,C] +' \dots +' [Z,Y]$$

16. Suppose that one byte in a buffer covered by the Internet checksum algorithm needs to be decremented (e.g., a header hop count field). Give an algorithm to compute the revised checksum without rescanning the entire buffer. Your algorithm should consider whether the byte in question is low order or high order.
17. Show that the Internet checksum can be computed by first taking the 32-bit ones complement sum of the buffer in 32-bit units, then taking the 16-bit ones complement sum of the upper and lower halfwords, and finishing as before by complementing the result. (To take a 32-bit ones complement sum on 32-bit twos complement hardware, you need access to the “overflow” bit.)
18. Suppose we want to transmit the message 11100011 and protect it from errors using the CRC polynomial  $x^3 + 1$ .
  - (a) Use polynomial long division to determine the message that should be transmitted.

- (b) Suppose the leftmost bit of the message is inverted due to noise on the transmission link. What is the result of the receiver's CRC calculation? How does the receiver know that an error has occurred?

-  19. Suppose we want to transmit the message 1011 0010 0100 1011 and protect it from errors using the CRC8 polynomial  $x^8 + x^2 + x^1 + 1$ .
- Use polynomial long division to determine the message that should be transmitted.
  - Suppose the leftmost bit of the message is inverted due to noise on the transmission link. What is the result of the receiver's CRC calculation? How does the receiver know that an error has occurred?
20. The CRC algorithm as presented in this chapter requires lots of bit manipulations. It is, however, possible to do polynomial long division taking multiple bits at a time, via a table-driven method, that enables efficient software implementations of CRC. We outline the strategy here for long division 3 bits at a time (see Table 2.5); in practice, we would divide 8 bits at a time, and the table would have 256 entries.

Let the divisor polynomial  $C = C(x)$  be  $x^3 + x^2 + 1$ , or 1101. To build the table for  $C$ , we take each 3-bit sequence,  $p$ , append three trailing 0s, and then find the quotient  $q = p \curvearrowright 000 \div C$ ,

**Table 2.5 Table-Driven CRC Calculation**

$p$	$q = p \curvearrowright 000 \div C$	$C \times q$
000	000	000 000
001	001	001 101
010	011	010 ____
011	0____	011 ____
100	111	100 011
101	110	101 110
110	100	110 ____
111	____	111 ____

ignoring the remainder. The third column is the product  $C \times q$ , the first 3 bits of which should equal  $p$ .

- (a) Verify, for  $p = 110$ , that the quotients  $p \div C$  and  $p \div C$  are the same; that is, it doesn't matter what the trailing bits are.
- (b) Fill in the missing entries in the table.
- (c) Use the table to divide 101 001 011 001 100 by  $C$ . Hint: The first 3 bits of the dividend are  $p = 101$ , so from the table the corresponding first 3 bits of the quotient are  $q = 110$ . Write the 110 above the second 3 bits of the dividend, and subtract  $C \times q = 101\ 110$ , again from the table, from the first 6 bits of the dividend. Keep going in groups of 3 bits. There should be no remainder.



21. With 1 parity bit we can detect all 1-bit errors. Show that at least one generalization fails, as follows:
  - (a) Show that if messages  $m$  are 8 bits long, then there is no error detection code  $e = e(m)$  of size 2 bits that can detect all 2-bit errors. Hint: Consider the set  $M$  of all 8-bit messages with a single 1 bit; note that any message from  $M$  can be transmuted into any other with a 2-bit error, and show that some pair of messages  $m_1$  and  $m_2$  in  $M$  must have the same error code  $e$ .
  - (b) Find an  $N$  (not necessarily minimal) such that no 32-bit error detection code applied to  $N$ -bit blocks can detect all errors altering up to 8 bits.
22. Consider an ARQ protocol that uses only negative acknowledgments (NAKs), but no positive acknowledgments (ACKs). Describe what timeouts would have to be scheduled. Explain why an ACK-based protocol is usually preferred to a NAK-based protocol.
23. Consider an ARQ algorithm running over a 40-km point-to-point fiber link.
  - (a) Compute the one-way propagation delay for this link, assuming that the speed of light is  $2 \times 10^8$  m/s in the fiber.
  - (b) Suggest a suitable timeout value for the ARQ algorithm to use.
  - (c) Why might it still be possible for the ARQ algorithm to time out and retransmit a frame, given this timeout value?

24. Suppose you are designing a sliding window protocol for a 1-Mbps point-to-point link to the moon, which has a one-way latency of 1.25 seconds. Assuming that each frame carries 1 KB of data, what is the minimum number of bits you need for the sequence number?

✓ 25. Suppose you are designing a sliding window protocol for a 1-Mbps point-to-point link to the stationary satellite revolving around the Earth at an altitude of  $3 \times 10^4$  km. Assuming that each frame carries 1 KB of data, what is the minimum number of bits you need for the sequence number in the following cases? Assume the speed of light is  $3 \times 10^8$  m/s.

(a) RWS=1

(b) RWS=SWS

26. The text suggests that the sliding window protocol can be used to implement flow control. We can imagine doing this by having the receiver delay ACKs, that is, not send the ACK until there is free buffer space to hold the next frame. In doing so, each ACK would simultaneously acknowledge the receipt of the last frame and tell the source that there is now free buffer space available to hold the next frame. Explain why implementing flow control in this way is not a good idea.

27. Implicit in the stop-and-wait scenarios of Figure 2.17 is the notion that the receiver will retransmit its ACK immediately on receipt of the duplicate data frame. Suppose instead that the receiver keeps its own timer and retransmits its ACK only after the next expected frame has not arrived within the timeout interval. Draw timelines illustrating the scenarios in Figure 2.17(b) to (d); assume the receiver's timeout value is twice the sender's. Also redraw (c) assuming the receiver's timeout value is half the sender's.

28. In stop-and-wait transmission, suppose that both sender and receiver retransmit their last frame immediately on receipt of a duplicate ACK or data frame; such a strategy is superficially reasonable because receipt of such a duplicate is most likely to mean the other side has experienced a timeout.

(a) Draw a timeline showing what will happen if the first data frame is somehow duplicated, but no frame is lost. How long

will the duplications continue? This situation is known as the Sorcerer's Apprentice bug.

- (b) Suppose that, like data, ACKs are retransmitted if there is no response within the timeout period. Suppose also that both sides use the same timeout interval. Identify a reasonably likely scenario for triggering the Sorcerer's Apprentice bug.
29. Give some details of how you might augment the sliding window protocol with flow control by having ACKs carry additional information that reduces the SWS as the receiver runs out of buffer space. Illustrate your protocol with a timeline for a transmission; assume the initial SWS and RWS are 4, the link speed is instantaneous, and the receiver can free buffers at the rate of one per second (i.e., the receiver is the bottleneck). Show what happens at  $T = 0, T = 1, \dots, T = 4$  seconds.
30. Describe a protocol combining the sliding window algorithm with selective ACKs. Your protocol should retransmit promptly, but not if a frame simply arrives one or two positions out of order. Your protocol should also make explicit what happens if several consecutive frames are lost.
31. Draw a timeline diagram for the sliding window algorithm with  $SWS = RWS = 3$  frames, for the following two situations. Use a timeout interval of about  $2 \times RTT$ .
- (a) Frame 4 is lost.
  - (b) Frames 4 to 6 are lost.
- ✓ 32. Draw a timeline diagram for the sliding window algorithm with  $SWS = RWS = 4$  frames in the following two situations. Assume the receiver sends a duplicate acknowledgment if it does not receive the expected frame. For example, it sends DUPACK[2] when it expects to see Frame[2] but receives Frame[3] instead. Also, the receiver sends a cumulative acknowledgment after it receives all the outstanding frames. For example, it sends ACK[5] when it receives the lost frame Frame[2] after it already received Frame[3], Frame[4], and Frame[5]. Use a timeout interval of about  $2 \times RTT$ .
- (a) Frame 2 is lost. Retransmission takes place upon timeout (as usual).

- (b) Frame 2 is lost. Retransmission takes place either upon receipt of the first DUPACK or upon timeout. Does this scheme reduce the transaction time? (Note that some end-to-end protocols, such as variants of TCP, use similar schemes for fast retransmission.)
33. Suppose that we attempt to run the sliding window algorithm with  $SWS = RWS = 3$  and with  $\text{MaxSeqNum} = 5$ . The  $N$ th packet  $\text{DATA}[N]$  thus actually contains  $N \bmod 5$  in its sequence number field. Give an example in which the algorithm becomes confused; that is, a scenario in which the receiver expects  $\text{DATA}[5]$  and accepts  $\text{DATA}[0]$ —which has the same transmitted sequence number—in its stead. No packets may arrive out of order. Note that this implies  $\text{MaxSeqNum} \geq 6$  is necessary as well as sufficient.
34. Consider the sliding window algorithm with  $SWS = RWS = 3$ , with no out-of-order arrivals and with infinite-precision sequence numbers.
- Show that if  $\text{DATA}[6]$  is in the receive window, then  $\text{DATA}[0]$  (or in general any older data) cannot arrive at the receiver (and hence that  $\text{MaxSeqNum} = 6$  would have sufficed).
  - Show that if  $\text{ACK}[6]$  may be sent (or, more literally, that  $\text{DATA}[5]$  is in the sending window), then  $\text{ACK}[2]$  (or earlier) cannot be received.
- These amount to a proof of the formula given in Section 2.5.2, particularized to the case  $SWS = 3$ . Note that part (b) implies that the scenario of the previous problem cannot be reversed to involve a failure to distinguish  $\text{ACK}[0]$  and  $\text{ACK}[5]$ .
35. Suppose that we run the sliding window algorithm with  $SWS = 5$  and  $RWS = 3$ , and no out-of-order arrivals.
- Find the smallest value for  $\text{MaxSeqNum}$ . You may assume that it suffices to find the smallest  $\text{MaxSeqNum}$  such that if  $\text{DATA}[\text{MaxSeqNum}]$  is in the receive window, then  $\text{DATA}[0]$  can no longer arrive.
  - Give an example showing that  $\text{MaxSeqNum} - 1$  is not sufficient.
  - State a general rule for the minimum  $\text{MaxSeqNum}$  in terms of  $SWS$  and  $RWS$ .



■ FIGURE 2.37 Diagram for Exercises 36 to 38.

36. Suppose A is connected to B via an intermediate router R, as shown in Figure 2.37. The A–R and R–B links each accept and transmit only one packet per second in each direction (so two packets take 2 seconds), and the two directions transmit independently. Assume A sends to B using the sliding window protocol with  $SWS = 4$ .
- For Time = 0, 1, 2, 3, 4, 5, state what packets arrive at and leave each node, or label them on a timeline.
  - What happens if the links have a propagation delay of 1.0 second, but accept immediately as many packets as are offered (i.e., latency = 1 second but bandwidth is infinite)?
37. Suppose A is connected to B via an intermediate router R, as in the previous problem. The A–R link is instantaneous, but the R–B link transmits only one packet each second, one at a time (so two packets take 2 seconds). Assume A sends to B using the sliding window protocol with  $SWS = 4$ . For Time = 0, 1, 2, 3, 4, state what packets arrive at and are sent from A and B. How large does the queue at R grow?
38. Consider the situation in the previous exercise, except this time assume that the router has a queue size of 1; that is, it can hold one packet in addition to the one it is sending (in each direction). Let A's timeout be 5 seconds, and let SWS again be 4. Show what happens at each second from Time = 0 until all four packets from the first window-full are successfully delivered.
39. What kind of problems can arise when two hosts on the same Ethernet share the same hardware address? Describe what happens and why that behavior is a problem.
40. The 1982 Ethernet specification allowed between any two stations up to 1500 m of coaxial cable, 1000 m of other point-to-point link cable, and two repeaters. Each station or repeater connects to the coaxial cable via up to 50 m of “drop

**Table 2.6 Typical Delays Associated with Various Devices (Exercise 40)**

Item	Delay
Coaxial cable	Propagation speed .77c
Link/drop cable	Propagation speed .65c
Repeaters	Approximately 0.6 $\mu\text{s}$ each
Transceivers	Approximately 0.2 $\mu\text{s}$ each

cable.” Typical delays associated with each device are given in Table 2.6 (where  $c$  = speed of light in a vacuum =  $3 \times 10^8$  m/s). What is the worst-case round-trip propagation delay, measured in bits, due to the sources listed? (This list is not complete; other sources of delay include sense time and signal rise time.)

- ★ 41. Coaxial cable Ethernet was limited to a maximum of 500 m between repeaters, which regenerate the signal to 100% of its original amplitude. Along one 500-m segment, the signal could decay to no less than 14% of its original value (8.5 dB). Along 1500 m, then, the decay might be  $(0.14)^3 = 0.3\%$ . Such a signal, even along 2500 m, is still strong enough to be read; why then are repeaters required every 500 m?
42. Suppose the round-trip propagation delay for Ethernet is 46.4  $\mu\text{s}$ . This yields a minimum packet size of 512 bits (464 bits corresponding to propagation delay + 48 bits of jam signal).
- (a) What happens to the minimum packet size if the delay time is held constant, and the signalling rate rises to 100 Mbps?
  - (b) What are the drawbacks to so large a minimum packet size?
  - (c) If compatibility were not an issue, how might the specifications be written so as to permit a smaller minimum packet size?
- ★ 43. Let A and B be two stations attempting to transmit on an Ethernet. Each has a steady queue of frames ready to send; A’s frames will be numbered  $A_1, A_2$ , and so on, and B’s similarly. Let  $T = 51.2 \mu\text{s}$  be the exponential backoff base unit.
- Suppose A and B simultaneously attempt to send frame 1, collide, and happen to choose backoff times of  $0 \times T$  and  $1 \times T$ , respectively, meaning A wins the race and transmits  $A_1$  while B

waits. At the end of this transmission, B will attempt to retransmit  $B_1$  while A will attempt to transmit  $A_2$ . These first attempts will collide, but now A backs off for either  $0 \times T$  or  $1 \times T$ , while B backs off for time equal to one of  $0 \times T, \dots, 3 \times T$ .

- (a) Give the probability that A wins this second backoff race immediately after this first collision; that is, A's first choice of backoff time  $k \times 51.2$  is less than B's.
- (b) Suppose A wins this second backoff race. A transmits  $A_3$ , and when it is finished, A and B collide again as A tries to transmit  $A_4$  and B tries once more to transmit  $B_1$ . Give the probability that A wins this third backoff race immediately after the first collision.
- (c) Give a reasonable lower bound for the probability that A wins all the remaining backoff races.
- (d) What then happens to the frame  $B_1$ ?

This scenario is known as the Ethernet *capture effect*.

44. Suppose the Ethernet transmission algorithm is modified as follows: After each successful transmission attempt, a host waits one or two slot times before attempting to transmit again, and otherwise backs off the usual way.
- (a) Explain why the capture effect of the previous exercise is now much less likely.
  - (b) Show how the strategy above can now lead to a pair of hosts capturing the Ethernet, alternating transmissions, and locking out a third.
  - (c) Propose an alternative approach, for example, by modifying the exponential backoff. What aspects of a station's history might be used as parameters to the modified backoff?
45. Ethernets use Manchester encoding. Assuming that hosts sharing the Ethernet are not perfectly synchronized, why does this allow collisions to be detected soon after they occur, without waiting for the CRC at the end of the packet?
46. Suppose A, B, and C all make their first carrier sense, as part of an attempt to transmit, while a fourth station D is transmitting. Draw a timeline showing one possible sequence of transmissions, attempts, collisions, and exponential backoff choices. Your timeline should also meet the following criteria: (i) initial

transmission attempts should be in the order A, B, C but successful transmissions should be in the order C, B, A, and (ii) there should be at least four collisions.

- 47.** Repeat the previous exercise, now with the assumption that Ethernet is  $p$ -persistent with  $p = 0.33$  (that is, a waiting station transmits immediately with probability  $p$  when the line goes idle and otherwise defers one  $51.2\text{-}\mu\text{s}$  slot time and repeats the process). Your timeline should meet criterion (i) of the previous problem, but in lieu of criterion (ii) you should show at least one collision and at least one run of four deferrals on an idle line. Again, note that many solutions are possible.

- 48.** Suppose Ethernet physical addresses are chosen at random (using true random bits).
- What is the probability that on a 1024-host network, two addresses will be the same?
  - What is the probability that the above event will occur on one or more of  $2^{20}$  networks?
  - What is the probability that, of the  $2^{30}$  hosts in all the networks of (b), some pair has the same address?

Hint: The calculation for (a) and (c) is a variant of that used in solving the so-called Birthday Problem: Given  $N$  people, what is the probability that two of their birthdays (addresses) will be the same? The second person has probability  $1 - \frac{1}{365}$  of having a different birthday from the first, the third has probability  $1 - \frac{2}{365}$  of having a different birthday from the first two, and so on. The probability that all birthdays are different is thus

$$\left(1 - \frac{1}{365}\right) \times \left(1 - \frac{2}{365}\right) \times \cdots \times \left(1 - \frac{N-1}{365}\right)$$

which for smallish  $N$  is about

$$1 - \frac{1 + 2 + \cdots + (N-1)}{365}$$

- 49.** Suppose five stations are waiting for another packet to finish on an Ethernet. All transmit at once when the packet is finished and collide.
- Simulate this situation up until the point when one of the five waiting stations succeeds. Use coin flips or some other

genuine random source to determine backoff times. Make the following simplifications: Ignore inter-frame spacing, ignore variability in collision times (so that retransmission is always after an exact integral multiple of the 51.2- $\mu$ s slot time), and assume that each collision uses up exactly one slot time.

- (b) Discuss the effect of the listed simplifications in your simulation versus the behavior you might encounter on a real Ethernet.
50. Write a program to implement the simulation discussed above, this time with  $N$  stations waiting to transmit. Again, model time as an integer,  $T$ , in units of slot times, and again treat collisions as taking one slot time (so a collision at time  $T$  followed by a backoff of  $k = 0$  would result in a retransmission attempt at time  $T + 1$ ). Find the average delay before *one* station transmits successfully, for  $N = 20$ ,  $N = 40$ , and  $N = 100$ . Does your data support the notion that the delay is linear in  $N$ ? Hint: For each station, keep track of that station's `NextTimeToSend` and `CollisionCount`. You are done when you reach a time  $T$  for which there is only one station with `NextTimeToSend == T`. If there is no such station, increment  $T$ . If there are two or more, schedule the retransmissions and try again.
51. Suppose that  $N$  Ethernet stations, all trying to send at the same time, require  $N/2$  slot times to sort out who transmits next. Assuming the average packet size is 5 slot times, express the available bandwidth as a function of  $N$ .
52. Consider the following Ethernet model. Transmission attempts are at random times with an average spacing of  $\lambda$  slot times; specifically, the interval between consecutive attempts is an exponential random variable  $x = -\lambda \log u$ , where  $u$  is chosen randomly in the interval  $0 \leq u \leq 1$ . An attempt at time  $t$  results in a collision if there is another attempt in the range from  $t - 1$  to  $t + 1$ , where  $t$  is measured in units of the 51.2- $\mu$ s slot time; otherwise, the attempt succeeds.
- (a) Write a program to simulate, for a given value of  $\lambda$ , the average number of slot times needed before a successful transmission, called the *contention interval*. Find the

minimum value of the contention interval. Note that you will have to find one attempt past the one that succeeds in order to determine if there was a collision. Ignore retransmissions, which probably do not fit the random model above.

- (b) The Ethernet alternates between contention intervals and successful transmissions. Suppose the average successful transmission lasts 8 slot times (512 bytes). Using your minimum length of the contention interval from above, what fraction of the theoretical 10-Mbps bandwidth is available for transmissions?
53. How can a wireless node interfere with the communications of another node when the two nodes are separated by a distance greater than the transmission range of either node?
54. Why is collision detection more complex in wireless networks than in wired networks such as Ethernet?
55. How can hidden terminals be detected in 802.11 networks?
56. Why might a wireless mesh topology be superior to a base station topology for communications in a natural disaster?
57. Why isn't it practical for each node in a sensor net to learn its location by using GPS? Describe a practical alternative.

This page intentionally left blank



# 3

## Internetworking

*Nature seems ... to reach many of her ends by long circuitous routes.*

—Rudolph Lotze

In the previous chapter, we saw how to connect one node to another or to an existing network. Many technologies can be used to build “last-mile” links or to connect a modest number of nodes together, but how do we build networks of global scale? A single Ethernet can interconnect no more than 1024 hosts; a point-to-point link connects only two. Wireless networks are limited by the ranges of their radios. To build a global network, we need a way to interconnect these different types of links and networks. The concept of interconnecting different types of networks to

### PROBLEM: NOT ALL NETWORKS ARE DIRECTLY CONNECTED

build a large, global network is the core idea of the Internet and is often referred to as *internetworking*.

We can divide the internetworking problem up into a few subproblems. First of all, we need a way to interconnect links. Devices that interconnect links of the same type are often called *switches*, and these devices are the first topic of this chapter. A particularly important class of switches today are those used to interconnect Ethernet segments; these switches are also sometimes called *bridges*. The core job of a switch is to take packets that arrive on an input and *forward*

(or *switch*) them to the right output so that they will reach their appropriate destination. There are a variety of ways that the switch can determine the “right” output for a packet, which can be broadly categorized as connectionless and connection-oriented approaches. These two approaches have both found important application areas over the years.

Given the enormous diversity of network types, we also need a way to interconnect disparate networks and links (i.e., deal with *heterogeneity*). Devices that perform this task, once called *gateways*, are now mostly known as *routers*. The protocol that was invented to deal with interconnection of disparate network types, the Internet Protocol (IP), is the topic of our second section.

Once we interconnect a whole lot of links and networks with switches and routers, there are likely to be many different possible ways to get from one point to another. Finding a suitable path or *route* through a network is one of the fundamental problems of networking. Such paths should be efficient (e.g., no longer than necessary), loop free, and able to respond to the fact that networks are not static—nodes may fail or reboot, links may break, and new nodes or links may be added. Our third section looks at some of the algorithms and protocols that have been developed to address these issues.

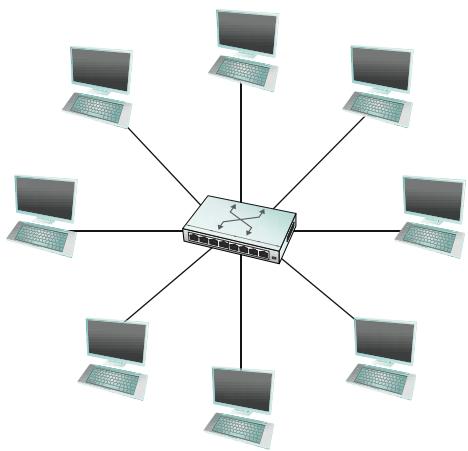
Once we understand the problems of switching and routing, we need some devices to perform those functions. This chapter concludes with some discussion of the ways switches and routers are built. While many packet switches and routers are quite similar to a general-purpose computer, there are many situations where more specialized designs are used. This is particularly the case at the high end, where there seems to be a never-ending need for bigger and faster routers to handle the ever-increasing traffic load in the Internet’s core.



### 3.1 SWITCHING AND BRIDGING

In the simplest terms, a switch is a mechanism that allows us to interconnect links to form a larger network. A switch is a multi-input, multi-output device that transfers packets from an input to one or more outputs. Thus, a switch adds the star topology (see Figure 3.1) to the point-to-point link, bus (Ethernet), and ring topologies established in the last chapter. A star topology has several attractive properties:

- Even though a switch has a fixed number of inputs and outputs, which limits the number of hosts that can be connected to a single switch, large networks can be built by interconnecting a number of switches.



■ FIGURE 3.1 A switch provides a star topology.

- We can connect switches to each other and to hosts using point-to-point links, which typically means that we can build networks of large geographic scope.
- Adding a new host to the network by connecting it to a switch does not necessarily reduce the performance of the network for other hosts already connected.

This last claim cannot be made for the shared-media networks discussed in the last chapter. For example, it is impossible for two hosts on the same 10-Mbps Ethernet segment to transmit continuously at 10 Mbps because they share the same transmission medium. Every host on a switched network has its own link to the switch, so it may be entirely possible for many hosts to transmit at the full link speed (bandwidth), provided that the switch is designed with enough aggregate capacity. Providing high aggregate throughput is one of the design goals for a switch; we return to this topic later. In general, switched networks are considered more *scalable* (i.e., more capable of growing to large numbers of nodes) than shared-media networks because of this ability to support many hosts at full speed.

A switch is connected to a set of links and, for each of these links, runs the appropriate data link protocol to communicate with the node at the other end of the link. A switch's primary job is to receive incoming packets on one of its links and to transmit them on some other link. This function is sometimes referred to as either *switching* or *forwarding*, and in

terms of the Open Systems Interconnection (OSI) architecture, it is the main function of the network layer.

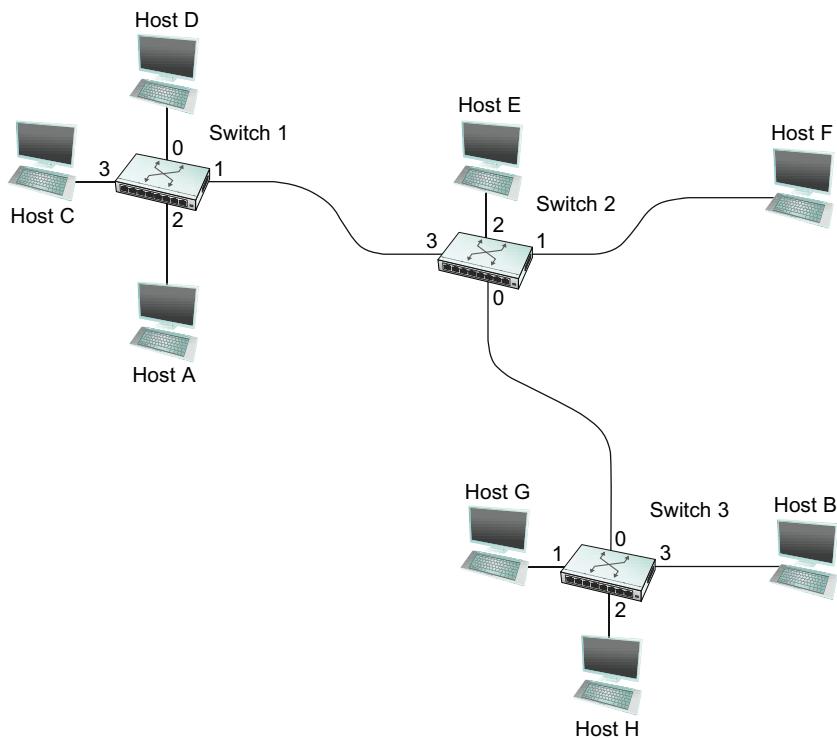
The question, then, is how does the switch decide which output link to place each packet on? The general answer is that it looks at the header of the packet for an identifier that it uses to make the decision. The details of how it uses this identifier vary, but there are two common approaches. The first is the *datagram* or *connectionless* approach. The second is the *virtual circuit* or *connection-oriented* approach. A third approach, *source routing*, is less common than these other two, but it does have some useful applications.

One thing that is common to all networks is that we need to have a way to identify the end nodes. Such identifiers are usually called *addresses*. We have already seen examples of addresses in the previous chapter, such as the 48-bit address used for Ethernet. The only requirement for Ethernet addresses is that no two nodes on a network have the same address. This is accomplished by making sure that all Ethernet cards are assigned a *globally unique* identifier. For the following discussions, we assume that each host has a globally unique address. Later on, we consider other useful properties that an address might have, but global uniqueness is adequate to get us started.

Another assumption that we need to make is that there is some way to identify the input and output ports of each switch. There are at least two sensible ways to identify ports: One is to number each port, and the other is to identify the port by the name of the node (switch or host) to which it leads. For now, we use numbering of the ports.

### 3.1.1 Datagrams

The idea behind datagrams is incredibly simple: You just include in every packet enough information to enable any switch to decide how to get it to its destination. That is, every packet contains the complete destination address. Consider the example network illustrated in Figure 3.2, in which the hosts have addresses A, B, C, and so on. To decide how to forward a packet, a switch consults a *forwarding table* (sometimes called a *routing table*), an example of which is depicted in Table 3.1. This particular table shows the forwarding information that switch 2 needs to forward datagrams in the example network. It is pretty easy to figure out such a table when you have a complete map of a simple network like that depicted here; we could imagine a network operator configuring



■ FIGURE 3.2 Datagram forwarding: an example network.

**Table 3.1 Forwarding Table for Switch 2**

Destination	Port
A	3
B	0
C	3
D	3
E	2
F	1
G	0
H	0

the tables statically. It is a lot harder to create the forwarding tables in large, complex networks with dynamically changing topologies and multiple paths between destinations. That harder problem is known as *routing* and is the topic of Section 3.3. We can think of routing as a process that takes place in the background so that, when a data packet turns up, we will have the right information in the forwarding table to be able to forward, or switch, the packet.

Datagram networks have the following characteristics:

- A host can send a packet anywhere at any time, since any packet that turns up at a switch can be immediately forwarded (assuming a correctly populated forwarding table). For this reason, datagram networks are often called *connectionless*; this contrasts with the *connection-oriented* networks described below, in which some *connection state* needs to be established before the first data packet is sent.
- When a host sends a packet, it has no way of knowing if the network is capable of delivering it or if the destination host is even up and running.
- Each packet is forwarded independently of previous packets that might have been sent to the same destination. Thus, two successive packets from host A to host B may follow completely different paths (perhaps because of a change in the forwarding table at some switch in the network).
- A switch or link failure might not have any serious effect on communication if it is possible to find an alternate route around the failure and to update the forwarding table accordingly.

This last fact is particularly important to the history of datagram networks. One of the important design goals of the Internet is robustness to failures, and history has shown it to be quite effective at meeting this goal.<sup>1</sup>

### 3.1.2 Virtual Circuit Switching

A second technique for packet switching, which differs significantly from the datagram model, uses the concept of a *virtual circuit* (VC).

---

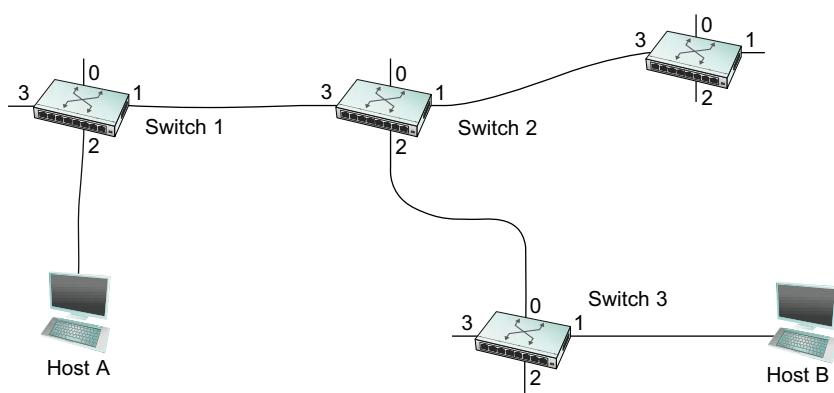
<sup>1</sup>The oft-repeated claim that the ARPANET was built to withstand nuclear attack does not appear to be substantiated by those who actually worked on its design, but robustness to failure of individual components was certainly a goal.

This approach, which is also referred to as a *connection-oriented model*, requires setting up a virtual connection from the source host to the destination host before any data is sent. To understand how this works, consider Figure 3.3, where host A again wants to send packets to host B. We can think of this as a two-stage process. The first stage is “connection setup.” The second is data transfer. We consider each in turn.

In the connection setup phase, it is necessary to establish a “connection state” in each of the switches between the source and destination hosts. The connection state for a single connection consists of an entry in a “VC table” in each switch through which the connection passes. One entry in the VC table on a single switch contains:

- A *virtual circuit identifier* (VCI) that uniquely identifies the connection at this switch and which will be carried inside the header of the packets that belong to this connection
- An incoming interface on which packets for this VC arrive at the switch
- An outgoing interface in which packets for this VC leave the switch
- A potentially different VCI that will be used for outgoing packets

The semantics of one such entry is as follows: If a packet arrives on the designated incoming interface and that packet contains the designated VCI value in its header, then that packet should be sent out the specified outgoing interface with the specified outgoing VCI value having been first placed in its header.



■ FIGURE 3.3 An example of a virtual circuit network.

Note that the combination of the VCI of packets as they are received at the switch *and* the interface on which they are received uniquely identifies the virtual connection. There may of course be many virtual connections established in the switch at one time. Also, we observe that the incoming and outgoing VCI values are generally not the same. Thus, the VCI is not a globally significant identifier for the connection; rather, it has significance only on a given link (i.e., it has *link-local scope*).

Whenever a new connection is created, we need to assign a new VCI for that connection on each link that the connection will traverse. We also need to ensure that the chosen VCI on a given link is not currently in use on that link by some existing connection.

There are two broad approaches to establishing connection state. One is to have a network administrator configure the state, in which case the virtual circuit is “permanent.” Of course, it can also be deleted by the administrator, so a permanent virtual circuit (PVC) might best be thought of as a long-lived or administratively configured VC. Alternatively, a host can send messages into the network to cause the state to be established. This is referred to as *signalling*, and the resulting virtual circuits are said to be *switched*. The salient characteristic of a switched virtual circuit (SVC) is that a host may set up and delete such a VC dynamically without the involvement of a network administrator. Note that an SVC should more accurately be called a *signalled* VC, since it is the use of signalling (not switching) that distinguishes an SVC from a PVC.

Let’s assume that a network administrator wants to manually create a new virtual connection from host A to host B.<sup>2</sup> First, the administrator needs to identify a path through the network from A to B. In the example network of Figure 3.3, there is only one such path, but in general this may not be the case. The administrator then picks a VCI value that is currently unused on each link for the connection. For the purposes of our example, let’s suppose that the VCI value 5 is chosen for the link from host A to switch 1, and that 11 is chosen for the link from switch 1 to switch 2. In that case, switch 1 needs to have an entry in its VC table configured as shown in Table 3.2.

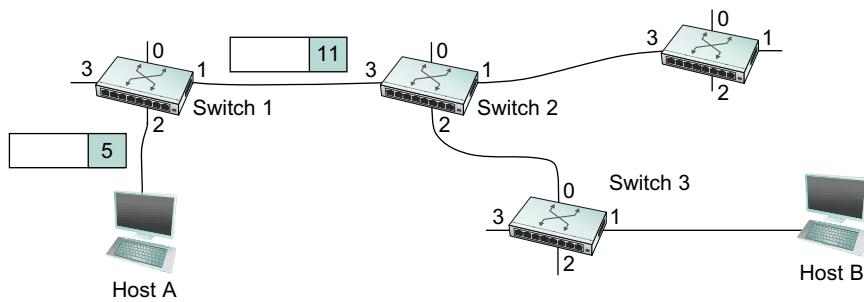
Similarly, suppose that the VCI of 7 is chosen to identify this connection on the link from switch 2 to switch 3 and that a VCI of 4 is chosen for

---

<sup>2</sup>In practice, the process would likely be much more automated than described here, perhaps using some sort of graphical network management tool. The following steps illustrate the state that has to be established in any case.

**Table 3.2** Virtual Circuit Table Entry for Switch 1

Incoming Interface	Incoming VCI	Outgoing Interface	Outgoing VCI
2	5	1	11

**FIGURE 3.4** A packet is sent into a virtual circuit network.

the link from switch 3 to host B. In that case, switches 2 and 3 need to be configured with VC table entries as shown in Table 3.3. Note that the “outgoing” VCI value at one switch is the “incoming” VCI value at the next switch.

Once the VC tables have been set up, the data transfer phase can proceed, as illustrated in Figure 3.4. For any packet that it wants to send to host B, A puts the VCI value of 5 in the header of the packet and sends it to switch 1. Switch 1 receives any such packet on interface 2, and it uses the combination of the interface and the VCI in the packet header to find the appropriate VC table entry. As shown in Table 3.2, the table entry in this

**Table 3.3** Virtual Circuit Table Entries for Switches 2 and 3

VC Table Entry at Switch 2

Incoming Interface	Incoming VCI	Outgoing Interface	Outgoing VCI
3	11	2	7

VC Table Entry at Switch 3

Incoming Interface	Incoming VCI	Outgoing Interface	Outgoing VCI
0	7	1	4

case tells switch 1 to forward the packet out of interface 1 and to put the VCI value 11 in the header when the packet is sent. Thus, the packet will arrive at switch 2 on interface 3 bearing VCI 11. Switch 2 looks up interface 3 and VCI 11 in its VC table (as shown in Table 3.3) and sends the packet on to switch 3 after updating the VCI value in the packet header appropriately, as shown in Figure 3.5. This process continues until it arrives at host B with the VCI value of 4 in the packet. To host B, this identifies the packet as having come from host A.

In real networks of reasonable size, the burden of configuring VC tables correctly in a large number of switches would quickly become excessive using the above procedures. Thus, either a network management tool or some sort of signalling (or both) is almost always used, even when setting up “permanent” VCs. In the case of PVCs, signalling is initiated by the network administrator, while SVCs are usually set up using signalling by one of the hosts. We consider now how the same VC just described could be set up by signalling from the host.

To start the signalling process, host A sends a setup message into the network—that is, to switch 1. The setup message contains, among other things, the complete destination address of host B. The setup message needs to get all the way to B to create the necessary connection state in every switch along the way. We can see that getting the setup message to B is a lot like getting a datagram to B, in that the switches have to know which output to send the setup message to so that it eventually reaches B. For now, let’s just assume that the switches know enough about the network topology to figure out how to do that, so that the setup message flows on to switches 2 and 3 before finally reaching host B.

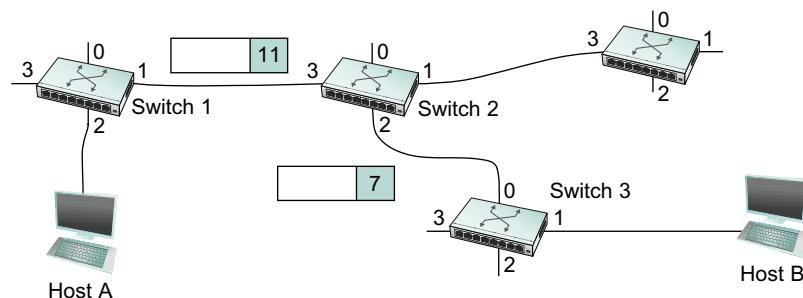


FIGURE 3.5 A packet makes its way through a virtual circuit network.

When switch 1 receives the connection request, in addition to sending it on to switch 2, it creates a new entry in its virtual circuit table for this new connection. This entry is exactly the same as shown previously in Table 3.2. The main difference is that now the task of assigning an unused VCI value on the interface is performed by the switch for that port. In this example, the switch picks the value 5. The virtual circuit table now has the following information: “When packets arrive on port 2 with identifier 5, send them out on port 1.” Another issue is that, somehow, host A will need to learn that it should put the VCI value of 5 in packets that it wants to send to B; we will see how that happens below.

When switch 2 receives the setup message, it performs a similar process; in this example, it picks the value 11 as the incoming VCI value. Similarly, switch 3 picks 7 as the value for its incoming VCI. Each switch can pick any number it likes, as long as that number is not currently in use for some other connection on that port of that switch. As noted above, VCIs have link-local scope; that is, they have no global significance.

Finally, the setup message arrives as host B. Assuming that B is healthy and willing to accept a connection from host A, it too allocates an incoming VCI value, in this case 4. This VCI value can be used by B to identify all packets coming from host A.

Now, to complete the connection, everyone needs to be told what their downstream neighbor is using as the VCI for this connection. Host B sends an acknowledgment of the connection setup to switch 3 and includes in that message the VCI that it chose (4). Now switch 3 can complete the virtual circuit table entry for this connection, since it knows the outgoing value must be 4. Switch 3 sends the acknowledgment on to switch 2, specifying a VCI of 7. Switch 2 sends the message on to switch 1, specifying a VCI of 11. Finally, switch 1 passes the acknowledgment on to host A, telling it to use the VCI of 5 for this connection.

At this point, everyone knows all that is necessary to allow traffic to flow from host A to host B. Each switch has a complete virtual circuit table entry for the connection. Furthermore, host A has a firm acknowledgment that everything is in place all the way to host B. At this point, the connection table entries are in place in all three switches just as in the administratively configured example above, but the whole process happened automatically in response to the signalling message sent from A. The data transfer phase can now begin and is identical to that used in the PVC case.

When host A no longer wants to send data to host B, it tears down the connection by sending a teardown message to switch 1. The switch removes the relevant entry from its table and forwards the message on to the other switches in the path, which similarly delete the appropriate table entries. At this point, if host A were to send a packet with a VCI of 5 to switch 1, it would be dropped as if the connection had never existed.

There are several things to note about virtual circuit switching:

- Since host A has to wait for the connection request to reach the far side of the network and return before it can send its first data packet, there is at least one round-trip time (RTT) of delay before data is sent.<sup>3</sup>
- While the connection request contains the full address for host B (which might be quite large, being a global identifier on the network), each data packet contains only a small identifier, which is only unique on one link. Thus, the per-packet overhead caused by the header is reduced relative to the datagram model.
- If a switch or a link in a connection fails, the connection is broken and a new one will need to be established. Also, the old one needs to be torn down to free up table storage space in the switches.
- The issue of how a switch decides which link to forward the connection request on has been glossed over. In essence, this is the same problem as building up the forwarding table for datagram forwarding, which requires some sort of *routing algorithm*. Routing is described in Section 3.3, and the algorithms described there are generally applicable to routing setup requests as well as datagrams.

One of the nice aspects of virtual circuits is that by the time the host gets the go-ahead to send data, it knows quite a lot about the network—for example, that there really is a route to the receiver and that the receiver is willing and able to receive data. It is also possible to allocate resources to the virtual circuit at the time it is established. For example, X.25 was an early (and now largely obsolete) virtual-circuit-based

---

<sup>3</sup>This is not strictly true. Some people have proposed “optimistically” sending a data packet immediately after sending the connection request. However, most current implementations wait for connection setup to complete before sending data.

networking technology. X.25 networks employ the following three-part strategy:

1. Buffers are allocated to each virtual circuit when the circuit is initialized.
2. The sliding window protocol (Section 2.5) is run between each pair of nodes along the virtual circuit, and this protocol is augmented with flow control to keep the sending node from over-running the buffers allocated at the receiving node.
3. The circuit is rejected by a given node if not enough buffers are available at that node when the connection request message is processed.

In doing these three things, each node is ensured of having the buffers it needs to queue the packets that arrive on that circuit. This basic strategy is usually called *hop-by-hop flow control*.

By comparison, a datagram network has no connection establishment phase, and each switch processes each packet independently, making it less obvious how a datagram network would allocate resources in a meaningful way. Instead, each arriving packet competes with all other packets for buffer space. If there are no free buffers, the incoming packet must be discarded. We observe, however, that even in a datagram-based network a source host often sends a sequence of packets to the same destination host. It is possible for each switch to distinguish among the set of packets it currently has queued, based on the source/destination pair, and thus for the switch to ensure that the packets belonging to each source/destination pair are receiving a fair share of the switch's buffers. We discuss this idea in much greater depth in Chapter 6.

In the virtual circuit model, we could imagine providing each circuit with a different *quality of service* (QoS). In this setting, the term *quality of service* is usually taken to mean that the network gives the user some kind of performance-related guarantee, which in turn implies that switches set aside the resources they need to meet this guarantee. For example, the switches along a given virtual circuit might allocate a percentage of each outgoing link's bandwidth to that circuit. As another example, a sequence of switches might ensure that packets belonging to a particular circuit not be delayed (queued) for more than a certain amount of time. We return to the topic of quality of service in Section 6.5.

### Introduction to Congestion

One important issue that switch designers face is *contention*. Contention occurs when multiple packets have to be queued at a switch because they are competing for the same output link. We'll look at how switches deal with this issue in [Section 1.4](#). You can think of contention as something that happens at the timescale of individual packet arrivals. Congestion, by contrast, happens at a slightly longer timescale, when a switch has so many packets queued that it runs out of buffer space and has to start dropping packets. We'll return to the topic of congestion in [Chapter 6](#), after we have seen the transport protocol component of the network architecture. At this point, however, we observe that how you deal with congestion is related to the issue of whether your network uses virtual circuits or datagrams.

On the one hand, suppose that each switch allocates enough buffers to handle the packets belonging to each virtual circuit it supports, as is done in an X.25 network. In this case, the network has defined away the problem of congestion—a switch never encounters a situation in which it has more packets to queue than it has buffer space, since it does not allow the connection to be established in the first place unless it can dedicate enough resources to it to avoid this situation. The problem with this approach, however, is that it is extremely conservative—it is unlikely that all the circuits will need to use all of their buffers at the same time, and as a consequence the switch is potentially underutilized.

On the other hand, the datagram model seemingly invites congestion—you do not know that there is enough contention at a switch to cause congestion until you run out of buffers. At that point, it is too late to prevent the congestion, and your only choice is to try to recover from it. The good news, of course, is that you may be able to get better utilization out of your switches since you are not holding buffers in reserve for a worst-case scenario that is unlikely to happen.

As is quite often the case, nothing is strictly black and white—there are design advantages for defining congestion away (as the X.25 model does) and for doing nothing about congestion until after it happens (as the simple datagram model does). There are also intermediate points between these two extremes. We describe some of these design points in [Chapter 6](#).

There have been a number of successful examples of virtual circuit technologies over the years, notably X.25, Frame Relay, and Asynchronous Transfer Mode (ATM). With the success of the Internet's connectionless model, however, none of them enjoys great popularity today. One of the most common applications of virtual circuits for many years was the construction of *virtual private networks* (VPNs), a subject discussed in [Section 3.2.9](#). Even that application is now mostly supported using Internet-based technologies today.

## Optical Switching

To a casual observer of the networking industry around the year 2000, it might have appeared that the most interesting sort of switching was optical switching. Indeed, optical switching did become an important technology in the late 1990s, due to a confluence of several factors. One factor was the commercial availability of dense wavelength division multiplexing (DWDM) equipment, which makes it possible to send a great deal of information down a single fiber by transmitting on a large number of optical wavelengths (or colors) at once. Thus, for example, one might send data on 100 or more different wavelengths, and each wavelength might carry as much as 10 Gbps of data.

A second factor was the commercial availability of optical amplifiers. Optical signals are attenuated as they pass through fiber, and after some distance (about 40 km or so) they need to be made stronger in some way. Before optical amplifiers, it was necessary to place *repeaters* in the path to recover the optical signal, convert it to a digital electronic signal, and then convert it back to optical again. Before you could get the data into a repeater, you would have to demultiplex it using a DWDM terminal. Thus, a large number of DWDM terminals would be needed just to drive a single fiber pair for a long distance. Optical amplifiers, unlike repeaters, are *analog* devices that boost whatever signal is sent along the fiber, even if it is sent on a hundred different wavelengths. Optical amplifiers therefore made DWDM gear much more attractive, because now a pair of DWDM terminals could talk to each other when separated by a distance of hundreds of kilometers. Furthermore, you could even upgrade the DWDM gear at the ends without touching the optical amplifiers in the middle of the path, because they will amplify 100 wavelengths as easily as 50 wavelengths.

With DWDM and optical amplifiers, it became possible to build optical networks of huge capacity. But at least one more type of device is needed to make these networks useful—the *optical switch*. Most so-called optical switches today actually perform their switching function electronically, and from an architectural point of view they have more in common with the circuit switches of the telephone network than the packet switches described in this chapter. A typical optical switch has a large number of interfaces that understand SONET framing and is able to cross-connect a SONET channel from an incoming interface to an outgoing interface. Thus, with an optical switch, it becomes possible to provide SONET channels from point A to point B via point C even if there is no direct fiber path from A to B—there just needs to be a path from A to C, a switch at C, and a path from C to B. In this respect, an optical switch bears some relationship to the switches in [Figure 3.3](#), in that it creates the illusion of a connection between two points even when there is no direct physical connection between them. However, optical switches do not provide virtual circuits, they provide “real” circuits (e.g., a SONET channel). There are even some newer types of optical

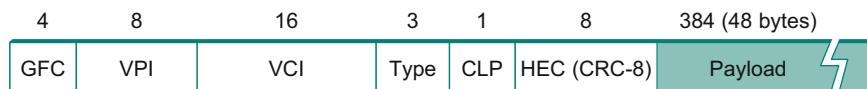
switches that use microscopic, electronically controlled mirrors to deflect all the light from one switch port to another, so that there could be an uninterrupted optical channel from point A to point B. The technology behind these devices is called *MEMS (microelectromechanical systems)*.

We don't cover optical networking extensively in this book, in part because of space considerations. For many practical purposes, you can think of optical networks as a piece of the infrastructure that enables telephone companies to provide SONET links or other types of circuits where and when you need them. However, it is worth noting that many of the technologies that are discussed later in this book, such as routing protocols and multiprotocol label switching, do have application to the world of optical networking.

#### Asynchronous Transfer Mode (ATM)

Asynchronous Transfer Mode (ATM) is probably the most well-known virtual circuit-based networking technology, although it is now somewhat past its peak in terms of deployment. ATM became an important technology in the 1980s and early 1990s for a variety of reasons, not the least of which is that it was embraced by the telephone industry, which had historically been less than active in data communications (other than as a supplier of links from which other people built networks). ATM also happened to be in the right place at the right time, as a high-speed switching technology that appeared on the scene just when shared media like Ethernet and token rings were starting to look a bit too slow for many users of computer networks. In some ways ATM was a competing technology with Ethernet switching, and it was seen by many as a competitor to IP as well.

There are a few aspects of ATM that are worth examining. The picture of the ATM packet format—more commonly called an ATM *cell*—in Figure 3.6 will illustrate the main points. We'll skip the generic flow control (GFC) bits, which never saw much use, and start with the 24 bits that are labelled VPI (virtual path identifier—8 bits) and VCI (virtual circuit identifier—16 bits). If you consider these bits together as a single 24-bit



■ FIGURE 3.6 ATM cell format at the UNI.

field, they correspond to the virtual circuit identifier introduced above. The reason for breaking the field into two parts was to allow for a level of hierarchy: All the circuits with the same VPI could, in some cases, be treated as a group (a virtual path) and could all be switched together looking only at the VPI, simplifying the work of a switch that could ignore all the VCI bits and reducing the size of the VC table considerably.

Skipping to the last header byte we find an 8-bit cyclic redundancy check (CRC), known as the *header error check* (HEC). It uses the CRC-8 polynomial given in Section 2.4.3 and provides error detection and single-bit error correction capability on the cell header only. Protecting the cell header is particularly important because an error in the VCI will cause the cell to be misdelivered.

Probably the most significant thing to notice about the ATM cell, and the reason it is called a cell and not a packet, is that it comes in only one size: 53 bytes. What was the reason for this? A big reason was to facilitate the implementation of hardware switches. When ATM was being created in the mid- and late 1980s, 10-Mbps Ethernet was the cutting-edge technology in terms of speed. To go much faster, most people thought in terms of hardware. Also, in the telephone world, people think big when they think of switches—telephone switches often serve tens of thousands of customers. Fixed-length packets turn out to be a very helpful thing if you want to build fast, highly scalable switches. There are two main reasons for this:

1. It is easier to build hardware to do simple jobs, and the job of processing packets is simpler when you already know how long each one will be.
2. If all packets are the same length, then you can have lots of switching elements all doing much the same thing in parallel, each of them taking the same time to do its job.

This second reason, the enabling of parallelism, greatly improves the scalability of switch designs. It would be overstating the case to say that fast parallel hardware switches can only be built using fixed-length cells. However, it is certainly true that cells ease the task of building such hardware and that there was a lot of knowledge available about how to build cell switches in hardware at the time the ATM standards were being defined. As it turns out, this same principle is still applied in many switches and routers today, even if they deal in variable length packets—they cut

those packets into some sort of cell in order to switch them, as we'll see in Section 3.4.

Having decided to use small, fixed-length packets, the next question is what is the right length to fix them at? If you make them too short, then the amount of header information that needs to be carried around relative to the amount of data that fits in one cell gets larger, so the percentage of link bandwidth that is actually used to carry data goes down. Even more seriously, if you build a device that processes cells at some maximum number of cells per second, then as cells get shorter the total data rate drops in direct proportion to cell size. An example of such a device might be a network adaptor that reassembles cells into larger units before handing them up to the host. The performance of such a device depends directly on cell size. On the other hand, if you make the cells too big, then there is a problem of wasted bandwidth caused by the need to pad transmitted data to fill a complete cell. If the cell payload size is 48 bytes and you want to send 1 byte, you'll need to send 47 bytes of padding. If this happens a lot, then the utilization of the link will be very low. The combination of relatively high header-to-payload ratio plus the frequency of sending partially filled cells did actually lead to some noticeable inefficiency in ATM networks that some detractors called the *cell tax*.

As it turns out, 48 bytes was picked for the ATM cell payload as a compromise. There were good arguments for both larger and smaller cells, but 48 made almost no one happy—a power of two would certainly have been better for computers to work with.

### 3.1.3 Source Routing

A third approach to switching that uses neither virtual circuits nor conventional datagrams is known as *source routing*. The name derives from the fact that all the information about network topology that is required to switch a packet across the network is provided by the source host.

There are various ways to implement source routing. One would be to assign a number to each output of each switch and to place that number in the header of the packet. The switching function is then very simple: For each packet that arrives on an input, the switch would read the port number in the header and transmit the packet on that output. However, since there will in general be more than one switch in the path between the sending and the receiving host, the header for the packet needs to contain enough information to allow every switch in the path to

## Where Are They Now?

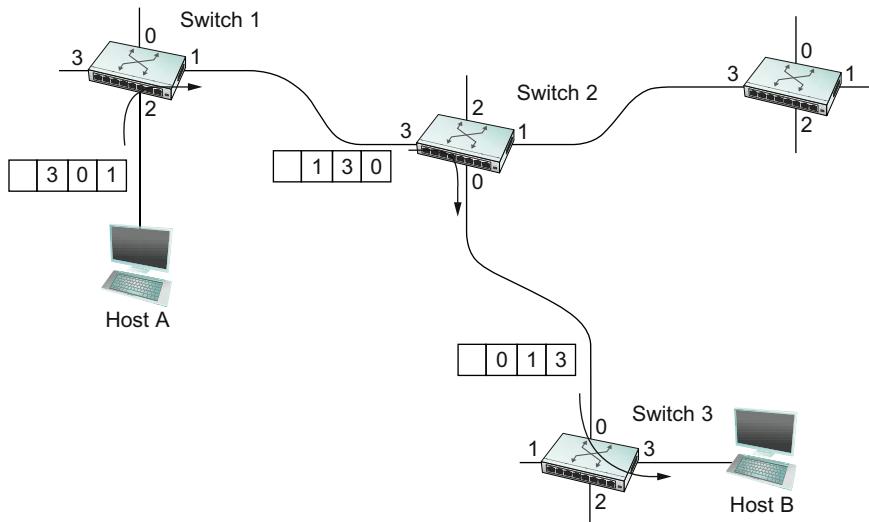
### ATM

There was a period of time in the late 1980s and early 1990s when ATM seemed (to many people) poised to take over the world. The major telecommunication companies were supporting it, and the promise of high-speed networks that could smoothly integrate voice, video, and data onto a common network seemed compelling. Proponents of ATM referred to anything that used variable-length packets—technologies such as Ethernet and IP—as “legacy” technologies. Today, however, Ethernet and IP dominate, and ATM is viewed as yesterday’s technology. You can still find pockets of ATM deployment, primarily as a way to get access to IP networks. Notably, a lot of Digital Subscriber Line (DSL) access networks were built using ATM, so some amount of broadband Internet access today is actually over ATM links, although this fact is completely hidden by the DSL modems, which take Ethernet frames and chop them into cells which are subsequently reassembled inside the access network.

There is room for debate as to why ATM didn’t take over the world. One thing that seems fundamentally important in retrospect was that IP was well on its way to becoming completely entrenched by the time ATM appeared. Even though the Internet wasn’t on the radar of a lot of people in the 1980s, it was already achieving global reach and the number of hosts connected was doubling every year. And, since the whole point of IP was to smoothly interconnect all sorts of different networks, when ATM appeared, rather than displace IP as its proponents imagined it might, ATM was quickly absorbed as just another network type over which IP could run. At that point, ATM was more directly in competition with Ethernet than with IP, and the arrival of inexpensive Ethernet switches and 100-Mbps Ethernet without expensive optics ensured that the Ethernet remained entrenched as a local area technology.

determine which output the packet needs to be placed on. One way to do this would be to put an ordered list of switch ports in the header and to rotate the list so that the next switch in the path is always at the front of the list. [Figure 3.7](#) illustrates this idea.

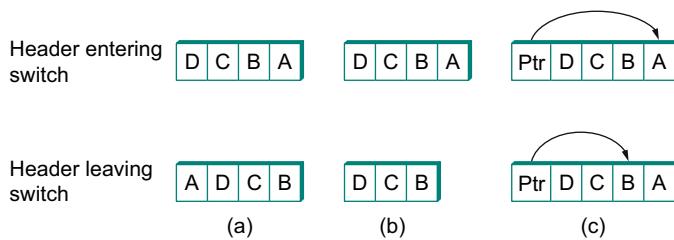
In this example, the packet needs to traverse three switches to get from host A to host B. At switch 1, it needs to exit on port 1, at the next switch it needs to exit at port 0, and at the third switch it needs to exit at port 3. Thus, the original header when the packet leaves host A contains the list of ports (3, 0, 1), where we assume that each switch reads the rightmost element of the list. To make sure that the next switch gets the appropriate information, each switch rotates the list after it has read its own entry. Thus, the packet header as it leaves switch 1 en route to switch 2 is now



■ FIGURE 3.7 Source routing in a switched network (where the switch reads the rightmost number).

(1, 3, 0); switch 2 performs another rotation and sends out a packet with (0, 1, 3) in the header. Although not shown, switch 3 performs yet another rotation, restoring the header to what it was when host A sent it.

There are several things to note about this approach. First, it assumes that host A knows enough about the topology of the network to form a header that has all the right directions in it for every switch in the path. This is somewhat analogous to the problem of building the forwarding tables in a datagram network or figuring out where to send a setup packet in a virtual circuit network. Second, observe that we cannot predict how big the header needs to be, since it must be able to hold one word of information for every switch on the path. This implies that headers are probably of variable length with no upper bound, unless we can predict with absolute certainty the maximum number of switches through which a packet will ever need to pass. Third, there are some variations on this approach. For example, rather than rotate the header, each switch could just strip the first element as it uses it. Rotation has an advantage over stripping, however: Host B gets a copy of the complete header, which may help it figure out how to get back to host A. Yet another alternative is to have the header carry a pointer to the current “next port” entry, so that each switch just updates the pointer rather than rotating the header; this may be more efficient to implement. We show these three approaches in



■ **FIGURE 3.8** Three ways to handle headers for source routing: (a) rotation; (b) stripping; (c) pointer. The labels are read right to left.

**Figure 3.8.** In each case, the entry that this switch needs to read is A, and the entry that the next switch needs to read is B.

Source routing can be used in both datagram networks and virtual circuit networks. For example, the Internet Protocol, which is a datagram protocol, includes a source route option that allows selected packets to be source routed, while the majority are switched as conventional datagrams. Source routing is also used in some virtual circuit networks as the means to get the initial setup request along the path from source to destination.

Source routes are sometimes categorized as “strict” or “loose.” In a strict source route, every node along the path must be specified, whereas a loose source route only specifies a set of nodes to be traversed, without saying exactly how to get from one node to the next. A loose source route can be thought of as a set of waypoints rather than a completely specified route. The loose option can be helpful to limit the amount of information that a source must obtain to create a source route. In any reasonably large network, it is likely to be hard for a host to get the complete path information it needs to construct a correct strict source route to any destination. But both types of source routes do find application in certain scenarios, one of which is described in [Section 4.3](#).

### 3.1.4 Bridges and LAN Switches

Having discussed some of the basic ideas behind switching, we now focus more closely on some specific switching technologies. We begin by considering a class of switch that is used to forward packets between LANs (local area networks) such as Ethernets. Such switches are sometimes known by the obvious name of LAN switches; historically, they have also



been referred to as *bridges*, and they are very widely used in campus and enterprise networks.

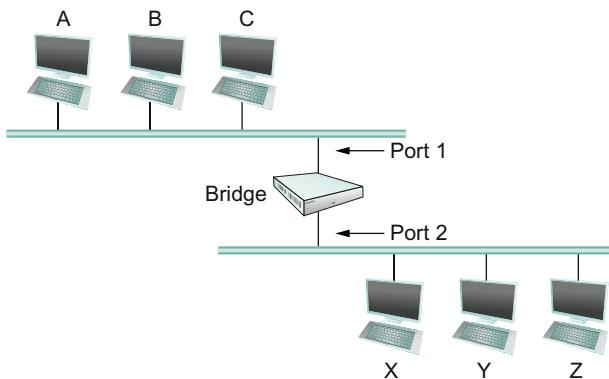
Suppose you have a pair of Ethernets that you want to interconnect. One approach you might try is to put a repeater between them, as described in Chapter 2. This would not be a workable solution, however, if doing so exceeded the physical limitations of the Ethernet. (Recall that no more than two repeaters between any pair of hosts and no more than a total of 2500 m in length are allowed.) An alternative would be to put a node with a pair of Ethernet adaptors between the two Ethernets and have the node forward frames from one Ethernet to the other. This node would differ from a repeater, which operates on bits, not frames, and just blindly copies the bits received on one interface to another. Instead, this node would fully implement the Ethernet's collision detection and media access protocols on each interface. Hence, the length and number-of-host restrictions of the Ethernet, which are all about managing collisions, would not apply to the combined pair of Ethernets connected in this way. This device operates in promiscuous mode, accepting all frames transmitted on either of the Ethernets, and forwarding them to the other.

The node we have just described is typically called a *bridge*, and a collection of LANs connected by one or more bridges is usually said to form an *extended LAN*. In their simplest variants, bridges simply accept LAN frames on their inputs and forward them out on all other outputs. This simple strategy was used by early bridges but has some pretty serious limitations as we'll see below. A number of refinements have been added over the years to make bridges an effective mechanism for interconnecting a set of LANs. The rest of this section fills in the more interesting details.

Note that a bridge meets our definition of a switch from the previous section: a multi-input, multi-output device, which transfers packets from an input to one or more outputs. And recall that this provides a way to increase the total bandwidth of a network. For example, while a single Ethernet segment might carry only 100 Mbps of total traffic, an Ethernet bridge can carry as much as  $100n$  Mbps, where  $n$  is the number of ports (inputs and outputs) on the bridge.

#### *Learning Bridges*

The first optimization we can make to a bridge is to observe that it need not forward all frames that it receives. Consider the bridge in Figure 3.9. Whenever a frame from host A that is addressed to host B arrives on port 1,



■ FIGURE 3.9 Illustration of a learning bridge.

**Table 3.4 Forwarding Table Maintained by a Bridge**

Host	Port
A	1
B	1
C	1
X	2
Y	2
Z	2

there is no need for the bridge to forward the frame out over port 2. The question, then, is how does a bridge come to learn on which port the various hosts reside?

One option would be to have a human download a table into the bridge similar to the one given in Table 3.4. Then, whenever the bridge receives a frame on port 1 that is addressed to host A, it would not forward the frame out on port 2; there would be no need because host A would have already directly received the frame on the LAN connected to port 1. Anytime a frame addressed to host A was received on port 2, the bridge would forward the frame out on port 1.

No one actually builds bridges in which the table is configured by hand. Having a human maintain this table is too burdensome, and there is a

simple trick by which a bridge can learn this information for itself. The idea is for each bridge to inspect the *source* address in all the frames it receives. Thus, when host A sends a frame to a host on either side of the bridge, the bridge receives this frame and records the fact that a frame from host A was just received on port 1. In this way, the bridge can build a table just like Table 3.4.

Note that a bridge using such a table implements a version of the datagram (or connectionless) model of forwarding described in Section 3.1.1. Each packet carries a global address, and the bridge decides which output to send a packet on by looking up that address in a table.

When a bridge first boots, this table is empty; entries are added over time. Also, a timeout is associated with each entry, and the bridge discards the entry after a specified period of time. This is to protect against the situation in which a host—and, as a consequence, its LAN address—is moved from one network to another. Thus, this table is not necessarily complete. Should the bridge receive a frame that is addressed to a host not currently in the table, it goes ahead and forwards the frame out on all the other ports. In other words, this table is simply an optimization that filters out some frames; it is not required for correctness.

#### *Implementation*

The code that implements the learning bridge algorithm is quite simple, and we sketch it here. Structure `BridgeEntry` defines a single entry in the bridge's forwarding table; these are stored in a `Map` structure (which supports `mapCreate`, `mapBind`, and `mapResolve` operations) to enable entries to be efficiently located when packets arrive from sources already in the table. The constant `MAX_TTL` specifies how long an entry is kept in the table before it is discarded.

```
#define BRIDGE_TAB_SIZE    1024 /* max. size of bridging
                                table */
#define MAX_TTL                120  /* time (in seconds) before
                                an entry is flushed */

typedef struct {
    MacAddr      destination;    /* MAC address of a node */
    int          ifnumber;        /* interface to reach it */
    u_short      TTL;            /* time to live */
    Binding      binding;        /* binding in the Map */
```

```
} BridgeEntry;

int      numEntries = 0;
Map     bridgeMap = mapCreate(BRIDGE_TAB_SIZE,
                           sizeof(BridgeEntry));
```

The routine that updates the forwarding table when a new packet arrives is given by `updateTable`. The arguments passed are the source media access control (MAC) address contained in the packet and the interface number on which it was received. Another routine, not shown here, is invoked at regular intervals, scans the entries in the forwarding table, and decrements the TTL (time to live) field of each entry, discarding any entries whose TTL has reached 0. Note that the TTL is reset to `MAX_TTL` every time a packet arrives to refresh an existing table entry and that the interface on which the destination can be reached is updated to reflect the most recently received packet.

```
void
updateTable (MacAddr src, int inif)
{
    BridgeEntry      *b;

    if (mapResolve(bridgeMap, &src, (void **)&b) == FALSE )
    {
        /* this address is not in the table, so try to add it */
        if (numEntries < BRIDGE_TAB_SIZE)
        {
            b = NEW(BridgeEntry);
            b->binding = mapBind( bridgeMap, &src, b );
            /* use source address of packet as dest. address in table */
            b->destination = src;
            numEntries++;
        }
        else
        {
            /* can't fit this address in the table now, so give up */
            return;
        }
    }
}
```

```

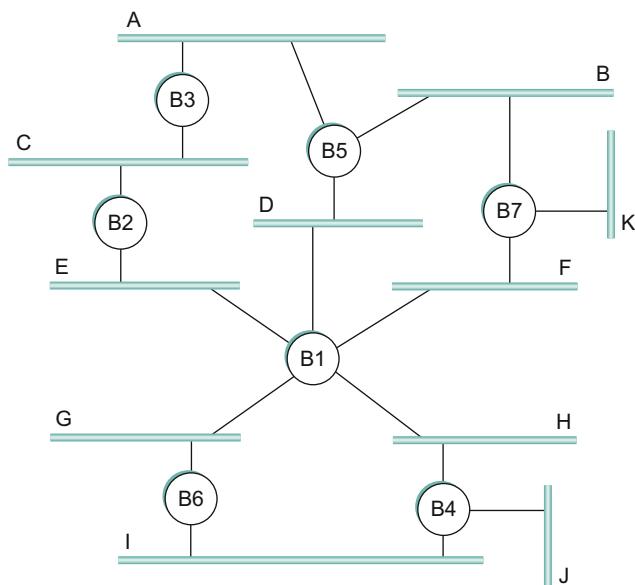
/* reset TTL and use most recent input interface */
b->TTL = MAX_TTL;
b->ifnumber = inif;
}

```

Note that this implementation adopts a simple strategy in the case where the bridge table has become full to capacity—it simply fails to add the new address. Recall that completeness of the bridge table is not necessary for correct forwarding; it just optimizes performance. If there is some entry in the table that is not currently being used, it will eventually time out and be removed, creating space for a new entry. An alternative approach would be to invoke some sort of cache replacement algorithm on finding the table full; for example, we might locate and remove the entry with the smallest TTL to accommodate the new entry.

#### *Spanning Tree Algorithm*

The preceding strategy works just fine until the extended LAN has a loop in it, in which case it fails in a horrible way—frames potentially loop through the extended LAN forever. This is easy to see in the example depicted in Figure 3.10, where, for example, bridges B1, B4, and B6 form

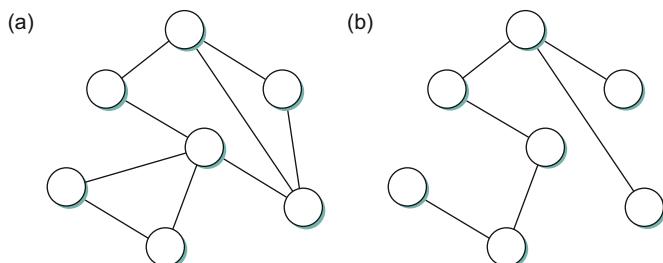


■ FIGURE 3.10 Extended LAN with loops.

a loop. Suppose that a packet enters bridge B4 from Ethernet J and that the destination address is one not yet in any bridge's forwarding table: B4 sends a copy of the packet out to Ethernets H and I. Now bridge B6 forwards the packet to Ethernet G, where B1 would see it and forward it back to Ethernet H; B4 still doesn't have this destination in its table, so it forwards the packet back to Ethernets I and J. There is nothing to stop this cycle from repeating endlessly, with packets looping in both directions among B1, B4, and B6.

Why would an extended LAN come to have a loop in it? One possibility is that the network is managed by more than one administrator, for example, because it spans multiple departments in an organization. In such a setting, it is possible that no single person knows the entire configuration of the network, meaning that a bridge that closes a loop might be added without anyone knowing. A second, more likely scenario is that loops are built into the network on purpose—to provide redundancy in case of failure. After all, a network with no loops needs only one link failure to become split into two separate partitions.

Whatever the cause, bridges must be able to correctly handle loops. This problem is addressed by having the bridges run a distributed *spanning tree* algorithm. If you think of the extended LAN as being represented by a graph that possibly has loops (cycles), then a spanning tree is a subgraph of this graph that covers (spans) all the vertices but contains no cycles. That is, a spanning tree keeps all of the vertices of the original graph but throws out some of the edges. For example, Figure 3.11 shows a cyclic graph on the left and one of possibly many spanning trees on the right.



■ FIGURE 3.11 Example of (a) a cyclic graph; (b) a corresponding spanning tree.

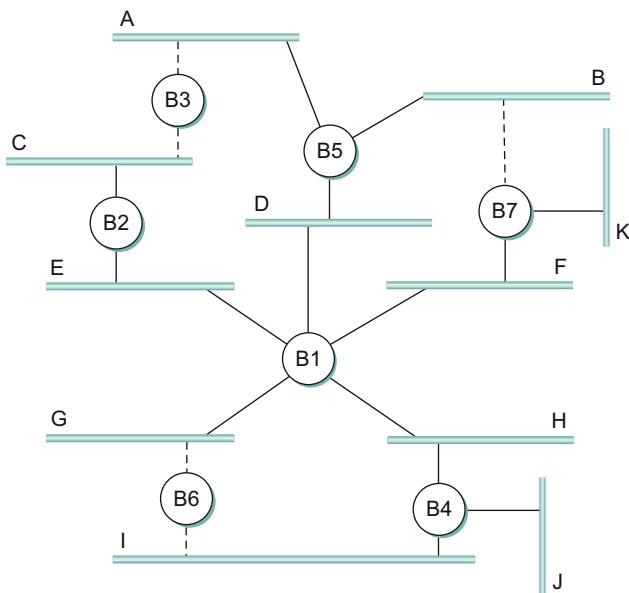
The idea of a spanning tree is simple enough: It's a subset of the actual network topology that has no loops and that reaches all the LANs in the extended LAN. The hard part is how all of the bridges coordinate their decisions to arrive at a single view of the spanning tree. After all, one topology is typically able to be covered by multiple spanning trees. The answer lies in the spanning tree protocol, which we'll describe now.

The spanning tree algorithm, which was developed by Radia Perlman at the Digital Equipment Corporation, is a protocol used by a set of bridges to agree upon a spanning tree for a particular extended LAN. (The IEEE 802.1 specification for LAN bridges is based on this algorithm.) In practice, this means that each bridge decides the ports over which it is and is not willing to forward frames. In a sense, it is by removing ports from the topology that the extended LAN is reduced to an acyclic tree.<sup>4</sup> It is even possible that an entire bridge will not participate in forwarding frames, which seems kind of strange at first glance. The algorithm is dynamic, however, meaning that the bridges are always prepared to reconfigure themselves into a new spanning tree should some bridge fail, and so those unused ports and bridges provide the redundant capacity needed to recover from failures.

The main idea of the spanning tree is for the bridges to select the ports over which they will forward frames. The algorithm selects ports as follows. Each bridge has a unique identifier; for our purposes, we use the labels B1, B2, B3, and so on. The algorithm first elects the bridge with the smallest ID as the root of the spanning tree; exactly how this election takes place is described below. The root bridge always forwards frames out over all of its ports. Next, each bridge computes the shortest path to the root and notes which of its ports is on this path. This port is also selected as the bridge's preferred path to the root. Finally, all the bridges connected to a given LAN elect a single *designated* bridge that will be responsible for forwarding frames toward the root bridge. Each LAN's designated bridge is the one that is closest to the root. If two or more bridges are equally close to the root, then the bridges' identifiers are used to break ties, and

---

<sup>4</sup>Representing an extended LAN as an abstract graph is a bit awkward. Basically, you let both the bridges and the LANs correspond to the vertices of the graph and the ports correspond to the graph's edges. However, the spanning tree we are going to compute for this graph needs to span only those nodes that correspond to networks. It is possible that nodes corresponding to bridges will be disconnected from the rest of the graph. This corresponds to a situation in which all the ports connecting a bridge to various networks get removed by the algorithm.



■ FIGURE 3.12 Spanning tree with some ports not selected.

the smallest ID wins. Of course, each bridge is connected to more than one LAN, so it participates in the election of a designated bridge for each LAN it is connected to. In effect, this means that each bridge decides if it is the designated bridge relative to each of its ports. The bridge forwards frames over those ports for which it is the designated bridge.

Figure 3.12 shows the spanning tree that corresponds to the extended LAN shown in Figure 3.10. In this example, B1 is the root bridge, since it has the smallest ID. Notice that both B3 and B5 are connected to LAN A, but B5 is the designated bridge since it is closer to the root. Similarly, both B5 and B7 are connected to LAN B, but in this case B5 is the designated bridge since it has the smaller ID; both are an equal distance from B1.

While it is possible for a human to look at the extended LAN given in Figure 3.10 and to compute the spanning tree given in Figure 3.12 according to the rules given above, the bridges in an extended LAN do not have the luxury of being able to see the topology of the entire network, let alone peek inside other bridges to see their ID. Instead, the bridges have to exchange configuration messages with each other and then decide

whether or not they are the root or a designated bridge based on these messages.

Specifically, the configuration messages contain three pieces of information:

1. The ID for the bridge that is sending the message
2. The ID for what the sending bridge believes to be the root bridge
3. The distance, measured in hops, from the sending bridge to the root bridge

Each bridge records the current *best* configuration message it has seen on each of its ports (“best” is defined below), including both messages it has received from other bridges and messages that it has itself transmitted.

Initially, each bridge thinks it is the root, and so it sends a configuration message out on each of its ports identifying itself as the root and giving a distance to the root of 0. Upon receiving a configuration message over a particular port, the bridge checks to see if that new message is better than the current best configuration message recorded for that port. The new configuration message is considered *better* than the currently recorded information if any of the following is true:

- It identifies a root with a smaller ID.
- It identifies a root with an equal ID but with a shorter distance.
- The root ID and distance are equal, but the sending bridge has a smaller ID

If the new message is better than the currently recorded information, the bridge discards the old information and saves the new information. However, it first adds 1 to the distance-to-root field since the bridge is one hop farther away from the root than the bridge that sent the message.

When a bridge receives a configuration message indicating that it is not the root bridge—that is, a message from a bridge with a smaller ID—the bridge stops generating configuration messages on its own and instead only forwards configuration messages from other bridges, after first adding 1 to the distance field. Likewise, when a bridge receives a configuration message that indicates it is not the designated bridge for that port—that is, a message from a bridge that is closer to the root or equally far from the root but with a smaller ID—the bridge stops sending

configuration messages over that port. Thus, when the system stabilizes, only the root bridge is still generating configuration messages, and the other bridges are forwarding these messages only over ports for which they are the designated bridge. At this point, a spanning tree has been built, and all the bridges are in agreement on which ports are in use for the spanning tree. Only those ports may be used for forwarding data packets in the extended LAN.

Let's see how this works with an example. Consider what would happen in Figure 3.12 if the power had just been restored to the building housing this network, so that all the bridges boot at about the same time. All the bridges would start off by claiming to be the root. We denote a configuration message from node  $X$  in which it claims to be distance  $d$  from root node  $Y$  as  $(Y, d, X)$ . Focusing on the activity at node B3, a sequence of events would unfold as follows:

1. B3 receives  $(B2, 0, B2)$ .
2. Since  $2 < 3$ , B3 accepts B2 as root.
3. B3 adds one to the distance advertised by B2 (0) and thus sends  $(B2, 1, B3)$  toward B5.
4. Meanwhile, B2 accepts B1 as root because it has the lower ID, and it sends  $(B1, 1, B2)$  toward B3.
5. B5 accepts B1 as root and sends  $(B1, 1, B5)$  toward B3.
6. B3 accepts B1 as root, and it notes that both B2 and B5 are closer to the root than it is; thus, B3 stops forwarding messages on both its interfaces.

This leaves B3 with both ports not selected, as shown in Figure 3.12.

Even after the system has stabilized, the root bridge continues to send configuration messages periodically, and the other bridges continue to forward these messages as described in the previous paragraph. Should a particular bridge fail, the downstream bridges will not receive these configuration messages, and after waiting a specified period of time they will once again claim to be the root, and the algorithm just described will kick in again to elect a new root and new designated bridges.

One important thing to notice is that although the algorithm is able to reconfigure the spanning tree whenever a bridge fails, it is not able to forward frames over alternative paths for the sake of routing around a congested bridge.

### *Broadcast and Multicast*

The preceding discussion has focused on how bridges forward unicast frames from one LAN to another. Since the goal of a bridge is to transparently extend a LAN across multiple networks, and since most LANs support both broadcast and multicast, then bridges must also support these two features. Broadcast is simple—each bridge forwards a frame with a destination broadcast address out on each active (selected) port other than the one on which the frame was received.

Multicast can be implemented in exactly the same way, with each host deciding for itself whether or not to accept the message. This is exactly what is done in practice. Notice, however, that since not all the LANs in an extended LAN necessarily have a host that is a member of a particular multicast group, it is possible to do better. Specifically, the spanning tree algorithm can be extended to prune networks over which multicast frames need not be forwarded. Consider a frame sent to group M by a host on LAN A in Figure 3.12. If there is no host on LAN J that belongs to group M, then there is no need for bridge B4 to forward the frames over that network. On the other hand, not having a host on LAN H that belongs to group M does not necessarily mean that bridge B1 can avoid forwarding multicast frames onto LAN H. It all depends on whether or not there are members of group M on LANs I and J.

How does a given bridge learn whether it should forward a multicast frame over a given port? It learns exactly the same way that a bridge learns whether it should forward a unicast frame over a particular port—by observing the *source* addresses that it receives over that port. Of course, groups are not typically the source of frames, so we have to cheat a little. In particular, each host that is a member of group M must periodically send a frame with the address for group M in the source field of the frame header. This frame would have as its destination address the multicast address for the bridges.

Note that, although the multicast extension just described has been proposed, it is not widely adopted. Instead, multicast is implemented in exactly the same way as broadcast on today's extended LANs.

### *Limitations of Bridges*

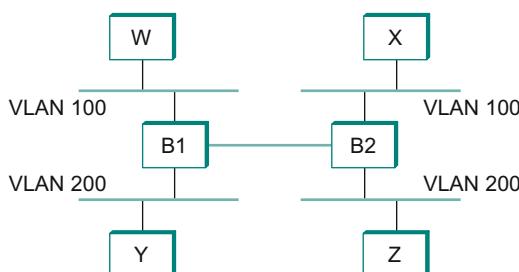
The bridge-based solution just described is meant to be used in only a fairly limited setting—to connect a handful of similar LANs. The main

limitations of bridges become apparent when we consider the issues of scale and heterogeneity.

On the issue of scale, it is not realistic to connect more than a few LANs by means of bridges, where in practice *few* typically means “tens of.” One reason for this is that the spanning tree algorithm scales linearly; that is, there is no provision for imposing a hierarchy on the extended LAN. A second reason is that bridges forward all broadcast frames. While it is reasonable for all hosts within a limited setting (say, a department) to see each other’s broadcast messages, it is unlikely that all the hosts in a larger environment (say, a large company or university) would want to have to be bothered by each other’s broadcast messages. Said another way, broadcast does not scale, and as a consequence extended LANs do not scale.

One approach to increasing the scalability of extended LANs is the *virtual LAN* (VLAN). VLANs allow a single extended LAN to be partitioned into several seemingly separate LANs. Each virtual LAN is assigned an identifier (sometimes called a *color*), and packets can only travel from one segment to another if both segments have the same identifier. This has the effect of limiting the number of segments in an extended LAN that will receive any given broadcast packet.

We can see how VLANs work with an example. Figure 3.13 shows four hosts on four different LAN segments. In the absence of VLANs, any broadcast packet from any host will reach all the other hosts. Now let’s suppose that we define the segments connected to hosts W and X as being in one VLAN, which we’ll call VLAN 100. We also define the segments that connect to hosts Y and Z as being in VLAN 200. To do this, we need to



■ FIGURE 3.13 Two virtual LANs share a common backbone.

configure a VLAN ID on each port of bridges B1 and B2. The link between B1 and B2 is considered to be in both VLANs.

When a packet sent by host X arrives at bridge B2, the bridge observes that it came in a port that was configured as being in VLAN 100. It inserts a VLAN header between the Ethernet header and its payload. The interesting part of the VLAN header is the VLAN ID; in this case, that ID is set to 100. The bridge now applies its normal rules for forwarding to the packet, with the extra restriction that the packet may not be sent out an interface that is not part of VLAN 100. Thus, under no circumstances will the packet—even a broadcast packet—be sent out the interface to host Z, which is in VLAN 200. The packet is, however, forwarded on to bridge B1, which follows the same rules and thus may forward the packet to host W but not to host Y.

An attractive feature of VLANs is that it is possible to change the logical topology without moving any wires or changing any addresses. For example, if we wanted to make the segment that connects to host Z be part of VLAN 100 and thus enable X, W, and Z to be on the same virtual LAN, then we would just need to change one piece of configuration on bridge B2.

On the issue of heterogeneity, bridges are fairly limited in the kinds of networks they can interconnect. In particular, bridges make use of the network's frame header and so can support only networks that have exactly the same format for addresses. Thus, bridges can be used to connect Ethernets to Ethernets, token rings to token rings, and one 802.11 network to another. It's also possible to put a bridge between, say, an Ethernet and an 802.11 network, since both networks support the same 48-bit address format. However, bridges do not readily generalize to other kinds of networks with different addressing formats, such as ATM.<sup>5</sup>

Despite their limitations, bridges are a very important part of the complete networking picture. Their main advantage is that they allow multiple LANs to be transparently connected; that is, the networks can be connected without the end hosts having to run any additional protocols (or even be aware, for that matter). The one potential exception is when the hosts are expected to announce their membership in a multicast group, as described in Section 3.1.4.

---

<sup>5</sup>Ultimately there was quite a lot of work done to make ATM networks behave more like Ethernets, called *LAN Emulation*, to get around this limitation, but this is rarely seen today.

Notice, however, that this transparency can be dangerous. If a host or, more precisely, the application and transport protocol running on that host is programmed under the assumption that it is running on a single LAN, then inserting bridges between the source and destination hosts can have unexpected consequences. For example, if a bridge becomes congested, it may have to drop frames; in contrast, it is rare that a single Ethernet ever drops a frame. As another example, the latency between any pair of hosts on an extended LAN becomes both larger and more highly variable; in contrast, the physical limitations of a single Ethernet make the latency both small and predictable. As a final example, it is possible (although unlikely) that frames will be reordered in an extended LAN; in contrast, frame order is never shuffled on a single Ethernet. The bottom line is that it is never safe to design network software under the assumption that it will run over a single Ethernet segment. Bridges happen.



## 3.2 BASIC INTERNETWORKING (IP)

In the previous section, we saw that it was possible to build reasonably large LANs using bridges and LAN switches, but that such approaches were limited in their ability to scale and to handle heterogeneity. In this section, we explore some ways to go beyond the limitations of bridged networks, enabling us to build large, highly heterogeneous networks with reasonably efficient routing. We refer to such networks as *internetworks*. We'll continue the discussion of how to build a truly global internetwork in the next chapter, but for now we'll explore the basics. We start by considering more carefully what the word *internetwork* means.

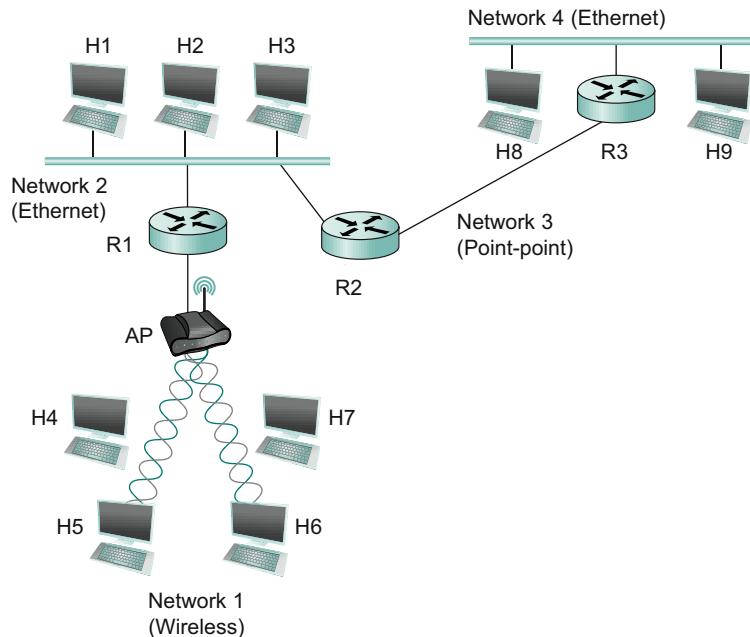
### 3.2.1 What Is an Internetwork?

We use the term *internetwork*, or sometimes just *internet* with a lowercase *i*, to refer to an arbitrary collection of networks interconnected to provide some sort of host-to-host packet delivery service. For example, a corporation with many sites might construct a private internetwork by interconnecting the LANs at their different sites with point-to-point links leased from the phone company. When we are talking about the widely used global internetwork to which a large percentage of networks are now connected, we call it the *Internet* with a capital *I*. In keeping with the first-principles approach of this book, we mainly want you to learn about the

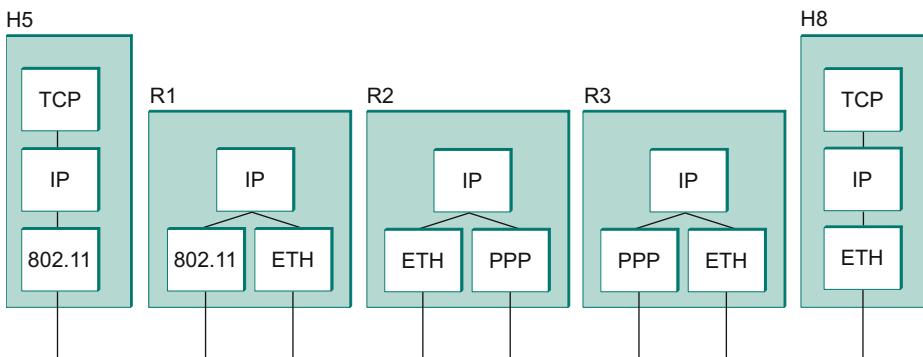
principles of “lowercase *i*” internetworking, but we illustrate these ideas with real-world examples from the “big *I*” Internet.

Another piece of terminology that can be confusing is the difference between networks, subnetworks, and internetworks. We are going to avoid subnetworks (or subnets) altogether until Section 3.2.5. For now, we use *network* to mean either a directly connected or a switched network of the kind described in the previous section and the previous chapter. Such a network uses one technology, such as 802.11 or Ethernet. An *internetwork* is an interconnected collection of such networks. Sometimes, to avoid ambiguity, we refer to the underlying networks that we are interconnecting as *physical* networks. An *internet* is a *logical* network built out of a collection of physical networks. In this context, a collection of Ethernets connected by bridges or switches would still be viewed as a single network.

Figure 3.14 shows an example internetwork. An internetwork is often referred to as a “network of networks” because it is made up of lots of smaller networks. In this figure, we see Ethernets, a wireless network,



■ FIGURE 3.14 A simple internetwork.  $Hn$  = host;  $Rn$  = router.



■ **FIGURE 3.15** A simple internetwork, showing the protocol layers used to connect H5 to H8 in Figure 3.14. ETH is the protocol that runs over the Ethernet.

and a point-to-point link. Each of these is a single-technology network. The nodes that interconnect the networks are called *routers*. They are also sometimes called *gateways*, but since this term has several other connotations, we restrict our usage to router.

The *Internet Protocol* is the key tool used today to build scalable, heterogeneous internetworks. It was originally known as the Kahn-Cerf protocol after its inventors.<sup>6</sup> One way to think of IP is that it runs on all the nodes (both hosts and routers) in a collection of networks and defines the infrastructure that allows these nodes and networks to function as a single logical internetwork. For example, Figure 3.15 shows how hosts H5 and H8 are logically connected by the internet in Figure 3.14, including the protocol graph running on each node. Note that higher-level protocols, such as TCP and UDP, typically run on top of IP on the hosts.

Most of the rest of this chapter is about various aspects of IP. While it is certainly possible to build an internetwork that does not use IP—for example, Novell created an internetworking protocol called IPX, which was in turn based on the XNS internet designed by Xerox—IP is the most interesting case to study simply because of the size of the Internet. Said another way, it is only the IP Internet that has really faced the issue of scale. Thus, it provides the best case study of a scalable internetworking protocol.

<sup>6</sup>Robert Kahn and Vint Cerf received the A.M. Turing award, often referred to as the Nobel Prize of computer science, in 2005 for their design of IP.

### 3.2.2 Service Model

A good place to start when you build an internetwork is to define its *service model*, that is, the host-to-host services you want to provide. The main concern in defining a service model for an internetwork is that we can provide a host-to-host service only if this service can somehow be provided over each of the underlying physical networks. For example, it would be no good deciding that our internetwork service model was going to provide guaranteed delivery of every packet in 1 ms or less if there were underlying network technologies that could arbitrarily delay packets. The philosophy used in defining the IP service model, therefore, was to make it undemanding enough that just about any network technology that might turn up in an internetwork would be able to provide the necessary service.

The IP service model can be thought of as having two parts: an addressing scheme, which provides a way to identify all hosts in the internetwork, and a datagram (connectionless) model of data delivery. This service model is sometimes called *best effort* because, although IP makes every effort to deliver datagrams, it makes no guarantees. We postpone a discussion of the addressing scheme for now and look first at the data delivery model.

#### Datagram Delivery

The IP datagram is fundamental to the Internet Protocol. Recall from Section 3.1.1 that a datagram is a type of packet that happens to be sent in a connectionless manner over a network. Every datagram carries enough information to let the network forward the packet to its correct destination; there is no need for any advance setup mechanism to tell the network what to do when the packet arrives. You just send it, and the network makes its best effort to get it to the desired destination. The “best-effort” part means that if something goes wrong and the packet gets lost, corrupted, misdelivered, or in any way fails to reach its intended destination, the network does nothing—it made its best effort, and that is all it has to do. It does not make any attempt to recover from the failure. This is sometimes called an *unreliable* service.

Best-effort, connectionless service is about the simplest service you could ask for from an internetwork, and this is a great strength. For example, if you provide best-effort service over a network that provides a reliable service, then that’s fine—you end up with a best-effort service that just happens to always deliver the packets. If, on the other hand, you had a reliable service model over an unreliable network, you would

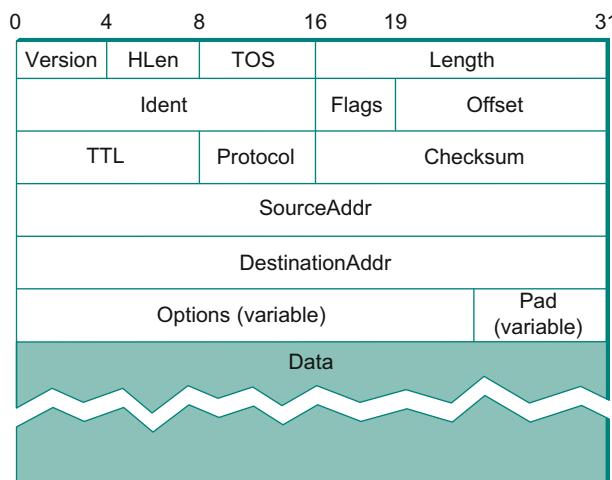
have to put lots of extra functionality into the routers to make up for the deficiencies of the underlying network. Keeping the routers as simple as possible was one of the original design goals of IP.

The ability of IP to “run over anything” is frequently cited as one of its most important characteristics. It is noteworthy that many of the technologies over which IP runs today did not exist when IP was invented. So far, no networking technology has been invented that has proven too bizarre for IP; it has even been claimed that IP can run over a network that transports messages using carrier pigeons.

Best-effort delivery does not just mean that packets can get lost. Sometimes they can get delivered out of order, and sometimes the same packet can get delivered more than once. The higher-level protocols or applications that run above IP need to be aware of all these possible failure modes.

#### Packet Format

Clearly, a key part of the IP service model is the type of packets that can be carried. The IP datagram, like most packets, consists of a header followed by a number of bytes of data. The format of the header is shown in Figure 3.16. Note that we have adopted a different style of representing packets than the one we used in previous chapters. This is because packet formats at the internetworking layer and above, where we will be focusing our attention



■ FIGURE 3.16 IPv4 packet header.

for the next few chapters, are almost invariably designed to align on 32-bit boundaries to simplify the task of processing them in software. Thus, the common way of representing them (used in Internet Requests for Comments, for example) is to draw them as a succession of 32-bit words. The top word is the one transmitted first, and the leftmost byte of each word is the one transmitted first. In this representation, you can easily recognize fields that are a multiple of 8 bits long. On the odd occasion when fields are not an even multiple of 8 bits, you can determine the field lengths by looking at the bit positions marked at the top of the packet.

Looking at each field in the IP header, we see that the “simple” model of best-effort datagram delivery still has some subtle features. The Version field specifies the version of IP. The current version of IP is 4, and it is sometimes called *IPv4*.<sup>7</sup> Observe that putting this field right at the start of the datagram makes it easy for everything else in the packet format to be redefined in subsequent versions; the header processing software starts off by looking at the version and then branches off to process the rest of the packet according to the appropriate format. The next field, HLen, specifies the length of the header in 32-bit words. When there are no options, which is most of the time, the header is 5 words (20 bytes) long. The 8-bit TOS (type of service) field has had a number of different definitions over the years, but its basic function is to allow packets to be treated differently based on application needs. For example, the TOS value might determine whether or not a packet should be placed in a special queue that receives low delay. We discuss the use of this field (which has evolved somewhat over the years) in more detail in Sections 6.4.2 and 6.5.3.

The next 16 bits of the header contain the Length of the datagram, including the header. Unlike the HLen field, the Length field counts bytes rather than words. Thus, the maximum size of an IP datagram is 65,535 bytes. The physical network over which IP is running, however, may not support such long packets. For this reason, IP supports a fragmentation and reassembly process. The second word of the header contains information about fragmentation, and the details of its use are presented in the following section entitled “Fragmentation and Reassembly.”

Moving on to the third word of the header, the next byte is the TTL (time to live) field. Its name reflects its historical meaning rather than the

---

<sup>7</sup>The next major version of IP, which is discussed in Chapter 4, has the new version number 6 and is known as IPv6. The version number 5 was used for an experimental protocol called ST-II that was not widely used.

way it is commonly used today. The intent of the field is to catch packets that have been going around in routing loops and discard them, rather than let them consume resources indefinitely. Originally, TTL was set to a specific number of seconds that the packet would be allowed to live, and routers along the path would decrement this field until it reached 0. However, since it was rare for a packet to sit for as long as 1 second in a router, and routers did not all have access to a common clock, most routers just decremented the TTL by 1 as they forwarded the packet. Thus, it became more of a hop count than a timer, which is still a perfectly good way to catch packets that are stuck in routing loops. One subtlety is in the initial setting of this field by the sending host: Set it too high and packets could circulate rather a lot before getting dropped; set it too low and they may not reach their destination. The value 64 is the current default.

The Protocol field is simply a demultiplexing key that identifies the higher-level protocol to which this IP packet should be passed. There are values defined for the TCP (Transmission Control Protocol—6), UDP (User Datagram Protocol—17), and many other protocols that may sit above IP in the protocol graph.

The Checksum is calculated by considering the entire IP header as a sequence of 16-bit words, adding them up using ones complement arithmetic, and taking the ones complement of the result. This is the IP checksum algorithm described in [Section 2.4](#). Thus, if any bit in the header is corrupted in transit, the checksum will not contain the correct value upon receipt of the packet. Since a corrupted header may contain an error in the destination address—and, as a result, may have been misdelivered—it makes sense to discard any packet that fails the checksum. It should be noted that this type of checksum does not have the same strong error detection properties as a CRC, but it is much easier to calculate in software.

The last two required fields in the header are the SourceAddr and the DestinationAddr for the packet. The latter is the key to datagram delivery: Every packet contains a full address for its intended destination so that forwarding decisions can be made at each router. The source address is required to allow recipients to decide if they want to accept the packet and to enable them to reply. IP addresses are discussed in [Section 3.2.3](#)—for now, the important thing to know is that IP defines its own global address space, independent of whatever physical networks it runs over. As we will see, this is one of the keys to supporting heterogeneity.

Finally, there may be a number of options at the end of the header. The presence or absence of options may be determined by examining the header length (HLen) field. While options are used fairly rarely, a complete IP implementation must handle them all.

#### *Fragmentation and Reassembly*

One of the problems of providing a uniform host-to-host service model over a heterogeneous collection of networks is that each network technology tends to have its own idea of how large a packet can be. For example, an Ethernet can accept packets up to 1500 bytes long, while FDDI (Fiber Distributed Data Interface) packets may be 4500 bytes long. This leaves two choices for the IP service model: Make sure that all IP datagrams are small enough to fit inside one packet on any network technology, or provide a means by which packets can be fragmented and reassembled when they are too big to go over a given network technology. The latter turns out to be a good choice, especially when you consider the fact that new network technologies are always turning up, and IP needs to run over all of them; this would make it hard to pick a suitably small bound on datagram size. This also means that a host will not send needlessly small packets, which wastes bandwidth and consumes processing resources by requiring more headers per byte of data sent. For example, two hosts connected to FDDI networks that are interconnected by a point-to-point link would not need to send packets small enough to fit on an Ethernet.

The central idea here is that every network type has a *maximum transmission unit* (MTU), which is the largest IP datagram that it can carry in a frame. Note that this value is smaller than the largest packet size on that network because the IP datagram needs to fit in the *payload* of the link-layer frame.<sup>8</sup>

When a host sends an IP datagram, therefore, it can choose any size that it wants. A reasonable choice is the MTU of the network to which the host is directly attached. Then, fragmentation will only be necessary if the path to the destination includes a network with a smaller MTU. Should the transport protocol that sits on top of IP give IP a packet larger than the local MTU, however, then the source host must fragment it.

Fragmentation typically occurs in a router when it receives a datagram that it wants to forward over a network that has an MTU that is smaller

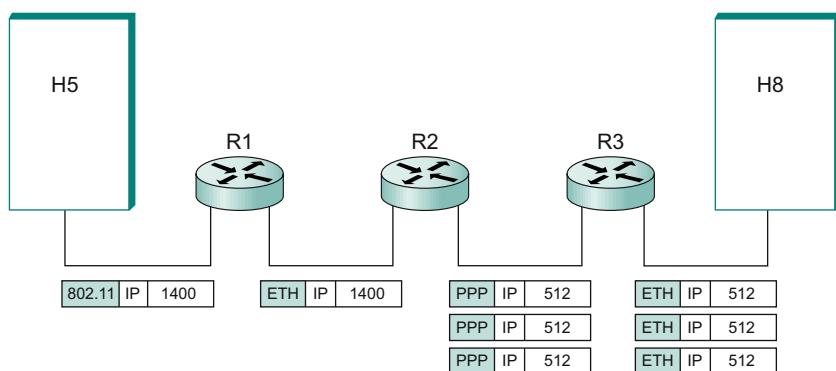
---

<sup>8</sup>In ATM networks, the MTU is, fortunately, much larger than a single cell, as ATM has its own fragmentation mechanisms. The link-layer frame in ATM is called a *convergence-sublayer protocol data unit* (CS-PDU).

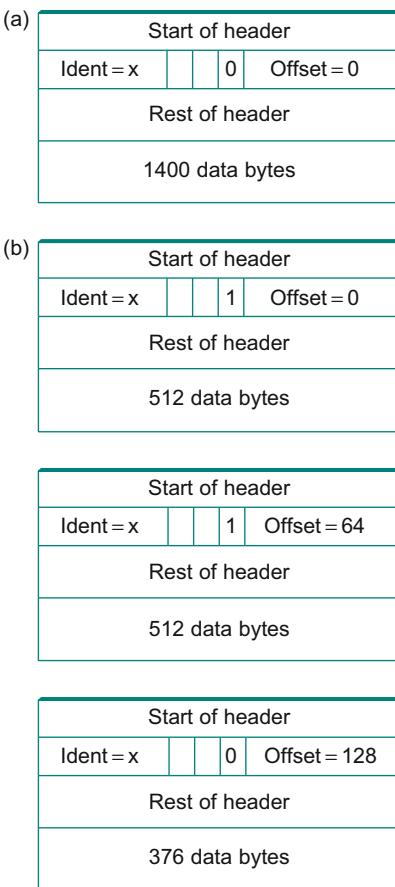
than the received datagram. To enable these fragments to be reassembled at the receiving host, they all carry the same identifier in the Ident field. This identifier is chosen by the sending host and is intended to be unique among all the datagrams that might arrive at the destination from this source over some reasonable time period. Since all fragments of the original datagram contain this identifier, the reassembling host will be able to recognize those fragments that go together. Should all the fragments not arrive at the receiving host, the host gives up on the reassembly process and discards the fragments that did arrive. IP does not attempt to recover from missing fragments.

To see what this all means, consider what happens when host H5 sends a datagram to host H8 in the example internet shown in Figure 3.14. Assuming that the MTU is 1500 bytes for the two Ethernets and the 802.11 network, and 532 bytes for the point-to-point network, then a 1420-byte datagram (20-byte IP header plus 1400 bytes of data) sent from H5 makes it across the 802.11 network and the first Ethernet without fragmentation but must be fragmented into three datagrams at router R2. These three fragments are then forwarded by router R3 across the second Ethernet to the destination host. This situation is illustrated in Figure 3.17. This figure also serves to reinforce two important points:

1. Each fragment is itself a self-contained IP datagram that is transmitted over a sequence of physical networks, independent of the other fragments.
2. Each IP datagram is re-encapsulated for each physical network over which it travels.



■ FIGURE 3.17 IP datagrams traversing the sequence of physical networks graphed in Figure 3.14.



■ FIGURE 3.18 Header fields used in IP fragmentation: (a) unfragmented packet; (b) fragmented packets.

The fragmentation process can be understood in detail by looking at the header fields of each datagram, as is done in Figure 3.18. The unfragmented packet, shown at the top, has 1400 bytes of data and a 20-byte IP header. When the packet arrives at router R2, which has an MTU of 532 bytes, it has to be fragmented. A 532-byte MTU leaves 512 bytes for data after the 20-byte IP header, so the first fragment contains 512 bytes of data. The router sets the M bit in the Flags field (see Figure 3.16), meaning that there are more fragments to follow, and it sets the Offset to 0, since this fragment contains the first part of the original datagram. The data carried in the second fragment starts with the 513th byte of the original

data, so the Offset field in this header is set to 64, which is  $512 \div 8$ . Why the division by 8? Because the designers of IP decided that fragmentation should always happen on 8-byte boundaries, which means that the Offset field counts 8-byte chunks, not bytes. (We leave it as an exercise for you to figure out why this design decision was made.) The third fragment contains the last 376 bytes of data, and the offset is now  $2 \times 512 \div 8 = 128$ . Since this is the last fragment, the M bit is not set.

Observe that the fragmentation process is done in such a way that it could be repeated if a fragment arrived at another network with an even smaller MTU. Fragmentation produces smaller, valid IP datagrams that can be readily reassembled into the original datagram upon receipt, independent of the order of their arrival. Reassembly is done at the receiving host and not at each router.

IP reassembly is far from a simple process. For example, if a single fragment is lost, the receiver will still attempt to reassemble the datagram, and it will eventually give up and have to garbage-collect the resources that were used to perform the failed reassembly.<sup>9</sup> For this reason, among others, IP fragmentation is generally considered a good thing to avoid. Hosts are now strongly encouraged to perform “path MTU discovery,” a process by which fragmentation is avoided by sending packets that are small enough to traverse the link with the smallest MTU in the path from sender to receiver.

### 3.2.3 Global Addresses

In the above discussion of the IP service model, we mentioned that one of the things that it provides is an addressing scheme. After all, if you want to be able to send data to any host on any network, there needs to be a way of identifying all the hosts. Thus, we need a global addressing scheme—one in which no two hosts have the same address. Global uniqueness is the first property that should be provided in an addressing scheme.<sup>10</sup>

Ethernet addresses are globally unique, but that alone does not suffice for an addressing scheme in a large internetwork. Ethernet addresses are also *flat*, which means that they have no structure and provide very few clues to routing protocols. (In fact, Ethernet addresses do have a

<sup>9</sup>As we will see in Chapter 8, getting a host to tie up resources needlessly can be the basis of a denial-of-service attack.

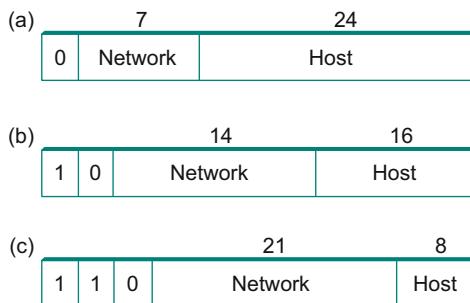
<sup>10</sup>For better or worse, global addressing isn't guaranteed anymore in the modern Internet, for a range of reasons, touched on in Section 4.1.

structure for the purposes of *assignment*—the first 24 bits identify the manufacturer—but this provides no useful information to routing protocols since this structure has nothing to do with network topology.) In contrast, IP addresses are *hierarchical*, by which we mean that they are made up of several parts that correspond to some sort of hierarchy in the internetwork. Specifically, IP addresses consist of two parts, usually referred to as a *network* part and a *host* part. This is a fairly logical structure for an internetwork, which is made up of many interconnected networks. The network part of an IP address identifies the network to which the host is attached; all hosts attached to the same network have the same network part in their IP address. The host part then identifies each host uniquely on that particular network. Thus, in the simple internetwork of Figure 3.14, the addresses of the hosts on network 1, for example, would all have the same network part and different host parts.

Note that the routers in Figure 3.14 are attached to two networks. They need to have an address on each network, one for each interface. For example, router R1, which sits between the wireless network and an Ethernet, has an IP address on the interface to the wireless network whose network part is the same as all the hosts on that network. It also has an IP address on the interface to the Ethernet that has the same network part as the hosts on that Ethernet. Thus, bearing in mind that a router might be implemented as a host with two network interfaces, it is more precise to think of IP addresses as belonging to interfaces than to hosts.

Now, what do these hierarchical addresses look like? Unlike some other forms of hierarchical address, the sizes of the two parts are not the same for all addresses. Originally, IP addresses were divided into three different classes, as shown in Figure 3.19, each of which defines different-sized network and host parts. (There are also class D addresses that specify a multicast group, discussed in Section 4.2, and class E addresses that are currently unused.) In all cases, the address is 32 bits long.

The class of an IP address is identified in the most significant few bits. If the first bit is 0, it is a class A address. If the first bit is 1 and the second is 0, it is a class B address. If the first two bits are 1 and the third is 0, it is a class C address. Thus, of the approximately 4 billion possible IP addresses, half are class A, one-quarter are class B, and one-eighth are class C. Each class allocates a certain number of bits for the network part of the address and the rest for the host part. Class A networks have 7 bits



■ FIGURE 3.19 IP addresses: (a) class A; (b) class B; (c) class C.

for the network part and 24 bits for the host part, meaning that there can be only 126 class A networks (the values 0 and 127 are reserved), but each of them can accommodate up to  $2^{24} - 2$  (about 16 million) hosts (again, there are two reserved values). Class B addresses allocate 14 bits for the network and 16 bits for the host, meaning that each class B network has room for 65,534 hosts. Finally, class C addresses have only 8 bits for the host and 21 for the network part. Therefore, a class C network can have only 256 unique host identifiers, which means only 254 attached hosts (one host identifier, 255, is reserved for broadcast, and 0 is not a valid host number). However, the addressing scheme supports  $2^{21}$  class C networks.

On the face of it, this addressing scheme has a lot of flexibility, allowing networks of vastly different sizes to be accommodated fairly efficiently. The original idea was that the Internet would consist of a small number of wide area networks (these would be class A networks), a modest number of site- (campus-) sized networks (these would be class B networks), and a large number of LANs (these would be class C networks). However, it turned out not to be flexible enough, as we will see in a moment. Today, IP addresses are normally “classless”; the details of this are explained below.

Before we look at how IP addresses get used, it is helpful to look at some practical matters, such as how you write them down. By convention, IP addresses are written as four *decimal* integers separated by dots. Each integer represents the decimal value contained in 1 byte of the address, starting at the most significant. For example, the address of the computer on which this sentence was typed is 171.69.210.245.

It is important not to confuse IP addresses with Internet domain names, which are also hierarchical. Domain names tend to be ASCII

strings separated by dots, such as cs.princeton.edu. We will be talking about those in Section 9.3.1. The important thing about IP addresses is that they are what is carried in the headers of IP packets, and it is those addresses that are used in IP routers to make forwarding decisions.

### 3.2.4 Datagram Forwarding in IP

We are now ready to look at the basic mechanism by which IP routers forward datagrams in an internetwork. Recall from Section 3.1 that *forwarding* is the process of taking a packet from an input and sending it out on the appropriate output, while *routing* is the process of building up the tables that allow the correct output for a packet to be determined. The discussion here focuses on forwarding; we take up routing in Section 3.3.

The main points to bear in mind as we discuss the forwarding of IP datagrams are the following:

- Every IP datagram contains the IP address of the destination host.
- The network part of an IP address uniquely identifies a single physical network that is part of the larger Internet.
- All hosts and routers that share the same network part of their address are connected to the same physical network and can thus communicate with each other by sending frames over that network.
- Every physical network that is part of the Internet has at least one router that, by definition, is also connected to at least one other physical network; this router can exchange packets with hosts or routers on either network.

Forwarding IP datagrams can therefore be handled in the following way. A datagram is sent from a source host to a destination host, possibly passing through several routers along the way. Any node, whether it is a host or a router, first tries to establish whether it is connected to the same physical network as the destination. To do this, it compares the network part of the destination address with the network part of the address of each of its network interfaces. (Hosts normally have only one interface, while routers normally have two or more, since they are typically connected to two or more networks.) If a match occurs, then that means that the destination lies on the same physical network as the interface, and the packet can be directly delivered over that network. Section 3.2.6 explains some of the details of this process.

If the node is not connected to the same physical network as the destination node, then it needs to send the datagram to a router. In general, each node will have a choice of several routers, and so it needs to pick the best one, or at least one that has a reasonable chance of getting the datagram closer to its destination. The router that it chooses is known as the *next hop* router. The router finds the correct next hop by consulting its forwarding table. The forwarding table is conceptually just a list of  $\langle \text{NetworkNum}, \text{NextHop} \rangle$  pairs. (As we will see below, forwarding tables in practice often contain some additional information related to the next hop.) Normally, there is also a default router that is used if none of the entries in the table matches the destination's network number. For a host, it may be quite acceptable to have a default router and nothing else—this means that all datagrams destined for hosts not on the physical network to which the sending host is attached will be sent out through the default router.

We can describe the datagram forwarding algorithm in the following way:

```
if (NetworkNum of destination = NetworkNum of one of my interfaces) then
    deliver packet to destination over that interface
else
    if (NetworkNum of destination is in my forwarding table) then
        deliver packet to NextHop router
    else
        deliver packet to default router
```

For a host with only one interface and only a default router in its forwarding table, this simplifies to

```
if (NetworkNum of destination = my NetworkNum) then
    deliver packet to destination directly
else
    deliver packet to default router
```

Let's see how this works in the example internetwork of Figure 3.14. First, suppose that H1 wants to send a datagram to H2. Since they are on the same physical network, H1 and H2 have the same network number in their IP address. Thus, H1 deduces that it can deliver the datagram directly to H2 over the Ethernet. The one issue that needs to be resolved is

how H1 finds out the correct Ethernet address for H2—this is the address resolution mechanism described in Section 3.2.6.

### Bridges, Switches, and Routers

It is easy to become confused about the distinction between bridges, switches, and routers. There is good reason for such confusion, since at some level they all forward messages from one link to another. One distinction people make is based on layering: Bridges are link-level nodes (they forward frames from one link to another to implement an extended LAN), switches are network-level nodes (they forward packets from one link to another to implement a packet-switched network), and routers are internet-level nodes (they forward datagrams from one network to another to implement an internet).

The distinction between bridges and switches is now pretty much obsolete. For example, we have already seen that a multiport bridge is usually called an Ethernet switch or LAN switch. For this reason, bridges and switches are often grouped together as “layer 2 devices,” where layer 2 in this context means “above the physical layer, below the internet layer.”

Historically, there have been important distinctions between LAN switches (or bridges) and WAN switches (such as those based on ATM or Frame Relay). LAN switches traditionally depend on the spanning tree algorithm, while WAN switches generally run routing protocols that allow each switch to learn the topology of the whole network. This is an important distinction because knowing the whole network topology allows the switches to discriminate among different routes, while, in contrast, the spanning tree algorithm locks in a single tree over which messages are forwarded. It is also the case that the spanning tree approach does not scale as well. Again, this distinction is under threat as routing protocols from the wide area start to make their way into LAN switches.

What about switches and routers? Internally, they look quite similar (as the section on switch and router implementation will illustrate). The key distinction is the sort of packet they forward: IP datagrams in the case of routers and Layer 2 packets (Ethernet frames or ATM cells) in the case of switches.

One big difference between a network built from switches and the Internet built from routers is that the Internet is able to accommodate heterogeneity, whereas switched networks typically consists of homogeneous links. This support for heterogeneity is one of the key reasons why the Internet is so widely deployed. It is also the fact that IP runs over virtually every other protocol (including ATM and Ethernet) that now causes those protocols to be viewed as Layer 2 technologies.

Now suppose H5 wants to send a datagram to H8. Since these hosts are on different physical networks, they have different network numbers, so H5 deduces that it needs to send the datagram to a router. R1 is the only choice—the default router—so H1 sends the datagram over the wireless network to R1. Similarly, R1 knows that it cannot deliver a datagram directly to H8 because neither of R1’s interfaces is on the same network as H8. Suppose R1’s default router is R2; R1 then sends the datagram to R2 over the Ethernet. Assuming R2 has the forwarding table shown in Table 3.5, it looks up H8’s network number (network 4) and forwards the datagram over the point-to-point network to R3. Finally, R3, since it is on the same network as H8, forwards the datagram directly to H8.

**Table 3.5 Example Forwarding Table for Router R2 in Figure 3.14**

NetworkNum	NextHop
1	R1
4	R3

**Table 3.6 Complete Forwarding Table for Router R2 in Figure 3.14**

NetworkNum	NextHop
1	R1
2	Interface 1
3	Interface 0
4	R3

Note that it is possible to include the information about directly connected networks in the forwarding table. For example, we could label the network interfaces of router R2 as interface 0 for the point-to-point link (network 3) and interface 1 for the Ethernet (network 2). Then R2 would have the forwarding table shown in Table 3.6.

Thus, for any network number that R2 encounters in a packet, it knows what to do. Either that network is directly connected to R2, in which case the packet can be delivered to its destination over that network, or the

network is reachable via some next hop router that R2 can reach over a network to which it is connected. In either case, R2 will use ARP, described below, to find the MAC address of the node to which the packet is to be sent next.

The forwarding table used by R2 is simple enough that it could be manually configured. Usually, however, these tables are more complex and would be built up by running a routing protocol such as one of those described in Section 3.3. Also note that, in practice, the network numbers are usually longer (e.g., 128.96).

We can now see how hierarchical addressing—splitting the address into network and host parts—has improved the scalability of a large network. Routers now contain forwarding tables that list only a set of network numbers rather than all the nodes in the network. In our simple example, that meant that R2 could store the information needed to reach all the hosts in the network (of which there were eight) in a four-entry table. Even if there were 100 hosts on each physical network, R2 would still only need those same four entries. This is a good first step (although by no means the last) in achieving scalability.



This illustrates one of the most important principles of building scalable networks: To achieve scalability, you need to reduce the amount of information that is stored in each node and that is exchanged between nodes. The most common way to do that is *hierarchical aggregation*. IP introduces a two-level hierarchy, with networks at the top level and nodes at the bottom level. We have aggregated information by letting routers deal only with reaching the right network; the information that a router needs to deliver a datagram to any node on a given network is represented by a single aggregated piece of information.

### 3.2.5 Subnetting and Classless Addressing

The original intent of IP addresses was that the network part would uniquely identify exactly one physical network. It turns out that this approach has a couple of drawbacks. Imagine a large campus that has lots of internal networks and decides to connect to the Internet. For every network, no matter how small, the site needs at least a class C network address. Even worse, for any network with more than 255 hosts, they need a class B address. This may not seem like a big deal, and indeed it wasn't when the Internet was first envisioned, but there are only a finite

number of network numbers, and there are far fewer class B addresses than class Cs. Class B addresses tend to be in particularly high demand because you never know if your network might expand beyond 255 nodes, so it is easier to use a class B address from the start than to have to renumber every host when you run out of room on a class C network. The problem we observe here is address assignment inefficiency: A network with two nodes uses an entire class C network address, thereby wasting 253 perfectly useful addresses; a class B network with slightly more than 255 hosts wastes over 64,000 addresses.

Assigning one network number per physical network, therefore, uses up the IP address space potentially much faster than we would like. While we would need to connect over 4 billion hosts to use up all the valid addresses, we only need to connect  $2^{14}$  (about 16,000) class B networks before that part of the address space runs out. Therefore, we would like to find some way to use the network numbers more efficiently.

Assigning many network numbers has another drawback that becomes apparent when you think about routing. Recall that the amount of state that is stored in a node participating in a routing protocol is proportional to the number of other nodes, and that routing in an internet consists of building up forwarding tables that tell a router how to reach different networks. Thus, the more network numbers there are in use, the bigger the forwarding tables get. Big forwarding tables add costs to routers, and they are potentially slower to search than smaller tables for a given technology, so they degrade router performance. This provides another motivation for assigning network numbers carefully.

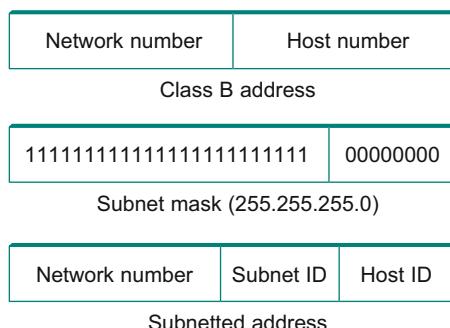
*Subnetting* provides a first step to reducing total number of network numbers that are assigned. The idea is to take a single IP network number and allocate the IP addresses with that network number to several physical networks, which are now referred to as *subnets*. Several things need to be done to make this work. First, the subnets should be close to each other. This is because at a distant point in the Internet, they will all look like a single network, having only one network number between them. This means that a router will only be able to select one route to reach any of the subnets, so they had better all be in the same general direction. A perfect situation in which to use subnetting is a large campus or corporation that has many physical networks. From outside the campus, all you need to know to reach any subnet inside the campus is where the campus connects to the rest of the Internet. This is often at a single point,

so one entry in your forwarding table will suffice. Even if there are multiple points at which the campus is connected to the rest of the Internet, knowing how to get to one point in the campus network is still a good start.

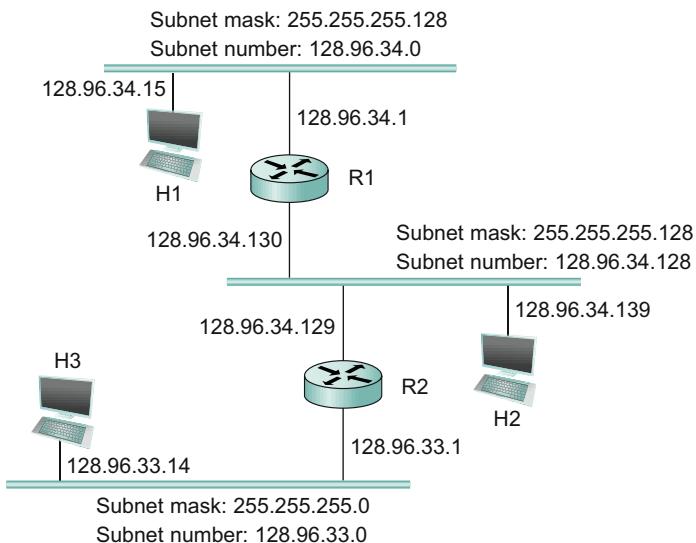
The mechanism by which a single network number can be shared among multiple networks involves configuring all the nodes on each subnet with a *subnet mask*. With simple IP addresses, all hosts on the same network must have the same network number. The subnet mask enables us to introduce a *subnet number*; all hosts on the same physical network will have the same subnet number, which means that hosts may be on different physical networks but share a single network number. This concept is illustrated in Figure 3.20.

What subnetting means to a host is that it is now configured with both an IP address and a subnet mask for the subnet to which it is attached. For example, host H1 in Figure 3.21 is configured with an address of 128.96.34.15 and a subnet mask of 255.255.255.128. (All hosts on a given subnet are configured with the same mask; that is, there is exactly one subnet mask per subnet.) The bitwise AND of these two numbers defines the subnet number of the host and of all other hosts on the same subnet. In this case, 128.96.34.15 AND 255.255.255.128 equals 128.96.34.0, so this is the subnet number for the topmost subnet in the figure.

When the host wants to send a packet to a certain IP address, the first thing it does is to perform a bitwise AND between its own subnet mask and the destination IP address. If the result equals the subnet number of the sending host, then it knows that the destination host is on the same subnet and the packet can be delivered directly over the subnet.



■ FIGURE 3.20 Subnet addressing.



■ FIGURE 3.21 An example of subnetting.

If the results are not equal, the packet needs to be sent to a router to be forwarded to another subnet. For example, if H1 is sending to H2, then H1 ANDs its subnet mask (255.255.255.128) with the address for H2 (128.96.34.139) to obtain 128.96.34.128. This does not match the subnet number for H1 (128.96.34.0) so H1 knows that H2 is on a different subnet. Since H1 cannot deliver the packet to H2 directly over the subnet, it sends the packet to its default router R1.

The forwarding table of a router also changes slightly when we introduce subnetting. Recall that we previously had a forwarding table that consisted of entries of the form  $\langle \text{NetworkNum}, \text{NextHop} \rangle$ . To support subnetting, the table must now hold entries of the form  $\langle \text{SubnetNumber}, \text{SubnetMask}, \text{NextHop} \rangle$ . To find the right entry in the table, the router ANDs the packet's destination address with the SubnetMask for each entry in turn; if the result matches the SubnetNumber of the entry, then this is the right entry to use, and it forwards the packet to the next hop router indicated. In the example network of Figure 3.21, router R1 would have the entries shown in Table 3.7.

Continuing with the example of a datagram from H1 being sent to H2, R1 would AND H2's address (128.96.34.139) with the subnet mask of the first entry (255.255.255.128) and compare the result (128.96.34.128) with

**Table 3.7 Example Forwarding Table with Subnetting for Figure 3.21**

SubnetNumber	SubnetMask	NextHop
128.96.34.0	255.255.255.128	Interface 0
128.96.34.128	255.255.255.128	Interface 1
128.96.33.0	255.255.255.0	R2

the network number for that entry (128.96.34.0). Since this is not a match, it proceeds to the next entry. This time a match does occur, so R1 delivers the datagram to H2 using interface 1, which is the interface connected to the same network as H2.

We can now describe the datagram forwarding algorithm in the following way:

```

D = destination IP address
for each forwarding table entry <SubnetNumber, SubnetMask, NextHop>
    D1 = SubnetMask & D
    if D1 = SubnetNumber
        if NextHop is an interface
            deliver datagram directly to destination
        else
            deliver datagram to NextHop (a router)
    
```

Although not shown in this example, a default route would usually be included in the table and would be used if no explicit matches were found. We note in passing that a naive implementation of this algorithm—one involving repeated ANDing of the destination address with a subnet mask that may not be different every time, and a linear table search—would be very inefficient.

An important consequence of subnetting is that different parts of the internet see the world differently. From outside our hypothetical campus, routers see a single network. In the example above, routers outside the campus see the collection of networks in Figure 3.21 as just the network 128.96, and they keep one entry in their forwarding tables to tell them how to reach it. Routers within the campus, however, need to be able to route packets to the right subnet. Thus, not all parts of the internet see exactly the same routing information. This is an example of *aggregation*.

of routing information, which is fundamental to scaling of the routing system. The next section shows how aggregation can be taken to another level.

### *Classless Addressing*

Subnetting has a counterpart, sometimes called *supernetting*, but more often called *Classless Interdomain Routing* or CIDR, pronounced “cider.” CIDR takes the subnetting idea to its logical conclusion by essentially doing away with address classes altogether. Why isn’t subnetting alone sufficient? In essence, subnetting only allows us to split a classful address among multiple subnets, while CIDR allows us to coalesce several classful addresses into a single “supernet.” This further tackles the address space inefficiency noted above, and does so in a way that keeps the routing system from being overloaded.

To see how the issues of address space efficiency and scalability of the routing system are coupled, consider the hypothetical case of a company whose network has 256 hosts on it. That is slightly too many for a Class C address, so you would be tempted to assign a class B. However, using up a chunk of address space that could address 65,535 to address 256 hosts has an efficiency of only  $256/65,535 = 0.39\%$ . Even though subnetting can help us to assign addresses carefully, it does not get around the fact that any organization with more than 255 hosts, or an expectation of eventually having that many, wants a class B address.

The first way you might deal with this issue would be to refuse to give a class B address to any organization that requests one unless they can show a need for something close to 64K addresses, and instead giving them an appropriate number of class C addresses to cover the expected number of hosts. Since we would now be handing out address space in chunks of 256 addresses at a time, we could more accurately match the amount of address space consumed to the size of the organization. For any organization with at least 256 hosts, we can guarantee an address utilization of at least 50%, and typically much more.

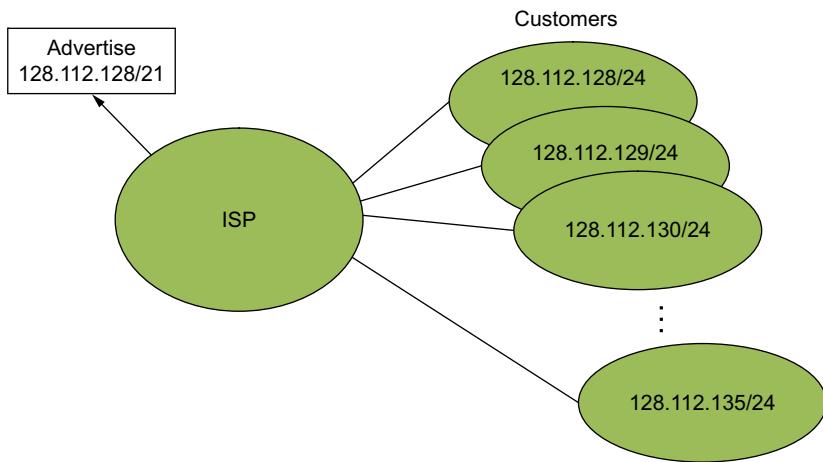
This solution, however, raises a problem that is at least as serious: excessive storage requirements at the routers. If a single site has, say, 16 class C network numbers assigned to it, that means every Internet backbone router needs 16 entries in its routing tables to direct packets to that site. This is true even if the path to every one of those networks is the same. If we had assigned a class B address to the site, the same routing

information could be stored in one table entry. However, our address assignment efficiency would then be only  $16 \times 255 / 65,536 = 6.2\%$ .

CIDR, therefore, tries to balance the desire to minimize the number of routes that a router needs to know against the need to hand out addresses efficiently. To do this, CIDR helps us to *aggregate* routes. That is, it lets us use a single entry in a forwarding table to tell us how to reach a lot of different networks. As noted above it does this by breaking the rigid boundaries between address classes. To understand how this works, consider our hypothetical organization with 16 class C network numbers. Instead of handing out 16 addresses at random, we can hand out a block of *contiguous* class C addresses. Suppose we assign the class C network numbers from 192.4.16 through 192.4.31. Observe that the top 20 bits of all the addresses in this range are the same (11000000 00000100 0001). Thus, what we have effectively created is a 20-bit network number—something that is between a class B network number and a class C number in terms of the number of hosts that it can support. In other words, we get both the high address efficiency of handing out addresses in chunks smaller than a class B network, and a single network prefix that can be used in forwarding tables. Observe that, for this scheme to work, we need to hand out blocks of class C addresses that share a common prefix, which means that each block must contain a number of class C networks that is a power of two.

CIDR requires a new type of notation to represent network numbers, or *prefixes* as they are known, because the prefixes can be of any length. The convention is to place a /X after the prefix, where X is the prefix length in bits. So, for the example above, the 20-bit prefix for all the networks 192.4.16 through 192.4.31 is represented as 192.4.16/20. By contrast, if we wanted to represent a single class C network number, which is 24 bits long, we would write it 192.4.16/24. Today, with CIDR being the norm, it is more common to hear people talk about “slash 24” prefixes than class C networks. Note that representing a network address in this way is similar to the  $\langle \text{mask}, \text{value} \rangle$  approach used in subnetting, as long as masks consist of contiguous bits starting from the most significant bit (which in practice is almost always the case).

The ability to aggregate routes at the edge of the network as we have just seen is only the first step. Imagine an Internet service provider network, whose primary job is to provide Internet connectivity to a large number of corporations and campuses (customers). If we assign prefixes



■ FIGURE 3.22 Route aggregation with CIDR.

to the customers in such a way that many different customer networks connected to the provider network share a common, shorter address prefix, then we can get even greater aggregation of routes. Consider the example in Figure 3.22. Assume that eight customers served by the provider network have each been assigned adjacent 24-bit network prefixes. Those prefixes all start with the same 21 bits. Since all of the customers are reachable through the same provider network, it can advertise a single route to all of them by just advertising the common 21-bit prefix they share. And it can do this even if not all the 24-bit prefixes have been handed out, as long as the provider ultimately *will* have the right to hand out those prefixes to a customer. One way to accomplish that is to assign a portion of address space to the provider in advance and then to let the network provider assign addresses from that space to its customers as needed. Note that, in contrast to this simple example, there is no need for all customer prefixes to be the same length.

#### IP Forwarding Revisited

In all our discussion of IP forwarding so far, we have assumed that we could find the network number in a packet and then look up that number in a forwarding table. However, now that we have introduced CIDR, we need to reexamine this assumption. CIDR means that prefixes may be of any length, from 2 to 32 bits. Furthermore, it is sometimes possible

to have prefixes in the forwarding table that “overlap,” in the sense that some addresses may match more than one prefix. For example, we might find both 171.69 (a 16-bit prefix) and 171.69.10 (a 24-bit prefix) in the forwarding table of a single router. In this case, a packet destined to, say, 171.69.10.5 clearly matches both prefixes. The rule in this case is based on the principle of “longest match”; that is, the packet matches the longest prefix, which would be 171.69.10 in this example. On the other hand, a packet destined to 171.69.20.5 would match 171.69 and *not* 171.69.10, and in the absence of any other matching entry in the routing table 171.69 would be the longest match.

The task of efficiently finding the longest match between an IP address and the variable-length prefixes in a forwarding table has been a fruitful field of research in recent years, and the Further Reading section of this chapter provides some references. The most well-known algorithm uses an approach known as a *PATRICIA tree*, which was actually developed well in advance of CIDR.

### 3.2.6 Address Translation (ARP)

In the previous section we talked about how to get IP datagrams to the right physical network but glossed over the issue of how to get a datagram to a particular host or router on that network. The main issue is that IP datagrams contain IP addresses, but the physical interface hardware on the host or router to which you want to send the datagram only understands the addressing scheme of that particular network. Thus, we need to translate the IP address to a link-level address that makes sense on this network (e.g., a 48-bit Ethernet address). We can then encapsulate the IP datagram inside a frame that contains that link-level address and send it either to the ultimate destination or to a router that promises to forward the datagram toward the ultimate destination.

One simple way to map an IP address into a physical network address is to encode a host’s physical address in the host part of its IP address. For example, a host with physical address 00100001 01001001 (which has the decimal value 33 in the upper byte and 81 in the lower byte) might be given the IP address 128.96.33.81. While this solution has been used on some networks, it is limited in that the network’s physical addresses can be no more than 16 bits long in this example; they can be only 8 bits long on a class C network. This clearly will not work for 48-bit Ethernet addresses.

A more general solution would be for each host to maintain a table of address pairs; that is, the table would map IP addresses into physical addresses. While this table could be centrally managed by a system administrator and then copied to each host on the network, a better approach would be for each host to dynamically learn the contents of the table using the network. This can be accomplished using the Address Resolution Protocol (ARP). The goal of ARP is to enable each host on a network to build up a table of mappings between IP addresses and link-level addresses. Since these mappings may change over time (e.g., because an Ethernet card in a host breaks and is replaced by a new one with a new address), the entries are timed out periodically and removed. This happens on the order of every 15 minutes. The set of mappings currently stored in a host is known as the ARP cache or ARP table.

ARP takes advantage of the fact that many link-level network technologies, such as Ethernet, support broadcast. If a host wants to send an IP datagram to a host (or router) that it knows to be on the same network (i.e., the sending and receiving node have the same IP network number), it first checks for a mapping in the cache. If no mapping is found, it needs to invoke the Address Resolution Protocol over the network. It does this by broadcasting an ARP query onto the network. This query contains the IP address in question (the target IP address). Each host receives the query and checks to see if it matches its IP address. If it does match, the host sends a response message that contains its link-layer address back to the originator of the query. The originator adds the information contained in this response to its ARP table.

The query message also includes the IP address and link-layer address of the sending host. Thus, when a host broadcasts a query message, each host on the network can learn the sender's link-level and IP addresses and place that information in its ARP table. However, not every host adds this information to its ARP table. If the host already has an entry for that host in its table, it "refreshes" this entry; that is, it resets the length of time until it discards the entry. If that host is the target of the query, then it adds the information about the sender to its table, even if it did not already have an entry for that host. This is because there is a good chance that the source host is about to send it an application-level message, and it may eventually have to send a response or ACK back to the source; it will need the source's physical address to do this. If a host is not the target and does not already have an entry for the source in its ARP table, then it does not

0	8	16	31		
Hardware type = 1		ProtocolType = 0x0800			
HLen = 48	PLen = 32	Operation			
SourceHardwareAddr (bytes 0–3)					
SourceHardwareAddr (bytes 4–5)		SourceProtocolAddr (bytes 0–1)			
SourceProtocolAddr (bytes 2–3)		TargetHardwareAddr (bytes 0–1)			
TargetHardwareAddr (bytes 2–5)					
TargetProtocolAddr (bytes 0–3)					

■ FIGURE 3.23 ARP packet format for mapping IP addresses into Ethernet addresses.

add an entry for the source. This is because there is no reason to believe that this host will ever need the source’s link-level address; there is no need to clutter its ARP table with this information.

Figure 3.23 shows the ARP packet format for IP-to-Ethernet address mappings. In fact, ARP can be used for lots of other kinds of mappings—the major differences are in the address sizes. In addition to the IP and link-layer addresses of both sender and target, the packet contains

- A **HardwareType** field, which specifies the type of physical network (e.g., Ethernet)
- A **ProtocolType** field, which specifies the higher-layer protocol (e.g., IP)
- **HLen** (“hardware” address length) and **PLen** (“protocol” address length) fields, which specify the length of the link-layer address and higher-layer protocol address, respectively
- An **Operation** field, which specifies whether this is a request or a response
- The source and target hardware (Ethernet) and protocol (IP) addresses

Note that the results of the ARP process can be added as an extra column in a forwarding table like the one in Table 4.1. Thus, for example, when R2 needs to forward a packet to network 2, it not only finds that the next hop is R1, but also finds the MAC address to place on the packet to send it to R1.

We have now seen the basic mechanisms that IP provides for dealing with both heterogeneity and scale. On the issue of heterogeneity, IP begins by defining a best-effort service model that makes minimal assumptions about the underlying networks; most notably, this service model is based on unreliable datagrams. IP then makes two important additions to this starting point: (1) a common packet format (fragmentation/reassembly) is the mechanism that makes this format work over networks with different MTUs) and (2) a global address space for identifying all hosts (ARP is the mechanism that makes this global address space work over networks with different physical addressing schemes). On the issue of scale, IP uses hierarchical aggregation to reduce the amount of information needed to forward packets. Specifically, IP addresses are partitioned into network and host components, with packets first routed toward the destination network and then delivered to the correct host on that network.

### 3.2.7 Host Configuration (DHCP)

In Section 2.6, we observed that Ethernet addresses are configured into the network adaptor by the manufacturer, and this process is managed in such a way to ensure that these addresses are globally unique. This is clearly a sufficient condition to ensure that any collection of hosts connected to a single Ethernet (including an extended LAN) will have unique addresses. Furthermore, uniqueness is all we ask of Ethernet addresses.

IP addresses, by contrast, not only must be unique on a given internetwork but also must reflect the structure of the internetwork. As noted above, they contain a network part and a host part, and the network part must be the same for all hosts on the same network. Thus, it is not possible for the IP address to be configured once into a host when it is manufactured, since that would imply that the manufacturer knew which hosts were going to end up on which networks, and it would mean that a host, once connected to one network, could never move to another. For this reason, IP addresses need to be reconfigurable.

In addition to an IP address, there are some other pieces of information a host needs to have before it can start sending packets. The most notable of these is the address of a default router—the place to which it can send packets whose destination address is not on the same network as the sending host.

Most host operating systems provide a way for a system administrator, or even a user, to manually configure the IP information needed by

a host; however, there are some obvious drawbacks to such manual configuration. One is that it is simply a lot of work to configure all the hosts in a large network directly, especially when you consider that such hosts are not reachable over a network until they are configured. Even more importantly, the configuration process is very error prone, since it is necessary to ensure that every host gets the correct network number and that no two hosts receive the same IP address. For these reasons, automated configuration methods are required. The primary method uses a protocol known as the *Dynamic Host Configuration Protocol* (DHCP).

DHCP relies on the existence of a DHCP server that is responsible for providing configuration information to hosts. There is at least one DHCP server for an administrative domain. At the simplest level, the DHCP server can function just as a centralized repository for host configuration information. Consider, for example, the problem of administering addresses in the internetwork of a large company. DHCP saves the network administrators from having to walk around to every host in the company with a list of addresses and network map in hand and configuring each host manually. Instead, the configuration information for each host could be stored in the DHCP server and automatically retrieved by each host when it is booted or connected to the network. However, the administrator would still pick the address that each host is to receive; he would just store that in the server. In this model, the configuration information for each host is stored in a table that is indexed by some form of unique client identifier, typically the hardware address (e.g., the Ethernet address of its network adaptor).

A more sophisticated use of DHCP saves the network administrator from even having to assign addresses to individual hosts. In this model, the DHCP server maintains a pool of available addresses that it hands out to hosts on demand. This considerably reduces the amount of configuration an administrator must do, since now it is only necessary to allocate a range of IP addresses (all with the same network number) to each network.

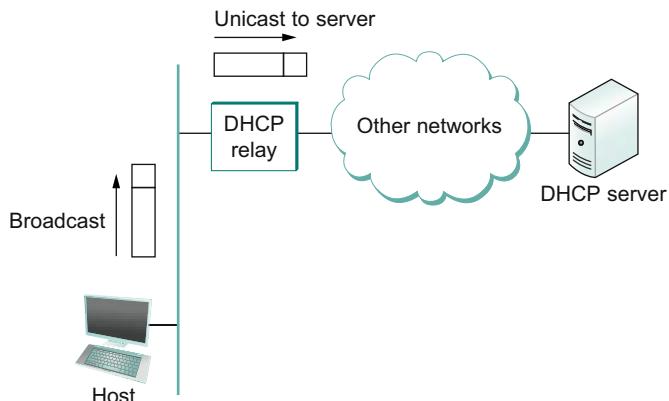
Since the goal of DHCP is to minimize the amount of manual configuration required for a host to function, it would rather defeat the purpose if each host had to be configured with the address of a DHCP server. Thus, the first problem faced by DHCP is that of server discovery.

To contact a DHCP server, a newly booted or attached host sends a DHCPDISCOVER message to a special IP address (255.255.255.255) that

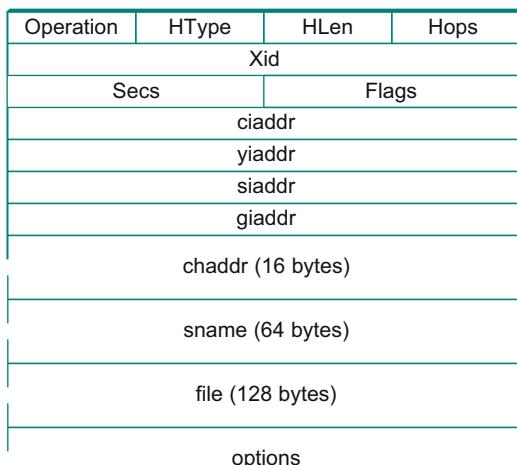
is an IP broadcast address. This means it will be received by all hosts and routers on that network. (Routers do not forward such packets onto other networks, preventing broadcast to the entire Internet.) In the simplest case, one of these nodes is the DHCP server for the network. The server would then reply to the host that generated the discovery message (all the other nodes would ignore it). However, it is not really desirable to require one DHCP server on every network, because this still creates a potentially large number of servers that need to be correctly and consistently configured. Thus, DHCP uses the concept of a *relay agent*. There is at least one relay agent on each network, and it is configured with just one piece of information: the IP address of the DHCP server. When a relay agent receives a **DHCPDISCOVER** message, it unicasts it to the DHCP server and awaits the response, which it will then send back to the requesting client. The process of relaying a message from a host to a remote DHCP server is shown in Figure 3.24.

Figure 3.25 shows the format of a DHCP message. The message is actually sent using a protocol called the *User Datagram Protocol* (UDP) that runs over IP. UDP is discussed in detail in the next chapter, but the only interesting thing it does in this context is to provide a demultiplexing key that says, “This is a DHCP packet.”

DHCP is derived from an earlier protocol called BOOTP, and some of the packet fields are thus not strictly relevant to host configuration. When



■ FIGURE 3.24 A DHCP relay agent receives a broadcast **DHCPDISCOVER** message from a host and sends a unicast **DHCPDISCOVER** to the DHCP server.



■ FIGURE 3.25 DHCP packet format.

trying to obtain configuration information, the client puts its hardware address (e.g., its Ethernet address) in the chaddr field. The DHCP server replies by filling in the yiaddr (“your” IP address) field and sending it to the client. Other information such as the default router to be used by this client can be included in the options field.

In the case where DHCP dynamically assigns IP addresses to hosts, it is clear that hosts cannot keep addresses indefinitely, as this would eventually cause the server to exhaust its address pool. At the same time, a host cannot be depended upon to give back its address, since it might have crashed, been unplugged from the network, or been turned off. Thus, DHCP allows addresses to be leased for some period of time. Once the lease expires, the server is free to return that address to its pool. A host with a leased address clearly needs to renew the lease periodically if in fact it is still connected to the network and functioning correctly.

DHCP illustrates an important aspect of scaling: the scaling of network management. While discussions of scaling often focus on keeping the state in network devices from growing too fast, it is important to pay attention to growth of network management complexity. By allowing network managers to configure a range of IP addresses per network rather than one IP address per host, DHCP improves the manageability of a network.

Note that DHCP may also introduce some more complexity into network management, since it makes the binding between physical hosts and IP addresses much more dynamic. This may make the network manager's job more difficult if, for example, it becomes necessary to locate a malfunctioning host.

### 3.2.8 Error Reporting (ICMP)

The next issue is how the Internet treats errors. While IP is perfectly willing to drop datagrams when the going gets tough—for example, when a router does not know how to forward the datagram or when one fragment of a datagram fails to arrive at the destination—it does not necessarily fail silently. IP is always configured with a companion protocol, known as the *Internet Control Message Protocol* (ICMP), that defines a collection of error messages that are sent back to the source host whenever a router or host is unable to process an IP datagram successfully. For example, ICMP defines error messages indicating that the destination host is unreachable (perhaps due to a link failure), that the reassembly process failed, that the TTL had reached 0, that the IP header checksum failed, and so on.

ICMP also defines a handful of control messages that a router can send back to a source host. One of the most useful control messages, called an *ICMP-Redirect*, tells the source host that there is a better route to the destination. ICMP-Redirects are used in the following situation. Suppose a host is connected to a network that has two routers attached to it, called *R1* and *R2*, where the host uses *R1* as its default router. Should *R1* ever receive a datagram from the host, where based on its forwarding table it knows that *R2* would have been a better choice for a particular destination address, it sends an ICMP-Redirect back to the host, instructing it to use *R2* for all future datagrams addressed to that destination. The host then adds this new route to its forwarding table.

ICMP also provides the basis for two widely used debugging tools, ping and traceroute. ping uses ICMP echo messages to determine if a node is reachable and alive. traceroute uses a slightly non-intuitive technique to determine the set of routers along the path to a destination, which is the topic for one of the exercises at the end of this chapter.

### 3.2.9 Virtual Networks and Tunnels

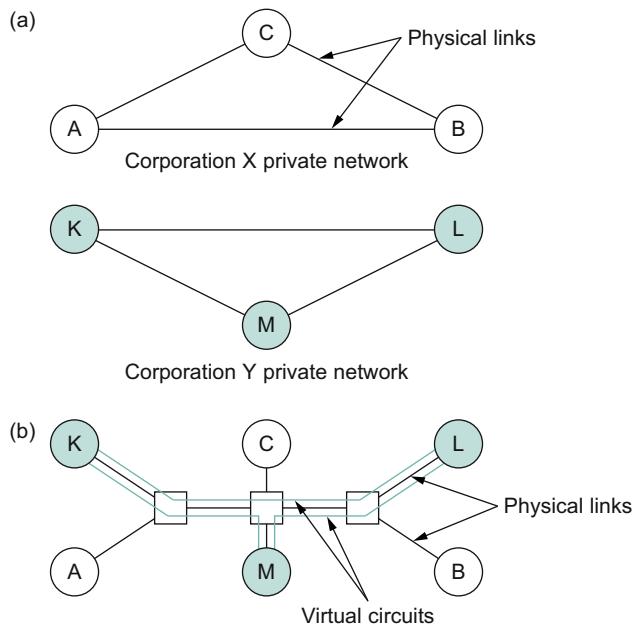
We conclude our introduction to IP by considering an issue you might not have anticipated, but one that is becoming increasingly important.

Our discussion up to this point has focused on making it possible for nodes on different networks to communicate with each other in an unrestricted way. This is usually the goal in the Internet—everybody wants to be able to send email to everybody, and the creator of a new website wants to reach the widest possible audience. However, there are many situations where more controlled connectivity is required. An important example of such a situation is the *virtual private network* (VPN).

The term VPN is heavily overused and definitions vary, but intuitively we can define a VPN by considering first the idea of a private network. Corporations with many sites often build private networks by leasing transmission lines from the phone companies and using those lines to interconnect sites. In such a network, communication is restricted to take place only among the sites of that corporation, which is often desirable for security reasons. To make a private network *virtual*, the leased transmission lines—which are not shared with any other corporations—would be replaced by some sort of shared network. A virtual circuit (VC) is a very reasonable replacement for a leased line because it still provides a logical point-to-point connection between the corporation's sites. For example, if corporation X has a VC from site A to site B, then clearly it can send packets between sites A and B. But there is no way that corporation Y can get its packets delivered to site B without first establishing its own virtual circuit to site B, and the establishment of such a VC can be administratively prevented, thus preventing unwanted connectivity between corporation X and corporation Y.

Figure 3.26(a) shows two private networks for two separate corporations. In Figure 3.26(b) they are both migrated to a virtual circuit network. The limited connectivity of a real private network is maintained, but since the private networks now share the same transmission facilities and switches we say that two virtual private networks have been created.

In Figure 3.26, a virtual circuit network (using Frame Relay or ATM, for example) is used to provide the controlled connectivity among sites. It is also possible to provide a similar function using an IP network—an internetwork—to provide the connectivity. However, we cannot just connect the various corporations' sites to a single internetwork because that would provide connectivity between corporation X and corporation Y, which we wish to avoid. To solve this problem, we need to introduce a new concept, the *IP tunnel*.

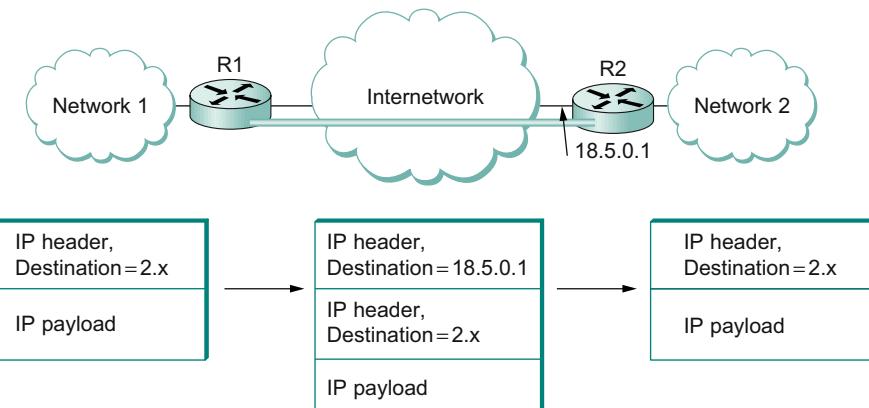


**FIGURE 3.26** An example of virtual private networks: (a) two separate private networks; (b) two virtual private networks sharing common switches.

We can think of an IP tunnel as a virtual point-to-point link between a pair of nodes that are actually separated by an arbitrary number of networks. The virtual link is created within the router at the entrance to the tunnel by providing it with the IP address of the router at the far end of the tunnel. Whenever the router at the entrance of the tunnel wants to send a packet over this virtual link, it encapsulates the packet inside an IP datagram. The destination address in the IP header is the address of the router at the far end of the tunnel, while the source address is that of the encapsulating router.

In the forwarding table of the router at the entrance to the tunnel, this virtual link looks much like a normal link. Consider, for example, the network in Figure 3.27. A tunnel has been configured from R1 to R2 and assigned a virtual interface number of 0. The forwarding table in R1 might therefore look like Table 3.8.

R1 has two physical interfaces. Interface 0 connects to network 1; interface 1 connects to a large internetwork and is thus the default for all traffic



**FIGURE 3.27** A tunnel through an internetwork. 18.5.0.1 is the address of R2 that can be reached from R1 across the internetwork.

**Table 3.8 Forwarding Table for Router R1 in Figure 3.27**

NetworkNum	NextHop
1	Interface 0
2	Virtual interface 0
Default	Interface 1

that does not match something more specific in the forwarding table. In addition, R1 has a virtual interface, which is the interface to the tunnel. Suppose R1 receives a packet from network 1 that contains an address in network 2. The forwarding table says this packet should be sent out virtual interface 0. In order to send a packet out this interface, the router takes the packet, adds an IP header addressed to R2, and then proceeds to forward the packet as if it had just been received. R2's address is 18.5.0.1; since the network number of this address is 18, not 1 or 2, a packet destined for R2 will be forwarded out the default interface into the internetwork.

Once the packet leaves R1, it looks to the rest of the world like a normal IP packet destined to R2, and it is forwarded accordingly. All the routers in the internetwork forward it using normal means, until it arrives at R2. When R2 receives the packet, it finds that it carries its own address, so it

removes the IP header and looks at the payload of the packet. What it finds is an inner IP packet whose destination address is in network 2. R2 now processes this packet like any other IP packet it receives. Since R2 is directly connected to network 2, it forwards the packet on to that network. Figure 3.27 shows the change in encapsulation of the packet as it moves across the network.

While R2 is acting as the endpoint of the tunnel, there is nothing to prevent it from performing the normal functions of a router. For example, it might receive some packets that are not tunneled, but that are addressed to networks that it knows how to reach, and it would forward them in the normal way.

You might wonder why anyone would want to go to all the trouble of creating a tunnel and changing the encapsulation of a packet as it goes across an internetwork. One reason is security, which we will discuss in more detail in Chapter 8. Supplemented with encryption, a tunnel can become a very private sort of link across a public network. Another reason may be that R1 and R2 have some capabilities that are not widely available in the intervening networks, such as multicast routing. By connecting these routers with a tunnel, we can build a virtual network in which all the routers with this capability appear to be directly connected. This in fact is how the MBone (multicast backbone) is built, as we will see in Section 4.2. A third reason to build tunnels is to carry packets from protocols other than IP across an IP network. As long as the routers at either end of the tunnel know how to handle these other protocols, the IP tunnel looks to them like a point-to-point link over which they can send non-IP packets. Tunnels also provide a mechanism by which we can force a packet to be delivered to a particular place even if its original header—the one that gets encapsulated inside the tunnel header—might suggest that it should go somewhere else. We will see an application of this when we consider mobile hosts in Section 4.4.2. Thus, we see that tunneling is a powerful and quite general technique for building virtual links across internetworks.

Tunneling does have its downsides. One is that it increases the length of packets; this might represent a significant waste of bandwidth for short packets. Longer packets might be subject to fragmentation, which has its own set of drawbacks. There may also be performance implications for the routers at either end of the tunnel, since they need to do more work than normal forwarding as they add and remove the tunnel header.

Finally, there is a management cost for the administrative entity that is responsible for setting up the tunnels and making sure they are correctly handled by the routing protocols.

### 3.3 ROUTING

So far in this chapter we have assumed that the switches and routers have enough knowledge of the network topology so they can choose the right port onto which each packet should be output. In the case of virtual circuits, routing is an issue only for the connection request packet; all subsequent packets follow the same path as the request. In datagram networks, including IP networks, routing is an issue for every packet. In either case, a switch or router needs to be able to look at a destination address and then to determine which of the output ports is the best choice to get a packet to that address. As we saw in Section 3.1.1, the switch makes this decision by consulting a forwarding table. The fundamental problem of routing is how switches and routers acquire the information in their forwarding tables.



We restate an important distinction, which is often neglected, between *forwarding* and *routing*. Forwarding consists of taking a packet, looking at its destination address, consulting a table, and sending the packet in a direction determined by that table. We saw several examples of forwarding in the preceding section. Routing is the process by which forwarding tables are built. We also note that forwarding is a relatively simple and well-defined process performed locally at a node, whereas routing depends on complex distributed algorithms that have continued to evolve throughout the history of networking.

While the terms *forwarding table* and *routing table* are sometimes used interchangeably, we will make a distinction between them here. The forwarding table is used when a packet is being forwarded and so must contain enough information to accomplish the forwarding function. This means that a row in the forwarding table contains the mapping from a network prefix to an outgoing interface and some MAC information, such as the Ethernet address of the next hop. The routing table, on the other hand, is the table that is built up by the routing algorithms as a precursor to building the forwarding table. It generally contains mappings from network prefixes to next hops. It may also contain information about how this

**Table 3.9 Example Rows from (a) Routing and (b) Forwarding Tables**

(a)	
Prefix/Length	Next Hop
18/8	171.69.245.10

(b)		
Prefix/Length	Interface	MAC Address
18/8	if0	8:0:2b:e4:b:1:2

information was learned, so that the router will be able to decide when it should discard some information.

Whether the routing table and forwarding table are actually separate data structures is something of an implementation choice, but there are numerous reasons to keep them separate. For example, the forwarding table needs to be structured to optimize the process of looking up an address when forwarding a packet, while the routing table needs to be optimized for the purpose of calculating changes in topology. In many cases, the forwarding table may even be implemented in specialized hardware, whereas this is rarely if ever done for the routing table. Table 3.9 provides an example of a row from each sort of table. In this case, the routing table tells us that network prefix 18/8 is to be reached by a next hop router with the IP address 171.69.245.10, while the forwarding table contains the information about exactly how to forward a packet to that next hop: Send it out interface number 0 with a MAC address of 8:0:2b:e4:b:1:2. Note that the last piece of information is provided by the Address Resolution Protocol.

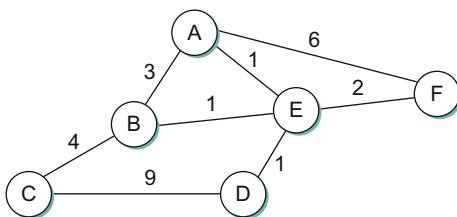
Before getting into the details of routing, we need to remind ourselves of the key question we should be asking anytime we try to build a mechanism for the Internet: “Does this solution scale?” The answer for the algorithms and protocols described in this section is “not so much.” They are designed for networks of fairly modest size—up to a few hundred nodes, in practice. However, the solutions we describe do serve as a building block for a hierarchical routing infrastructure that is used in the Internet today. Specifically, the protocols described in this section are

collectively known as *intradomain* routing protocols, or *interior gateway protocols* (IGPs). To understand these terms, we need to define a routing *domain*. A good working definition is an internetwork in which all the routers are under the same administrative control (e.g., a single university campus, or the network of a single Internet Service Provider). The relevance of this definition will become apparent in the next chapter when we look at *interdomain* routing protocols. For now, the important thing to keep in mind is that we are considering the problem of routing in the context of small to midsized networks, not for a network the size of the Internet.

### 3.3.1 Network as a Graph

Routing is, in essence, a problem of graph theory. Figure 3.28 shows a graph representing a network. The nodes of the graph, labeled A through F, may be hosts, switches, routers, or networks. For our initial discussion, we will focus on the case where the nodes are routers. The edges of the graph correspond to the network links. Each edge has an associated *cost*, which gives some indication of the desirability of sending traffic over that link. A discussion of how edge costs are assigned is given in Section 3.3.4.<sup>11</sup>

The basic problem of routing is to find the lowest-cost path between any two nodes, where the cost of a path equals the sum of the costs of all the edges that make up the path. For a simple network like the one in Figure 3.28, you could imagine just calculating all the shortest paths and



■ FIGURE 3.28 Network represented as a graph.

<sup>11</sup>In the example networks (graphs) used throughout this chapter, we use undirected edges and assign each edge a single cost. This is actually a slight simplification. It is more accurate to make the edges directed, which typically means that there would be a pair of edges between each node—one flowing in each direction, and each with its own edge cost.

loading them into some nonvolatile storage on each node. Such a static approach has several shortcomings:

- It does not deal with node or link failures.
- It does not consider the addition of new nodes or links.
- It implies that edge costs cannot change, even though we might reasonably wish to have link costs change over time (e.g., assigning high cost to a link that is heavily loaded).

For these reasons, routing is achieved in most practical networks by running routing protocols among the nodes. These protocols provide a distributed, dynamic way to solve the problem of finding the lowest-cost path in the presence of link and node failures and changing edge costs. Note the word *distributed* in the previous sentence; it is difficult to make centralized solutions scalable, so all the widely used routing protocols use distributed algorithms.<sup>12</sup>

The distributed nature of routing algorithms is one of the main reasons why this has been such a rich field of research and development—there are a lot of challenges in making distributed algorithms work well. For example, distributed algorithms raise the possibility that two routers will at one instant have different ideas about the shortest path to some destination. In fact, each one may think that the other one is closer to the destination and decide to send packets to the other one. Clearly, such packets will be stuck in a loop until the discrepancy between the two routers is resolved, and it would be good to resolve it as soon as possible. This is just one example of the type of problem routing protocols must address.

To begin our analysis, we assume that the edge costs in the network are known. We will examine the two main classes of routing protocols: *distance vector* and *link state*. In Section 3.3.4, we return to the problem of calculating edge costs in a meaningful way.

### 3.3.2 Distance-Vector (RIP)

The idea behind the distance-vector algorithm is suggested by its name.<sup>13</sup> Each node constructs a one-dimensional array (a vector) containing the “distances” (costs) to all other nodes and distributes that vector to its

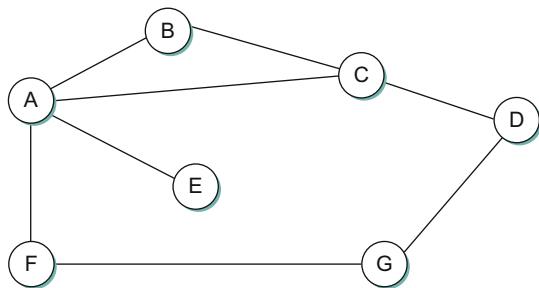


<sup>12</sup>This widely held assumption, however, has been re-examined in recent years—see the Further Reading section.

<sup>13</sup>The other common name for this class of algorithm is Bellman-Ford, after its inventors.

immediate neighbors. The starting assumption for distance-vector routing is that each node knows the cost of the link to each of its directly connected neighbors. These costs may be provided when the router is configured by a network manager. A link that is down is assigned an infinite cost.

To see how a distance-vector routing algorithm works, it is easiest to consider an example like the one depicted in Figure 3.29. In this example, the cost of each link is set to 1, so that a least-cost path is simply the one with the fewest hops. (Since all edges have the same cost, we do not show the costs in the graph.) We can represent each node's knowledge about the distances to all other nodes as a table like Table 3.10. Note that each



■ FIGURE 3.29 Distance-vector routing: an example network.

**Table 3.10 Initial Distances Stored at Each Node (Global View)**

Information Stored at Node	Distance to Reach Node						
	A	B	C	D	E	F	G
A	0	1	1	$\infty$	1	1	$\infty$
B	1	0	1	$\infty$	$\infty$	$\infty$	$\infty$
C	1	1	0	1	$\infty$	$\infty$	$\infty$
D	$\infty$	$\infty$	1	0	$\infty$	$\infty$	1
E	1	$\infty$	$\infty$	$\infty$	0	$\infty$	$\infty$
F	1	$\infty$	$\infty$	$\infty$	$\infty$	0	1
G	$\infty$	$\infty$	$\infty$	1	$\infty$	1	0

node knows only the information in one row of the table (the one that bears its name in the left column). The global view that is presented here is not available at any single point in the network.

We may consider each row in Table 3.10 as a list of distances from one node to all other nodes, representing the current beliefs of that node. Initially, each node sets a cost of 1 to its directly connected neighbors and  $\infty$  to all other nodes. Thus, A initially believes that it can reach B in one hop and that D is unreachable. The routing table stored at A reflects this set of beliefs and includes the name of the next hop that A would use to reach any reachable node. Initially, then, A's routing table would look like Table 3.11.

The next step in distance-vector routing is that every node sends a message to its directly connected neighbors containing its personal list of distances. For example, node F tells node A that it can reach node G at a cost of 1; A also knows it can reach F at a cost of 1, so it adds these costs to get the cost of reaching G by means of F. This total cost of 2 is less than the current cost of infinity, so A records that it can reach G at a cost of 2 by going through F. Similarly, A learns from C that D can be reached from C at a cost of 1; it adds this to the cost of reaching C (1) and decides that D can be reached via C at a cost of 2, which is better than the old cost of infinity. At the same time, A learns from C that B can be reached from C at a cost of 1, so it concludes that the cost of reaching B via C is 2. Since this is worse than the current cost of reaching B (1), this new information is ignored.

**Table 3.11 Initial Routing Table at Node A**

Destination	Cost	NextHop
B	1	B
C	1	C
D	$\infty$	—
E	1	E
F	1	F
G	$\infty$	—

At this point, A can update its routing table with costs and next hops for all nodes in the network. The result is shown in Table 3.12.

In the absence of any topology changes, it takes only a few exchanges of information between neighbors before each node has a complete routing table. The process of getting consistent routing information to all the nodes is called *convergence*. Table 3.13 shows the final set of costs from each node to all other nodes when routing has converged. We must stress that there is no one node in the network that has all the information in this table—each node only knows about the contents of its own routing table. The beauty of a distributed algorithm like this is that it enables all

**Table 3.12 Final Routing Table at Node A**

Destination	Cost	NextHop
B	1	B
C	1	C
D	2	C
E	1	E
F	1	F
G	2	F

**Table 3.13 Final Distances Stored at Each Node (Global View)**

Information Stored at Node	Distance to Reach Node						
	A	B	C	D	E	F	G
A	0	1	1	2	1	1	2
B	1	0	1	2	2	2	3
C	1	1	0	1	2	2	2
D	2	2	1	0	3	2	1
E	1	2	2	3	0	2	3
F	1	2	2	2	2	0	1
G	2	3	2	1	3	1	0

nodes to achieve a consistent view of the network in the absence of any centralized authority.

There are a few details to fill in before our discussion of distance-vector routing is complete. First we note that there are two different circumstances under which a given node decides to send a routing update to its neighbors. One of these circumstances is the *periodic* update. In this case, each node automatically sends an update message every so often, even if nothing has changed. This serves to let the other nodes know that this node is still running. It also makes sure that they keep getting information that they may need if their current routes become unviable. The frequency of these periodic updates varies from protocol to protocol, but it is typically on the order of several seconds to several minutes. The second mechanism, sometimes called a *triggered* update, happens whenever a node notices a link failure or receives an update from one of its neighbors that causes it to change one of the routes in its routing table. Whenever a node's routing table changes, it sends an update to its neighbors, which may lead to a change in their tables, causing them to send an update to their neighbors.

Now consider what happens when a link or node fails. The nodes that notice first send new lists of distances to their neighbors, and normally the system settles down fairly quickly to a new state. As to the question of how a node detects a failure, there are a couple of different answers. In one approach, a node continually tests the link to another node by sending a control packet and seeing if it receives an acknowledgment. In another approach, a node determines that the link (or the node at the other end of the link) is down if it does not receive the expected periodic routing update for the last few update cycles.

To understand what happens when a node detects a link failure, consider what happens when F detects that its link to G has failed. First, F sets its new distance to G to infinity and passes that information along to A. Since A knows that its 2-hop path to G is through F, A would also set its distance to G to infinity. However, with the next update from C, A would learn that C has a 2-hop path to G. Thus, A would know that it could reach G in 3 hops through C, which is less than infinity, and so A would update its table accordingly. When it advertises this to F, node F would learn that it can reach G at a cost of 4 through A, which is less than infinity, and the system would again become stable.

Unfortunately, slightly different circumstances can prevent the network from stabilizing. Suppose, for example, that the link from A to E goes down. In the next round of updates, A advertises a distance of infinity to E, but B and C advertise a distance of 2 to E. Depending on the exact timing of events, the following might happen: Node B, upon hearing that E can be reached in 2 hops from C, concludes that it can reach E in 3 hops and advertises this to A; node A concludes that it can reach E in 4 hops and advertises this to C; node C concludes that it can reach E in 5 hops; and so on. This cycle stops only when the distances reach some number that is large enough to be considered infinite. In the meantime, none of the nodes actually knows that E is unreachable, and the routing tables for the network do not stabilize. This situation is known as the *count to infinity* problem.

There are several partial solutions to this problem. The first one is to use some relatively small number as an approximation of infinity. For example, we might decide that the maximum number of hops to get across a certain network is never going to be more than 16, and so we could pick 16 as the value that represents infinity. This at least bounds the amount of time that it takes to count to infinity. Of course, it could also present a problem if our network grew to a point where some nodes were separated by more than 16 hops.

One technique to improve the time to stabilize routing is called *split horizon*. The idea is that when a node sends a routing update to its neighbors, it does not send those routes it learned from each neighbor back to that neighbor. For example, if B has the route (E, 2, A) in its table, then it knows it must have learned this route from A, and so whenever B sends a routing update to A, it does not include the route (E, 2) in that update. In a stronger variation of split horizon, called *split horizon with poison reverse*, B actually sends that route back to A, but it puts negative information in the route to ensure that A will not eventually use B to get to E. For example, B sends the route (E,  $\infty$ ) to A. The problem with both of these techniques is that they only work for routing loops that involve two nodes. For larger routing loops, more drastic measures are called for. Continuing the above example, if B and C had waited for a while after hearing of the link failure from A before advertising routes to E, they would have found that neither of them really had a route to E. Unfortunately, this approach delays the convergence of the protocol; speed of convergence is one of the key advantages of its competitor, link-state routing, the subject of Section 3.3.3.

### Implementation

The code that implements this algorithm is very straightforward; we give only some of the basics here. Structure `Route` defines each entry in the routing table, and constant `MAX_TTL` specifies how long an entry is kept in the table before it is discarded.

```
#define MAX_ROUTES      128      /* maximum size of routing table */
#define MAX_TTL          120      /* time (in seconds) until route expires */

typedef struct {
    NodeAddr   Destination;    /* address of destination */
    NodeAddr   NextHop;        /* address of next hop */
    int        Cost;           /* distance metric */
    u_short    TTL;            /* time to live */
} Route;

int      numRoutes = 0;
Route   routingTable[MAX_ROUTES];
```

The routine that updates the local node's routing table based on a new route is given by `mergeRoute`. Although not shown, a timer function periodically scans the list of routes in the node's routing table, decrements the TTL (time to live) field of each route, and discards any routes that have a time to live of 0. Notice, however, that the TTL field is reset to `MAX_TTL` any time the route is reconfirmed by an update message from a neighboring node.

```
void
mergeRoute (Route *new)
{
    int i;

    for (i = 0; i < numRoutes; ++i)
    {
        if (new->Destination == routingTable[i].Destination)
        {
            if (new->Cost + 1 < routingTable[i].Cost)
            {
                /* found a better route: */
                break;
            } else if (new->NextHop == routingTable[i].NextHop) {
```

```

        /* metric for current next-hop may have
           changed: */
        break;
    } else {
        /* route is uninteresting---just ignore
           it */
        return;
    }
}
if (i == numRoutes)
{
    /* this is a completely new route; is there room
       for it? */
    if (numRoutes < MAXROUTES)
    {
        ++numRoutes;
    } else {
        /* can't fit this route in table so give up */
        return;
    }
}
routingTable[i] = *new;
/* reset TTL */
routingTable[i].TTL = MAX_TTL;
/* account for hop to get to next node */
++routingTable[i].Cost;

}

```

Finally, the procedure `updateRoutingTable` is the main routine that calls `mergeRoute` to incorporate all the routes contained in a routing update that is received from a neighboring node.

```

void
updateRoutingTable (Route *newRoute, int numNewRoutes)
{
    int i;

    for (i=0; i < numNewRoutes; ++i)
    {

```

```
    mergeRoute(&newRoute[i]);  
}  
}
```

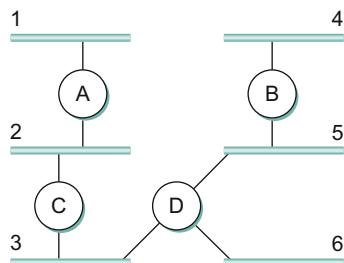
### Routing Information Protocol (RIP)

One of the more widely used routing protocols in IP networks is the Routing Information Protocol (RIP). Its widespread use in the early days of IP was due in no small part to the fact that it was distributed along with the popular Berkeley Software Distribution (BSD) version of Unix, from which many commercial versions of Unix were derived. It is also extremely simple. RIP is the canonical example of a routing protocol built on the distance-vector algorithm just described.

Routing protocols in internetworks differ very slightly from the idealized graph model described above. In an internetwork, the goal of the routers is to learn how to forward packets to various *networks*. Thus, rather than advertising the cost of reaching other routers, the routers advertise the cost of reaching networks. For example, in Figure 3.30, router C would advertise to router A the fact that it can reach networks 2 and 3 (to which it is directly connected) at a cost of 0, networks 5 and 6 at cost 1, and network 4 at cost 2.

We can see evidence of this in the RIP (version 2) packet format in Figure 3.31. The majority of the packet is taken up with  $\langle$ address, mask, distance $\rangle$  triples. However, the principles of the routing algorithm are just the same. For example, if router A learns from router B that network X can be reached at a lower cost via B than via the existing next hop in the routing table, A updates the cost and next hop information for the network number accordingly.

RIP is in fact a fairly straightforward implementation of distance-vector routing. Routers running RIP send their advertisements every 30 seconds;



■ FIGURE 3.30 Example network running RIP.

0	8	16	31
Command	Version	Must be zero	
Family of net 1		Route Tags	
Address prefix of net 1			
Mask of net 1			
Distance to net 1			
Family of net 2		Route Tags	
Address prefix of net 2			
Mask of net 2			
Distance to net 2			

■ FIGURE 3.31 RIPv2 packet format.

a router also sends an update message whenever an update from another router causes it to change its routing table. One point of interest is that it supports multiple address families, not just IP—that is the reason for the Family part of the advertisements. RIP version 2 (RIPv2) also introduced the subnet masks described in Section 3.2.5, whereas RIP version 1 worked with the old classful addresses of IP.

As we will see below, it is possible to use a range of different metrics or costs for the links in a routing protocol. RIP takes the simplest approach, with all link costs being equal to 1, just as in our example above. Thus, it always tries to find the minimum hop route. Valid distances are 1 through 15, with 16 representing infinity. This also limits RIP to running on fairly small networks—those with no paths longer than 15 hops.



### 3.3.3 Link State (OSPF)

Link-state routing is the second major class of intradomain routing protocol. The starting assumptions for link-state routing are rather similar to those for distance-vector routing. Each node is assumed to be capable of finding out the state of the link to its neighbors (up or down) and the cost of each link. Again, we want to provide each node with enough information to enable it to find the least-cost path to any destination. The basic

idea behind link-state protocols is very simple: Every node knows how to reach its directly connected neighbors, and if we make sure that the totality of this knowledge is disseminated to every node, then every node will have enough knowledge of the network to build a complete map of the network. This is clearly a sufficient condition (although not a necessary one) for finding the shortest path to any point in the network. Thus, link-state routing protocols rely on two mechanisms: reliable dissemination of link-state information, and the calculation of routes from the sum of all the accumulated link-state knowledge.

### *Reliable Flooding*

*Reliable flooding* is the process of making sure that all the nodes participating in the routing protocol get a copy of the link-state information from all the other nodes. As the term *flooding* suggests, the basic idea is for a node to send its link-state information out on all of its directly connected links; each node that receives this information then forwards it out on all of *its* links. This process continues until the information has reached all the nodes in the network.

More precisely, each node creates an update packet, also called a *link-state packet* (LSP), which contains the following information:

- The ID of the node that created the LSP
- A list of directly connected neighbors of that node, with the cost of the link to each one
- A sequence number
- A time to live for this packet

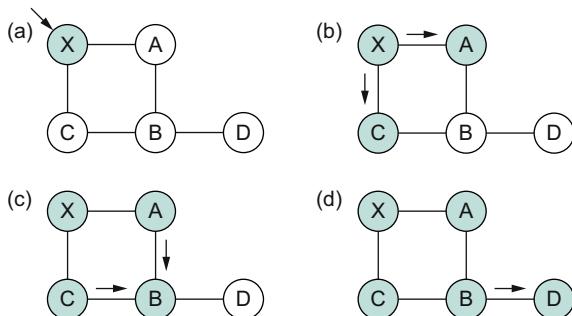
The first two items are needed to enable route calculation; the last two are used to make the process of flooding the packet to all nodes reliable. Reliability includes making sure that you have the most recent copy of the information, since there may be multiple, contradictory LSPs from one node traversing the network. Making the flooding reliable has proven to be quite difficult. (For example, an early version of link-state routing used in the ARPANET caused that network to fail in 1981.)

Flooding works in the following way. First, the transmission of LSPs between adjacent routers is made reliable using acknowledgments and retransmissions just as in the reliable link-layer protocol described in Section 2.5. However, several more steps are necessary to reliably flood an LSP to all nodes in a network.

Consider a node X that receives a copy of an LSP that originated at some other node Y. Note that Y may be any other router in the same routing domain as X. X checks to see if it has already stored a copy of an LSP from Y. If not, it stores the LSP. If it already has a copy, it compares the sequence numbers; if the new LSP has a larger sequence number, it is assumed to be the more recent, and that LSP is stored, replacing the old one. A smaller (or equal) sequence number would imply an LSP older (or not newer) than the one stored, so it would be discarded and no further action would be needed. If the received LSP was the newer one, X then sends a copy of that LSP to all of its neighbors except the neighbor from which the LSP was just received. The fact that the LSP is not sent back to the node from which it was received helps to bring an end to the flooding of an LSP. Since X passes the LSP on to all its neighbors, who then turn around and do the same thing, the most recent copy of the LSP eventually reaches all nodes.

Figure 3.32 shows an LSP being flooded in a small network. Each node becomes shaded as it stores the new LSP. In Figure 3.32(a) the LSP arrives at node X, which sends it to neighbors A and C in Figure 3.32(b). A and C do not send it back to X, but send it on to B. Since B receives two identical copies of the LSP, it will accept whichever arrived first and ignore the second as a duplicate. It then passes the LSP onto D, which has no neighbors to flood it to, and the process is complete.

Just as in RIP, each node generates LSPs under two circumstances. Either the expiry of a periodic timer or a change in topology can cause a node to generate a new LSP. However, the only topology-based reason for a node to generate an LSP is if one of its directly connected links or



**FIGURE 3.32** Flooding of link-state packets: (a) LSP arrives at node X; (b) X floods LSP to A and C; (c) A and C flood LSP to B (but not X); (d) flooding is complete.

immediate neighbors has gone down. The failure of a link can be detected in some cases by the link-layer protocol. The demise of a neighbor or loss of connectivity to that neighbor can be detected using periodic “hello” packets. Each node sends these to its immediate neighbors at defined intervals. If a sufficiently long time passes without receipt of a “hello” from a neighbor, the link to that neighbor will be declared down, and a new LSP will be generated to reflect this fact.

One of the important design goals of a link-state protocol’s flooding mechanism is that the newest information must be flooded to all nodes as quickly as possible, while old information must be removed from the network and not allowed to circulate. In addition, it is clearly desirable to minimize the total amount of routing traffic that is sent around the network; after all, this is just overhead from the perspective of those who actually use the network for their applications. The next few paragraphs describe some of the ways that these goals are accomplished.

One easy way to reduce overhead is to avoid generating LSPs unless absolutely necessary. This can be done by using very long timers—often on the order of hours—for the periodic generation of LSPs. Given that the flooding protocol is truly reliable when topology changes, it is safe to assume that messages saying “nothing has changed” do not need to be sent very often.

To make sure that old information is replaced by newer information, LSPs carry sequence numbers. Each time a node generates a new LSP, it increments the sequence number by 1. Unlike most sequence numbers used in protocols, these sequence numbers are not expected to wrap, so the field needs to be quite large (say, 64 bits). If a node goes down and then comes back up, it starts with a sequence number of 0. If the node was down for a long time, all the old LSPs for that node will have timed out (as described below); otherwise, this node will eventually receive a copy of its own LSP with a higher sequence number, which it can then increment and use as its own sequence number. This will ensure that its new LSP replaces any of its old LSPs left over from before the node went down.

LSPs also carry a time to live. This is used to ensure that old link-state information is eventually removed from the network. A node always decrements the TTL of a newly received LSP before flooding it to its neighbors. It also “ages” the LSP while it is stored in the node. When the TTL reaches 0, the node refloods the LSP with a TTL of 0, which is interpreted by all the nodes in the network as a signal to delete that LSP.

### Route Calculation

Once a given node has a copy of the LSP from every other node, it is able to compute a complete map for the topology of the network, and from this map it is able to decide the best route to each destination. The question, then, is exactly how it calculates routes from this information. The solution is based on a well-known algorithm from graph theory—Dijkstra's shortest-path algorithm.

We first define Dijkstra's algorithm in graph-theoretic terms. Imagine that a node takes all the LSPs it has received and constructs a graphical representation of the network, in which  $N$  denotes the set of nodes in the graph,  $l(i, j)$  denotes the nonnegative cost (weight) associated with the edge between nodes  $i, j \in N$  and  $l(i, j) = \infty$  if no edge connects  $i$  and  $j$ . In the following description, we let  $s \in N$  denote this node, that is, the node executing the algorithm to find the shortest path to all the other nodes in  $N$ . Also, the algorithm maintains the following two variables:  $M$  denotes the set of nodes incorporated so far by the algorithm, and  $C(n)$  denotes the cost of the path from  $s$  to each node  $n$ . Given these definitions, the algorithm is defined as follows:

```
 $M = \{s\}$ 
for each  $n$  in  $N - \{s\}$ 
   $C(n) = l(s, n)$ 
while ( $N \neq M$ )
   $M = M \cup \{w\}$  such that  $C(w)$  is the minimum for all  $w$  in  $(N - M)$ 
  for each  $n$  in  $(N - M)$ 
     $C(n) = \text{MIN}(C(n), C(w) + l(w, n))$ 
```

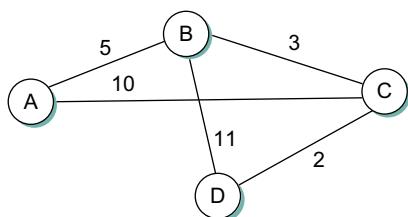
Basically, the algorithm works as follows. We start with  $M$  containing this node  $s$  and then initialize the table of costs (the  $C(n)$ s) to other nodes using the known costs to directly connected nodes. We then look for the node that is reachable at the lowest cost ( $w$ ) and add it to  $M$ . Finally, we update the table of costs by considering the cost of reaching nodes through  $w$ . In the last line of the algorithm, we choose a new route to node  $n$  that goes through node  $w$  if the total cost of going from the source to  $w$  and then following the link from  $w$  to  $n$  is less than the old route we had to  $n$ . This procedure is repeated until all nodes are incorporated in  $M$ .

In practice, each switch computes its routing table directly from the LSPs it has collected using a realization of Dijkstra's algorithm called the *forward search* algorithm. Specifically, each switch maintains two lists,

known as Tentative and Confirmed. Each of these lists contains a set of entries of the form (Destination, Cost, NextHop). The algorithm works as follows:

1. Initialize the Confirmed list with an entry for myself; this entry has a cost of 0.
2. For the node just added to the Confirmed list in the previous step, call it node **Next** and select its LSP.
3. For each neighbor (**Neighbor**) of **Next**, calculate the cost (**Cost**) to reach this **Neighbor** as the sum of the cost from myself to **Next** and from **Next** to **Neighbor**.
  - (a) If **Neighbor** is currently on neither the Confirmed nor the Tentative list, then add (**Neighbor**, **Cost**, **NextHop**) to the Tentative list, where **NextHop** is the direction I go to reach **Next**.
  - (b) If **Neighbor** is currently on the Tentative list, and the **Cost** is less than the currently listed cost for **Neighbor**, then replace the current entry with (**Neighbor**, **Cost**, **NextHop**), where **NextHop** is the direction I go to reach **Next**.
4. If the Tentative list is empty, stop. Otherwise, pick the entry from the Tentative list with the lowest cost, move it to the Confirmed list, and return to step 2.

This will become a lot easier to understand when we look at an example. Consider the network depicted in Figure 3.33. Note that, unlike our previous example, this network has a range of different edge costs. Table 3.14 traces the steps for building the routing table for node D. We denote the two outputs of D by using the names of the nodes to which they connect, B and C. Note the way the algorithm seems to head off on



■ FIGURE 3.33 Link-state routing: an example network.

**Table 3.14 Steps for Building Routing Table for Node D (Figure 3.33)**

<b>Step</b>	<b>Confirmed</b>	<b>Tentative</b>	<b>Comments</b>
1	(D,0,-)		Since D is the only new member of the confirmed list, look at its LSP.
2	(D,0,-)	(B,11,B) (C,2,C)	D's LSP says we can reach B through B at cost 11, which is better than anything else on either list, so put it on Tentative list; same for C.
3	(D,0,-) (C,2,C)	(B,11,B)	Put lowest-cost member of Tentative (C) onto Confirmed list. Next, examine LSP of newly confirmed member (C).
4	(D,0,-) (C,2,C)	(B,5,C) (A,12,C)	Cost to reach B through C is 5, so replace (B,11,B). C's LSP tells us that we can reach A at cost 12.
5	(D,0,-) (C,2,C) (B,5,C)	(A,12,C)	Move lowest-cost member of Tentative (B) to Confirmed, then look at its LSP.
6	(D,0,-) (C,2,C) (B,5,C)	(A,10,C)	Since we can reach A at cost 5 through B, replace the Tentative entry.
7	(D,0,-) (C,2,C) (B,5,C) (A,10,C)		Move lowest-cost member of Tentative (A) to Confirmed, and we are all done.

false leads (like the 11-unit cost path to B that was the first addition to the Tentative list) but ends up with the least-cost paths to all nodes.

The link-state routing algorithm has many nice properties: It has been proven to stabilize quickly, it does not generate much traffic, and it responds rapidly to topology changes or node failures. On the downside, the amount of information stored at each node (one LSP for every other node in the network) can be quite large. This is one of the fundamental problems of routing and is an instance of the more general problem of scalability. Some solutions to both the specific problem (the amount of storage potentially required at each node) and the general problem (scalability) will be discussed in the next section.



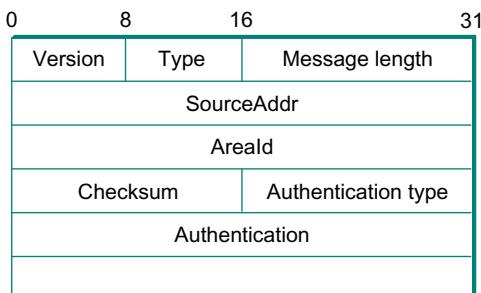
The difference between the distance-vector and link-state algorithms can be summarized as follows. In distance-vector, each node talks only to its directly connected neighbors, but it tells them everything it has learned (i.e., distance to all nodes). In link-state, each node talks to all other nodes, but it tells them only what it knows for sure (i.e., only the state of its directly connected links).

### The Open Shortest Path First Protocol (OSPF)

One of the most widely used link-state routing protocols is OSPF. The first word, “Open,” refers to the fact that it is an open, nonproprietary standard, created under the auspices of the Internet Engineering Task Force (IETF). The “SPF” part comes from an alternative name for link-state routing. OSPF adds quite a number of features to the basic link-state algorithm described above, including the following:

- *Authentication of routing messages*—One feature of distributed routing algorithms is that they disperse information from one node to many other nodes, and the entire network can thus be impacted by bad information from one node. For this reason, it’s a good idea to be sure that all the nodes taking part in the protocol can be trusted. Authenticating routing messages helps achieve this. Early versions of OSPF used a simple 8-byte password for authentication. This is not a strong enough form of authentication to prevent dedicated malicious users, but it alleviates some problems caused by misconfiguration or casual attacks. (A similar form of authentication was added to RIP in version 2.) Strong cryptographic authentication of the sort discussed in [Section 8.3](#) was later added.
- *Additional hierarchy*—Hierarchy is one of the fundamental tools used to make systems more scalable. OSPF introduces another layer of hierarchy into routing by allowing a domain to be partitioned into *areas*. This means that a router within a domain does not necessarily need to know how to reach every network within that domain—it may be able to get by knowing only how to get to the right area. Thus, there is a reduction in the amount of information that must be transmitted to and stored in each node. We examine areas in detail in [Section 4.1.1](#).
- *Load balancing*—OSPF allows multiple routes to the same place to be assigned the same cost and will cause traffic to be distributed evenly over those routes, thus making better use of available network capacity.

There are several different types of OSPF messages, but all begin with the same header, as shown in [Figure 3.34](#). The Version field is currently set to 2, and the Type field may take the values 1 through 5. The SourceAddr identifies the sender of the message, and the Areald is a 32-bit identifier



■ FIGURE 3.34 OSPF header format.

of the area in which the node is located. The entire packet, except the authentication data, is protected by a 16-bit checksum using the same algorithm as the IP header (see Section 2.4). The Authentication type is 0 if no authentication is used; otherwise, it may be 1, implying that a simple password is used, or 2, which indicates that a cryptographic authentication checksum, of the sort described in Section 8.3, is used. In the latter cases, the Authentication field carries the password or cryptographic checksum.

Of the five OSPF message types, type 1 is the “hello” message, which a router sends to its peers to notify them that it is still alive and connected as described above. The remaining types are used to request, send, and acknowledge the receipt of link-state messages. The basic building block of link-state messages in OSPF is the link-state advertisement (LSA). One message may contain many LSAs. We provide a few details of the LSA here.

Like any internetwork routing protocol, OSPF must provide information about how to reach networks. Thus, OSPF must provide a little more information than the simple graph-based protocol described above. Specifically, a router running OSPF may generate link-state packets that advertise one or more of the networks that are directly connected to that router. In addition, a router that is connected to another router by some link must advertise the cost of reaching that router over the link. These two types of advertisements are necessary to enable all the routers in a domain to determine the cost of reaching all networks in that domain and the appropriate next hop for each network.

Figure 3.35 shows the packet format for a type 1 link-state advertisement. Type 1 LSAs advertise the cost of links between routers. Type 2

LS Age	Options	Type = 1
Link-state ID		
Advertising router		
LS sequence number		
LS checksum	Length	
0	Flags	0
Number of links		
Link ID		
Link data		
Link type	Num_TOS	Metric
Optional TOS information		
More links		

■ FIGURE 3.35 OSPF link-state advertisement.

LSAs are used to advertise networks to which the advertising router is connected, while other types are used to support additional hierarchy as described in the next section. Many fields in the LSA should be familiar from the preceding discussion. The LS Age is the equivalent of a time to live, except that it counts up and the LSA expires when the age reaches a defined maximum value. The Type field tells us that this is a type 1 LSA.

In a type 1 LSA, the Link state ID and the Advertising router field are identical. Each carries a 32-bit identifier for the router that created this LSA. While a number of assignment strategies may be used to assign this ID, it is essential that it be unique in the routing domain and that a given router consistently uses the same router ID. One way to pick a router ID that meets these requirements would be to pick the lowest IP address among all the IP addresses assigned to that router. (Recall that a router may have a different IP address on each of its interfaces.)

The LS sequence number is used exactly as described above to detect old or duplicate LSAs. The LS checksum is similar to others we have seen in Section 2.4 and in other protocols; it is, of course, used to verify that data has not been corrupted. It covers all fields in the packet except LS Age, so it is not necessary to recompute a checksum every time LS Age is incremented. Length is the length in bytes of the complete LSA.

Now we get to the actual link-state information. This is made a little complicated by the presence of TOS (type of service) information. Ignoring that for a moment, each link in the LSA is represented by a Link ID, some Link Data, and a metric. The first two of these fields identify the link;

a common way to do this would be to use the router ID of the router at the far end of the link as the Link ID and then use the Link Data to disambiguate among multiple parallel links if necessary. The metric is of course the cost of the link. Type tells us something about the link—for example, if it is a point-to-point link.

The TOS information is present to allow OSPF to choose different routes for IP packets based on the value in their TOS field. Instead of assigning a single metric to a link, it is possible to assign different metrics depending on the TOS value of the data. For example, if we had a link in our network that was very good for delay-sensitive traffic, we could give it a low metric for the TOS value representing low delay and a high metric for everything else. OSPF would then pick a different shortest path for those packets that had their TOS field set to that value. It is worth noting that, at the time of writing, this capability has not been widely deployed.<sup>14</sup>

### 3.3.4 Metrics

The preceding discussion assumes that link costs, or metrics, are known when we execute the routing algorithm. In this section, we look at some ways to calculate link costs that have proven effective in practice. One example that we have seen already, which is quite reasonable and very simple, is to assign a cost of 1 to all links—the least-cost route will then be the one with the fewest hops. Such an approach has several drawbacks, however. First, it does not distinguish between links on a latency basis. Thus, a satellite link with 250-ms latency looks just as attractive to the routing protocol as a terrestrial link with 1-ms latency. Second, it does not distinguish between routes on a capacity basis, making a 9.6-kbps link look just as good as a 45-Mbps link. Finally, it does not distinguish between links based on their current load, making it impossible to route around overloaded links. It turns out that this last problem is the hardest because you are trying to capture the complex and dynamic characteristics of a link in a single scalar cost.

The ARPANET was the testing ground for a number of different approaches to link-cost calculation. (It was also the place where the superior stability of link-state over distance-vector routing was demonstrated;

---

<sup>14</sup>Note also that the meaning of the TOS field has changed since the OSPF specification was written. This topic is discussed in Section 6.5.3.

the original mechanism used distance vector while the later version used link state.) The following discussion traces the evolution of the ARPANET routing metric and, in so doing, explores the subtle aspects of the problem.

The original ARPANET routing metric measured the number of packets that were queued waiting to be transmitted on each link, meaning that a link with 10 packets queued waiting to be transmitted was assigned a larger cost weight than a link with 5 packets queued for transmission. Using queue length as a routing metric did not work well, however, since queue length is an artificial measure of load—it moves packets toward the shortest queue rather than toward the destination, a situation all too familiar to those of us who hop from line to line at the grocery store. Stated more precisely, the original ARPANET routing mechanism suffered from the fact that it did not take either the bandwidth or the latency of the link into consideration.

A second version of the ARPANET routing algorithm, sometimes called the *new routing mechanism*, took both link bandwidth and latency into consideration and used delay, rather than just queue length, as a measure of load. This was done as follows. First, each incoming packet was timestamped with its time of arrival at the router (*ArrivalTime*); its departure time from the router (*DepartTime*) was also recorded. Second, when the link-level ACK was received from the other side, the node computed the delay for that packet as

$$\text{Delay} = (\text{DepartTime} - \text{ArrivalTime}) + \text{TransmissionTime} + \text{Latency}$$

where *TransmissionTime* and *Latency* were statically defined for the link and captured the link's bandwidth and latency, respectively. Notice that in this case, *DepartTime* – *ArrivalTime* represents the amount of time the packet was delayed (queued) in the node due to load. If the ACK did not arrive, but instead the packet timed out, then *DepartTime* was reset to the time the packet was *retransmitted*. In this case, *DepartTime* – *ArrivalTime* captures the reliability of the link—the more frequent the retransmission of packets, the less reliable the link, and the more we want to avoid it. Finally, the weight assigned to each link was derived from the average delay experienced by the packets recently sent over that link.

Although an improvement over the original mechanism, this approach also had a lot of problems. Under light load, it worked reasonably well,

since the two static factors of delay dominated the cost. Under heavy load, however, a congested link would start to advertise a very high cost. This caused all the traffic to move off that link, leaving it idle, so then it would advertise a low cost, thereby attracting back all the traffic, and so on. The effect of this instability was that, under heavy load, many links would in fact spend a great deal of time being idle, which is the last thing you want under heavy load.

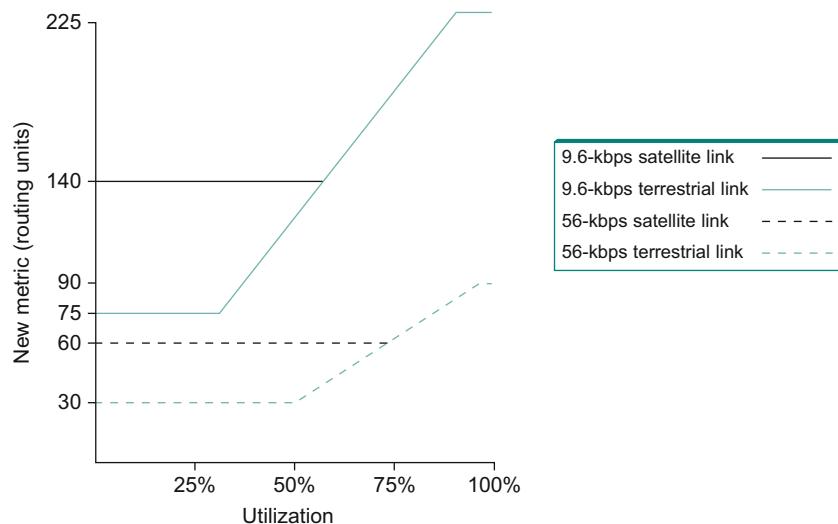
Another problem was that the range of link values was much too large. For example, a heavily loaded 9.6-kbps link could look 127 times more costly than a lightly loaded 56-kbps link. This means that the routing algorithm would choose a path with 126 hops of lightly loaded 56-kbps links in preference to a 1-hop 9.6-kbps path. While shedding some traffic from an overloaded line is a good idea, making it look so unattractive that it loses all its traffic is excessive. Using 126 hops when 1 hop will do is in general a bad use of network resources. Also, satellite links were unduly penalized, so that an idle 56-kbps satellite link looked considerably more costly than an idle 9.6-kbps terrestrial link, even though the former would give better performance for high-bandwidth applications.

A third approach, called the “revised ARPANET routing metric,” addressed these problems. The major changes were to compress the dynamic range of the metric considerably, to account for the link type, and to smooth the variation of the metric with time.

The smoothing was achieved by several mechanisms. First, the delay measurement was transformed to a link utilization, and this number was averaged with the last reported utilization to suppress sudden changes. Second, there was a hard limit on how much the metric could change from one measurement cycle to the next. By smoothing the changes in the cost, the likelihood that all nodes would abandon a route at once is greatly reduced.

The compression of the dynamic range was achieved by feeding the measured utilization, the link type, and the link speed into a function that is shown graphically in Figure 3.36. Observe the following:

- A highly loaded link never shows a cost of more than three times its cost when idle.
- The most expensive link is only seven times the cost of the least expensive.



■ FIGURE 3.36 Revised ARPANET routing metric versus link utilization.

- A high-speed satellite link is more attractive than a low-speed terrestrial link.
- Cost is a function of link utilization only at moderate to high loads.

All of these factors mean that a link is much less likely to be universally abandoned, since a threefold increase in cost is likely to make the link unattractive for some paths while letting it remain the best choice for others. The slopes, offsets, and breakpoints for the curves in Figure 3.36 were arrived at by a great deal of trial and error, and they were carefully tuned to provide good performance.

We end our discussion of routing metrics with a dose of reality. In the majority of real-world network deployments at the time of writing, metrics change rarely if at all and only under the control of a network administrator, not automatically as was described above. The reason for this is partly that conventional wisdom now holds that dynamically changing metrics are too unstable, even though this probably need not be true. Perhaps more significantly, many networks today lack the great disparity of link speeds and latencies that prevailed in the ARPANET. Thus, static metrics are the norm. One common approach to setting metrics is to use a constant multiplied by  $(1/\text{link\_bandwidth})$ .

### Monitoring Routing Behavior

Given the complexity of routing packets through a network of the scale of the Internet, we might wonder how well the system works. We know it works some of the time because we are able to connect to sites all over the world. We suspect it doesn't work all the time, though, because sometimes we are unable to connect to certain sites. The real problem is determining what part of the system is at fault when our connections fail: Has some routing machinery failed to work properly, is the remote server too busy, or has some link or machine simply gone down?

This is really an issue of network management, and while there are tools that system administrators use to keep tabs on their own networks—for example, see the Simple Network Management Protocol (SNMP) described in [Section 9.3.2](#)—it is a largely unresolved problem for the Internet as a whole. In fact, the Internet has grown so large and complex that, even though it is constructed from a collection of man-made, largely deterministic parts, we have come to view it almost as a living organism or natural phenomenon that is to be studied. That is, we try to understand the Internet's dynamic behavior by performing experiments on it and proposing models that explain our observations.

An excellent example of this kind of study has been conducted by Vern Paxson. Paxson used the Unix traceroute tool to study 40,000 end-to-end routes between 37 Internet sites in 1995. He was attempting to answer questions about how routes fail, how stable routes are over time, and whether or not they are symmetric. Among other things, Paxson found that the likelihood of a user encountering a serious end-to-end routing problem was 1 in 30, and that such problems usually lasted about 30 seconds. He also found that two-thirds of the Internet's routes persisted for days or weeks, and that about one-third of the time the route used to get from host A to host B included at least one different routing domain than the route used to get from host B to host A. Paxson's overall conclusion was that Internet routing was becoming less and less predictable over time.

## 3.4 IMPLEMENTATION AND PERFORMANCE

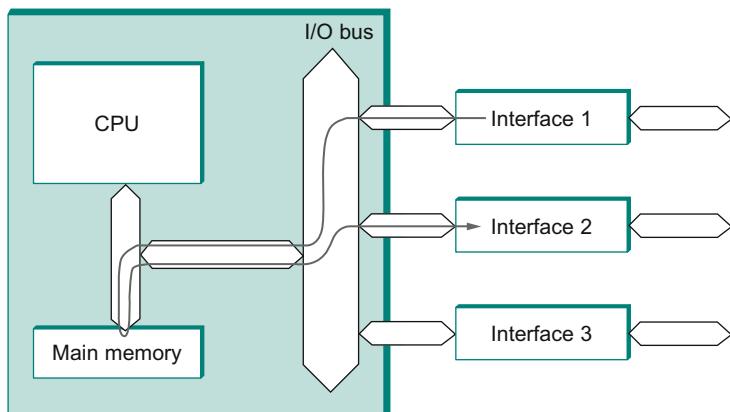
So far, we have talked about what switches and routers must do without discussing how to do it. There is a very simple way to build a switch or router: Buy a general-purpose processor and equip it with a number of network interfaces. Such a device, running suitable software, can receive packets on one of its interfaces, perform any of the switching or forwarding functions described above, and send packets out another of its

interfaces. This is, in fact, a popular way to build experimental routers and switches when you want to be able to do things like develop new routing protocols because it offers extreme flexibility and a familiar programming environment. It is also not too far removed from the architecture of many commercial mid- to low-end routers.

### 3.4.1 Switch Basics

Switches and routers use similar implementation techniques, so we'll start this section by looking at those common techniques, then move on to look at the specific issues affecting router implementation in Section 3.4.4. For most of this section, we'll use the word *switch* to cover both types of devices, since their internal designs are so similar (and it's tedious to say "switch or router" all the time).

Figure 3.37 shows a processor with three network interfaces used as a switch. The figure shows a path that a packet might take from the time it arrives on interface 1 until it is output on interface 2. We have assumed here that the processor has a mechanism to move data directly from an interface to its main memory without having to be directly copied by the CPU, a technique called *direct memory access* (DMA). Once the packet is in memory, the CPU examines its header to determine which interface the packet should be sent out on. It then uses DMA to move the packet out to the appropriate interface. Note that Figure 3.37 does not show the packet



■ FIGURE 3.37 A general-purpose processor used as a packet switch.

going to the CPU because the CPU inspects only the header of the packet; it does not have to read every byte of data in the packet.

The main problem with using a general-purpose processor as a switch is that its performance is limited by the fact that all packets must pass through a single point of contention: In the example shown, each packet crosses the I/O bus twice and is written to and read from main memory once. The upper bound on aggregate throughput of such a device (the total sustainable data rate summed over all inputs) is, thus, either half the main memory bandwidth or half the I/O bus bandwidth, whichever is less. (Usually, it's the I/O bus bandwidth.) For example, a machine with a 133-MHz, 64-bit-wide I/O bus can transmit data at a peak rate of a little over 8 Gbps. Since forwarding a packet involves crossing the bus twice, the actual limit is 4 Gbps—enough to build a switch with a fair number of 100-Mbps Ethernet ports, for example, but hardly enough for a high-end router in the core of the Internet.

Moreover, this upper bound also assumes that moving data is the only problem—a fair approximation for long packets but a bad one when packets are short. In the latter case, the cost of processing each packet—parsing its header and deciding which output link to transmit it on—is likely to dominate. Suppose, for example, that a processor can perform all the necessary processing to switch 2 million packets each second. This is sometimes called the packet per second (pps) rate. (This number is representative of what is achievable on an inexpensive PC.) If the average packet is short, say, 64 bytes, this would imply

$$\begin{aligned}\text{Throughput} &= \text{pps} \times (\text{BitsPerPacket}) \\ &= 2 \times 10^6 \times 64 \times 8 \\ &= 1024 \times 10^6\end{aligned}$$

that is, a throughput of about 1 Gbps—substantially below the range that users are demanding from their networks today. Bear in mind that this 1 Gbps would be shared by all users connected to the switch, just as the bandwidth of a single (unswitched) Ethernet segment is shared among all users connected to the shared medium. Thus, for example, a 20-port switch with this aggregate throughput would only be able to cope with an average data rate of about 50 Mbps on each port.

To address this problem, hardware designers have come up with a large array of switch designs that reduce the amount of contention and provide

high aggregate throughput. Note that some contention is unavoidable: If every input has data to send to a single output, then they cannot all send it at once. However, if data destined for different outputs is arriving at different inputs, then a well-designed switch will be able to move data from inputs to outputs in parallel, thus increasing the aggregate throughput.

### Defining Throughput

It turns out to be difficult to define precisely the throughput of a switch. Intuitively, we might think that if a switch has  $n$  inputs that each support a link speed of  $s_i$ , then the throughput would just be the sum of all the  $s_i$ . This is actually the best possible throughput that such a switch could provide, but in practice almost no real switch can guarantee that level of performance. One reason for this is simple to understand. Suppose that, for some period of time, all the traffic arriving at the switch needed to be sent to the same output. As long as the bandwidth of that output is less than the sum of the input bandwidths, then some of the traffic will need to be either buffered or dropped. With this particular traffic pattern, the switch could not provide a sustained throughput higher than the link speed of that one output. However, a switch might be able to handle traffic arriving at the full link speed on all inputs if it is distributed across all the outputs evenly; this would be considered optimal.

Another factor that affects the performance of switches is the size of packets arriving on the inputs. For an ATM switch, this is normally not an issue because all “packets” (cells) are the same length. But, for Ethernet switches or IP routers, packets of widely varying sizes are possible. Some of the operations that a switch must perform have a constant overhead per packet, so a switch is likely to perform differently depending on whether all arriving packets are very short, very long, or mixed. For this reason, routers or switches that forward variable-length packets are often characterized by a *packet per second* rate as well as a throughput in bits per second. The pps rate is usually measured with minimum-sized packets.

The first thing to notice about this discussion is that the throughput of the switch is a function of the traffic to which it is subjected. One of the things that switch designers spend a lot of their time doing is trying to come up with traffic models that approximate the behavior of real data traffic. It turns out that it is extremely difficult to achieve accurate models. There are several elements to a traffic model. The main ones are (1) when the packets arrive, (2) what outputs they are destined for, and (3) how big they are.

Traffic modeling is a well-established science that has been extremely successful in the world of telephony, enabling telephone companies to engineer their networks to carry expected loads quite efficiently. This is partly because the way people use the phone network does not change that much

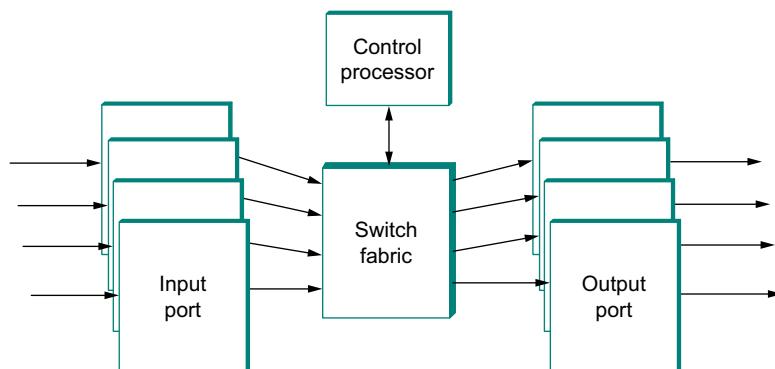
over time: The frequency with which calls are placed, the amount of time taken for a call, and the tendency of everyone to make calls on Mother's Day have stayed fairly constant for many years. By contrast, the rapid evolution of computer communications, where a new application like BitTorrent can change the traffic patterns almost overnight, has made effective modeling of computer networks much more difficult. Nevertheless, there are some excellent books and articles on the subject that we list at the end of the chapter.

To give you a sense of the range of throughputs that designers need to be concerned about, a single rack router used in the core of the Internet at the time of writing might support 16 OC-768 links for a throughput of approximately 640 Gbps. A 640-Gbps switch, if called upon to handle a steady stream of 64-byte packets, would need a packet per second rate of

$$640 \times 10^9 \div (64 \times 8) = 1.25 \times 10^9 \text{ pps}$$

### 3.4.2 Ports

Most switches look conceptually similar to the one shown in Figure 3.38. They consist of a number of *input* and *output ports* and a *fabric*. There is usually at least one control processor in charge of the whole switch that communicates with the ports either directly or, as shown here, via the switch fabric. The ports communicate with the outside world. They may contain fiber optic receivers and lasers, buffers to hold packets that are waiting to be switched or transmitted, and often a significant amount of other circuitry that enables the switch to function. The fabric has a very



■ FIGURE 3.38 A  $4 \times 4$  switch.

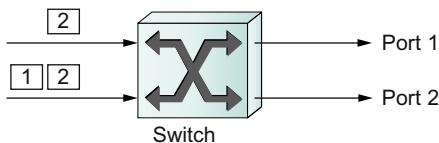
simple and well-defined job: When presented with a packet, deliver it to the right output port.

One of the jobs of the ports, then, is to deal with the complexity of the real world in such a way that the fabric can do its relatively simple job. For example, suppose that this switch is supporting a virtual circuit model of communication. In general, the virtual circuit mapping tables described in Section 3.1.2 are located in the ports. The ports maintain lists of virtual circuit identifiers that are currently in use, with information about what output a packet should be sent out on for each VCI and how the VCI needs to be remapped to ensure uniqueness on the outgoing link. Similarly, the ports of an Ethernet switch store tables that map between Ethernet addresses and output ports (bridge forwarding tables as described in Section 3.1.4). In general, when a packet is handed from an input port to the fabric, the port has figured out where the packet needs to go, and either the port sets up the fabric accordingly by communicating some control information to it, or it attaches enough information to the packet itself (e.g., an output port number) to allow the fabric to do its job automatically. Fabrics that switch packets by looking only at the information in the packet are referred to as *self-routing*, since they require no external control to route packets. An example of a self-routing fabric is discussed below.

The input port is the first place to look for performance bottlenecks. The input port has to receive a steady stream of packets, analyze information in the header of each one to determine which output port (or ports) the packet must be sent to, and pass the packet on to the fabric. The type of header analysis that it performs can range from a simple table lookup on a VCI to complex matching algorithms that examine many fields in the header. This is the type of operation that sometimes becomes a problem when the average packet size is very small. Consider, for example, 64-byte packets arriving on a port connected to an OC-48 (2.48 Gbps) link. Such a port needs to process packets at a rate of

$$2.48 \times 10^9 \div (64 \times 8) = 4.83 \times 10^6 \text{ pps}$$

In other words, when small packets are arriving as fast as possible on this link (the worst-case scenario that most ports are engineered to handle), the input port has approximately 200 nanoseconds to process each packet.



■ FIGURE 3.39 Simple illustration of head-of-line blocking.

Another key function of ports is buffering. Observe that buffering can happen in either the input or the output port; it can also happen within the fabric (sometimes called *internal buffering*). Simple input buffering has some serious limitations. Consider an input buffer implemented as a FIFO. As packets arrive at the switch, they are placed in the input buffer. The switch then tries to forward the packets at the front of each FIFO to their appropriate output port. However, if the packets at the front of several different input ports are destined for the same output port at the same time, then only one of them can be forwarded;<sup>15</sup> the rest must stay in their input buffers.

The drawback of this feature is that those packets left at the front of the input buffer prevent other packets further back in the buffer from getting a chance to go to their chosen outputs, even though there may be no contention for those outputs. This phenomenon is called *head-of-line blocking*. A simple example of head-of-line blocking is given in Figure 3.39, where we see a packet destined for port 1 blocked behind a packet contending for port 2. It can be shown that when traffic is uniformly distributed among outputs, head-of-line blocking limits the throughput of an input-buffered switch to 59% of the theoretical maximum (which is the sum of the link bandwidths for the switch). Thus, the majority of switches use either pure output buffering or a mixture of internal and output buffering. Those that do rely on input buffers use more advanced buffer management schemes to avoid head-of-line blocking.

Buffers actually perform a more complex task than just holding onto packets that are waiting to be transmitted. Buffers are the main source of delay in a switch, and also the place where packets are most likely to

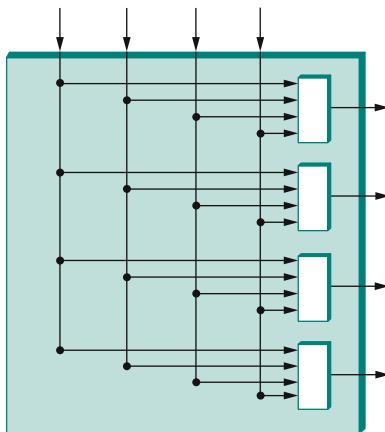
<sup>15</sup>For a simple input-buffered switch, exactly one packet at a time can be sent to a given output port. It is possible to design switches that can forward more than one packet to the same output at once, at a cost of higher switch complexity, but there is always some upper limit on the number.

get dropped due to lack of space to store them. The buffers therefore are the main place where the quality of service characteristics of a switch are determined. For example, if a certain packet has been sent along a VC that has a guaranteed delay, it cannot afford to sit in a buffer for very long. This means that the buffers, in general, must be managed using packet scheduling and discard algorithms that meet a wide range of QoS requirements. We talk more about these issues in [Chapter 6](#).

### 3.4.3 Fabrics

While there has been an abundance of impressive research conducted on the design of efficient and scalable fabrics, it is sufficient for our purposes here to understand only the high-level properties of a switch fabric. A switch fabric should be able to move packets from input ports to output ports with minimal delay and in a way that meets the throughput goals of the switch. That usually means that fabrics display some degree of parallelism. A high-performance fabric with  $n$  ports can often move one packet from each of its  $n$  ports to one of the output ports at the same time. A sample of fabric types includes the following:

- **Shared Bus**—This is the type of “fabric” found in a conventional processor used as a switch, as described above. Because the bus bandwidth determines the throughput of the switch, high-performance switches usually have specially designed busses rather than the standard busses found in PCs.
- **Shared Memory**—In a shared memory switch, packets are written into a memory location by an input port and then read from memory by the output ports. Here it is the memory bandwidth that determines switch throughput, so wide and fast memory is typically used in this sort of design. A shared memory switch is similar in principle to the shared bus switch, except it usually uses a specially designed, high-speed memory bus rather than an I/O bus.
- **Crossbar**—A crossbar switch is a matrix of pathways that can be configured to connect any input port to any output port. Figure 3.40 shows a  $4 \times 4$  crossbar switch. The main problem with crossbars is that, in their simplest form, they require each output port to be able to accept packets from all inputs at once, implying that each port would have a memory bandwidth equal to the total switch throughput. In reality, more complex designs are typically

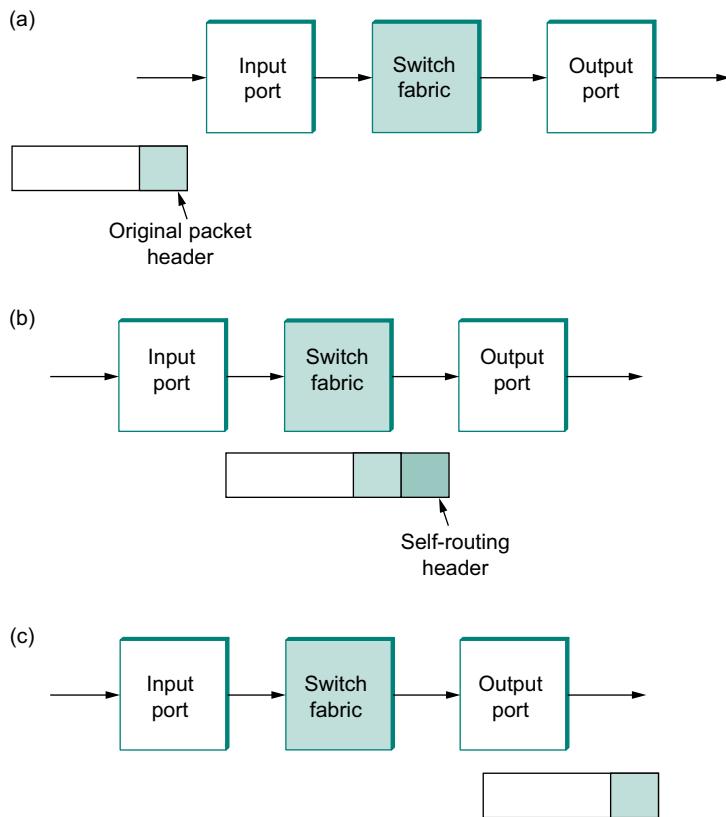


■ FIGURE 3.40 A  $4 \times 4$  crossbar switch.

used to address this issue (see, for example, the Knockout switch and McKeown’s virtual output-buffered approach in the Further Reading section.)

- *Self-routing*—As noted above, self-routing fabrics rely on some information in the packet header to direct each packet to its correct output. Usually a special “self-routing header” is appended to the packet by the input port after it has determined which output the packet needs to go to, as illustrated in Figure 3.41; this extra header is removed before the packet leaves the switch. Self-routing fabrics are often built from large numbers of very simple  $2 \times 2$  switching elements interconnected in regular patterns, such as the *banyan* switching fabric shown in Figure 3.42. For some examples of self-routing fabric designs, see the Further Reading section at the end of this chapter.

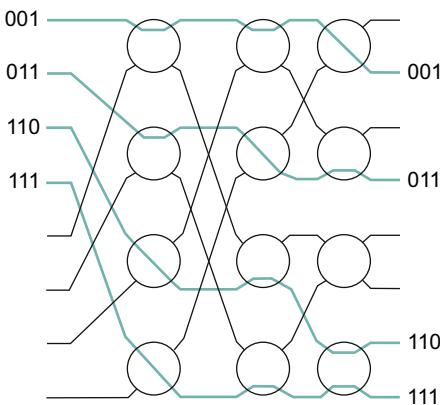
Self-routing fabrics are among the most scalable approaches to fabric design, and there has been a wealth of research on the topic, some of which is listed in the Further Reading section. Many self-routing fabrics resemble the one shown in Figure 3.42, consisting of regularly interconnected  $2 \times 2$  switching elements. For example, the  $2 \times 2$  switches in the banyan network perform a simple task: They look at 1 bit in each self-routing header and route packets toward the upper output if it is zero or toward the lower output if it is one. Obviously, if two packets arrive at a



**FIGURE 3.41** A self-routing header is applied to a packet at input to enable the fabric to send the packet to the correct output, where it is removed: (a) Packet arrives at input port; (b) input port attaches self-routing header to direct packet to correct output; (c) self-routing header is removed at output port before packet leaves switch.

banyan element at the same time and both have the bit set to the same value, then they want to be routed to the same output and a collision will occur. Either preventing or dealing with these collisions is a main challenge for self-routing switch design. The banyan network is a clever arrangement of  $2 \times 2$  switching elements that routes all packets to the correct output without collisions if the packets are presented in ascending order.

We can see how this works in an example, as shown in Figure 3.42, where the self-routing header contains the output port number encoded in binary. The switch elements in the first column look at the most significant bit of the output port number and route packets to the top if that bit



■ FIGURE 3.42 Routing packets through a banyan network. The 3-bit numbers represent values in the self-routing headers of four arriving packets.

is a 0 or the bottom if it is a 1. Switch elements in the second column look at the second bit in the header, and those in the last column look at the least significant bit. You can see from this example that the packets are routed to the correct destination port without collisions. Notice how the top outputs from the first column of switches all lead to the top half of the network, thus getting packets with port numbers 0 to 3 into the right half of the network. The next column gets packets to the right quarter of the network, and the final column gets them to the right output port. The clever part is the way switches are arranged to avoid collisions. Part of the arrangement includes the “perfect shuffle” wiring pattern at the start of the network. To build a complete switch fabric around a banyan network would require additional components to sort packets before they are presented to the banyan. The Batcher-banyan switch design is a notable example of such an approach. The Batcher network, which is also built from a regular interconnection of  $2 \times 2$  switching elements, sorts packets into descending order. On leaving the Batcher network, the packets are then ready to be directed to the correct output, with no risk of collisions, by the banyan network.

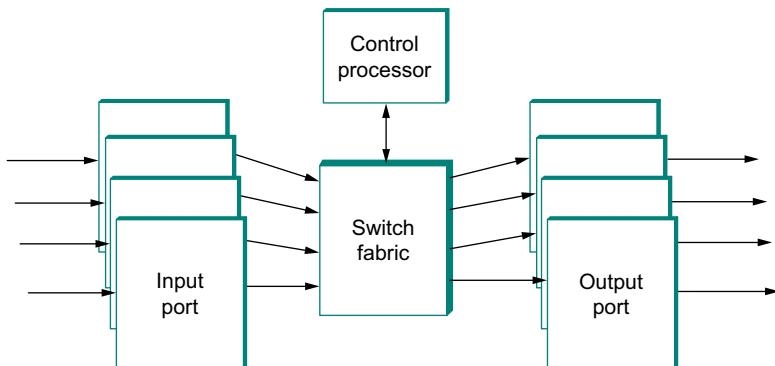
One of the interesting things about switch design is the wide range of different types of switches that can be built using the same basic technology. For example, Ethernet switches, ATM switches, and Internet routers, discussed below, have all been built using designs such as those outlined in this section.

### 3.4.4 Router Implementation

We have now seen a variety of ways to build a switch, ranging from a general-purpose processor with a suitable number of network interfaces to some sophisticated hardware designs. In general, the same range of options is available for building routers, many of which look something like Figure 3.43. The control processor is responsible for running the routing protocols discussed above, among other things, and generally acts as the central point of control of the router. The switching fabric transfers packets from one port to another, just as in a switch; and the ports provide a range of functionality to allow the router to interface to links of various types (e.g., Ethernet, SONET).

A few points are worth noting about router design and how it differs from switch design. First, routers must be designed to handle variable-length packets, a constraint that does not apply to ATM switches but is certainly applicable to Ethernet or Frame Relay switches. It turns out that many high-performance routers are designed using a switching fabric that is cell based. In such cases, the ports must be able to convert variable-length packets into cells and back again. This is known as *segmentation and re-assembly* (SAR), a problem also faced by network adaptors for ATM networks.

Another consequence of the variable length of IP datagrams is that it can be harder to characterize the performance of a router than a switch that forwards only cells. Routers can usually forward a certain number of packets per second, and this implies that the total throughput in *bits*



■ FIGURE 3.43 Block diagram of a router.

per second depends on packet size. Router designers generally have to make a choice as to what packet length they will support at *line rate*. That is, if *pps* (packets per second) is the rate at which packets arriving on a particular port can be forwarded, and *linerate* is the physical speed of the port in bits per second, then there will be some *packetsize* in bits such that:

$$\text{packetsize} \times \text{pps} = \text{linerate}$$

This is the packet size at which the router can forward at line rate; it is likely to be able to sustain line rate for longer packets but not for shorter packets. Sometimes a designer might decide that the right packet size to support is 40 bytes, since that is the minimum size of an IP packet that has a TCP header attached. Another choice might be the expected *average* packet size, which can be determined by studying traces of network traffic. For example, measurements of the Internet backbone suggest that the average IP packet is around 300 bytes long. However, such a router would fall behind and perhaps start dropping packets when faced with a long sequence of short packets, which is statistically likely from time to time and also very possible if the router is subject to an active attack (see Chapter 8). Design decisions of this type depend heavily on cost considerations and the intended application of the router.

When it comes to the task of forwarding IP packets, routers can be broadly characterized as having either a *centralized* or *distributed* forwarding model. In the centralized model, the IP forwarding algorithm, outlined earlier in this chapter, is done in a single processing engine that handles the traffic from all ports. In the distributed model, there are several processing engines, perhaps one per port, or more often one per line card, where a line card may serve one or more physical ports. Each model has advantages and disadvantages. All things being equal, a distributed forwarding model should be able to forward more packets per second through the router as a whole, because there is more processing power in total. But a distributed model also complicates the software architecture, because each forwarding engine typically needs its own copy of the forwarding table, and thus it is necessary for the control processor to ensure that the forwarding tables are updated consistently and in a timely manner.

Another aspect of router implementation that is significantly different from that of switches is the IP forwarding algorithm itself. In bridges and

most ATM switches, the forwarding algorithm simply involves looking up a fixed-length identifier (MAC address or VCI) in a table, finding the correct output port in the table, and sending the packet to that port. We have already seen in Section 3.2.4 that the IP forwarding algorithm is a little more complicated than that, in part because the relevant number of bits that need to be examined when forwarding a packet is not fixed but variable, typically ranging from 8 bits to 32 bits.

Because of the relatively high complexity of the IP forwarding algorithm, there have been periods of time when it seemed IP routers might be running up against fundamental upper limits of performance. However, as we discuss in the Further Reading section of this chapter, there have been many innovative approaches to IP forwarding developed over the years, and at the time of writing there are commercial routers that can forward 40 Gbps of IP traffic *per interface*. By combining many such high-performance IP forwarding engines with the sort of very scalable switch fabrics discussed in Section 3.4, it has now become possible to build routers with many terabits of total throughput. That is more than enough to see us through the next few years of growth in Internet traffic.

Another technology of interest in the field of router implementation is the *network processor*. A network processor is intended to be a device that is just about as programmable as a standard PC processor, but that is more highly optimized for networking tasks. For example, a network processor might have instructions that are particularly well suited to performing lookups on IP addresses, or calculating checksums on IP datagrams. Such devices could be used in routers and other networking devices (e.g., firewalls).

One of the interesting and ongoing debates about network processors is whether they can do a better job than the alternatives. For example, given the continuous and remarkable improvements in performance of conventional processors, and the huge industry that drives those improvements, can network processors keep up? And can a device that strives for generality do as good a job as a custom-designed application-specific integrated circuit (ASIC) that does nothing except, say, IP forwarding? Part of the answer to questions like these depends on what you mean by “do a better job.” For example, there will always be trade-offs to be made between cost of hardware, time to market, performance, power consumption, and flexibility—the ability to change the features

supported by a router after it is built. We will see in later chapters just how diverse the requirements for router functionality can be. It is safe to assume that a wide range of router designs will exist for the foreseeable future and that network processors will have some role to play.

### 3.5 SUMMARY

This chapter has begun to look at some of the issues involved in building scalable and heterogeneous networks by using switches and routers to interconnect links and networks. The most common use of switching is the interconnection of LANs, especially Ethernet segments. LAN switches, or bridges, use techniques such as source address learning to improve forwarding efficiency and spanning tree algorithms to avoid looping. These switches are extensively used in data centers, campuses, and corporate networks.

To deal with heterogeneous networks, the Internetworking Protocol (IP) was invented and forms the basis of today's routers. IP tackles heterogeneity by defining a simple, common service model for an internetwork, which is based on the best-effort delivery of IP datagrams. An important part of the service model is the global addressing scheme, which enables any two nodes in an internetwork to uniquely identify each other for the purposes of exchanging data. The IP service model is simple enough to be supported by any known networking technology, and the ARP mechanism is used to translate global IP addresses into local link-layer addresses.

A crucial aspect of the operation of an internetwork is the determination of efficient routes to any destination in the internet. Internet routing algorithms solve this problem in a distributed fashion; this chapter introduced the two major classes of algorithms—link-state and distance-vector—along with examples of their application (RIP and OSPF).

Both switches and routers need to forward packets from inputs to outputs at a high rate and, in some circumstances, grow to a large size to accommodate hundreds or thousands of ports. Building switches that both scale and offer high performance at acceptable cost is complicated by the problem of contention; as a consequence, switches and routers often employ special-purpose hardware rather than being built from general-purpose workstations.

The Internet has without doubt been an enormous success, and it can be easy to forget that there was ever a time when it didn't exist. However, the inventors of the Internet developed it in part because the networks available at the time, such as the circuit-switched telephone network, were not well suited to the things they wanted to do. Now that the Internet is an established artifact, just as the telephone network was in the 1960s, it is reasonable to ask: What comes after the Internet?

No one knows the answer to that question at the moment, but some significant research efforts are underway to try to enable some sort of "Future Internet." While it is difficult to imagine that today's Internet will be replaced by something new any time soon (after all, the telephone network is still around, although increasingly its traffic is moving onto the

### WHAT'S NEXT: THE FUTURE INTERNET

Internet), thinking beyond the constraints of incrementally deployable tweaks to today's Internet could enable some new innovations that we would otherwise miss. It is popular to talk about "clean slate" research in this context—such research looks at what might be possible if we *could* start from scratch, postponing deployment considerations for later.

For example, what if we assumed that every node in the Internet was mobile? We would probably start with a different way of identifying nodes—rather than an IP address, which includes information about what network the node is currently attached to, we might use some other form of identifier. Or, as another example, we might consider a different trust model than the one built into the current Internet. When the Internet was originally developed, it seemed reasonable to assume that every host should be able to send to every other host by default, but today in the world of spammers, phishers, and denial-of-service attacks, a different trust model—with more limited initial capabilities for newly connected or unknown nodes perhaps—might be considered. These two examples illustrate cases where, knowing today some things that were not apparent in the '70s (like the importance of mobility and security to networking), we might want to come up with a very different design for an internetwork.

A couple of points can be made here. First, you should not assume that the Internet is “done.” Its architecture is inherently flexible and it will continue to evolve. We will see some examples of its evolution in the next chapter. The other point is that there’s more than one way to do networking research: Developing incrementally deployable ideas is great, but in the words of Internet pioneer David Clark, “To conceive the future, it helps to let go of the present.”

## ■ FURTHER READING

The seminal paper on bridges, in particular the spanning tree algorithm, is the article by Perlman below. Not surprisingly, countless papers have been written on various aspects of the Internet; the paper by Cerf and Kahn is the one that originally introduced the TCP/IP architecture and is worth reading for its historical perspective. Finally, McKeown’s paper, one of many on switch design, describes an approach to switch design that uses cells internally but has been used commercially as the basis for high-performance routers forwarding variable-length IP packets.

- Perlman, R. An algorithm for distributed computation of spanning trees in an extended LAN. *Proceedings of the Ninth Data Communications Symposium*, pages 44–53, September 1985.
- Cerf, V., and R. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications* COM-22(5):637–648, May 1974.
- McKeown, N. The iSLIP scheduling algorithm for input-queued switches. *IEEE Transactions on Networking* 7(2):188–201, April 1999.

A good general overview of bridges and routers can be found in another work by Perlman [Per00]. There is a wealth of papers on ATM; Turner [Tur85], an ATM pioneer, was one of the first to propose the use of a cell-based network for integrated services.

Many of the techniques and protocols that are central to today’s Internet are described in requests for comments (RFCs): Subnetting is described in Mogul and Postel [MP85], CIDR is described in Fuller and Li [FL06], RIPv2 is defined in Malkin [Mal98], and OSPF is defined in Moy [Moy98]. The OSPF specification, at over 200 pages, is one of the longer RFCs around, but it contains an unusual wealth of detail about

how to implement a protocol. The reasons to avoid IP fragmentation are examined in Kent and Mogul [KM87] and the Path MTU discovery technique is described in Mogul and Deering [MD90].

A forward-looking paper about research on the future Internet was written by Clark et al. [CPB<sup>+</sup>05]. This paper is related to the ongoing research efforts around a future Internet for which we provide live references below.

Literally thousands of papers have been published on switch architectures. One early paper that explains Batcher networks well is, not surprisingly, one by Batcher himself [Bat68]. Sorting networks are explained by Drysdale and Young [DY75], and an interesting form of cross-bar switch is described by Yeh et al. [YHA87]. Giacopelli et al. [GHMS91] describe the “Sunshine” switch, which provides insights into the important role of traffic analysis in switch design. In particular, the Sunshine designers were among the first to realize that cells were likely to arrive at a switch in bursts and thus were able to factor correlated arrivals into their design. A good overview of the performance of different switching fabrics can be found in Robertazzi [Rob93]. An example of the design of a switch based on variable-length packets can be found in Gopal and Guerin [GG94].

There has been a lot of work aimed at developing algorithms that can be used by routers to do fast lookup of IP addresses. (Recall that the problem is that the router needs to match the longest prefix in the forwarding table.) PATRICIA trees are one of the first algorithms applied to this problem [Mor68]. More recent work is reported in [DBCP97], [WVTP97], [LS98], [SVSM98], and [EVD04]. For an overview of how algorithms like these can be used to build a high-speed router, see Partridge et al. [Par98].

Optical networking is a rich field in its own right, with its own journals, conferences, etc. We recommend Ramaswami et al. [RS01] as a good introductory text in that field.

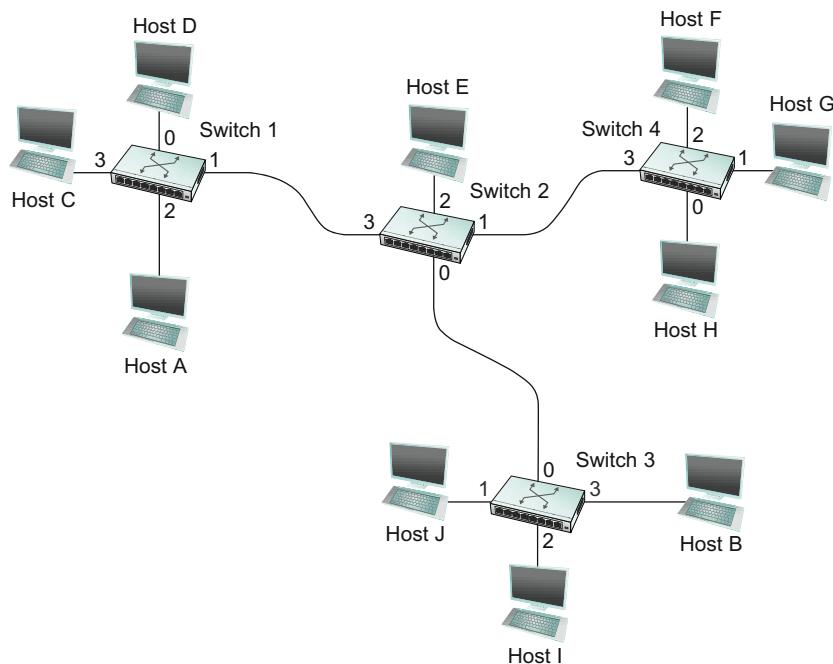
An excellent text to read if you want to learn about the mathematical analysis of network performance is by Kleinrock [Kle75], one of the pioneers of the ARPANET. Many papers have been published on the applications of queuing theory to packet switching. We recommend the article by Paxson and Floyd [PF94] as a significant contribution focused on the Internet, and one by Leland et al. [LTWW94], a paper that introduces the important concept of “long-range dependence” and shows the inadequacy of many traditional approaches to traffic modeling.

Finally, we recommend the following live references:

- <http://www.nets-find.net/>: A website of the U.S. National Science Foundation that covers the “Future Internet Design” research program.
- <http://www.geni.net/>: A site describing the GENI networking testbed, which has been created to enable some of the “clean slate” research described above.

### EXERCISES

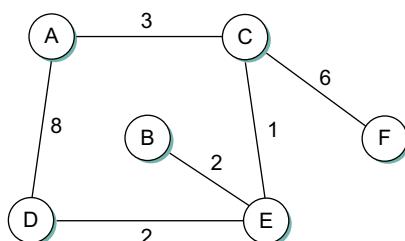
1. Using the example network given in Figure 3.44, give the virtual circuit tables for all the switches after each of the following connections is established. Assume that the sequence of connections is cumulative; that is, the first connection is still up when the second connection is established, and so on. Also assume that the VCI assignment always picks the lowest unused



■ FIGURE 3.44 Example network for Exercises 1 and 2.

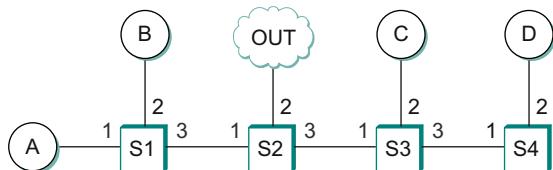
VCI on each link, starting with 0, and that a VCI is consumed for both directions of a virtual circuit.

- (a) Host A connects to host C.
  - (b) Host D connects to host B.
  - (c) Host D connects to host I.
  - (d) Host A connects to host B.
  - (e) Host F connects to host J.
  - (f) Host H connects to host A.
- ✓ 2. Using the example network given in Figure 3.44, give the virtual circuit tables for all the switches after each of the following connections is established. Assume that the sequence of connections is cumulative; that is, the first connection is still up when the second connection is established, and so on. Also assume that the VCI assignment always picks the lowest unused VCI on each link, starting with 0, and that a VCI is consumed for both directions of a virtual circuit.
- (a) Host D connects to host H.
  - (b) Host B connects to host G.
  - (c) Host F connects to host A.
  - (d) Host H connects to host C.
  - (e) Host I connects to host E.
  - (f) Host H connects to host J.
3. For the network given in Figure 3.45, give the datagram forwarding table for each node. The links are labeled with relative costs; your tables should forward each packet via the lowest-cost path to its destination.



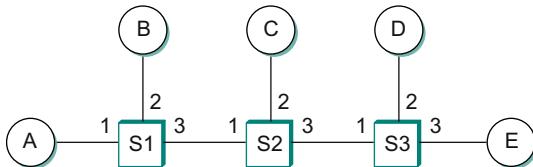
■ FIGURE 3.45 Network for Exercise 3.

4. Give forwarding tables for switches S1 to S4 in Figure 3.46. Each switch should have a default routing entry, chosen to forward packets with unrecognized destination addresses toward OUT. Any specific-destination table entries duplicated by the default entry should then be eliminated.



■ FIGURE 3.46 Diagram for Exercise 4.

5. Consider the virtual circuit switches in Figure 3.47. Table 3.15 lists, for each switch, what  $\langle$ port, VCI $\rangle$  (or  $\langle$ VCI, interface $\rangle$ ) pairs are connected to what other. Connections are bidirectional. List all endpoint-to-endpoint connections.



■ FIGURE 3.47 Diagram for Exercise 5.

6. In the source routing example of Section 3.1.3, the address received by B is not reversible and doesn't help B know how to reach A. Propose a modification to the delivery mechanism that does allow for reversibility. Your mechanism should *not* require giving all switches globally unique names.
7. Propose a mechanism that virtual circuit switches might use so that if one switch loses all its state regarding connections then a sender of packets along a path through that switch is informed of the failure.
8. Propose a mechanism that might be used by datagram switches so that if one switch loses all or part of its forwarding table affected senders are informed of the failure.

**Table 3.15 VCI Tables for Switches in Figure 3.47**

Switch S1			
Port	VCI	Port	VCI
1	2	3	1
1	1	2	3
2	1	3	2

Switch S2			
Port	VCI	Port	VCI
1	1	3	3
1	2	3	2

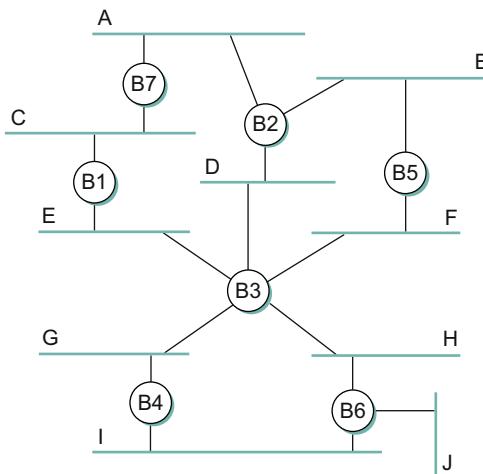
  

Switch S3			
Port	VCI	Port	VCI
1	3	2	1
1	2	2	2

9. The virtual circuit mechanism described in Section 3.1.2 assumes that each link is point-to-point. Extend the forwarding algorithm to work in the case that links are shared-media connections (e.g., Ethernet).
10. Suppose, in Figure 3.2, that a new link has been added, connecting switch 3 port 1 (where G is now) and switch 1 port 0 (where D is now); neither switch is informed of this link. Furthermore, switch 3 mistakenly thinks that host B is reached via port 1.
  - (a) What happens if host A attempts to send to host B, using datagram forwarding?
  - (b) What happens if host A attempts to connect to host B, using the virtual circuit setup mechanism discussed in the text?
11. Give an example of a working virtual circuit whose path traverses some link twice. Packets sent along this path should *not*, however, circulate indefinitely.
12. In Section 3.1.2, each switch chose the VCI value for the incoming link. Show that it is also possible for each switch to choose the VCI value for the outbound link and that the same VCI values will be

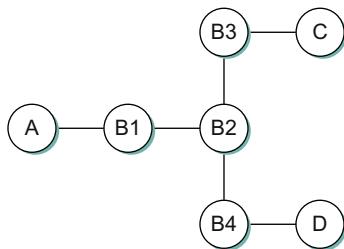
chosen by each approach. If each switch chooses the outbound VCI, is it still necessary to wait one RTT before data is sent?

- 13.** Given the extended LAN shown in Figure 3.48, indicate which ports are not selected by the spanning tree algorithm.



■ FIGURE 3.48 Network for Exercises 13 and 14.

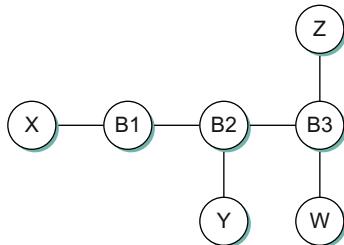
- 14.** Given the extended LAN shown in Figure 3.48, assume that bridge B1 suffers catastrophic failure. Indicate which ports are not selected by the spanning tree algorithm after the recovery process and a new tree has been formed.
- 15.** Consider the arrangement of learning bridges shown in Figure 3.49. Assuming all are initially empty, give the forwarding tables for each of the bridges B1 to B4 after the following transmissions:
- A sends to C.
  - C sends to A.
  - D sends to C.
- Identify ports with the unique neighbor reached directly from that port; that is, the ports for B1 are to be labeled “A” and “B2.”
- 16.** As in the previous problem, consider the arrangement of learning bridges shown in Figure 3.49. Assuming all are initially empty, give



■ FIGURE 3.49 Network for Exercises 15 and 16.

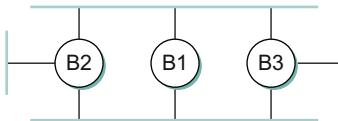
the forwarding tables for each of the bridges B1 to B4 after the following transmissions:

- D sends to C.
  - C sends to D.
  - A sends to C.
17. Consider hosts X, Y, Z, W and learning bridges B1, B2, B3, with initially empty forwarding tables, as in Figure 3.50.
- Suppose X sends to W. Which bridges learn where X is? Does Y's network interface see this packet?
  - Suppose Z now sends to X. Which bridges learn where Z is? Does Y's network interface see this packet?
  - Suppose Y now sends to X. Which bridges learn where Y is? Does Z's network interface see this packet?
  - Finally, suppose W sends to Y. Which bridges learn where W is? Does Z's network interface see this packet?



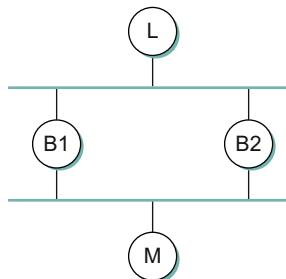
■ FIGURE 3.50 Diagram for Exercise 17.

18. Give the spanning tree generated for the extended LAN shown in Figure 3.51, and discuss how any ties are resolved.



■ FIGURE 3.51 Extended LAN for Exercise 18.

- 19.** Suppose learning bridges B1 and B2 form a loop as shown in Figure 3.52, and do *not* implement the spanning tree algorithm. Each bridge maintains a single table of  $\langle \text{address}, \text{interface} \rangle$  pairs.
- What will happen if M sends to L?
  - Suppose a short while later L replies to M. Give a sequence of events that leads to one packet from M and one packet from L circling the loop in opposite directions.



■ FIGURE 3.52 Loop for Exercises 19 and 20.

- 20.** Suppose that M in Figure 3.52 sends to itself (this normally would never happen). State what would happen, assuming:
- The bridges' learning algorithm is to install (or update) the new  $\langle \text{sourceaddress}, \text{interface} \rangle$  entry *before* searching the table for the destination address.
  - The new source address was installed *after* destination address lookup.
- 21.** Consider the extended LAN of Figure 3.10. What happens in the spanning tree algorithm if bridge B1 does not participate and
- Simply forwards all spanning tree algorithm messages?
  - Drops all spanning tree messages?

22. Suppose some repeaters (hubs), rather than bridges, are connected into a loop.
- (a) What will happen when somebody transmits?
  - (b) Why would the spanning tree mechanism be difficult or impossible to implement for repeaters?
  - (c) Propose a mechanism by which repeaters might detect loops and shut down some ports to break the loop. Your solution is not required to work 100% of the time.
23. Suppose a bridge has two of its ports on the same network. How might the bridge detect and correct this?
24. What percentage of an ATM link's total bandwidth is consumed by the ATM cell headers? Ignore padding to fill cells or ATM adaptation layer headers.
25. Cell switching methods (like ATM) essentially always use virtual circuit switching rather than datagram forwarding. Give a specific argument why this is so (consider the preceding question).
26. Suppose a workstation has an I/O bus speed of 800 Mbps and memory bandwidth of 2 Gbps. Assuming direct memory access (DMA) is used to move data in and out of main memory, how many interfaces to 100-Mbps Ethernet links could a switch based on this workstation handle?
- ✓ 27. Suppose a workstation has an I/O bus speed of 1 Gbps and memory bandwidth of 2 Gbps. Assuming DMA is used to move data in and out of main memory, how many interfaces to 100-Mbps Ethernet links could a switch based on this workstation handle?
28. Suppose a switch is built using a computer workstation and that it can forward packets at a rate of 500,000 packets per second, regardless (within limits) of size. Assume the workstation uses direct memory access (DMA) to move data in and out of its main memory, which has a bandwidth of 2 Gbps, and that the I/O bus has a bandwidth of 1 Gbps. At what packet size would the bus bandwidth become the limiting factor?
29. Suppose that a switch is designed to have both input and output FIFO buffering. As packets arrive on an input port they are inserted

at the tail of the FIFO. The switch then tries to forward the packets at the head of each FIFO to the tail of the appropriate output FIFO.

- (a) Explain under what circumstances such a switch can lose a packet destined for an output port whose FIFO is empty.
- (b) What is this behavior called?
- (c) Assume that the FIFO buffering memory can be redistributed freely. Suggest a reshuffling of the buffers that avoids the above problem, and explain why it does so.

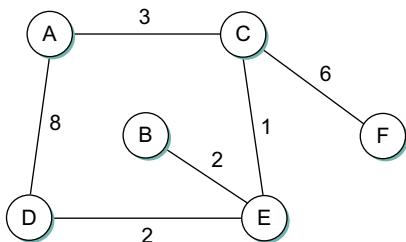
-  30. A stage of an  $n \times n$  banyan network consists of  $(n/2)$   $2 \times 2$  switching elements. The first stage directs packets to the correct half of the network, the next stage to the correct quarter, and so on, until the packet is routed to the correct output. Derive an expression for the number of  $2 \times 2$  switching elements needed to make an  $n \times n$  banyan network. Verify your answer for  $n = 8$ .
-  31. Describe how a Batcher network works. (See the Further Reading section.) Explain how a Batcher network can be used in combination with a banyan network to build a switching fabric.
- 32. Suppose a 10-Mbps Ethernet hub (repeater) is replaced by a 10-Mbps switch, in an environment where all traffic is between a single server and  $N$  “clients.” Because all traffic must still traverse the server–switch link, nominally there is no improvement in bandwidth.
  - (a) Would you expect *any* improvement in bandwidth? If so, why?
  - (b) What other advantages and drawbacks might a switch offer versus a hub?
- 33. What aspect of IP addresses makes it necessary to have one address per network interface, rather than just one per host? In light of your answer, why does IP tolerate point-to-point interfaces that have nonunique addresses or no addresses?
- 34. Why does the Offset field in the IP header measure the offset in 8-byte units? (Hint: Recall that the Offset field is 13 bits long.)
- 35. Some signalling errors can cause entire ranges of bits in a packet to be overwritten by all 0s or all 1s. Suppose all the bits in the packet, including the Internet checksum, are overwritten. Could a packet with all 0s or all 1s be a legal IPv4 packet? Will the Internet checksum catch that error? Why or why not?

36. Suppose a TCP message that contains 1024 bytes of data and 20 bytes of TCP header is passed to IP for delivery across two networks interconnected by a router (i.e., it travels from the source host to a router to the destination host). The first network has an MTU of 1024 bytes; the second has an MTU of 576 bytes. Each network's MTU gives the size of the largest IP datagram that can be carried in a link-layer frame. Give the sizes and offsets of the sequence of fragments delivered to the network layer at the destination host. Assume all IP headers are 20 bytes.
- ✓ 37. Path MTU is the smallest MTU of any link on the current path (route) between two hosts. Assume we could discover the path MTU of the path used in the previous exercise, and that we use this value as the MTU for all the path segments. Give the sizes and offsets of the sequence of fragments delivered to the network layer at the destination host.
- ★ 38. Suppose an IP packet is fragmented into 10 fragments, each with a 1% (independent) probability of loss. To a reasonable approximation, this means there is a 10% chance of losing the whole packet due to loss of a fragment. What is the probability of net loss of the whole packet if the packet is transmitted twice,  
**(a)** Assuming all fragments received must have been part of the same transmission?  
**(b)** Assuming any given fragment may have been part of either transmission?  
**(c)** Explain how use of the Ident field might be applicable here.
39. Suppose the fragments of Figure 3.18(b) all pass through another router onto a link with an MTU of 380 bytes, not counting the link header. Show the fragments produced. If the packet were originally fragmented for this MTU, how many fragments would be produced?
40. What is the maximum bandwidth at which an IP host can send 576-byte packets without having the Ident field wrap around within 60 seconds? Suppose that IP's maximum segment lifetime (MSL) is 60 seconds; that is, delayed packets can arrive up to 60 seconds late but no later. What might happen if this bandwidth were exceeded?

41. Why do you think IPv4 has fragment reassembly done at the endpoint, rather than at the next router? Why do you think IPv6 abandoned fragmentation entirely? (Hint: Think about the differences between IP-layer fragmentation and link-layer fragmentation).
42. Having ARP table entries time out after 10 to 15 minutes is an attempt at a reasonable compromise. Describe the problems that can occur if the timeout value is too small or too large.
43. IP currently uses 32-bit addresses. If we could redesign IP to use the 6-byte MAC address instead of the 32-bit address, would we be able to eliminate the need for ARP? Explain why or why not.
44. Suppose hosts A and B have been assigned the same IP address on the same Ethernet, on which ARP is used. B starts up after A. What will happen to A's existing connections? Explain how "self-ARP" (querying the network on start-up for one's own IP address) might help with this problem.
45. Suppose an IP implementation adheres literally to the following algorithm on receipt of a packet, P, destined for IP address D:

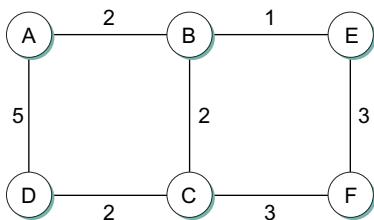
```
if ((Ethernet address for D is in ARP cache))
    (send P)
else
    (send out an ARP query for D)
    (put P into a queue until the response comes back)
```

  - (a) If the IP layer receives a burst of packets destined for D, how might this algorithm waste resources unnecessarily?
  - (b) Sketch an improved version.
  - (c) Suppose we simply drop P, after sending out a query, when cache lookup fails. How would this behave? (Some early ARP implementations allegedly did this.)
46. For the network shown in Figure 3.53, give global distance-vector tables like those of Tables 3.10 and 3.13 when
  - (a) Each node knows only the distances to its immediate neighbors.
  - (b) Each node has reported the information it had in the preceding step to its immediate neighbors.
  - (c) Step (b) happens a second time.



■ FIGURE 3.53 Network for Exercises 46, 48, and 54.

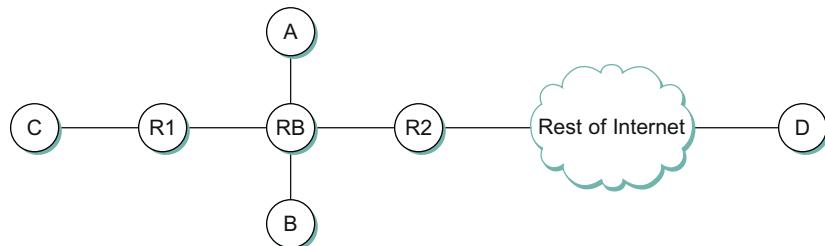
- ✓ 47. For the network given in Figure 3.54, give global distance-vector tables like those of Tables 3.10 and 3.13 when
- Each node knows only the distances to its immediate neighbors.
  - Each node has reported the information it had in the preceding step to its immediate neighbors.
  - Step (b) happens a second time.



■ FIGURE 3.54 Network for Exercise 47.

48. For the network given in Figure 3.53, show how the *link-state* algorithm builds the routing table for node D.
49. Use the Unix utility traceroute (Windows tracert) to determine how many hops it is from your host to other hosts in the Internet (e.g., cs.princeton.edu or www.cisco.com). How many routers do you traverse just to get out of your local site? Read the man page or other documentation for traceroute and explain how it is implemented.
50. What will happen if traceroute is used to find the path to an unassigned address? Does it matter if the network portion or only the host portion is unassigned?

51. A site is shown in Figure 3.55. R1 and R2 are routers; R2 connects to the outside world. Individual LANs are Ethernets. RB is a *bridge-router*; it routes traffic addressed to it and acts as a bridge for other traffic. Subnetting is used inside the site; ARP is used on each subnet. Unfortunately, host A has been misconfigured and doesn't use subnets. Which of B, C, and D can A reach?



■ FIGURE 3.55 Site for Exercise 51.

52. Suppose we have the forwarding tables shown in Table 3.16 for nodes A and F, in a network where all links have cost 1. Give a diagram of the smallest network consistent with these tables.

Table 3.16 Forwarding Tables for Exercise 52

A		
Node	Cost	Nexthop
B	1	B
C	2	B
D	1	D
E	2	B
F	3	D

F		
Node	Cost	Nexthop
A	3	E
B	2	C
C	1	C
D	2	E
E	1	E

- ✓ 53. Suppose we have the forwarding tables shown in Table 3.17 for nodes A and F, in a network where all links have cost 1. Give a diagram of the smallest network consistent with these tables.

**Table 3.17 Forwarding Tables for Exercise 53**

<b>A</b>		
<b>Node</b>	<b>Cost</b>	<b>Nexthop</b>
B	1	B
C	1	C
D	2	B
F	2	C

<b>F</b>		
<b>Node</b>	<b>Cost</b>	<b>Nexthop</b>
A	2	C
B	3	C
C	1	C
D	2	C
E	1	E

54. For the network in Figure 3.53, suppose the forwarding tables are all established as in Exercise 46 and then the C–E link fails. Give:
- (a) The tables of A, B, D, and F after C and E have reported the news.
  - (b) The tables of A and D after their next mutual exchange.
  - (c) The table of C after A exchanges with it.
55. Suppose a router has built up the routing table shown in Table 3.18. The router can deliver packets directly over interfaces 0 and 1, or it can forward packets to routers R2, R3, or R4. Describe what the router does with a packet addressed to each of the following destinations:
- (a) 128.96.39.10
  - (b) 128.96.40.12
  - (c) 128.96.40.151
  - (d) 192.4.153.17
  - (e) 192.4.153.90

**Table 3.18 Routing Table for Exercise 55**

<b>SubnetNumber</b>	<b>SubnetMask</b>	<b>NextHop</b>
128.96.39.0	255.255.255.128	Interface 0
128.96.39.128	255.255.255.128	Interface 1
128.96.40.0	255.255.255.128	R2
192.4.153.0	255.255.255.192	R3
<i>(default)</i>		R4

- ✓ 56. Suppose a router has built up the routing table shown in Table 3.19. The router can deliver packets directly over interfaces 0 and 1, or it can forward packets to routers R2, R3, or R4. Assume the router does the longest prefix match. Describe what the router does with a packet addressed to each of the following destinations:
- (a) 128.96.171.92
  - (b) 128.96.167.151
  - (c) 128.96.163.151
  - (d) 128.96.169.192
  - (e) 128.96.165.121

**Table 3.19 Routing Table for Exercise 56**

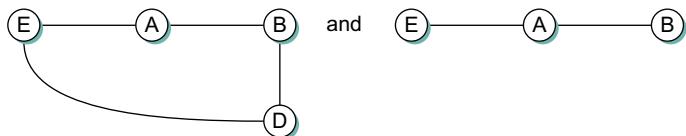
<b>SubnetNumber</b>	<b>SubnetMask</b>	<b>NextHop</b>
128.96.170.0	255.255.254.0	Interface 0
128.96.168.0	255.255.254.0	Interface 1
128.96.166.0	255.255.254.0	R2
128.96.164.0	255.255.252.0	R3
<i>(default)</i>		R4

- ★ 57. Consider the simple network in Figure 3.56, in which A and B exchange distance-vector routing information. All links have cost 1. Suppose the A–E link fails.

**FIGURE 3.56** Simple network for Exercise 57.

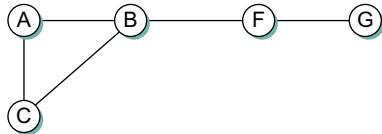
- (a) Give a sequence of routing table updates that leads to a routing loop between A and B.
  - (b) Estimate the probability of the scenario in (a), assuming A and B send out routing updates at random times, each at the same average rate.
  - (c) Estimate the probability of a loop forming if A broadcasts an updated report within 1 second of discovering the A–E failure, and B broadcasts every 60 seconds uniformly.
58. Consider the situation involving the creation of a routing loop in the network of Figure 3.29 when the A–E link goes down. List *all* sequences of table updates among A, B, and C, pertaining to destination E, that lead to the loop. Assume that table updates are done one at a time, that the split-horizon technique is observed by all participants, and that A sends its initial report of E’s unreachability to B before C. You may ignore updates that don’t result in changes.
59. Suppose a set of routers all use the split-horizon technique; we consider here under what circumstances it makes a difference if they use poison reverse in addition.
- (a) Show that poison reverse makes no difference in the evolution of the routing loop in the two examples described in Section 3.3.2, given that the hosts involved use split horizon.
  - (b) Suppose split-horizon routers A and B somehow reach a state in which they forward traffic for a given destination X toward each other. Describe how this situation will evolve with and without the use of poison reverse.
  - (c) Give a sequence of events that leads A and B to a looped state as in (b), even if poison reverse is used. (Hint: Suppose B and A connect through a very slow link. They each reach X through a third node, C, and simultaneously advertise their routes to each other.)
60. *Hold down* is another distance-vector loop-avoidance technique, whereby hosts ignore updates for a period of time until link failure news has had a chance to propagate. Consider the networks in Figure 3.57, where all links have cost 1 except E–D, with cost 10. Suppose that the E–A link breaks and B reports its loop-forming E route to A immediately afterwards (this is the false route, via A).

Specify the details of a hold-down interpretation, and use this to describe the evolution of the routing loop in both networks. To what extent can hold down prevent the loop in the EAB network without delaying the discovery of the alternative route in the EABD network?



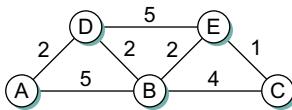
■ FIGURE 3.57 Networks for Exercise 60.

- 61.** Consider the network in Figure 3.58, using link-state routing. Suppose the B–F link fails, and the following then occur in sequence:
- Node H is added to the right side with a connection to G.
  - Node D is added to the left side with a connection to C.
  - A new link, D–A, is added.
- The failed B–F link is now restored. Describe what link-state packets will flood back and forth. Assume that the initial sequence number at all nodes is 1, that no packets time out, and that both ends of a link use the same sequence number in their LSP for that link, greater than any sequence number used before.

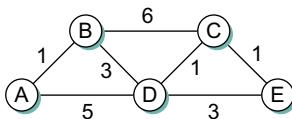


■ FIGURE 3.58 Network for Exercise 61.

- 62.** Give the steps as in Table 3.14 in the forward search algorithm as it builds the routing database for node A in the network shown in Figure 3.59.
- ✓ **63.** Give the steps as in Table 3.14 in the forward search algorithm as it builds the routing database for node A in the network shown in Figure 3.60.

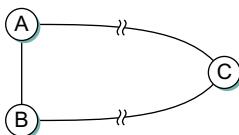


■ FIGURE 3.59 Network for Exercise 62.



■ FIGURE 3.60 Network for Exercise 63.

64. Suppose that nodes in the network shown in Figure 3.61 participate in link-state routing, and C receives contradictory LSPs: One from A arrives claiming the A–B link is down, but one from B arrives claiming the A–B link is up.
- How could this happen?
  - What should C do? What can C expect?
- Do not assume that the LSPs contain any synchronized timestamp.



■ FIGURE 3.61 Network for Exercise 64.

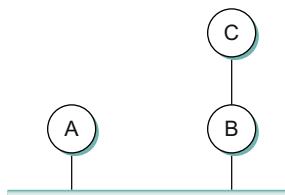
65. Suppose IP routers learned about IP networks and subnets the way Ethernet learning bridges learn about hosts: by noting the appearance of new ones and the interface by which they arrive. Compare this with existing distance-vector router learning
- For a leaf site with a single attachment to the Internet.
  - For internal use at an organization that did not connect to the Internet.
- Assume that routers only receive new-network notices from other routers and that the originating routers receive their IP network information via configuration.

66. IP hosts that are not designated routers are *required* to drop packets misaddressed to them, even if they would otherwise be able to forward them correctly. In the absence of this requirement, what would happen if a packet addressed to IP address A were inadvertently broadcast at the link layer? What other justifications for this requirement can you think of?
67. Read the man page or other documentation for the Unix/Windows utility netstat. Use netstat to display the current IP routing table on your host. Explain the purpose of each entry. What is the practical minimum number of entries?
68. An organization has been assigned the prefix 212.1.1/24 (class C) and wants to form subnets for four departments, with hosts as follows:

A	75 hosts
B	35 hosts
C	20 hosts
D	18 hosts

There are 148 hosts in all.

- (a) Give a possible arrangement of subnet masks to make this possible.
- (b) Suggest what the organization might do if department D grows to 32 hosts.
69. Suppose hosts A and B are on an Ethernet LAN with IP network address 200.0.0/24. It is desired to attach a host C to the network via a direct connection to B (see Figure 3.62). Explain how to do this with subnets; give sample subnet assignments. Assume that



■ FIGURE 3.62 Network for Exercise 69.

an additional network prefix is not available. What does this do to the size of the Ethernet LAN?

70. An alternative method for connecting host C in Exercise 69 is to use *proxy ARP* and routing: B agrees to route traffic to and from C and also answers ARP queries for C received over the Ethernet.
- (a) Give all packets sent, with physical addresses, as A uses ARP to locate and then send one packet to C.
  - (b) Give B's routing table. What peculiarity must it contain?
71. Suppose two subnets share the same physical LAN; hosts on each subnet will see the other subnet's broadcast packets.
- (a) How will DHCP fare if two servers, one for each subnet, coexist on the shared LAN? What problems might [*do!*] arise?
  - (b) Will ARP be affected by such sharing?
72. Table 3.20 is a routing table using CIDR. Address bytes are in hexadecimal. The notation “/12” in C4.50.0.0/12 denotes a netmask with 12 leading 1 bits: FFF0.0.0. Note that the last three entries cover every address and thus serve in lieu of a default route. State to what next hop the following will be delivered:
- (a) C4.5E.13.87
  - (b) C4.5E.22.09
  - (c) C3.41.80.02
  - (d) 5E.43.91.12
  - (e) C4.6D.31.2E
  - (f) C4.6B.31.2E

**Table 3.20 Routing Table for Exercise 72**

Net/MaskLength	Nexthop
C4.50.0.0/12	A
C4.5E.10.0/20	B
C4.60.0.0/12	C
C4.68.0.0/14	D
80.0.0.0/1	E
40.0.0.0/2	F
00.0.0.0/2	G

- ✓ 73. Table 3.21 is a routing table using CIDR. Address bytes are in hexadecimal. The notation “/12” in C4.50.0.0/12 denotes a netmask with 12 leading 1 bits: FFF0.0.0. State to what next hop the following will be delivered:
- (a) C4.4B.31.2E
  - (b) C4.5E.05.09
  - (c) C4.4D.31.2E
  - (d) C4.5E.03.87
  - (e) C4.5E.7F.12
  - (f) C4.5E.D1.02

**Table 3.21 Routing Table for Exercise 73**

Net/MaskLength	Nexthop
C4.5E.2.0/23	A
C4.5E.4.0/22	B
C4.5E.C0.0/19	C
C4.5E.40.0/18	D
C4.4C.0.0/14	E
C0.0.0.0/2	F
80.0.0.0/1	G

74. An ISP that has authority to assign addresses from a /16 prefix (an old class B address) is working with a new company to allocate it a portion of address space based on CIDR. The new company needs IP addresses for machines in three divisions of its corporate network: Engineering, Marketing, and Sales. These divisions plan to grow as follows: Engineering has 5 machines as of the start of year 1 and intends to add 1 machine every week, Marketing will never need more than 16 machines, and Sales needs 1 machine for every 2 clients. As of the start of year 1, the company has no clients, but the sales model indicates that, by the start of year 2, the company will have 6 clients and each week thereafter will get one new client with probability 60%, will lose one client with probability 20%, or will maintain the same number with probability 20%.

- (a) What address range would be required to support the company's growth plans for at least 7 years if Marketing uses all 16 of its addresses and the Sales and Engineering plans behave as expected?
  - (b) How long would this address assignment last? At the time when the company runs out of address space, how would the addresses be assigned to the three groups?
  - (c) If, instead of using CIDR addressing, it was necessary to use old-style classful addresses, what options would the new company have in terms of getting address space?
75. Propose a lookup algorithm for an IP forwarding table containing prefixes of varying lengths that does not require a linear search of the entire table to find the longest match.



# Advanced Internetworking

# 4

*Every seeming equality conceals a hierarchy.*

—Mason Cooley

We have now seen how to build an internetwork that consists of a number of networks of different types. That is, we have dealt with the problem of *heterogeneity*. The second critical problem in internetworking—arguably the fundamental problem for all networking—is *scale*. To understand the problem of scaling, it is worth considering the growth of the Internet, which has roughly doubled in size each year for 30 years. This sort of growth forces us to face a number of challenges.

Chief among these is how do you build a routing system that can handle hundreds of thousands of networks and billions

## PROBLEM: SCALING TO BILLIONS

of end nodes? As we will see in this chapter, most approaches to tackling the scalability of routing depend on the introduction of hierarchy. We can introduce hierarchy in the form of areas within a domain; we also use hierarchy to scale the routing system among domains. The interdomain routing protocol that has enabled the Internet to scale to its current size is BGP. We will take a look at how BGP operates, and consider the challenges faced by BGP as the Internet continues to grow.

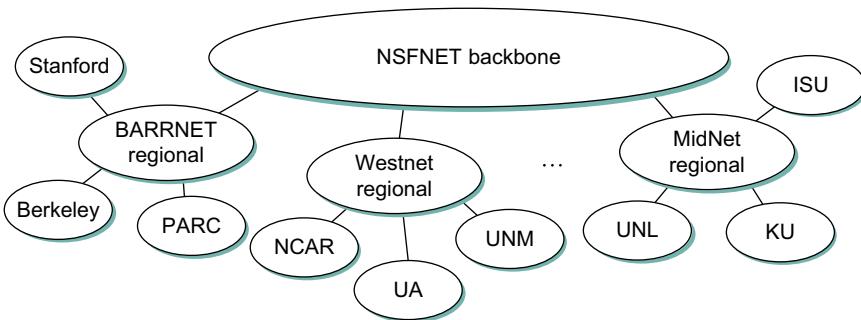
Closely related to the scalability of routing is the problem of addressing. Even two decades ago it had become apparent that the 32-bit addressing scheme of IP version 4 would not last forever. That led to the definition of a new version of IP—version 6, since version 5 had been used in an earlier experiment. IPv6 primarily expands the address space but also adds a number of new features, some of which have been retrofitted to IPv4.

While the Internet continues to grow in size, it also needs to evolve its functionality. The final sections of this chapter cover some significant enhancements to the Internet's capabilities. The first, multicast, is an enhancement of the basic service model. We show how multicast—the ability to deliver the same packets to a group of receivers efficiently—can be incorporated into an internet, and we describe several of the routing protocols that have been developed to support multicast. The second enhancement, Multiprotocol Label Switching (MPLS), modifies the forwarding mechanism of IP networks. This modification has enabled some changes in the way IP routing is performed and in the services offered by IP networks. Finally, we look at the effects of mobility on routing and describe some enhancements to IP to support mobile hosts and routers. For each of these enhancements, issues of scalability continue to be important.

#### 4.1 THE GLOBAL INTERNET

At this point, we have seen how to connect a heterogeneous collection of networks to create an internetwork and how to use the simple hierarchy of the IP address to make routing in an internet somewhat scalable. We say “somewhat” scalable because, even though each router does not need to know about all the hosts connected to the internet, it does, in the model described so far, need to know about all the networks connected to the internet. Today’s Internet has hundreds of thousands of networks connected to it (or more, depending on how you count). Routing protocols such as those we have just discussed do not scale to those kinds of numbers. This section looks at a variety of techniques that greatly improve scalability and that have enabled the Internet to grow as far as it has.

Before getting to these techniques, we need to have a general picture in our heads of what the global Internet looks like. It is not just a random interconnection of Ethernets, but instead it takes on a shape that reflects the fact that it interconnects many different organizations. Figure 4.1 gives a simple depiction of the state of the Internet in 1990. Since that



■ FIGURE 4.1 The tree structure of the Internet in 1990.

time, the Internet’s topology has grown much more complex than this figure suggests—we present a slightly more accurate picture of the current Internet in [Section 4.1.2](#) and [Figure 4.4](#)—but this picture will do for now.

One of the salient features of this topology is that it consists of end-user sites (e.g., Stanford University) that connect to service provider networks (e.g., BARRNET was a provider network that served sites in the San Francisco Bay Area). In 1990, many providers served a limited geographic region and were thus known as *regional networks*. The regional networks were, in turn, connected by a nationwide backbone. In 1990, this backbone was funded by the National Science Foundation (NSF) and was therefore called the *NSFNET backbone*. Although the detail is not shown in this figure, the provider networks are typically built from a large number of point-to-point links (e.g., T1 and DS3 links in the past, OC-48 and OC-192 SONET links today) that connect to routers; similarly, each end-user site is typically not a single network but instead consists of multiple physical networks connected by routers and bridges.

Notice in [Figure 4.1](#) that each provider and end-user is likely to be an administratively independent entity. This has some significant consequences on routing. For example, it is quite likely that different providers will have different ideas about the best routing protocol to use within their networks and on how metrics should be assigned to links in their network. Because of this independence, each provider’s network is usually a single *autonomous system* (AS). We will define this term more precisely in [Section 4.1.2](#), but for now it is adequate to think of an AS as a network that is administered independently of other ASs.

The fact that the Internet has a discernible structure can be used to our advantage as we tackle the problem of scalability. In fact, we need to deal with two related scaling issues. The first is the scalability of routing. We need to find ways to minimize the number of network numbers that get carried around in routing protocols and stored in the routing tables of routers. The second is address utilization—that is, making sure that the IP address space does not get consumed too quickly.

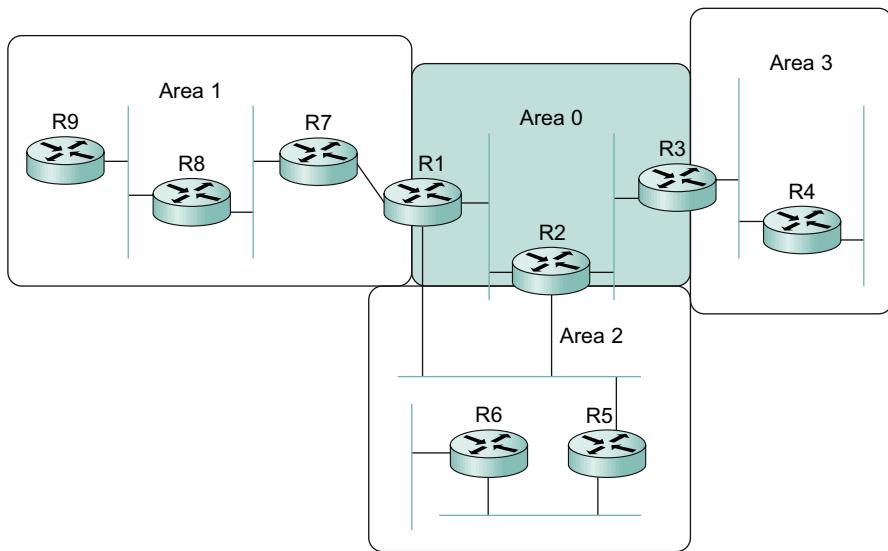
Throughout this book, we see the principle of hierarchy used again and again to improve scalability. We saw in the previous chapter how the hierarchical structure of IP addresses, especially with the flexibility provided by Classless Interdomain Routing (CIDR) and subnetting, can improve the scalability of routing. In the next two sections, we'll see further uses of hierarchy (and its partner, aggregation) to provide greater scalability, first in a single domain and then between domains. Our final subsection looks at the emerging standards for IP version 6, the invention of which was largely the result of scalability concerns.

### 4.1.1 Routing Areas

As a first example of using hierarchy to scale up the routing system, we'll examine how link-state routing protocols (such as OSPF and IS-IS) can be used to partition a routing domain into subdomains called *areas*. (The terminology varies somewhat among protocols—we use the OSPF terminology here.) By adding this extra level of hierarchy, we enable single domains to grow larger without overburdening the routing protocols or resorting to the more complex interdomain routing protocols described below.

An area is a set of routers that are administratively configured to exchange link-state information with each other. There is one special area—the backbone area, also known as area 0. An example of a routing domain divided into areas is shown in [Figure 4.2](#). Routers R1, R2, and R3 are members of the backbone area. They are also members of at least one nonbackbone area; R1 is actually a member of both area 1 and area 2. A router that is a member of both the backbone area and a nonbackbone area is an area border router (ABR). Note that these are distinct from the routers that are at the edge of an AS, which are referred to as AS border routers for clarity.

Routing within a single area is exactly as described in [Section 3.3.3](#). All the routers in the area send link-state advertisements to each other



■ FIGURE 4.2 A domain divided into areas.

and thus develop a complete, consistent map of the area. However, the link-state advertisements of routers that are not area border routers do not leave the area in which they originated. This has the effect of making the flooding and route calculation processes considerably more scalable. For example, router R4 in area 3 will never see a link-state advertisement from router R8 in area 1. As a consequence, it will know nothing about the detailed topology of areas other than its own.

How, then, does a router in one area determine the right next hop for a packet destined to a network in another area? The answer to this becomes clear if we imagine the path of a packet that has to travel from one nonbackbone area to another as being split into three parts. First, it travels from its source network to the backbone area, then it crosses the backbone, then it travels from the backbone to the destination network. To make this work, the area border routers summarize routing information that they have learned from one area and make it available in their advertisements to other areas. For example, R1 receives link-state advertisements from all the routers in area 1 and can thus determine the cost of reaching any network in area 1. When R1 sends link-state advertisements into area 0, it advertises the costs of reaching the networks in area 1 much as if all those networks were directly connected to R1. This enables all the

area 0 routers to learn the cost to reach all networks in area 1. The area border routers then summarize this information and advertise it into the nonbackbone areas. Thus, all routers learn how to reach all networks in the domain.

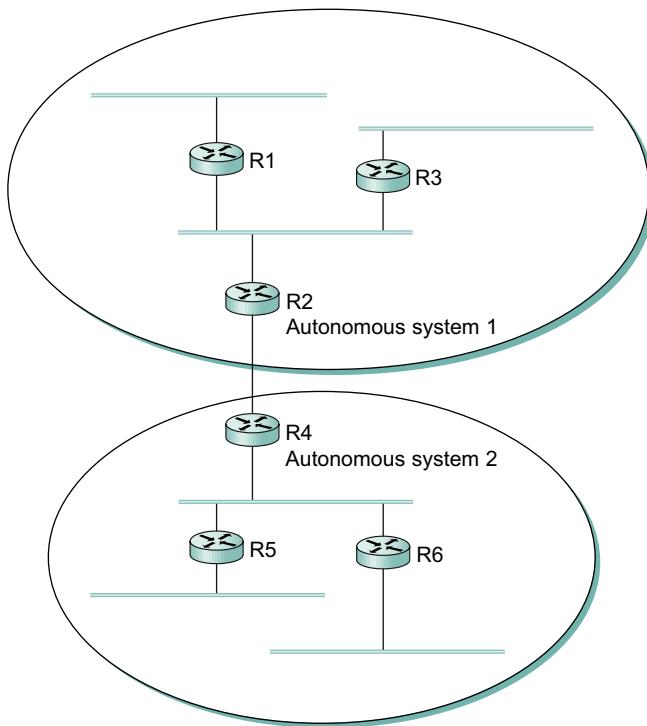
Note that, in the case of area 2, there are two ABRs and that routers in area 2 will thus have to make a choice as to which one they use to reach the backbone. This is easy enough, since both R1 and R2 will be advertising costs to various networks, so it will become clear which is the better choice as the routers in area 2 run their shortest-path algorithm. For example, it is pretty clear that R1 is going to be a better choice than R2 for destinations in area 1.

When dividing a domain into areas, the network administrator makes a tradeoff between scalability and optimality of routing. The use of areas forces all packets traveling from one area to another to go via the backbone area, even if a shorter path might have been available. For example, even if R4 and R5 were directly connected, packets would not flow between them because they are in different nonbackbone areas. It turns out that the need for scalability is often more important than the need to use the absolute shortest path.



This illustrates an important principle in network design. There is frequently a trade-off between some sort of optimality and scalability. When hierarchy is introduced, information is hidden from some nodes in the network, hindering their ability to make perfectly optimal decisions. However, information hiding is essential to scalability, since it saves all nodes from having global knowledge. It is invariably true in large networks that scalability is a more pressing design goal than perfect optimality.

Finally, we note that there is a trick by which network administrators can more flexibly decide which routers go in area 0. This trick uses the idea of a *virtual link* between routers. Such a virtual link is obtained by configuring a router that is not directly connected to area 0 to exchange backbone routing information with a router that is. For example, a virtual link could be configured from R8 to R1, thus making R8 part of the backbone. R8 would now participate in link-state advertisement flooding with the other routers in area 0. The cost of the virtual link from R8 to R1 is determined by the exchange of routing information that takes place in area 1. This technique can help to improve the optimality of routing.



■ FIGURE 4.3 A network with two autonomous systems.

#### 4.1.2 Interdomain Routing (BGP)

At the beginning of this chapter, we introduced the notion that the Internet is organized as autonomous systems, each of which is under the control of a single administrative entity. A corporation's complex internal network might be a single AS, as may the network of a single Internet Service Provider (ISP). Figure 4.3 shows a simple network with two autonomous systems.



LAB 08:

BGP

The basic idea behind autonomous systems is to provide an additional way to hierarchically aggregate routing information in a large internet, thus improving scalability. We now divide the routing problem into two parts: routing within a single autonomous system and routing between autonomous systems. Since another name for autonomous systems in the Internet is routing *domains*, we refer to the two parts of the routing problem as interdomain routing and intradomain routing. In addition to

improving scalability, the AS model decouples the intradomain routing that takes place in one AS from that taking place in another. Thus, each AS can run whatever intradomain routing protocols it chooses. It can even use static routes or multiple protocols, if desired. The interdomain routing problem is then one of having different ASs share reachability information—descriptions of the set of IP addresses that can be reached via a given AS—with each other.

#### *Challenges in Interdomain Routing*

Perhaps the most important challenge of interdomain routing today is the need for each AS to determine its own routing *policies*. A simple example routing policy implemented at a particular AS might look like this: “Whenever possible, I prefer to send traffic via AS X than via AS Y, but I’ll use AS Y if it is the only path, and I never want to carry traffic from AS X to AS Y or *vice versa*.” Such a policy would be typical when I have paid money to both AS X and AS Y to connect my AS to the rest of the Internet, and AS X is my preferred provider of connectivity, with AS Y being the fallback. Because I view both AS X and AS Y as providers (and presumably I paid them to play this role), I don’t expect to help them out by carrying traffic between them across my network (this is called *transit* traffic). The more autonomous systems I connect to, the more complex policies I might have, especially when you consider backbone providers, who may interconnect with dozens of other providers and hundreds of customers and have different economic arrangements (which affect routing policies) with each one.

A key design goal of interdomain routing is that policies like the example above, and much more complex ones, should be supported by the interdomain routing system. To make the problem harder, I need to be able to implement such a policy without any help from other autonomous systems, and in the face of possible misconfiguration or malicious behavior by other autonomous systems. Furthermore, there is often a desire to keep the policies *private*, because the entities that run the autonomous systems—mostly ISPs—are often in competition with each other and don’t want their economic arrangements made public.

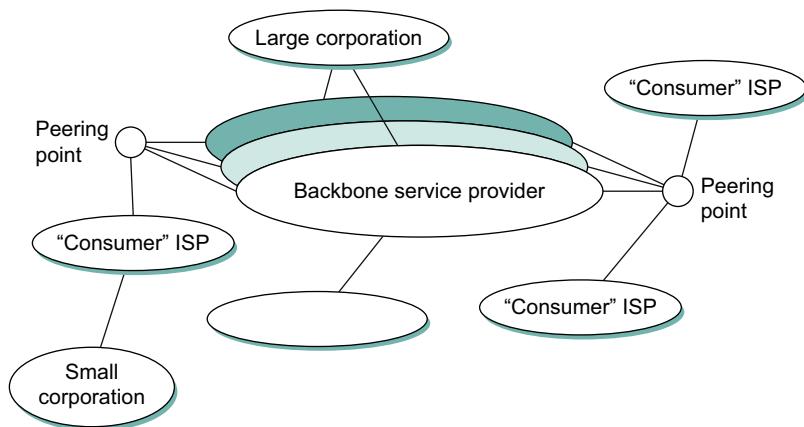
There have been two major interdomain routing protocols in the history of the Internet. The first was the Exterior Gateway Protocol (EGP), which had a number of limitations, perhaps the most severe of which

was that it constrained the topology of the Internet rather significantly. EGP was designed when the Internet had a treelike topology, such as that illustrated in Figure 4.1, and did not allow for the topology to become more general. Note that in this simple treelike structure there is a single backbone, and autonomous systems are connected only as parents and children and not as peers.

The replacement for EGP is the Border Gateway Protocol (BGP), which is in its fourth version at the time of this writing (BGP-4). BGP is often regarded as one of the more complex parts of the Internet. We'll cover some of its high points here.

Unlike its predecessor EGP, BGP makes virtually no assumptions about how autonomous systems are interconnected—they form an arbitrary graph. This model is clearly general enough to accommodate non-tree-structured internetworks, like the simplified picture of a multi-provider Internet shown in Figure 4.4. (It turns out there is still some sort of structure to the Internet, as we'll see below, but it's nothing like as simple as a tree, and BGP makes no assumptions about such structure.)

Unlike the simple tree-structured Internet shown in Figure 4.1, or even the fairly simple picture in Figure 4.4, today's Internet consists of a richly interconnected set of networks, mostly operated by private companies (ISPs) rather than governments. Many Internet Service Providers (ISPs)



■ FIGURE 4.4 A simple multi-provider Internet.

exist mainly to provide service to “consumers” (i.e., individuals with computers in their homes), while others offer something more like the old backbone service, interconnecting other providers and sometimes larger corporations. Often, many providers arrange to interconnect with each other at a single *peering point*.

To get a better sense of how we might manage routing among this complex interconnection of autonomous systems, we can start by defining a few terms. We define *local traffic* as traffic that originates at or terminates on nodes within an AS, and *transit traffic* as traffic that passes through an AS. We can classify autonomous systems into three broad types:

- Stub AS—an AS that has only a single connection to one other AS; such an AS will only carry local traffic. The small corporation in Figure 4.4 is an example of a stub AS.
- Multihomed AS—an AS that has connections to more than one other AS but that refuses to carry transit traffic, such as the large corporation at the top of Figure 4.4.
- Transit AS—an AS that has connections to more than one other AS and that is designed to carry both transit and local traffic, such as the backbone providers in Figure 4.4.

Whereas the discussion of routing in Section 3.3 focused on finding optimal paths based on minimizing some sort of link metric, the goals of interdomain routing are rather more complex. First, it is necessary to find *some* path to the intended destination that is loop free. Second, paths must be compliant with the policies of the various autonomous systems along the path—and, as we have already seen, those policies might be almost arbitrarily complex. Thus, while intradomain focuses on a well-defined problem of optimizing the scalar cost of the path, interdomain focuses on finding a non-looping, *policy-compliant* path—a much more complex optimization problem.

There are additional factors that make interdomain routing hard. The first is simply a matter of scale. An Internet backbone router must be able to forward any packet destined anywhere in the Internet. That means having a routing table that will provide a match for any valid IP address. While CIDR has helped to control the number of distinct prefixes that are carried in the Internet’s backbone routing, there is inevitably a lot of routing

information to pass around—on the order of 300,000 prefixes at the time of writing.<sup>1</sup>

A further challenge in interdomain routing arises from the autonomous nature of the domains. Note that each domain may run its own interior routing protocols and use any scheme it chooses to assign metrics to paths. This means that it is impossible to calculate meaningful path costs for a path that crosses multiple autonomous systems. A cost of 1000 across one provider might imply a great path, but it might mean an unacceptably bad one from another provider. As a result, interdomain routing advertises only *reachability*. The concept of reachability is basically a statement that “you can reach this network through this AS.” This means that for interdomain routing to pick an optimal path is essentially impossible.

The autonomous nature of interdomain raises issue of trust. Provider A might be unwilling to believe certain advertisements from provider B for fear that provider B will advertise erroneous routing information. For example, trusting provider B when he advertises a great route to anywhere in the Internet can be a disastrous choice if provider B turns out to have made a mistake configuring his routers or to have insufficient capacity to carry the traffic.

The issue of trust is also related to the need to support complex policies as noted above. For example, I might be willing to trust a particular provider only when he advertises reachability to certain prefixes, and thus I would have a policy that says, “Use AS X to reach only prefixes *p* and *q*, if and only if AS X advertises reachability to those prefixes.”

#### *Basics of BGP*

Each AS has one or more *border routers* through which packets enter and leave the AS. In our simple example in [Figure 4.3](#), routers R2 and R4 would be border routers. (Over the years, routers have sometimes also been known as *gateways*, hence the names of the protocols BGP and EGP). A border router is simply an IP router that is charged with the task of forwarding packets between autonomous systems.

Each AS that participates in BGP must also have at least one *BGP speaker*, a router that “speaks” BGP to other BGP speakers in other

---

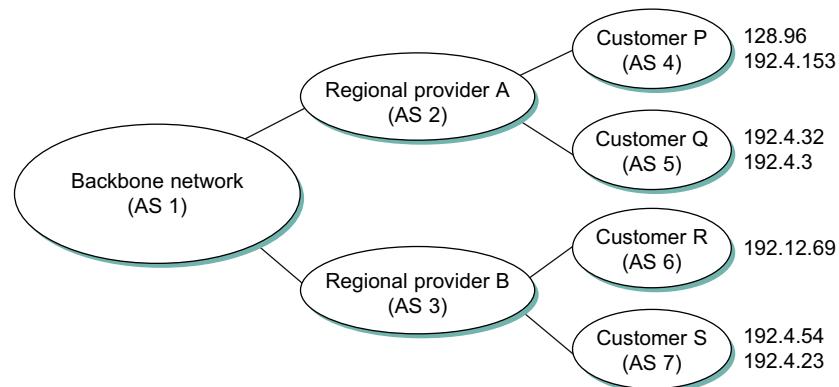
<sup>1</sup>Check the live references at the end of this chapter for a current estimate of this number.

autonomous systems. It is common to find that border routers are also BGP speakers, but that does not have to be the case.

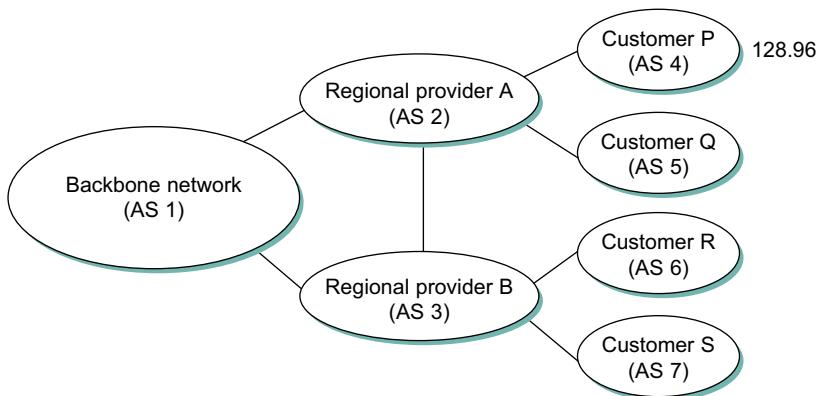
BGP does not belong to either of the two main classes of routing protocols (distance-vector and link-state protocols) described in Section 3.3. Unlike these protocols, BGP advertises *complete paths* as an enumerated list of autonomous systems to reach a particular network. It is sometimes called a *path-vector* protocol for this reason. The advertisement of complete paths is necessary to enable the sorts of policy decisions described above to be made in accordance with the wishes of a particular AS. It also enables routing loops to be readily detected.

To see how this works, consider the very simple example network in Figure 4.5. Assume that the providers are transit networks, while the customer networks are stubs. A BGP speaker for the AS of provider A (AS 2) would be able to advertise reachability information for each of the network numbers assigned to customers P and Q. Thus, it would say, in effect, “The networks 128.96, 192.4.153, 192.4.32, and 192.4.3 can be reached directly from AS 2.” The backbone network, on receiving this advertisement, can advertise, “The networks 128.96, 192.4.153, 192.4.32, and 192.4.3 can be reached along the path (AS 1, AS 2).” Similarly, it could advertise, “The networks 192.12.69, 192.4.54, and 192.4.23 can be reached along the path (AS 1, AS 3).”

An important job of BGP is to prevent the establishment of looping paths. For example, consider the network illustrated in Figure 4.6. It differs from Figure 4.5 only in the addition of an extra link between AS 2



■ FIGURE 4.5 Example of a network running BGP.

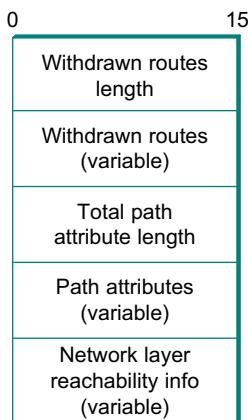


■ FIGURE 4.6 Example of loop among autonomous systems.

and AS 3, but the effect now is that the graph of autonomous systems has a loop in it. Suppose AS 1 learns that it can reach network 128.96 through AS 2, so it advertises this fact to AS 3, who in turn advertises it back to AS 2. In the absence of any loop prevention mechanism, AS 2 could now decide that AS 3 was the preferred route for packets destined for 128.96. If AS 2 starts sending packets addressed to 128.96 to AS 3, AS 3 would send them to AS 1; AS 1 would send them back to AS 2; and they would loop forever. This is prevented by carrying the complete AS path in the routing messages. In this case, the advertisement for a path to 128.96 received by AS 2 from AS 3 would contain an AS path of  $\langle \text{AS 3}, \text{AS 1}, \text{AS 2}, \text{AS 4} \rangle$ . AS 2 sees itself in this path, and thus concludes that this is not a useful path for it to use.

In order for this loop prevention technique to work, the AS numbers carried in BGP clearly need to be unique. For example, AS 2 can only recognize itself in the AS path in the above example if no other AS identifies itself in the same way. AS numbers have until recently been 16-bit numbers, and they are assigned by a central authority to assure uniqueness. While 16 bits only allows about 65,000 autonomous systems, which might not seem like a lot, we note that a stub AS does not need a unique AS number, and this covers the overwhelming majority of nonprovider networks.<sup>2</sup>

<sup>2</sup>32-bit AS numbers have also been defined and came into use around 2009, thus ensuring that AS number space will not become a scarce resource.

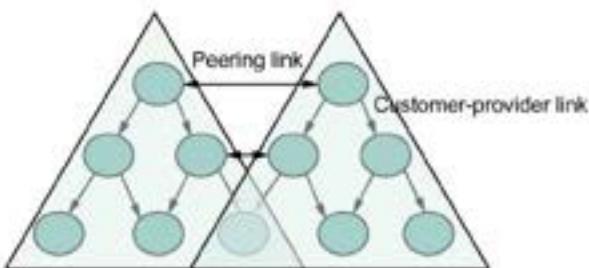


■ FIGURE 4.7 BGP-4 update packet format.

A given AS will only advertise routes that it considers good enough for itself. That is, if a BGP speaker has a choice of several different routes to a destination, it will choose the best one according to its own local policies, and then that will be the route it advertises. Furthermore, a BGP speaker is under no obligation to advertise any route to a destination, even if it has one. This is how an AS can implement a policy of not providing transit—by refusing to advertise routes to prefixes that are not contained within that AS, even if it knows how to reach them.

Given that links fail and policies change, BGP speakers need to be able to cancel previously advertised paths. This is done with a form of negative advertisement known as a *withdrawn route*. Both positive and negative reachability information are carried in a BGP update message, the format of which is shown in Figure 4.7. (Note that the fields in this figure are multiples of 16 bits, unlike other packet formats in this chapter.)

Unlike the routing protocols described in the previous chapter, BGP is defined to run on top of TCP, the reliable transport protocol described in Section 5.2. Because BGP speakers can count on TCP to be reliable, this means that any information that has been sent from one speaker to another does not need to be sent again. Thus, as long as nothing has changed, a BGP speaker can simply send an occasional *keepalive* message that says, in effect, “I’m still here and nothing has changed.” If that router were to crash or become disconnected from its peer, it would stop sending the keepalives, and the other routers that had learned routes from it would assume that those routes were no longer valid.



■ FIGURE 4.8 Common AS relationships.

#### Common AS Relationships and Policies

Having said that policies may be arbitrarily complex, there turn out to be a few common ones, reflecting common relationships between autonomous systems. The most common relationships are illustrated in Figure 4.8. The three common relationships and the policies that go with them are as follows:

- **Provider-Customer**—Providers are in the business of connecting their customers to the rest of the Internet. A customer might be a corporation, or it might be a smaller ISP (which may have customers of its own). So the common policy is to advertise all the routes I know about to my customer, and advertise routes I learn from my customer to everyone.
- **Customer-Provider**—In the other direction, the customer wants to get traffic directed to him (and his customers, if he has them) by his provider, and he wants to be able to send traffic to the rest of the Internet through his provider. So the common policy in this case is to advertise my own prefixes and routes learned from my customers to my provider, advertise routes learned from my provider to my customers, but don't advertise routes learned from one provider to another provider. That last part is to make sure the customer doesn't find himself in the business of carrying traffic from one provider to another, which isn't in his interests if he is paying the providers to carry traffic for him.
- **Peer**—The third option is a symmetrical peering between autonomous systems. Two providers who view themselves as equals usually peer so that they can get access to each other's customers without having to pay another provider. The typical policy here is to advertise routes learned from my customers to my

peer, advertise routes learned from my peer to my customers, but don't advertise routes from my peer to any provider or *vice versa*.

One thing to note about this figure is the way it has brought back some structure to the apparently unstructured Internet. At the bottom of the hierarchy we have the stub networks that are customers of one or more providers, and as we move up the hierarchy we see providers who have other providers as their customers. At the top, we have providers who have customers and peers but are not customers of anyone. These providers are known as the *Tier-1* providers.



Let's return to the real question: How does all this help us to build scalable networks? First, the number of nodes participating in BGP is on the order of the number of autonomous systems, which is much smaller than the number of networks. Second, finding a good interdomain route is only a matter of finding a path to the right border router, of which there are only a few per AS. Thus, we have neatly subdivided the routing problem into manageable parts, once again using a new level of hierarchy to increase scalability. The complexity of interdomain routing is now on the order of the number of autonomous systems, and the complexity of intradomain routing is on the order of the number of networks in a single AS.

#### *Integrating Interdomain and Intradomain Routing*

While the preceding discussion illustrates how a BGP speaker learns interdomain routing information, the question still remains as to how all the other routers in a domain get this information. There are several ways this problem can be addressed.

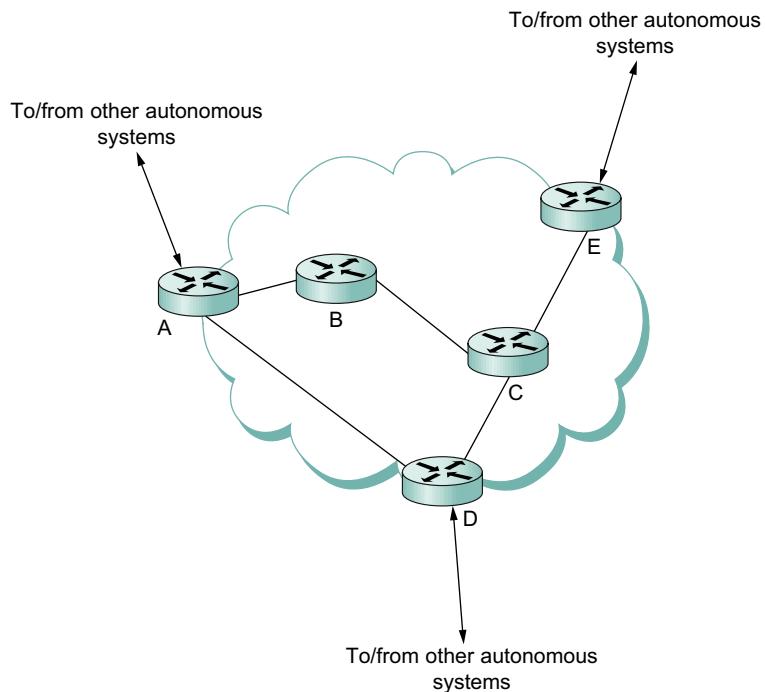
Let's start with a very simple situation, which is also very common. In the case of a stub AS that only connects to other autonomous systems at a single point, the border router is clearly the only choice for all routes that are outside the AS. Such a router can inject a *default route* into the intradomain routing protocol. In effect, this is a statement that any network that has not been explicitly advertised in the intradomain protocol is reachable through the border router. Recall from the discussion of IP forwarding in Section 3.2 that the default entry in the forwarding table comes after all the more specific entries, and it matches anything that failed to match a specific entry.

The next step up in complexity is to have the border routers inject specific routes they have learned from outside the AS. Consider, for example,

the border router of a provider AS that connects to a customer AS. That router could learn that the network prefix 192.4.54/24 is located inside the customer AS, either through BGP or because the information is configured into the border router. It could inject a route to that prefix into the routing protocol running inside the provider AS. This would be an advertisement of the sort, “I have a link to 192.4.54/24 of cost X.” This would cause other routers in the provider AS to learn that this border router is the place to send packets destined for that prefix.

The final level of complexity comes in backbone networks, which learn so much routing information from BGP that it becomes too costly to inject it into the intradomain protocol. For example, if a border router wants to inject 10,000 prefixes that it learned about from another AS, it will have to send very big link-state packets to the other routers in that AS, and their shortest-path calculations are going to become very complex. For this reason, the routers in a backbone network use a variant of BGP called *interior BGP* (iBGP) to effectively redistribute the information that is learned by the BGP speakers at the edges of the AS to all the other routers in the AS. (The other variant of BGP, discussed above, runs between autonomous systems and is called *exterior BGP*, or eBGP). iBGP enables any router in the AS to learn the best border router to use when sending a packet to any address. At the same time, each router in the AS keeps track of how to get to each border router using a conventional intradomain protocol with no injected information. By combining these two sets of information, each router in the AS is able to determine the appropriate next hop for all prefixes.

To see how this all works, consider the simple example network, representing a single AS, in [Figure 4.9](#). The three border routers, A, D, and E, speak eBGP to other autonomous systems and learn how to reach various prefixes. These three border routers communicate with other and with the interior routers B and C by building a mesh of iBGP sessions among all the routers in the AS. Let’s now focus in on how router B builds up its complete view of how to forward packets to any prefix. Look at the table at the top left of [Figure 4.10](#) which shows the information that router B learns from its iBGP sessions. It learns that some prefixes are best reached via router A, some via D, and some via E. At the same time, all the routers in the AS are also running some intradomain routing protocol such as Routing Information Protocol (RIP) or Open Shortest Path First (OSPF). (A generic term for intradomain protocols is an interior gateway protocol, or IGP.) From this completely separate protocol, B learns how



■ **FIGURE 4.9** Example of interdomain and intradomain routing. All routers run iBGP and an intradomain routing protocol. Border routers A, D, and E also run eBGP to other autonomous systems.

to reach other nodes *inside* the domain, as shown in the top right table. For example, to reach router E, B needs to send packets toward router C. Finally, in the bottom table, B puts the whole picture together, combining the information about external prefixes learned from iBGP with the information about interior routes to the border routers learned from the IGP. Thus, if a prefix like 18.0/16 is reachable via border router E, and the best interior path to E is via C, then it follows that any packet destined for 18.0/16 should be forwarded toward C. In this way, any router in the AS can build up a complete routing table for any prefix that is reachable via some border router of the AS.

#### 4.1.3 IP Version 6 (IPv6)

In many respects, the motivation for a new version of IP is simple: to deal with exhaustion of the IP address space. CIDR helped considerably to contain the rate at which the Internet address space is being consumed and also helped to control the growth of routing table information needed in the Internet's routers. However, there will come a point at which these

The diagram illustrates the merging of two routing tables at router B. It starts with two separate tables: the BGP table for the AS and the IGP table for router B. These are then combined into a single, larger table.

Prefix	BGP Next Hop
18.0/16	E
12.5.5/24	A
128.34/16	D
128.69./16	A

BGP table for the AS

Router	IGP Path
A	A
C	C
D	C
E	C

IGP table for router B

Prefix	IGP Path
18.0/16	C
12.5.5/24	A
128.34/16	C
128.69./16	A

Combined table for router B

**FIGURE 4.10** BGP routing table, IGP routing table, and combined table at router B.

techniques are no longer adequate. In particular, it is virtually impossible to achieve 100% address utilization efficiency, so the address space will be exhausted well before the 4 billionth host is connected to the Internet. Even if we were able to use all 4 billion addresses, it's not too hard to imagine ways that that number could be exhausted, now that IP addresses are assigned not just to full-blown computers but also to mobile phones, televisions, and other household appliances. All of these possibilities argue that a bigger address space than that provided by 32 bits will eventually be needed.

#### *Historical Perspective*

The IETF began looking at the problem of expanding the IP address space in 1991, and several alternatives were proposed. Since the IP address is carried in the header of every IP packet, increasing the size of the address dictates a change in the packet header. This means a new version of the Internet Protocol and, as a consequence, a need for new software for every host and router in the Internet. This is clearly not a trivial matter—it is a major change that needs to be thought about very carefully.

The effort to define a new version of IP was known as IP Next Generation, or IPng. As the work progressed, an official IP version number was assigned, so IPng is now known as IPv6. Note that the version of IP discussed so far in this chapter is version 4 (IPv4). The apparent discontinuity in numbering is the result of version number 5 being used for an experimental protocol some years ago.

The significance of changing to a new version of IP caused a snowball effect. The general feeling among network designers was that if you are going to make a change of this magnitude you might as well fix as many other things in IP as possible at the same time. Consequently, the IETF solicited white papers from anyone who cared to write one, asking for input on the features that might be desired in a new version of IP. In addition to the need to accommodate scalable routing and addressing, some of the other wish list items for IPng included:

- Support for real-time services
- Security support
- Autoconfiguration (i.e., the ability of hosts to automatically configure themselves with such information as their own IP address and domain name)
- Enhanced routing functionality, including support for mobile hosts

It is interesting to note that, while many of these features were absent from IPv4 at the time IPv6 was being designed, support for all of them has made its way into IPv4 in recent years, often using similar techniques in both protocols. It can be argued that the freedom to think of IPv6 as a clean slate facilitated the design of new capabilities for IP that were then retrofitted into IPv4.

In addition to the wish list, one absolutely non-negotiable feature for IPng was that there must be a transition plan to move from the current version of IP (version 4) to the new version. With the Internet being so large and having no centralized control, it would be completely impossible to have a “flag day” on which everyone shut down their hosts and routers and installed a new version of IP. Thus, there will probably be a long transition period in which some hosts and routers will run IPv4 only, some will run IPv4 and IPv6, and some will run IPv6 only.

The IETF appointed a committee called the IPng Directorate to collect all the inputs on IPng requirements and to evaluate proposals for a protocol to become IPng. Over the life of this committee there were

numerous proposals, some of which merged with other proposals, and eventually one was chosen by the Directorate to be the basis for IPng. That proposal was called *Simple Internet Protocol Plus* (SIPP). SIPP originally called for a doubling of the IP address size to 64 bits. When the Directorate selected SIPP, they stipulated several changes, one of which was another doubling of the address to 128 bits (16 bytes). It was around this time that version number 6 was assigned. The rest of this section describes some of the main features of IPv6. At the time of this writing, most of the key specifications for IPv6 are Proposed or Draft Standards in the IETF.

#### *Addresses and Routing*

First and foremost, IPv6 provides a 128-bit address space, as opposed to the 32 bits of version 4. Thus, while version 4 can potentially address 4 billion nodes if address assignment efficiency reaches 100%, IPv6 can address  $3.4 \times 10^{38}$  nodes, again assuming 100% efficiency. As we have seen, though, 100% efficiency in address assignment is not likely. Some analysis of other addressing schemes, such as those of the French and U.S. telephone networks, as well as that of IPv4, have turned up some empirical numbers for address assignment efficiency. Based on the most pessimistic estimates of efficiency drawn from this study, the IPv6 address space is predicted to provide over 1500 addresses per square foot of the Earth's surface, which certainly seems like it should serve us well even when toasters on Venus have IP addresses.

#### *Address Space Allocation*

Drawing on the effectiveness of CIDR in IPv4, IPv6 addresses are also classless, but the address space is still subdivided in various ways based on the leading bits. Rather than specifying different address classes, the leading bits specify different uses of the IPv6 address. The current assignment of prefixes is listed in Table 4.1.

This allocation of the address space warrants a little discussion. First, the entire functionality of IPv4's three main address classes (A, B, and C) is contained inside the "everything else" range. Global Unicast Addresses, as we will see shortly, are a lot like classless IPv4 addresses, only much longer. These are the main ones of interest at this point, with over 99% of the total IPv6 address space available to this important form of address. (At the time of writing, IPv6 unicast addresses are being allocated from

**Table 4.1 Address Prefix Assignments for IPv6**

Prefix	Use
00...0 (128 bits)	Unspecified
00...1 (128 bits)	Loopback
1111 1111	Multicast addresses
1111 1110 10	Link-local unicast
Everything else	Global Unicast Addresses

the block that begins 001, with the remaining address space—about 87%—being reserved for future use.)

The multicast address space is (obviously) for multicast, thereby serving the same role as class D addresses in IPv4. Note that multicast addresses are easy to distinguish—they start with a byte of all 1s. We will see how these addresses are used in Section 4.2.

The idea behind link-local use addresses is to enable a host to construct an address that will work on the network to which it is connected without being concerned about the global uniqueness of the address. This may be useful for autoconfiguration, as we will see below. Similarly, the site-local use addresses are intended to allow valid addresses to be constructed on a site (e.g., a private corporate network) that is not connected to the larger Internet; again, global uniqueness need not be an issue.

Within the global unicast address space are some important special types of addresses. A node may be assigned an IPv4-compatible IPv6 address by zero-extending a 32-bit IPv4 address to 128 bits. A node that is only capable of understanding IPv4 can be assigned an IPv4-mapped IPv6 address by prefixing the 32-bit IPv4 address with 2 bytes of all 1s and then zero-extending the result to 128 bits. These two special address types have uses in the IPv4-to-IPv6 transition (see the sidebar on this topic).

#### *Address Notation*

Just as with IPv4, there is some special notation for writing down IPv6 addresses. The standard representation is  $x:x:x:x:x:x$ , where each “x” is a hexadecimal representation of a 16-bit piece of the address. An example would be

47CD:1234:4422:AC02:0022:1234:A456:0124

## Transition from IPv4 to IPv6

The most important idea behind the transition from IPv4 to IPv6 is that the Internet is far too big and decentralized to have a “flag day”—one specified day on which every host and router is upgraded from IPv4 to IPv6. Thus, IPv6 needs to be deployed incrementally in such a way that hosts and routers that only understand IPv4 can continue to function for as long as possible. Ideally, IPv4 nodes should be able to talk to other IPv4 nodes and some set of other IPv6-capable nodes indefinitely. Also, IPv6 hosts should be capable of talking to other IPv6 nodes even when some of the infrastructure between them may only support IPv4. Two major mechanisms have been defined to help this transition: *dual-stack operation* and *tunneling*.

The idea of dual stacks is fairly straightforward: IPv6 nodes run both IPv6 and IPv4 and use the Version field to decide which stack should process an arriving packet. In this case, the IPv6 address could be unrelated to the IPv4 address, or it could be the IPv4-mapped IPv6 address described earlier in this section.

The basic tunneling technique, in which an IP packet is sent as the *payload* of another IP packet, was described in Section 3.2. For IPv6 transition, tunneling is used to send an IPv6 packet over a piece of the network that only understands IPv4. This means that the IPv6 packet is encapsulated within an IPv4 header that has the address of the tunnel endpoint in its header, is transmitted across the IPv4-only piece of network, and then is decapsulated at the endpoint. The endpoint could be either a router or a host; in either case, it must be IPv6 capable to be able to process the IPv6 packet after decapsulation. If the endpoint is a host with an IPv4-mapped IPv6 address, then tunneling can be done automatically by extracting the IPv4 address from the IPv6 address and using it to form the IPv4 header. Otherwise, the tunnel must be configured manually. In this case, the encapsulating node needs to know the IPv4 address of the other end of the tunnel, since it cannot be extracted from the IPv6 header. From the perspective of IPv6, the other end of the tunnel looks like a regular IPv6 node that is just one hop away, even though there may be many hops of IPv4 infrastructure between the tunnel endpoints.

Any IPv6 address can be written using this notation. Since there are a few special types of IPv6 addresses, there are some special notations that may be helpful in certain circumstances. For example, an address with a large number of contiguous 0s can be written more compactly by omitting all the 0 fields. Thus,

47CD:0000:0000:0000:0000:A456:0124

could be written

47CD::A456:0124

Clearly, this form of shorthand can only be used for one set of contiguous 0s in an address to avoid ambiguity.

The two types of IPv6 addresses that contain an embedded IPv4 address have their own special notation that makes extraction of the IPv4 address easier. For example, the IPv4-mapped IPv6 address of a host whose IPv4 address was 128.96.33.81 could be written as

::FFFF:128.96.33.81

That is, the last 32 bits are written in IPv4 notation, rather than as a pair of hexadecimal numbers separated by a colon. Note that the double colon at the front indicates the leading 0s.

#### *Global Unicast Addresses*

By far the most important sort of addressing that IPv6 must provide is plain old unicast addressing. It must do this in a way that supports the rapid rate of addition of new hosts to the Internet and that allows routing to be done in a scalable way as the number of physical networks in the Internet grows. Thus, at the heart of IPv6 is the unicast address allocation plan that determines how unicast addresses will be assigned to service providers, autonomous systems, networks, hosts, and routers.

In fact, the address allocation plan that is proposed for IPv6 unicast addresses is extremely similar to that being deployed with CIDR in IPv4. To understand how it works and how it provides scalability, it is helpful to define some new terms. We may think of a nontransit AS (i.e., a stub or multihomed AS) as a *subscriber*, and we may think of a transit AS as a *provider*. Furthermore, we may subdivide providers into *direct* and *indirect*. The former are directly connected to subscribers. The latter primarily connect other providers, are not connected directly to subscribers, and are often known as *backbone networks*.

With this set of definitions, we can see that the Internet is not just an arbitrarily interconnected set of autonomous systems; it has some intrinsic hierarchy. The difficulty lies in making use of this hierarchy without inventing mechanisms that fail when the hierarchy is not strictly observed, as happened with EGP. For example, the distinction between direct and indirect providers becomes blurred when a subscriber

connects to a backbone or when a direct provider starts connecting to many other providers.

As with CIDR, the goal of the IPv6 address allocation plan is to provide aggregation of routing information to reduce the burden on intradomain routers. Again, the key idea is to use an address prefix—a set of contiguous bits at the most significant end of the address—to aggregate reachability information to a large number of networks and even to a large number of autonomous systems. The main way to achieve this is to assign an address prefix to a direct provider and then for that direct provider to assign longer prefixes that begin with that prefix to its subscribers. This is exactly what we observed in [Figure 3.22](#). Thus, a provider can advertise a single prefix for all of its subscribers.

Of course, the drawback is that if a site decides to change providers, it will need to obtain a new address prefix and renumber all the nodes in the site. This could be a colossal undertaking, enough to dissuade most people from ever changing providers. For this reason, there is ongoing research on other addressing schemes, such as geographic addressing, in which a site's address is a function of its location rather than the provider to which it attaches. At present, however, provider-based addressing is necessary to make routing work efficiently.

Note that while IPv6 address assignment is essentially equivalent to the way address assignment has happened in IPv4 since the introduction of CIDR, IPv6 has the significant advantage of not having a large installed base of assigned addresses to fit into its plans.

One question is whether it makes sense for hierarchical aggregation to take place at other levels in the hierarchy. For example, should all providers obtain their address prefixes from within a prefix allocated to the backbone to which they connect? Given that most providers connect to multiple backbones, this probably doesn't make sense. Also, since the number of providers is much smaller than the number of sites, the benefits of aggregating at this level are much fewer.

One place where aggregation may make sense is at the national or continental level. Continental boundaries form natural divisions in the Internet topology. If all addresses in Europe, for example, had a common prefix, then a great deal of aggregation could be done, and most routers in other continents would only need one routing table entry for all networks with the Europe prefix. Providers in Europe would all select their prefixes such that they began with the European prefix. Using this scheme, an IPv6 address might look like [Figure 4.11](#). The RegistryID might be an identifier

3	m	n	o	p	125-m-n-o-p
010	RegistryID	ProviderID	SubscriberID	SubnetID	InterfaceID

■ FIGURE 4.11 An IPv6 provider-based unicast address.

assigned to a European address registry, with different IDs assigned to other continents or countries. Note that prefixes would be of different lengths under this scenario. For example, a provider with few customers could have a longer prefix (and thus less total address space available) than one with many customers.

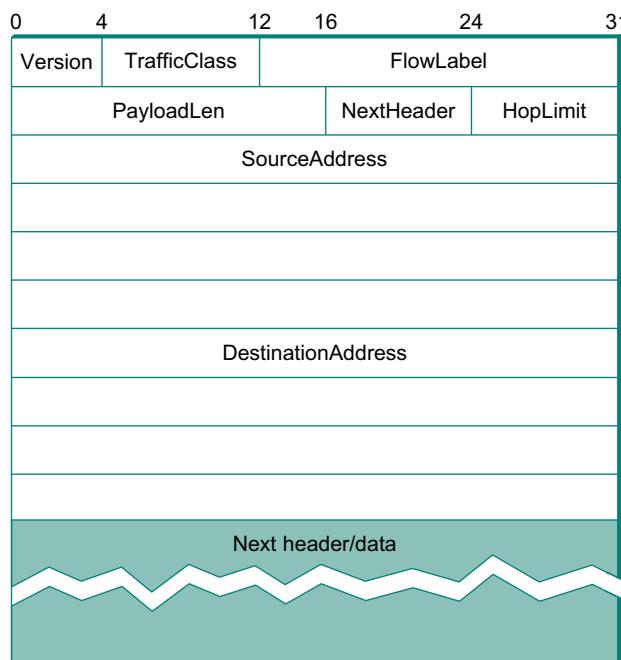
One tricky situation could occur when a subscriber is connected to more than one provider. Which prefix should the subscriber use for his or her site? There is no perfect solution to the problem. For example, suppose a subscriber is connected to two providers, X and Y. If the subscriber takes his prefix from X, then Y has to advertise a prefix that has no relationship to its other subscribers and that as a consequence cannot be aggregated. If the subscriber numbers part of his AS with the prefix of X and part with the prefix of Y, he runs the risk of having half his site become unreachable if the connection to one provider goes down. One solution that works fairly well if X and Y have a lot of subscribers in common is for them to have three prefixes between them: one for subscribers of X only, one for subscribers of Y only, and one for the sites that are subscribers of both X and Y.

#### *Packet Format*

Despite the fact that IPv6 extends IPv4 in several ways, its header format is actually simpler. This simplicity is due to a concerted effort to remove unnecessary functionality from the protocol. Figure 4.12 shows the result. (For comparison with IPv4, see the header format shown in Figure 3.16.)

As with many headers, this one starts with a Version field, which is set to 6 for IPv6. The Version field is in the same place relative to the start of the header as IPv4's Version field so that header-processing software can immediately decide which header format to look for. The TrafficClass and FlowLabel fields both relate to quality of service issues, as discussed in Section 6.5.

The PayloadLen field gives the length of the packet, excluding the IPv6 header, measured in bytes. The NextHeader field cleverly replaces both the IP options and the Protocol field of IPv4. If options are required, then



■ FIGURE 4.12 IPv6 packet header.

they are carried in one or more special headers following the IP header, and this is indicated by the value of the `NextHeader` field. If there are no special headers, the `NextHeader` field is the demux key identifying the higher-level protocol running over IP (e.g., TCP or UDP); that is, it serves the same purpose as the IPv4 `Protocol` field. Also, fragmentation is now handled as an optional header, which means that the fragmentation-related fields of IPv4 are not included in the IPv6 header. The `HopLimit` field is simply the TTL of IPv4, renamed to reflect the way it is actually used.

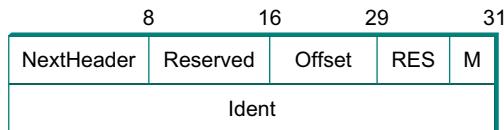
Finally, the bulk of the header is taken up with the source and destination addresses, each of which is 16 bytes (128 bits) long. Thus, the IPv6 header is always 40 bytes long. Considering that IPv6 addresses are four times longer than those of IPv4, this compares quite well with the IPv4 header, which is 20 bytes long in the absence of options.

The way that IPv6 handles options is quite an improvement over IPv4. In IPv4, if any options were present, every router had to parse the entire options field to see if any of the options were relevant. This is because

the options were all buried at the end of the IP header, as an unordered collection of  $\langle$ type, length, value $\rangle$  tuples. In contrast, IPv6 treats options as *extension headers* that must, if present, appear in a specific order. This means that each router can quickly determine if any of the options are relevant to it; in most cases, they will not be. Usually this can be determined by just looking at the NextHeader field. The end result is that option processing is much more efficient in IPv6, which is an important factor in router performance. In addition, the new formatting of options as extension headers means that they can be of arbitrary length, whereas in IPv4 they were limited to 44 bytes at most. We will see how some of the options are used below.

Each option has its own type of extension header. The type of each extension header is identified by the value of the NextHeader field in the header that precedes it, and each extension header contains a NextHeader field to identify the header following it. The last extension header will be followed by a transport-layer header (e.g., TCP) and in this case the value of the NextHeader field is the same as the value of the Protocol field would be in an IPv4 header. Thus, the NextHeader field does double duty; it may either identify the type of extension header to follow, or, in the last extension header, it serves as a demux key to identify the higher-layer protocol running over IPv6.

Consider the example of the fragmentation header, shown in Figure 4.13. This header provides functionality similar to the fragmentation fields in the IPv4 header described in Section 3.2.2, but it is only present if fragmentation is necessary. Assuming it is the only extension header present, then the NextHeader field of the IPv6 header would contain the value 44, which is the value assigned to indicate the fragmentation header. The NextHeader field of the fragmentation header itself contains a value describing the header that follows it. Again, assuming no other extension headers are present, then the next header might be the TCP header, which results in NextHeader containing the value 6, just as the Protocol field would in IPv4. If the fragmentation header



■ FIGURE 4.13 IPv6 fragmentation extension header.

were followed by, say, an authentication header, then the fragmentation header's NextHeader field would contain the value 51.

#### Autoconfiguration

While the Internet's growth has been impressive, one factor that has inhibited faster acceptance of the technology is the fact that getting connected to the Internet has typically required a fair amount of system administration expertise. In particular, every host that is connected to the Internet needs to be configured with a certain minimum amount of information, such as a valid IP address, a subnet mask for the link to which it attaches, and the address of a name server. Thus, it has not been possible to unpack a new computer and connect it to the Internet without some preconfiguration. One goal of IPv6, therefore, is to provide support for autoconfiguration, sometimes referred to as *plug-and-play* operation.

As we saw in Section 3.2.7, autoconfiguration is possible for IPv4, but it depends on the existence of a server that is configured to hand out addresses and other configuration information to Dynamic Host Configuration Protocol (DHCP) clients. The longer address format in IPv6 helps provide a useful, new form of autoconfiguration called *stateless* autoconfiguration, which does not require a server.

Recall that IPv6 unicast addresses are hierarchical, and that the least significant portion is the interface ID. Thus, we can subdivide the auto-configuration problem into two parts:

1. Obtain an interface ID that is unique on the link to which the host is attached.
2. Obtain the correct address prefix for this subnet.

#### Network Address Translation

While IPv6 was motivated by a concern that increased usage of IP would lead to exhaustion of the address space, another technology has become popular as a way to conserve IP address space. That technology is network address translation (NAT), and its widespread use is one main reason why IPv6 deployment remains in its early stages. NAT is viewed by some as "architecturally impure," but it is also a fact of networking life that cannot be ignored.

The basic idea behind NAT is that all the hosts that might communicate with each other over the Internet do not need to have globally unique addresses. Instead, a host could be assigned a "private address" that is not necessarily globally unique, but is unique within some more limited

scope—for example, within the corporate network where the host resides. The class A network number 10 is often used for this purpose, since that network number was assigned to the ARPANET and is no longer in use as a globally unique address. As long as the host communicates only with other hosts in the corporate network, a locally unique address is sufficient. If it should want to communicate with a host outside the corporate network, it does so via a *NAT box*, a device that is able to translate from the private address used by the host to some globally unique address that is assigned to the NAT box. Since it's likely that a small subset of the hosts in the corporation requires the services of the NAT box at any one time, the NAT box might be able to get by with a small pool of globally unique addresses, much smaller than the number of addresses that would be needed if every host in the corporation had a globally unique address.

So, we can imagine a NAT box receiving IP packets from a host inside the corporation and translating the IP source address from some private address (say, 10.0.1.5) to a globally unique address (say, 171.69.210.246). When packets come back from the remote host addressed to 171.69.210.246, the NAT box translates the destination address to 10.0.1.5 and forwards the packet on toward the host.

The chief drawback of NAT is that it breaks a key assumption of the IP service model—that all nodes have globally unique addresses. It turns out that lots of applications and protocols rely on this assumption. Some protocols that run over IP (e.g., application protocols such as FTP) carry IP addresses in their messages. These addresses also need to be translated by a NAT box if the higher-layer protocol is to work properly, and thus NAT boxes become much more complex than simple IP header translators. They potentially need to understand an ever-growing number of higher-layer protocols. This in turn presents an obstacle to deployment of new applications.

Even more serious is the fact that NATs make it difficult for an outside device to initiate a connection to a device on the private side of the NAT, since, in the absence of an established mapping in the NAT device, there is no public address to which to send the connection request. This situation has complicated the deployment of many applications such as Voice over IP.

It is probably safe to say that networks would be better off without NAT, but its disappearance seems unlikely. While widespread deployment of IPv6 would probably help, NAT is now popular for a range of other reasons beyond its original purpose. For example, it becomes easier to switch providers if your entire internal network has (private) IP addresses that bear no relation to the provider's address space. And, while NAT boxes cannot be considered a true solution to security threats, the fact that the addresses behind a NAT box are not globally meaningful provides a level of protection against simple attacks. It will be interesting to see how NAT fares in the future as IPv6 deployment gathers momentum.

The first part turns out to be rather easy, since every host on a link must have a unique link-level address. For example, all hosts on an Ethernet have a unique 48-bit Ethernet address. This can be turned into a valid link-local use address by adding the appropriate prefix from Table 4.1 (1111 1110 10) followed by enough 0s to make up 128 bits. For some devices—for example, printers or hosts on a small routerless network that do not connect to any other networks—this address may be perfectly adequate. Those devices that need a globally valid address depend on a router on the same link to periodically advertise the appropriate prefix for the link. Clearly, this requires that the router be configured with the correct address prefix, and that this prefix be chosen in such a way that there is enough space at the end (e.g., 48 bits) to attach an appropriate link-level address.

The ability to embed link-level addresses as long as 48 bits into IPv6 addresses was one of the reasons for choosing such a large address size. Not only does 128 bits allow the embedding, but it leaves plenty of space for the multilevel hierarchy of addressing that we discussed above.

#### *Advanced Routing Capabilities*

Another of IPv6's extension headers is the routing header. In the absence of this header, routing for IPv6 differs very little from that of IPv4 under CIDR. The routing header contains a list of IPv6 addresses that represent nodes or topological areas that the packet should visit en route to its destination. A topological area may be, for example, a backbone provider's network. Specifying that packets must visit this network would be a way of implementing provider selection on a packet-by-packet basis. Thus, a host could say that it wants some packets to go through a provider that is cheap, others through a provider that provides high reliability, and still others through a provider that the host trusts to provide security.

To provide the ability to specify topological entities rather than individual nodes, IPv6 defines an *anycast* address. An anycast address is assigned to a set of interfaces, and packets sent to that address will go to the “nearest” of those interfaces, with nearest being determined by the routing protocols. For example, all the routers of a backbone provider could be assigned a single anycast address, which would be used in the routing header.

The anycast address and the routing header are also expected to be used to provide enhanced routing support to mobile hosts. The detailed mechanisms for providing this support are still being defined.

#### *Other Features*

As mentioned at the beginning of this section, the primary motivation behind the development of IPv6 was to support the continued growth of the Internet. Once the IP header had to be changed for the sake of the addresses, however, the door was open for a wide variety of other changes, two of which we have just described—autoconfiguration and source-directed routing. IPv6 includes several additional features, most of which are covered elsewhere in this book—mobility is discussed in Section 4.4.2, network security is the topic of Chapter 8, and a new service model proposed for the Internet is described in Section 6.5. It is interesting to note that, in most of these areas, the IPv4 and IPv6 capabilities have become virtually indistinguishable, so that the main driver for IPv6 remains the need for larger addresses.

## 4.2 MULTICAST

As we saw in Chapter 2, multi-access networks like Ethernet implement multicast in hardware. There are, however, applications that need a broader multicasting capability that is effective at the scale of internetworks. For example, when a radio station is broadcast over the Internet, the same data must be sent to all the hosts where a user has tuned in to that station. In that example, the communication is one-to-many. Other examples of one-to-many applications include transmitting the same news, current stock prices, or software updates to multiple hosts. There are also applications whose communication is many-to-many, such as multimedia teleconferencing, online multiplayer gaming, or distributed simulations. In such cases, members of a group receive data from multiple senders, typically each other. From any particular sender, they all receive the same data.

Normal IP communication, in which each packet must be addressed and sent to a single host, is not well suited to such applications. If an application has data to send to a group, it would have to send a separate packet with the identical data to each member of the group. This redundancy consumes more bandwidth than necessary. Furthermore, the

redundant traffic is not distributed evenly but rather is focused around the sending host, and may easily exceed the capacity of the sending host and the nearby networks and routers.

To better support many-to-many and one-to-many communication, IP provides an IP-level multicast analogous to the link-level multicast provided by multi-access networks like Ethernet as we saw in [Chapter 2](#). Now that we are introducing the concept of multicast for IP, we also need a term for the traditional one-to-one service of IP that has been described so far: That service is referred to as *unicast*.

The basic IP multicast model is a many-to-many model based on multicast *groups*, where each group has its own IP *multicast address*. The hosts that are members of a group receive copies of any packets sent to that group's multicast address. A host can be in multiple groups, and it can join and leave groups freely by telling its local router using a protocol that we will discuss shortly. Thus, while we think of unicast addresses as being associated with a node or an interface, multicast addresses are associated with an abstract group, the membership of which changes dynamically over time. Further, the original IP multicast service model allows *any* host to send multicast traffic to a group; it doesn't have to be a member of the group, and there may be any number of such senders to a given group.

Using IP multicast to send the identical packet to each member of the group, a host sends a single copy of the packet addressed to the group's multicast address. The sending host doesn't need to know the individual unicast IP address of each member of the group because, as we will see, that knowledge is distributed among the routers in the internetwork. Similarly, the sending host doesn't need to send multiple copies of the packet because the routers will make copies whenever they have to forward the packet over more than one link. Compared to using unicast IP to deliver the same packets to many receivers, IP multicast is more scalable because it eliminates the redundant traffic (packets) that would have been sent many times over the same links, especially those near to the sending host.

IP's original many-to-many multicast has been supplemented with support for a form of one-to-many multicast. In this model of one-to-many multicast, called *Source-Specific Multicast* (SSM), a receiving host specifies both a multicast group and a specific sending host. The receiving

host would then receive multicasts addressed to the specified group, but only if they are from the specified sender. Many Internet multicast applications (e.g., radio broadcasts) fit the SSM model. To contrast it with SSM, IP's original many-to-many model is sometimes referred to as *Any Source Multicast* (ASM).

A host signals its desire to join or leave a multicast group by communicating with its local router using a special protocol for just that purpose. In IPv4, that protocol is the *Internet Group Management Protocol* (IGMP); in IPv6, it is *Multicast Listener Discovery* (MLD). The router then has the responsibility for making multicast behave correctly with regard to that host. Because a host may fail to leave a multicast group when it should (after a crash or other failure, for example), the router periodically polls the LAN to determine which groups are still of interest to the attached hosts.

### 4.2.1 Multicast Addresses

IP has a subrange of its address space reserved for multicast addresses. In IPv4, these addresses are assigned in the class D address space, and IPv6 also has a portion of its address space (see Table 4.1) reserved for multicast group addresses. Some subranges of the multicast ranges are reserved for intradomain multicast, so they can be reused independently by different domains.

There are thus 28 bits of possible multicast address in IPv4 when we ignore the prefix shared by all multicast addresses. This presents a problem when attempting to take advantage of hardware multicasting on a local area network (LAN). Let's take the case of Ethernet. Ethernet multicast addresses have only 23 bits when we ignore their shared prefix. In other words, to take advantage of Ethernet multicasting, IP has to map 28-bit IP multicast addresses into 23-bit Ethernet multicast addresses. This is implemented by taking the low-order 23 bits of any IP multicast address to use as its Ethernet multicast address and ignoring the high-order 5 bits. Thus, 32 ( $2^5$ ) IP addresses map into each one of the Ethernet addresses.

When a host on an Ethernet joins an IP multicast group, it configures its Ethernet interface to receive any packets with the corresponding Ethernet multicast address. Unfortunately, this causes the receiving host to receive not only the multicast traffic it desired but also traffic sent to any of the

other 31 IP multicast groups that map to the same Ethernet address, if they are routed to that Ethernet. Therefore, IP at the receiving host must examine the IP header of any multicast packet to determine whether the packet really belongs to the desired group. In summary, the mismatch of multicast address sizes means that multicast traffic may place a burden on hosts that are not even interested in the group to which the traffic was sent. Fortunately, in some switched networks (such as switched Ethernet) this problem can be mitigated by schemes wherein the switches recognize unwanted packets and discard them.

One perplexing question is how senders and receivers learn which multicast addresses to use in the first place. This is normally handled by out-of-band means, and there are some quite sophisticated tools to enable group addresses to be advertised on the Internet. One example is `sdr`, discussed in Section 9.2.1.

## 4.2.2 Multicast Routing (DVMRP, PIM, MSDP)

A router's unicast forwarding tables indicate, for any IP address, which link to use to forward the unicast packet. To support multicast, a router must additionally have multicast forwarding tables that indicate, based on multicast address, which links—possibly more than one—to use to forward the multicast packet (the router duplicates the packet if it is to be forwarded over multiple links). Thus, where unicast forwarding tables collectively specify a set of paths, multicast forwarding tables collectively specify a set of trees: *multicast distribution trees*. Furthermore, to support Source-Specific Multicast (and, it turns out, for some types of Any Source Multicast), the multicast forwarding tables must indicate which links to use based on the combination of multicast address and the (unicast) IP address of the source, again specifying a set of trees.

Multicast routing is the process by which the multicast distribution trees are determined or, more concretely, the process by which the multicast forwarding tables are built. As with unicast routing, it is not enough that a multicast routing protocol “work”; it must also scale reasonably well as the network grows, and it must accommodate the autonomy of different routing domains.

### DVMRP

Distance-vector routing, which we discussed in Section 3.3.2 for unicast, can be extended to support multicast. The resulting protocol is called

*Distance Vector Multicast Routing Protocol*, or DVMRP. DVMRP was the first multicast routing protocol to see widespread use.

Recall that, in the distance-vector algorithm, each router maintains a table of  $\langle \text{Destination}, \text{Cost}, \text{NextHop} \rangle$  tuples, and exchanges a list of  $\langle \text{Destination}, \text{Cost} \rangle$  pairs with its directly connected neighbors. Extending this algorithm to support multicast is a two-stage process. First, we create a broadcast mechanism that allows a packet to be forwarded to all the networks on the internet. Second, we need to refine this mechanism so that it prunes back networks that do not have hosts that belong to the multicast group. Consequently, DVMRP is one of several multicast routing protocols described as *flood-and-prune* protocols.

Given a unicast routing table, each router knows that the current shortest path to a given destination goes through `NextHop`. Thus, whenever it receives a multicast packet from source S, the router forwards the packet on all outgoing links (except the one on which the packet arrived) if and only if the packet arrived over the link that is on the shortest path to S (i.e., the packet came *from* the `NextHop` associated with S in the routing table). This strategy effectively floods packets outward from S but does not loop packets back toward S.

There are two major shortcomings to this approach. The first is that it truly floods the network; it has no provision for avoiding LANs that have no members in the multicast group. We address this problem below. The second limitation is that a given packet will be forwarded over a LAN by each of the routers connected to that LAN. This is due to the forwarding strategy of flooding packets on all links other than the one on which the packet arrived, without regard to whether or not those links are part of the shortest-path tree rooted at the source.

The solution to this second limitation is to eliminate the duplicate broadcast packets that are generated when more than one router is connected to a given LAN. One way to do this is to designate one router as the *parent* router for each link, relative to the source, where only the parent router is allowed to forward multicast packets from that source over the LAN. The router that has the shortest path to source S is selected as the parent; a tie between two routers would be broken according to which router has the smallest address. A given router can learn if it is the parent for the LAN (again relative to each possible source) based upon the distance-vector messages it exchanges with its neighbors.

Notice that this refinement requires that each router keep, for each source, a bit for each of its incident links indicating whether or not it is the parent for that source/link pair. Keep in mind that in an internet setting, a source is a network, not a host, since an internet router is only interested in forwarding packets between networks. The resulting mechanism is sometimes called *Reverse Path Broadcast* (RPB) or *Reverse Path Forwarding* (RPF). The path is reverse because we are considering the shortest path toward the *source* when making our forwarding decisions, as compared to unicast routing, which looks for the shortest path to a given *destination*.

The RPB mechanism just described implements shortest-path broadcast. We now want to prune the set of networks that receives each packet addressed to group G to exclude those that have no hosts that are members of G. This can be accomplished in two stages. First, we need to recognize when a *leaf* network has no group members. Determining that a network is a leaf is easy—if the parent router as described above is the only router on the network, then the network is a leaf. Determining if any group members reside on the network is accomplished by having each host that is a member of group G periodically announce this fact over the network, as described in our earlier description of link-state multicast. The router then uses this information to decide whether or not to forward a multicast packet addressed to G over this LAN.

The second stage is to propagate this “no members of G here” information up the shortest-path tree. This is done by having the router augment the  $\langle \text{Destination}, \text{Cost} \rangle$  pairs it sends to its neighbors with the set of groups for which the leaf network is interested in receiving multicast packets. This information can then be propagated from router to router, so that for each of its links a given router knows for what groups it should forward multicast packets.

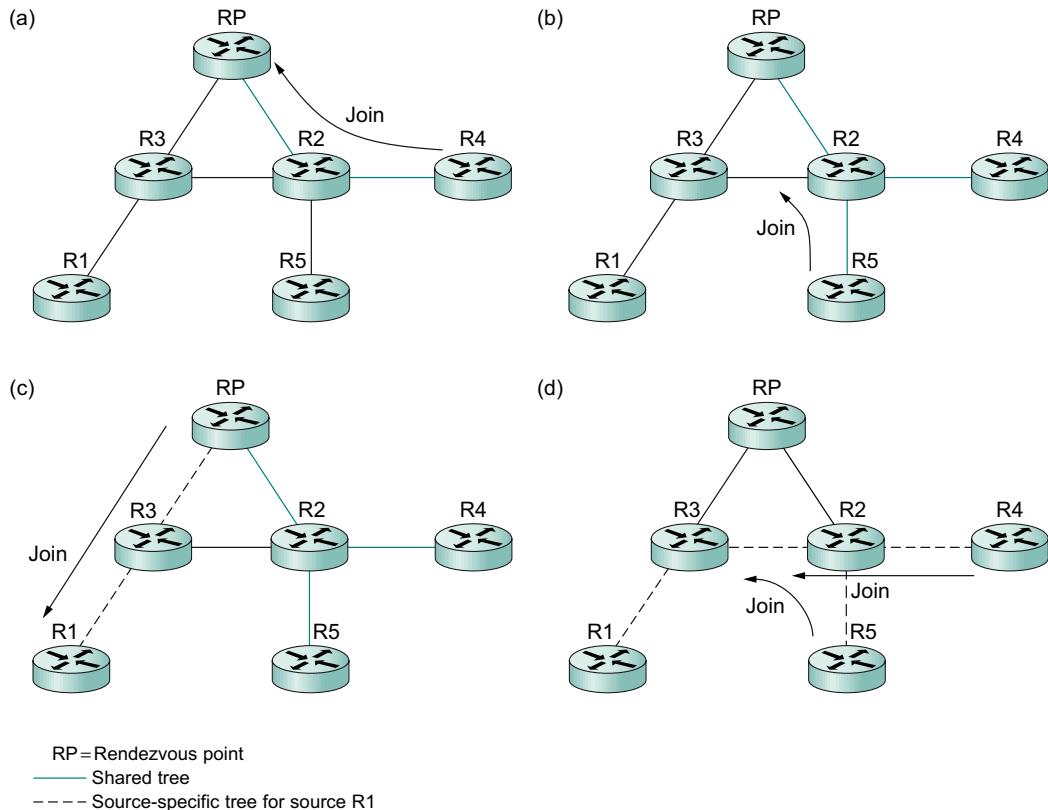
Note that including all of this information in the routing update is a fairly expensive thing to do. In practice, therefore, this information is exchanged only when some source starts sending packets to that group. In other words, the strategy is to use RPB, which adds a small amount of overhead to the basic distance-vector algorithm, until a particular multicast address becomes active. At that time, routers that are not interested in receiving packets addressed to that group speak up, and that information is propagated to the other routers.

### PIM-SM

*Protocol Independent Multicast*, or PIM, was developed in response to the scaling problems of earlier multicast routing protocols. In particular, it was recognized that the existing protocols did not scale well in environments where a relatively small proportion of routers want to receive traffic for a certain group. For example, broadcasting traffic to all routers until they explicitly ask to be removed from the distribution is not a good design choice if most routers don't want to receive the traffic in the first place. This situation is sufficiently common that PIM divides the problem space into *sparse mode* and *dense mode*, where sparse and dense refer to the proportion of routers that will want the multicast. PIM dense mode (PIM-DM) uses a flood-and-prune algorithm like DVMRP and suffers from the same scalability problem. PIM sparse mode (PIM-SM) has become the dominant multicast routing protocol and is the focus of our discussion here. The “protocol independent” aspect of PIM, by the way, refers to the fact that, unlike earlier protocols such as DVMRP, PIM does not depend on any particular sort of unicast routing—it can be used with any unicast routing protocol, as we will see below.

In PIM-SM, routers explicitly join the multicast distribution tree using PIM protocol messages known as *Join* messages. Note the contrast to DVMRP's approach of creating a broadcast tree first and then pruning the uninterested routers. The question that arises is where to send those Join messages because, after all, any host (and any number of hosts) could send to the multicast group. To address this, PIM-SM assigns to each group a special router known as the *rendezvous point* (RP). In general, a number of routers in a domain are configured to be candidate RPs, and PIM-SM defines a set of procedures by which all the routers in a domain can agree on the router to use as the RP for a given group. These procedures are rather complex, as they must deal with a wide variety of scenarios, such as the failure of a candidate RP and the partitioning of a domain into two separate networks due to a number of link or node failures. For the rest of this discussion, we assume that all routers in a domain know the unicast IP address of the RP for a given group.

A multicast forwarding tree is built as a result of routers sending Join messages to the RP. PIM-SM allows two types of trees to be constructed: a *shared* tree, which may be used by all senders, and a *source-specific* tree, which may be used only by a specific sending host. The normal mode of operation creates the shared tree first, followed by one or more source-specific trees if there is enough traffic to warrant it. Because



■ **FIGURE 4.14** PIM operation: (a) R4 sends **Join** to RP and joins shared tree; (b) R5 joins shared tree; (c) RP builds source-specific tree to R1 by sending **Join** to R1; (d) R4 and R5 build source-specific tree to R1 by sending **Joins** to R1.

building trees installs state in the routers along the tree, it is important that the default is to have only one tree for a group, not one for every sender to a group.

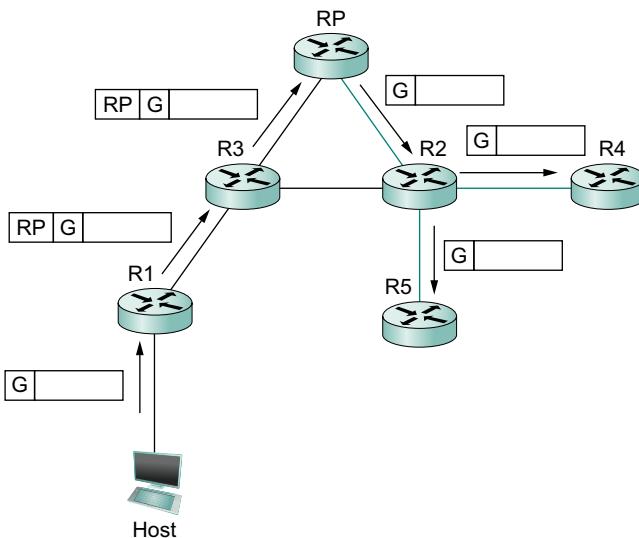
When a router sends a **Join** message toward the RP for a group G, it is sent using normal IP unicast transmission. This is illustrated in Figure 4.14(a), in which router R4 is sending a **Join** to the rendezvous point for some group. The initial **Join** message is “wildcarded”; that is, it applies to all senders. A **Join** message clearly must pass through some sequence of routers before reaching the RP (e.g., R2). Each router along the path looks at the **Join** and creates a forwarding table entry for the shared tree, called a  $(*, G)$  entry (where \* means “all senders”). To create the forwarding table

entry, it looks at the interface on which the Join arrived and marks that interface as one on which it should forward data packets for this group. It then determines which interface it will use to forward the Join toward the RP. This will be the only acceptable interface for incoming packets sent to this group. It then forwards the Join toward the RP. Eventually, the message arrives at the RP, completing the construction of the tree branch. The shared tree thus constructed is shown as a solid line from the RP to R4 in Figure 4.14(a).

As more routers send Joins toward the RP, they cause new branches to be added to the tree, as illustrated in Figure 4.14(b). Note that, in this case, the Join only needs to travel to R2, which can add the new branch to the tree simply by adding a new outgoing interface to the forwarding table entry created for this group. R2 need not forward the Join on to the RP. Note also that the end result of this process is to build a tree whose root is the RP.

At this point, suppose a host wishes to send a message to the group. To do so, it constructs a packet with the appropriate multicast group address as its destination and sends it to a router on its local network known as the *designated router* (DR). Suppose the DR is R1 in Figure 4.14. There is no state for this multicast group between R1 and the RP at this point, so instead of simply forwarding the multicast packet, R1 *tunnels* it to the RP. That is, R1 encapsulates the multicast packet inside a PIM Register message that it sends to the unicast IP address of the RP. Just like a tunnel endpoint of the sort described in Section 3.2.9, the RP receives the packet addressed to it, looks at the payload of the Register message, and finds inside an IP packet addressed to the multicast address of this group. The RP, of course, does know what to do with such a packet—it sends it out onto the shared tree of which the RP is the root. In the example of Figure 4.14, this means that the RP sends the packet on to R2, which is able to forward it on to R4 and R5. The complete delivery of a packet from R1 to R4 and R5 is shown in Figure 4.15. We see the tunneled packet travel from R1 to the RP with an extra IP header containing the unicast address of RP, and then the multicast packet addressed to G making its way along the shared tree to R4 and R5.

At this point, we might be tempted to declare success, since all hosts can send to all receivers this way. However, there is some bandwidth inefficiency and processing cost in the encapsulation and decapsulation of packets on the way to the RP, so the RP forces knowledge about this



**FIGURE 4.15** Delivery of a packet along a shared tree. R1 tunnels the packet to the RP, which forwards it along the shared tree to R4 and R5.

group into the intervening routers so tunneling can be avoided. It sends a Join message toward the sending host (Figure 4.14(c)). As this Join travels toward the host, it causes the routers along the path (R3) to learn about the group, so that it will be possible for the DR to send the packet to the group as *native* (i.e., not tunneled) multicast packets.

An important detail to note at this stage is that the Join message sent by the RP to the sending host is specific to that sender, whereas the previous ones sent by R4 and R5 applied to all senders. Thus, the effect of the new Join is to create *sender-specific* state in the routers between the identified source and the RP. This is referred to as (S, G) state, since it applies to one sender to one group, and contrasts with the (\*, G) state that was installed between the receivers and the RP that applies to all senders. Thus, in Figure 4.14(c), we see a source-specific route from R1 to the RP (indicated by the dashed line) and a tree that is valid for all senders from the RP to the receivers (indicated by the solid line).

The next possible optimization is to replace the entire shared tree with a source-specific tree. This is desirable because the path from sender to receiver via the RP might be significantly longer than the shortest possible path. This again is likely to be triggered by a high data rate being observed

from some sender. In this case, the router at the downstream end of the tree—say, R4 in our example—sends a source-specific Join toward the source. As it follows the shortest path toward the source, the routers along the way create (S, G) state for this tree, and the result is a tree that has its root at the source, rather than the RP. Assuming both R4 and R5 made the switch to the source-specific tree, we would end up with the tree shown in Figure 4.14(d). Note that this tree no longer involves the RP at all. We have removed the shared tree from this picture to simplify the diagram, but in reality all routers with receivers for a group must stay on the shared tree in case new senders show up.

We can now see why PIM is protocol independent. All of its mechanisms for building and maintaining trees take advantage of unicast routing without depending on any particular unicast routing protocol. The formation of trees is entirely determined by the paths that Join messages follow, which is determined by the choice of shortest paths made by unicast routing. Thus, to be precise, PIM is “unicast routing protocol independent,” as compared to DVMRP. Note that PIM is very much bound up with the Internet Protocol—it is not protocol independent in terms of network-layer protocols.

The design of PIM-SM again illustrates the challenges in building scalable networks and how scalability is sometimes pitted against some sort of optimality. The shared tree is certainly more scalable than a source-specific tree, in the sense that it reduces the total state in routers to be on the order of the number of groups rather than the number of senders times the number of groups. However, the source-specific tree is likely to be necessary to achieve efficient routing and effective use of link bandwidth.

#### *Interdomain Multicast (MSDP)*

PIM-SM has some significant shortcomings when it comes to interdomain multicast. In particular, the existence of a single RP for a group goes against the principle that domains are autonomous. For a given multicast group, all the participating domains would be dependent on the domain where the RP is located. Furthermore, if there is a particular multicast group for which a sender and some receivers shared a single domain, the multicast traffic would still have to be routed initially from the sender to those receivers via whatever domain has the RP for that multicast group. Consequently, the PIM-SM protocol is typically not used across domains, only within a domain.

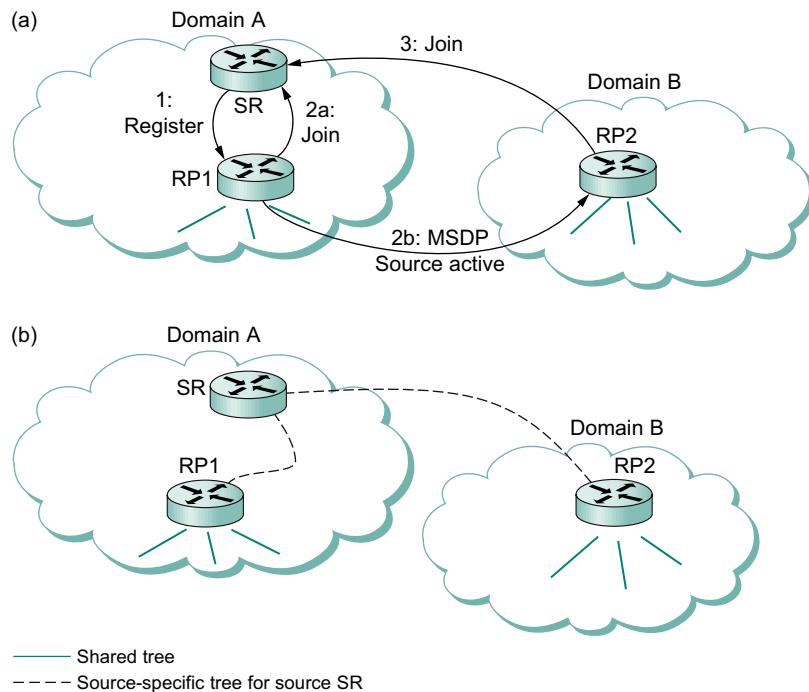
To extend multicast across domains using PIM-SM, the Multicast Source Discovery Protocol (MSDP) was devised. MSDP is used to connect different domains—each running PIM-SM internally, with its own RPs—by connecting the RPs of the different domains. Each RP has one or more MSDP peer RPs in other domains. Each pair of MSDP peers is connected by a TCP connection (Section 5.2) over which the MSDP protocol runs. Together, all the MSDP peers for a given multicast group form a loose mesh that is used as a broadcast network. MSDP messages are broadcast through the mesh of peer RPs using the Reverse Path Broadcast algorithm that we discussed in the context of DVMRP.

What information does MSDP broadcast through the mesh of RPs? Not group membership information; when a host joins a group, the furthest that information will flow is its own domain's RP. Instead, it is source—multicast sender—information. Each RP knows the sources in its own domain because it receives a Register message whenever a new source arises. Each RP periodically uses MSDP to broadcast Source Active messages to its peers, giving the IP address of the source, the multicast group address, and the IP address of the originating RP.

If an MSDP peer RP that receives one of these broadcasts has active receivers for that multicast group, it sends a source-specific Join, on that RP's own behalf, to the source host, as shown in Figure 4.16(a). The Join message builds a branch of the source-specific tree to this RP, as shown in Figure 4.16(b). The result is that every RP that is part of the MSDP network and has active receivers for a particular multicast group is added to the source-specific tree of the new source. When an RP receives a multicast from the source, the RP uses its shared tree to forward the multicast to the receivers in its domain.

#### *Source-Specific Multicast (PIM-SSM)*

The original service model of PIM was, like earlier multicast protocols, a many-to-many model. Receivers joined a group, and any host could send to the group. However, it was recognized in the late 1990s that it might be useful to add a one-to-many model. Lots of multicast applications, after all, have only one legitimate sender, such as the speaker at a conference being sent over the Internet. We already saw that PIM-SM can create source-specific shortest path trees as an optimization after using the shared tree initially. In the original PIM design, this optimization was invisible to hosts—only routers joined source-specific trees. However, once the need for a one-to-many service model was recognized, it was



**FIGURE 4.16** MSDP operation: (a) The source SR sends a **Register** to its domain’s RP, RP1; then RP1 sends a source-specific **Join** to SR and an **MSDP Source Active** to its MSDP peer in Domain B, RP2; then RP2 sends a source-specific **Join** to SR. (b) As a result, RP1 and RP2 are in the source-specific tree for source SR.

decided to make the source-specific routing capability of PIM-SM explicitly available to hosts. It turns out that this mainly required changes to IGMP and its IPv6 analog, MLD, rather than PIM itself. The newly exposed capability is now known as PIM-SSM (PIM Source-Specific Multicast).

PIM-SSM introduces a new concept, the *channel*, which is the combination of a source address S and a group address G. The group address G looks just like a normal IP multicast address, and both IPv4 and IPv6 have allocated subranges of the multicast address space for SSM. To use PIM-SSM, a host specifies both the group and the source in an IGMP Membership Report message to its local router. That router then sends a PIM-SM source-specific Join message toward the source, thereby adding a branch to itself in the source-specific tree, just as was described above for “normal” PIM-SM, but bypassing the whole shared-tree stage. Since the tree that results is source specific, only the designated source can send packets on that tree.

The introduction of PIM-SSM has provided some significant benefits, particularly since there is relatively high demand for one-to-many multicasting:

- Multicasts travel more directly to receivers.
- The address of a channel is effectively a multicast group address plus a source address. Therefore, given that a certain range of multicast group addresses will be used for SSM exclusively, multiple domains can use the same multicast group address independently and without conflict, as long as they use it only with sources in their own domains.
- Because only the specified source can send to an SSM group, there is less risk of attacks based on malicious hosts overwhelming the routers or receivers with bogus multicast traffic.
- PIM-SSM can be used across domains exactly as it is used within a domain, without reliance on anything like MSDP.

SSM, therefore, is quite a useful addition to the multicast service model.

#### *Bidirectional Trees (BIDIR-PIM)*

We round off our discussion of multicast with another enhancement to PIM known as *Bidirectional PIM*. BIDIR-PIM is a recent variant of PIM-SM that is well suited to many-to-many multicasting within a domain, especially when senders and receivers to a group may be the same, as in a multiparty videoconference, for example. As in PIM-SM, would-be receivers join groups by sending IGMP Membership Report messages (which must not be source specific), and a shared tree rooted at an RP is used to forward multicast packets to receivers. Unlike PIM-SM, however, the shared tree also has branches to the *sources*. That wouldn't make any sense with PIM-SM's unidirectional tree, but BIDIR-PIM's trees are bidirectional—a router that receives a multicast packet from a downstream branch can forward it both up the tree and down other branches. The route followed to deliver a packet to any particular receiver goes only as far up the tree as necessary before going down the branch to that receiver. See the multicast route from R1 to R2 in Figure 4.17(b) for an example. R4 forwards a multicast packet downstream to R2 at the same time that it forwards a copy of the same packet upstream to R5.

A surprising aspect of BIDIR-PIM is that there need not actually be an RP. All that is needed is a routable address, which is known as an RP

## Where Are They Now?

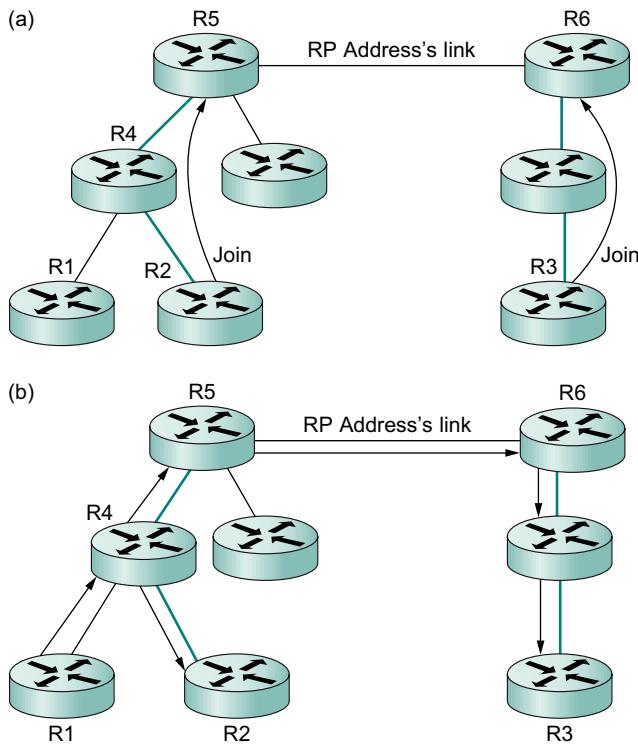
### THE FATE OF MULTICAST PROTOCOLS

A number of IP multicast protocols have fallen by the wayside since the 1991 publication of Steve Deering's doctoral thesis, "Multicast Routing in a Datagram Network." In most cases, their downfall had something to do with scaling. The most successful early multicast protocol was DVMRP, which we discussed at the start of the section. The *Multicast Open Shortest Path First* (MOSPF) protocol was based on the Open Shortest Path First (OSPF) unicast routing protocol. PIM dense mode (PIM-DM) has some similarity to DVMRP, in that it also uses a flood-and-prune approach; at the same time, it is like PIM-SM in being independent of the unicast routing protocol used. All of these protocols are more appropriate to a dense domain (i.e., one with a high proportion of routers interested in the multicast). These protocols all appeared relatively early in the history of multicast, before some of the scaling challenges were fully apparent. Although they would still make sense within a domain for multicast groups expected to be of dense interest, they are rarely used today, in part because the routers usually must support PIM-SM anyway.

*Core-Based Trees* (CBT) was another approach to multicast that was proposed at about the same time as PIM. The IETF was initially unable to choose between the two approaches, and both PIM and CBT were advanced as "experimental" protocols. However, PIM was more widely adopted by industry, and the main technical contributions of CBT—shared trees and bidirectional trees—were ultimately incorporated into PIM-SM and BiDIR-PIM, respectively.

*Border Gateway Multicast Protocol* (BGMP) also uses the concept of a bidirectional shared tree. In BGMP's case, however, the nodes of the tree are domains, with one of the domains as the root. In other words, BGMP is like MSDP in tying together domains to support interdomain multicasts. Unlike MSDP, the domains are free to choose their own intradomain protocols. BGMP was proposed at the IETF, and just a few years ago BGMP was expected to replace MSDP as the dominant interdomain routing protocol. BGMP is quite complex, however, and it requires the existence of a protocol that assigns ranges of multicast addresses to domains, in order for BGMP to know which domain is the root for a given address. Consequently, there have been, it appears, no implementations of BGMP, let alone deployments, at the time of writing.

address even though it need not be the address of an RP or anything at all. How can this be? A Join from a receiver is forwarded toward the RP address until it reaches a router with an interface on the link where the RP address would reside, where the Join terminates. Figure 4.17(a) shows a



**FIGURE 4.17** BIDIR-PIM operation: (a) R2 and R3 send **Joins** toward the RP address that terminate when they reach a router on the RP address's link. (b) A multicast packet from R1 is forwarded upstream to the RP address's link and downstream wherever it intersects a group member branch.

Join from R2 terminating at R5, and a Join from R3 terminating at R6. The upstream forwarding of a multicast packet similarly flows toward the RP address until it reaches a router with an interface on the link where the RP address would reside, but then the router forwards the multicast packet onto that link as the final step of upstream forwarding, ensuring that all other routers on that link receive the packet. Figure 4.17(b) illustrates the flow of multicast traffic originating at R1.

BIDIR-PIM cannot thus far be used across domains. On the other hand, it has several advantages over PIM-SM for many-to-many multicast within a domain:

- There is no source registration process because the routers already know how to route a multicast packet toward the RP address.

- The routes are more direct than those that use PIM-SM’s shared tree because they go only as far up the tree as necessary, not all the way to the RP.
- Bidirectional trees use much less state than the source-specific trees of PIM-SM because there is never any source-specific state. (On the other hand, the routes will be longer than those of source-specific trees.)
- The RP cannot be a bottleneck, and indeed no actual RP is needed.

One conclusion to draw from the fact that there are so many different approaches to multicast just within PIM is that multicast is a difficult problem space in which to find optimal solutions. You need to decide which criteria you want to optimize (bandwidth usage, router state, path length, etc.) and what sort of application you are trying to support (one-to-many, many-to-many, etc.) before you can make a choice of the “best” multicast mode for the task.

### 4.3 MULTIPROTOCOL LABEL SWITCHING (MPLS)

We continue our discussion of enhancements to IP by describing an addition to the Internet architecture that is very widely used but largely hidden from end users. The enhancement, called *Multiprotocol Label Switching* (MPLS), combines some of the properties of virtual circuits with the flexibility and robustness of datagrams. On the one hand, MPLS is very much associated with the Internet Protocol’s datagram-based architecture—it relies on IP addresses and IP routing protocols to do its job. On the other hand, MPLS-enabled routers also forward packets by examining relatively short, fixed-length labels, and these labels have local scope, just like in a virtual circuit network. It is perhaps this marriage of two seemingly opposed technologies that has caused MPLS to have a somewhat mixed reception in the Internet engineering community.

Before looking at how MPLS works, it is reasonable to ask “what is it good for?” Many claims have been made for MPLS, but there are three main things that it is used for today:

- To enable IP capabilities on devices that do not have the capability to forward IP datagrams in the normal manner

- To forward IP packets along explicit routes—precalculated routes that don't necessarily match those that normal IP routing protocols would select
- To support certain types of virtual private network services

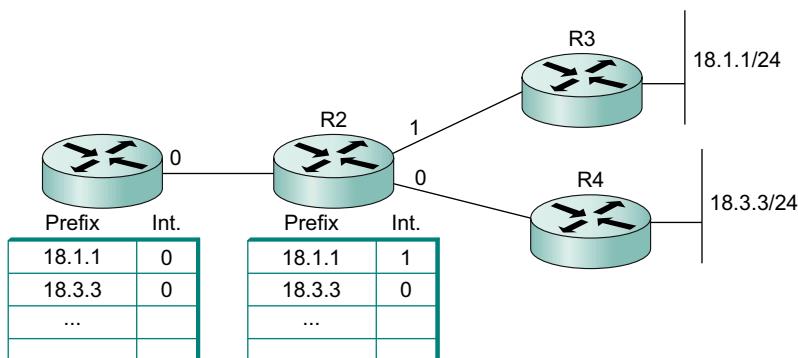
It is worth noting that one of the original goals—improving performance—is not on the list. This has a lot to do with the advances that have been made in forwarding algorithms for IP routers in recent years and with the complex set of factors beyond header processing that determine performance.

The best way to understand how MPLS works is to look at some examples of its use. In the next three sections, we will look at examples to illustrate the three applications of MPLS mentioned above.

### 4.3.1 Destination-Based Forwarding

One of the earliest publications to introduce the idea of attaching labels to IP packets was a paper by Chandranmenon and Vargese that described an idea called *threaded indices*. A very similar idea is now implemented in MPLS-enabled routers. The following example shows how this idea works.

Consider the network in Figure 4.18. Each of the two routers on the far right (R3 and R4) has one connected network, with prefixes 18.1.1/24 and 18.3.3/24. The remaining routers (R1 and R2) have routing tables that indicate which outgoing interface each router would use when forwarding packets to one of those two networks.



■ FIGURE 4.18 Routing tables in example network.

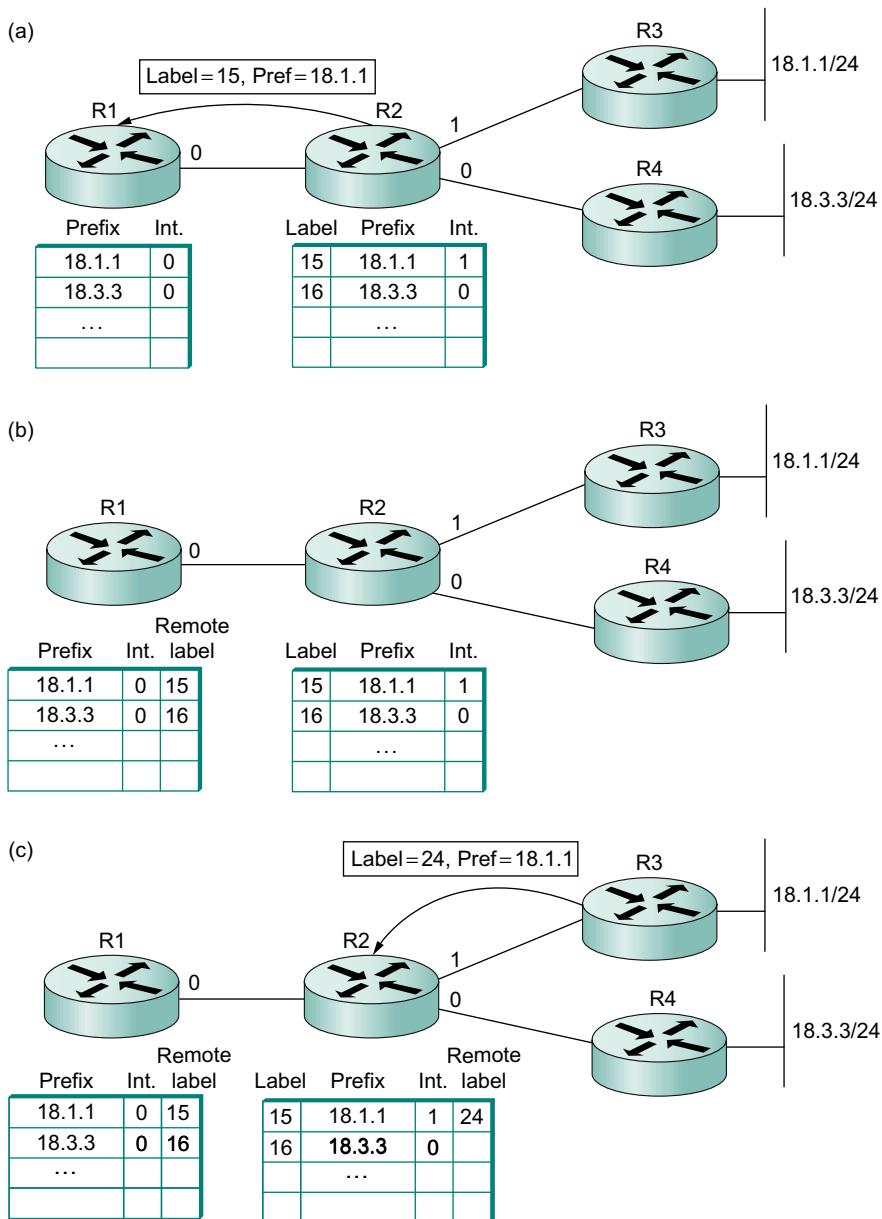
When MPLS is enabled on a router, the router allocates a label for each prefix in its routing table and advertises both the label and the prefix that it represents to its neighboring routers. This advertisement is carried in the Label Distribution Protocol. This is illustrated in Figure 4.19. Router R2 has allocated the label value 15 for the prefix 18.1.1 and the label value 16 for the prefix 18.3.3. These labels can be chosen at the convenience of the allocating router and can be thought of as indices into the routing table. After allocating the labels, R2 advertises the label bindings to its neighbors; in this case, we see R2 advertising a binding between the label 15 and the prefix 18.1.1 to R1. The meaning of such an advertisement is that R2 has said, in effect, “Please attach the label 15 to all packets sent to me that are destined to prefix 18.1.1.” R1 stores the label in a table alongside the prefix that it represents as the remote or outgoing label for any packets that it sends to that prefix.

In Figure 4.19(c), we see another label advertisement from router R3 to R2 for the prefix 18.1.1, and R2 places the remote label that it learned from R3 in the appropriate place in its table.

At this point, we can look at what happens when a packet is forwarded in this network. Suppose a packet destined to the IP address 18.1.1.5 arrives from the left to router R1. R1 in this case is referred to as a *Label Edge Router* (LER); an LER performs a complete IP lookup on arriving IP packets and then applies labels to them as a result of the lookup. In this case, R1 would see that 18.1.1.5 matches the prefix 18.1.1 in its forwarding table and that this entry contains both an outgoing interface and a remote label value. R1 therefore attaches the remote label 15 to the packet before sending it.

When the packet arrives at R2, R2 looks only at the label in the packet, not the IP address. The forwarding table at R2 indicates that packets arriving with a label value of 15 should be sent out interface 1 and that they should carry the label value 24, as advertised by router R3. R2 therefore rewrites, or swaps, the label and forwards it on to R3.

What has been accomplished by all this application and swapping of labels? Observe that when R2 forwarded the packet in this example it never actually needed to examine the IP address. Instead, R2 looked only at the incoming label. Thus, we have replaced the normal IP destination address lookup with a label lookup. To understand why this is significant, it helps to recall that, although IP addresses are always the same length, IP prefixes are of variable length, and the IP destination address



■ FIGURE 4.19 (a) R2 allocates labels and advertises bindings to R1. (b) R1 stores the received labels in a table. (c) R3 advertises another binding, and R2 stores the received label in a table.

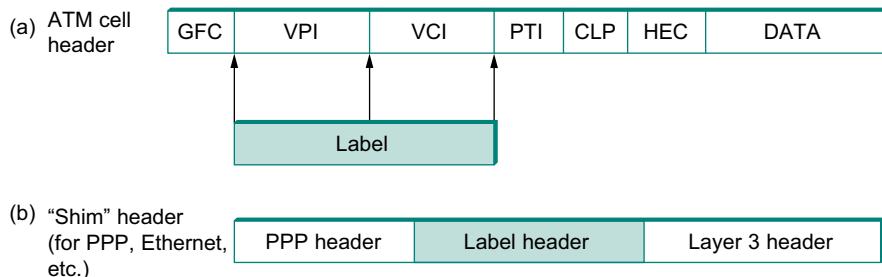
lookup algorithm needs to find the *longest match*—the longest prefix that matches the high order bits in the IP address of the packet being forwarded. By contrast, the label forwarding mechanism just described is an *exact match* algorithm. It is possible to implement a very simple exact match algorithm, for example, by using the label as an index into an array, where each element in the array is one line in the forwarding table.

Note that, while the forwarding algorithm has been changed from longest match to exact match, the routing algorithm can be any standard IP routing algorithm (e.g., OSPF). The path that a packet will follow in this environment is the exact same path that it would have followed if MPLS were not involved—the path chosen by the IP routing algorithms. All that has changed is the forwarding algorithm.

An important fundamental concept of MPLS is illustrated by this example. Every MPLS label is associated with a *forwarding equivalence class* (FEC)—a set of packets that are to receive the same forwarding treatment in a particular router. In this example, each prefix in the routing table is an FEC; that is, all packets that match the prefix 18.1.1—no matter what the low order bits of the IP address are—get forwarded along the same path. Thus, each router can allocate one label that maps to 18.1.1, and any packet that contains an IP address whose high order bits match that prefix can be forwarded using that label.

As we will see in the subsequent examples, FECs are a very powerful and flexible concept. FECs can be formed using almost any criteria; for example, all the packets corresponding to a particular customer could be considered to be in the same FEC.

Returning to the example at hand, we observe that changing the forwarding algorithm from normal IP forwarding to label swapping has an important consequence: Devices that previously didn't know how to forward IP packets can be used to forward IP traffic in an MPLS network. The most notable early application of this result was to ATM switches, which can support MPLS without any changes to their forwarding hardware. ATM switches support the label-swapping forwarding algorithm just described, and by providing these switches with IP routing protocols and a method to distribute label bindings they could be turned into *Label Switching Routers* (LSRs)—devices that run IP control protocols but use the label switching forwarding algorithm. More recently, the same idea has been applied to optical switches of the sort described in Section 3.1.2.

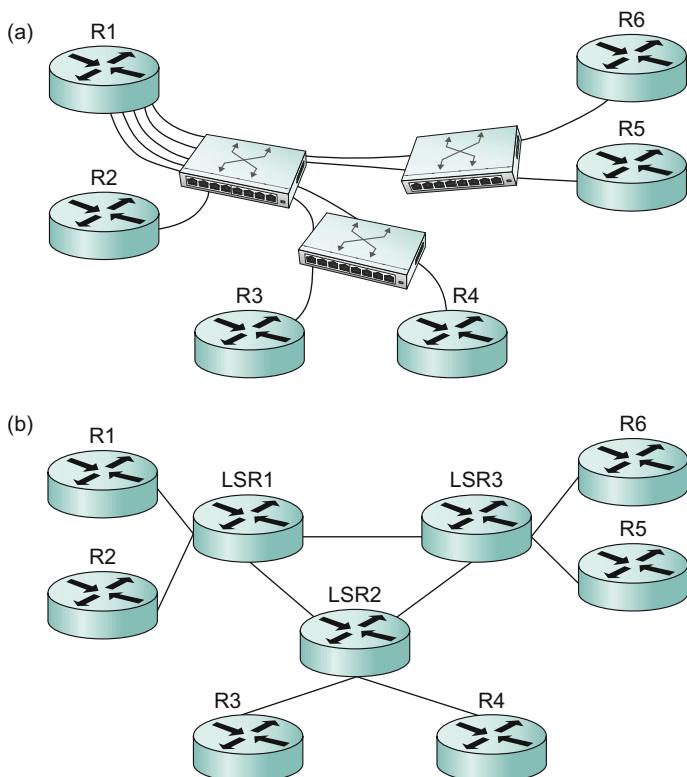


**FIGURE 4.20** (a) Label on an ATM-encapsulated packet; (b) label on a frame-encapsulated packet.

Before we consider the purported benefits of turning an ATM switch into an LSR, we should tie up some loose ends. We have said that labels are “attached” to packets, but where exactly are they attached? The answer depends on the type of link on which packets are carried. Two common methods for carrying labels on packets are shown in Figure 4.20. When IP packets are carried as complete frames, as they are on most link types including Ethernet and PPP, the label is inserted as a “shim” between the layer 2 header and the IP (or other layer 3) header, as shown in the lower part of the figure. However, if an ATM switch is to function as an MPLS LSR, then the label needs to be in a place where the switch can use it, and that means it needs to be in the ATM cell header, exactly where one would normally find the virtual circuit identifier (VCI) and virtual path identifier (VPI) fields.

### What Layer is MPLS?

There have been many debates about where MPLS belongs in the layered protocol architectures presented in Section 1.3. Since the MPLS header is normally found between the layer 3 and layer 2 headers in a packet, it is sometimes referred to as a layer 2.5 protocol. Some people argue that, since IP packets are encapsulated inside MPLS headers, MPLS must be “below” IP, making it a layer 2 protocol. Others argue that, since the control protocols for MPLS are, in large part, the same protocols as IP—MPLS uses IP routing protocols and IP addressing—then MPLS must be at the same layer as IP (i.e., layer 3). As we noted in Section 1.3, layered architectures are useful tools but they may not always exactly describe the real world, and MPLS is a good example of where strictly layerist views may be difficult to reconcile with reality.



■ FIGURE 4.21 (a) Routers connect to each other using an overlay of virtual circuits. (b) Routers peer directly with LSRs.

Having now devised a scheme by which an ATM switch can function as an LSR, what have we gained? One thing to note is that we could now build a network that uses a mixture of conventional IP routers, label edge routers, and ATM switches functioning as LSRs, and they would all use the same routing protocols. To understand the benefits of using the same protocols, consider the alternative. In Figure 4.21(a), we see a set of routers interconnected by virtual circuits over an ATM network, a configuration called an *overlay* network. At one point in time, networks of this type were often built because commercially available ATM switches supported higher total throughput than routers. Today, networks like this are less common because routers have caught up with and even surpassed ATM switches. However, these networks still exist

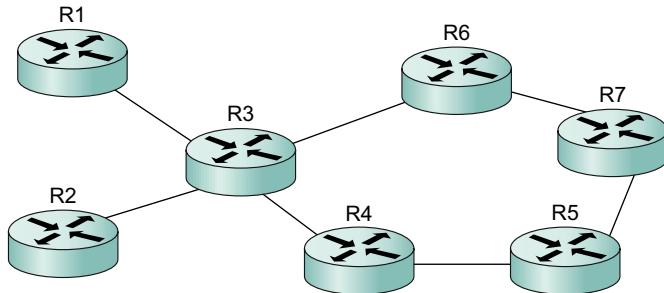
because of the significant installed base of ATM switches in network backbones, which in turn is partly a result of ATM's ability to support a range of capabilities such as circuit emulation and virtual circuit services.

In an overlay network, each router would potentially be connected to each of the other routers by a virtual circuit, but in this case for clarity we have just shown the circuits from R1 to all of its peer routers. R1 has five routing neighbors and needs to exchange routing protocol messages with all of them—we say that R1 has five routing adjacencies. By contrast, in [Figure 4.21\(b\)](#), the ATM switches have been replaced with LSRs. There are no longer virtual circuits interconnecting the routers. Thus, R1 has only one adjacency, with LSR1. In large networks, running MPLS on the switches leads to a significant reduction in the number of adjacencies that each router must maintain and can greatly reduce the amount of work that the routers have to do to keep each other informed of topology changes.

A second benefit of running the same routing protocols on edge routers and on the LSRs is that the edge routers now have a full view of the topology of the network. This means that if some link or node fails inside the network, the edge routers will have a better chance of picking a good new path than if the ATM switches rerouted the affected VCs without the knowledge of the edge routers.

Note that the step of “replacing” ATM switches with LSRs is actually achieved by changing the protocols running on the switches, but typically no change to the forwarding hardware is needed; that is, an ATM switch can often be converted to an MPLS LSR by upgrading only its software. Furthermore, an MPLS LSR might continue to support standard ATM capabilities at the same time as it runs the MPLS control protocols, in what is referred to as “ships in the night” mode.

More recently, the idea of running IP control protocols on devices that are unable to forward IP packets natively has been extended to optical switches and STDM devices such as SONET multiplexors. This is known as *Generalized MPLS* (GMPLS). Part of the motivation for GMPLS was to provide routers with topological knowledge of an optical network, just as in the ATM case. Even more important was the fact that there were no standard protocols for controlling optical devices, so MPLS seemed like a natural fit for that job.



■ FIGURE 4.22 A network requiring explicit routing.

### 4.3.2 Explicit Routing

In Section 3.1.3, we introduced the concept of source routing. IP has a source routing option, but it is not widely used for several reasons, including the fact that only a limited number of hops can be specified and because it is usually processed outside the “fast path” on most routers.

MPLS provides a convenient way to add capabilities similar to source-routing to IP networks, although the capability is more often referred to as *explicit routing* rather than *source routing*. One reason for the distinction is that it usually isn’t the real source of the packet that picks the route. More often it is one of the routers inside a service provider’s network. Figure 4.22 shows an example of how the explicit routing capability of MPLS might be applied. This sort of network is often called a *fish* network because of its shape (the routers R1 and R2 form the tail; R7 is at the head).

Suppose that the operator of the network in Figure 4.22 has determined that any traffic flowing from R1 to R7 should follow the path R1-R3-R6-R7 and that any traffic going from R2 to R7 should follow the path R2-R3-R4-R5-R7. One reason for such a choice would be to make good use of the capacity available along the two distinct paths from R3 to R7. We can think of the R1-to-R7 traffic as constituting one forwarding equivalence class, and the R2-to-R7 traffic constitutes a second FEC. Forwarding traffic in these two classes along different paths is difficult with normal IP routing, because R3 doesn’t normally look at where traffic came from in making its forwarding decisions.

Because MPLS uses label swapping to forward packets, it is easy enough to achieve the desired routing if the routers are MPLS enabled.

If R1 and R2 attach distinct labels to packets before sending them to R3—thus identifying them as being in different FECs—then R3 can forward packets from R1 and R2 along different paths. The question that then arises is how do all the routers in the network agree on what labels to use and how to forward packets with particular labels? Clearly, we can't use the same procedures as described in the preceding section to distribute labels, because those procedures establish labels that cause packets to follow the normal paths picked by IP routing, which is exactly what we are trying to avoid. Instead, a new mechanism is needed. It turns out that the protocol used for this task is the Resource Reservation Protocol (RSVP). We'll talk more about this protocol in [Section 6.5.2](#), but for now it suffices to say that it is possible to send an RSVP message along an explicitly specified path (e.g., R1-R3-R6-R7) and use it to set up label forwarding table entries all along that path. This is very similar to the process of establishing a virtual circuit described in [Section 3.3](#).

One of the applications of explicit routing is *traffic engineering*, which refers to the task of ensuring that sufficient resources are available in a network to meet the demands placed on it. Controlling exactly which paths the traffic flows on is an important part of traffic engineering. Explicit routing can also help to make networks more resilient in the face of failure, using a capability called *fast reroute*. For example, it is possible to precalculate a path from router A to router B that explicitly avoids a certain link L. In the event that link L fails, router A could send all traffic destined to B down the precalculated path. The combination of precalculation of the backup path and the explicit routing of packets along the path means that A doesn't need to wait for routing protocol packets to make their way across the network or for routing algorithms to be executed by various other nodes in the network. In certain circumstances, this can significantly reduce the time taken to reroute packets around a point of failure.

One final point to note about explicit routing is that explicit routes need not be calculated by a network operator as in the above example. Routers can use various algorithms to calculate explicit routes automatically. The most common of these is *constrained shortest path first* (CSPF), which is like the link-state algorithms described in [Section 3.3.3](#), but which also takes various *constraints* into account. For example, if it was required to find a path from R1 to R7 that could carry an offered load of 100 Mbps, we could say that the constraint is that each link must have at least 100 Mbps of available capacity. CSPF addresses this sort

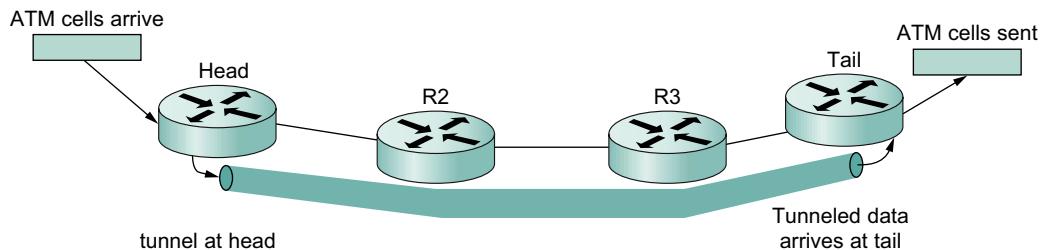
of problem. More details on CSPF, and the applications of explicit routing, are provided in the Further Reading section.

### 4.3.3 Virtual Private Networks and Tunnels

We first talked about virtual private networks (VPNs) in [Section 3.2.9](#), and we noted that one way to build them was using tunnels. It turns out that MPLS can be thought of as a way to build tunnels, and this makes it suitable for building VPNs of various types.

The simplest form of MPLS VPN to understand is a layer 2 VPN. In this type of VPN, MPLS is used to tunnel layer 2 data (such as Ethernet frames or ATM cells) across a network of MPLS-enabled routers. Recall from [Section 3.2.9](#) that one reason for tunnels is to provide some sort of network service (such as multicast) that is not supported by some routers in the network. The same logic applies here: IP routers are not ATM switches, so you cannot provide an ATM virtual circuit service across a network of conventional routers. However, if you had a pair of routers interconnected by a tunnel, they could send ATM cells across the tunnel and emulate an ATM circuit. The term for this technique within the IETF is *pseudowire emulation*. [Figure 4.23](#) illustrates the idea.

We have already seen how IP tunnels are built: The router at the entrance of the tunnel wraps the data to be tunneled in an IP header (the *tunnel header*), which represents the address of the router at the far end of the tunnel and sends the data like any other IP packet. The receiving router receives the packet with its own address in the header, strips the tunnel header, and finds the data that was tunneled, which it then processes. Exactly what it does with that data depends on what it is. For example, if it were another IP packet, it would then be forwarded on like a normal IP packet. However, it need not be an IP packet, as long as the



■ **FIGURE 4.23** An ATM circuit is emulated by a tunnel.

receiving router knows what to do with non-IP packets. We'll return to the issue of how to handle non-IP data in a moment.

An MPLS tunnel is not too different from an IP tunnel, except that the tunnel header consists of an MPLS header rather than an IP header. Looking back to our first example, in [Figure 4.19](#), we saw that router R1 attached a label (15) to every packet that it sent towards prefix 18.1.1. Such a packet would then follow the path R1-R2-R3, with each router in the path examining only the MPLS label. Thus, we observe that there was no requirement that R1 only send IP packets along this path—any data could be wrapped up in the MPLS header and it would follow the same path, because the intervening routers never look beyond the MPLS header. In this regard, an MPLS header is just like an IP tunnel header.<sup>3</sup> The only issue with sending non-IP traffic along a tunnel, MPLS or otherwise, is what to do with non-IP traffic when it reaches the end of the tunnel. The general solution is to carry some sort of demultiplexing identifier in the tunnel payload that tells the router at the end of the tunnel what to do. It turns out that an MPLS label is a perfect fit for such an identifier. An example will make this clear.

Let's assume we want to tunnel ATM cells from one router to another across a network of MPLS-enabled routers, as in [Figure 4.23](#). Further, we assume that the goal is to emulate an ATM virtual circuit; that is, cells arrive at the entrance, or head, of the tunnel on a certain input port with a certain VCI and should leave the tail end of the tunnel on a certain output port and potentially different VCI. This can be accomplished by configuring the head and tail routers as follows:

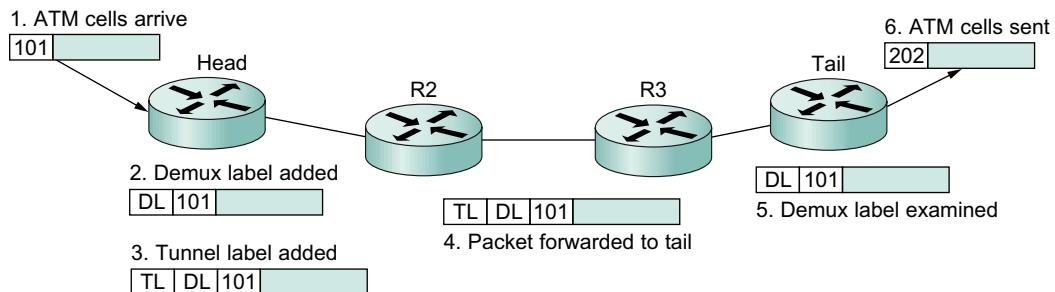
- The head router needs to be configured with the incoming port, the incoming VCI, the demultiplexing label for this emulated circuit, and the address of the tunnel end router.
- The tail router needs to be configured with the outgoing port, the outgoing VCI, and the demultiplexing label.

Once the routers are provided with this information, we can see how an ATM cell would be forwarded. [Figure 4.24](#) illustrates the steps.

1. An ATM cell arrives on the designated input port with the appropriate VCI value (101 in this example).

---

<sup>3</sup>Note, however, that an MPLS header is only 4 bytes long, compared to 20 for an IP header, which implies a bandwidth saving when MPLS is used.



■ FIGURE 4.24 Forward ATM cells along a tunnel.

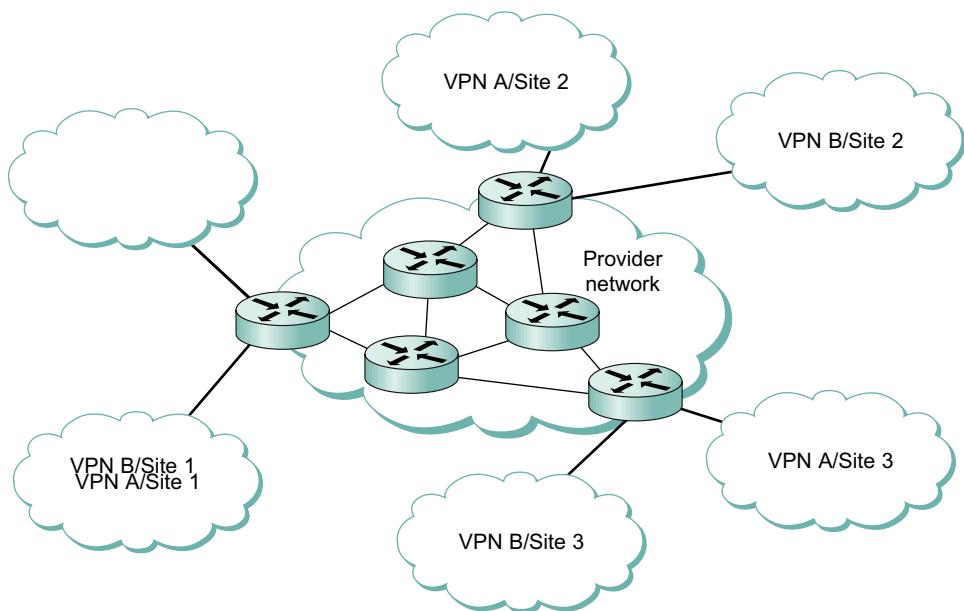
2. The head router attaches the demultiplexing label that identifies the emulated circuit.
3. The head router then attaches a second label, which is the tunnel label that will get the packet to the tail router. This label is learned by mechanisms just like those described in Section 4.3.1.
4. Routers between the head and tail forward the packet using only the tunnel label.
5. The tail router removes the tunnel label, finds the demultiplexing label, and recognizes the emulated circuit.
6. The tail router modifies the ATM VCI to the correct value (202 in this case) and sends it out the correct port.

One item in this example that might be surprising is that the packet has two labels attached to it. This is one of the interesting features of MPLS—labels may be stacked on a packet to any depth. This provides some useful scaling capabilities. In this example, it allows a single tunnel to carry a potentially large number of emulated circuits.

The same techniques described here can be applied to emulate many other layer 2 services, including Frame Relay and Ethernet. It is worth noting that virtually identical capabilities can be provided using IP tunnels; the main advantage of MPLS here is the shorter tunnel header.

Before MPLS was used to tunnel layer 2 services, it was also being used to support layer 3 VPNs. We won't go into the details of layer 3 VPNs, which are quite complex (see the Further Reading section for some good sources of more information), but we will note that they represent one of the most popular uses of MPLS today. Layer 3 VPNs also use stacks of MPLS labels to tunnel packets across an IP network. However, the packets

that are tunneled are themselves IP packets—hence, the name *layer 3* VPNs. In a layer 3 VPN, a single service provider operates a network of MPLS-enabled routers and provides a “virtually private” IP network service to any number of distinct customers. That is, each customer of the provider has some number of sites, and the service provider creates the illusion for each customer that there are no other customers on the network. The customer sees an IP network interconnecting his own sites and no other sites. This means that each customer is isolated from all other customers in terms of both routing and addressing. Customer A can't send packets directly to customer B, and *vice versa*.<sup>4</sup> Customer A can even use IP addresses that have also been used by customer B. The basic idea is illustrated in Figure 4.25. As in layer 2 VPNs, MPLS is used to



■ FIGURE 4.25 Example of a layer 3 VPN. Customers A and B each obtain a virtually private IP service from a single provider.

<sup>4</sup>Customer A in fact usually *can* send data to customer B in some restricted way. Most likely, both customer A and customer B have some connection to the global Internet, and thus it is probably possible for customer A to send email messages, for example, to the mail server inside customer B's network. The “privacy” offered by a VPN prevents customer A from having unrestricted access to all the machines and subnets inside customer B's network.

tunnel packets from one site to another; however, the configuration of the tunnels is performed automatically by some fairly elaborate use of BGP which is beyond the scope of this book.

In summary, MPLS is a rather versatile tool that has been applied to a wide range of different networking problems. It combines the label-swapping forwarding mechanism that is normally associated with virtual circuit networks with the routing and control protocols of IP datagram networks to produce a class of network that is somewhere between the two conventional extremes. This extends the capabilities of IP networks to enable, among other things, more precise control of routing and the support of a range of VPN services.

### Where Are They Now?

#### DEPLOYMENT OF MPLS

Originally conceived as a technology that would operate within the network of individual service providers, MPLS remains hidden from most consumer and academic users of the Internet today. However, it is now sufficiently popular among service providers that it has become almost mandatory for high-end router manufacturers to include MPLS capabilities in their products. The widespread success of MPLS is a relatively well-kept secret, at least to students and researchers focused on the public Internet.

Two main applications of MPLS account for most of its deployment. The layer 3 VPN application described in this section is the killer application for MPLS. Almost every service provider in the world now offers an MPLS-based layer 3 VPN service. This is often run on routers that are essentially separate from the Internet, since the main use of layer 3 VPNs is to provide private IP service to corporations, not to provide global Internet connectivity. Some providers do run their Internet service and VPN service over a common backbone, however.

The second popular usage of MPLS is explicit routing, either for traffic engineering or fast reroute, or both. Unlike the layer 3 VPN service, which is explicitly marketed to end customers, explicit routing is an internal capability that providers use to improve the reliability of their networks or reduce the cost. Providers do not usually publicize details of their internal network designs, making it more difficult to determine how many providers actually use this technology. It is clear that the explicit routing features of MPLS are used by fewer providers than the VPN features, but nevertheless there is evidence of significant usage, especially when bandwidth is expensive or when there is a strong desire to maintain low levels of congestion (e.g., to support real-time services).

## 4.4 ROUTING AMONG MOBILE DEVICES

It probably should not be a great surprise to learn that mobile devices present some challenges for the Internet architecture. The Internet was designed in an era when computers were large, immobile devices, and, while the Internet's designers probably had some notion that mobile devices might appear in the future, it's fair to assume it was not a top priority to accommodate them. Today, of course, mobile computers are everywhere, notably in the forms of laptops and IP-enabled mobile phones, and increasingly in other forms such as sensors. In this section, we will look at some of the challenges posed by the appearance of mobile devices and some of the current approaches to accommodating them.

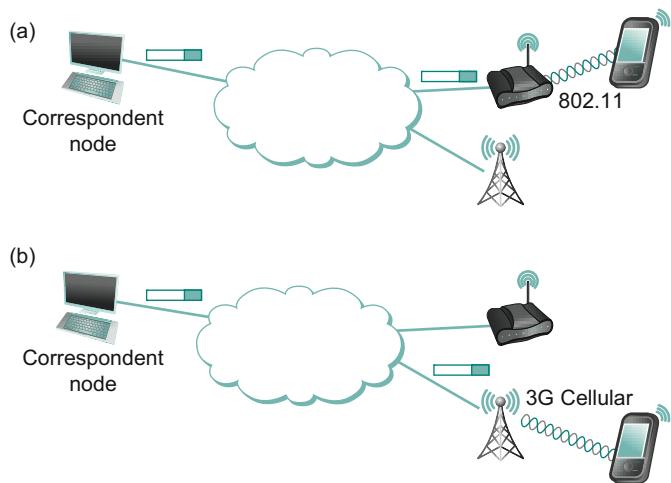


### 4.4.1 Challenges for Mobile Networking

Most readers of this book have probably used a networked mobile device at some point, and for many of us mobile devices have become the norm. So one might reasonably think that mobile networking is a solved problem. Certainly, it is easy enough today to turn up in a wireless hotspot, connect to the Internet using 802.11 or some other wireless networking protocol, and obtain pretty good Internet service. One key enabling technology that made the hotspot feasible is DHCP (see [Section 3.2.7](#) for details). You can settle in at a café, open your laptop, obtain an IP address for your laptop, and get your laptop talking to a default router and a Domain Name System (DNS) server (see [Section 9.3.1](#)), and for a broad class of applications you have everything you need.

If we look a little more closely, however, it's clear that for some application scenarios, just getting a new IP address every time you move—which is what DHCP does for you—isn't always enough. Suppose you are using your laptop or smartphone for a Voice over IP telephone call, and while talking on the phone you move from one hotspot to another, or even switch from 802.11 to 3G wireless for your Internet connection.

Clearly, when you move from one access network to another, you need to get a new IP address—one that corresponds to the new network. But, the computer or telephone at the other end of your conversation doesn't immediately know where you have moved or what your new IP address is. Consequently, in the absence of some other mechanism, packets would continue to be sent to the address where you *used* to be, not where you are now. This problem is illustrated in [Figure 4.26](#); as the mobile node moves from the 802.11 network in [Figure 4.26\(a\)](#) to the cellular



■ FIGURE 4.26 Forwarding packets from a correspondent node to a mobile node.

network in Figure 4.26(b), somehow packets from the *correspondent node* need to find their way to the new network and then on to the mobile node.

There are many different ways to tackle the problem just described, and we will look at some of them below. Assuming that there is some way to redirect packets so that they come to your new address rather than your old address, the next immediately apparent problems relate to security. For example, if there is a mechanism by which I can say, “My new IP address is X,” how do I prevent some attacker from making such a statement without my permission, thus enabling him to either receive my packets, or to redirect my packets to some unwitting third party? Thus, we see that security and mobility are quite closely related.

One issue that the above discussion highlights is the fact that IP addresses actually serve two tasks. They are used as an *identifier* of an endpoint, and they are also used to *locate* the endpoint. Think of the identifier as a long-lived name for the endpoint, and the locator as some possibly more temporary information about how to route packets to the endpoint. As long as devices do not move, or do not move often, using a single address for both jobs seem pretty reasonable. But once devices start

to move, you would rather like to have an identifier that does not change as you move—this is sometimes called an *Endpoint Identifier* or *Host Identifier*—and a separate *locator*. This idea of separating locators from identifiers has been around for a long time, and most of the approaches to handling mobility described below provide such a separation in some form.

The assumption that IP addresses don't change shows up in many different places. For example, as we'll see in the next chapter, transport protocols like TCP have historically made assumptions about the IP address staying constant for the life of a connection, so transport protocols operating in a mobile world require some re-evaluation of that assumption.

While we are all familiar with endpoints that move, it is worth noting that routers can also move. This is certainly less common today than endpoint mobility, but there are plenty of environments where a mobile router might make sense. One example might be an emergency response team trying to deploy a network after some natural disaster has knocked out all the fixed infrastructure. There are additional considerations when *all* the nodes in a network, not just the endpoints, are mobile, a topic we will discuss in [Section 4.4.2](#).

As with many technologies, support for mobility raises issues of incremental deployment. Given that, for its first couple of decades, the Internet consisted entirely of nodes that didn't move, it's fair to assume that there will be a lot of routers and hosts around for the foreseeable future that make that assumption. Hence, mobility solutions need to deal with incremental deployment. Conversely, IP version 6 had the ability to make mobility part of its design from the outset, which provides it with some advantages.

Before we start to look at some of the approaches to supporting mobile devices, a couple of points of clarification. It is common to find that people confuse wireless networks with mobility. After all, mobility and wireless often are found together for obvious reasons. But wireless communication is really about getting data from A to B without a wire, as discussed in some detail in [Chapter 2](#), while mobility is about dealing with what happens when a node moves around as it communicates. Certainly many nodes that use wireless communication channels are not mobile,

and sometimes mobile nodes will use wired communication (although this is less common).

Finally, in this chapter we are mostly interested in what we might call *network-layer mobility*. That is, we are interested in how to deal with nodes that move from one network to another. As we saw in Section 2.7, moving from one access point to another in the same 802.11 network can be handled by mechanisms specific to 802.11, and cellular telephone networks also have ways to handle mobility, of course, but in large heterogeneous systems like the Internet we need to support mobility more broadly across networks.

#### 4.4.2 Routing to Mobile Hosts (Mobile IP)

Mobile IP is the primary mechanism in today's Internet architecture to tackle the problem of routing packets to mobile hosts. It introduces a few new capabilities but does not require any change from non-mobile hosts or most routers—thus tackling the incremental deployment issue raised above.

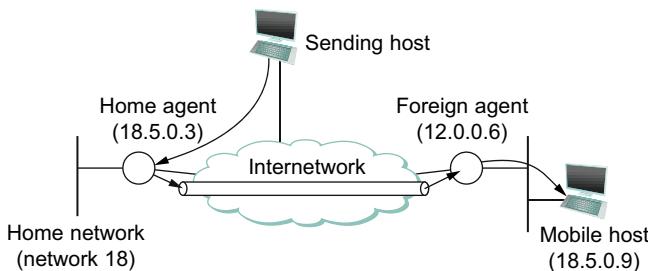
The mobile host is assumed to have a permanent IP address, called its *home address*, which has a network prefix equal to that of its *home network*. This is the address that will be used by other hosts when they initially send packets to the mobile host; because it does not change, it can be used by long-lived applications as the host roams. We can think of this as the long-lived identifier of the host.

When the host moves to a new foreign network away from its home network, it typically acquires a new address on that network using some means such as DHCP.<sup>5</sup> This address is going to change every time the host roams to a new network, so we can think of this as being more like the locator for the host, but it is important to note that the host does not lose its permanent home address when it acquires a new address on the foreign network. This home address is critical to its ability to sustain communications as it moves, as we'll see below.

While the majority of routers remain unchanged, mobility support does require some new functionality in at least one router, known as the

---

<sup>5</sup>Because DHCP was developed around the same time as Mobile IP, the original Mobile IP standards did not require DHCP, but DHCP is ubiquitous today.



■ FIGURE 4.27 Mobile host and mobility agents.

*home agent* of the mobile node. This router is located on the home network of the mobile host. In some cases, a second router with enhanced functionality, the *foreign agent*, is also required. This router is located on a network to which the mobile node attaches itself when it is away from its home network. We will consider first the operation of Mobile IP when a foreign agent is used. An example network with both home and foreign agents is shown in Figure 4.27.

Both home and foreign agents periodically announce their presence on the networks to which they are attached using agent advertisement messages. A mobile host may also solicit an advertisement when it attaches to a new network. The advertisement by the home agent enables a mobile host to learn the address of its home agent before it leaves its home network. When the mobile host attaches to a foreign network, it hears an advertisement from a foreign agent and registers with the agent, providing the address of its home agent. The foreign agent then contacts the home agent, providing a *care-of address*. This is usually the IP address of the foreign agent.

At this point, we can see that any host that tries to send a packet to the mobile host will send it with a destination address equal to the home address of that node. Normal IP forwarding will cause that packet to arrive on the home network of the mobile node on which the home agent is sitting. Thus, we can divide the problem of delivering the packet to the mobile node into three parts:

1. How does the home agent intercept a packet that is destined for the mobile node?

2. How does the home agent then deliver the packet to the foreign agent?
3. How does the foreign agent deliver the packet to the mobile node?

The first problem might look easy if you just look at Figure 4.27, in which the home agent is clearly the only path between the sending host and the home network and thus must receive packets that are destined to the mobile node. But what if the sending (correspondent) node were on network 18, or what if there were another router connected to network 18 that tried to deliver the packet without its passing through the home agent? To address this problem, the home agent actually impersonates the mobile node, using a technique called *proxy ARP*. This works just like Address Resolution Protocol (ARP) as described in Section 3.2.6, except that the home agent inserts the IP address of the mobile node, rather than its own, in the ARP messages. It uses its own hardware address, so that all the nodes on the same network learn to associate the hardware address of the home agent with the IP address of the mobile node. One subtle aspect of this process is the fact that ARP information may be cached in other nodes on the network. To make sure that these caches are invalidated in a timely way, the home agent issues an ARP message as soon as the mobile node registers with a foreign agent. Because the ARP message is not a response to a normal ARP request, it is termed a *gratuitous ARP*.

The second problem is the delivery of the intercepted packet to the foreign agent. Here we use the tunneling technique described in Section 3.2.9. The home agent simply wraps the packet inside an IP header that is destined for the foreign agent and transmits it into the internetwork. All the intervening routers just see an IP packet destined for the IP address of the foreign agent. Another way of looking at this is that an IP tunnel is established between the home agent and the foreign agent, and the home agent just drops packets destined for the mobile node into that tunnel.

When a packet finally arrives at the foreign agent, it strips the extra IP header and finds inside an IP packet destined for the home address of the mobile node. Clearly the foreign agent cannot treat this like any old IP packet because this would cause it to send it back to the home network.

Instead, it has to recognize the address as that of a registered mobile node. It then delivers the packet to the *hardware* address of the mobile node (e.g., its Ethernet address), which was learned as part of the registration process.

One observation that can be made about these procedures is that it is possible for the foreign agent and the mobile node to be in the same box; that is, a mobile node can perform the foreign agent function itself. To make this work, however, the mobile node must be able to dynamically acquire an IP address that is located in the address space of the foreign network (e.g., using DHCP). This address will then be used as the care-of address. In our example, this address would have a network number of 12. This approach has the desirable feature of allowing mobile nodes to attach to networks that don't have foreign agents; thus, mobility can be achieved with only the addition of a home agent and some new software on the mobile node (assuming DHCP is used on the foreign network).

What about traffic in the other direction (i.e., from mobile node to fixed node)? This turns out to be much easier. The mobile node just puts the IP address of the fixed node in the destination field of its IP packets while putting its permanent address in the source field, and the packets are forwarded to the fixed node using normal means. Of course, if both nodes in a conversation are mobile, then the procedures described above are used in each direction.

#### *Route Optimization in Mobile IP*

There is one significant drawback to the above approach: The route from the correspondent node to the mobile node can be significantly suboptimal. One of the most extreme examples is when a mobile node and the correspondent node are on the same network, but the home network for the mobile node is on the far side of the Internet. The sending correspondent node addresses all packets to the home network; they traverse the Internet to reach the home agent, which then tunnels them back across the Internet to reach the foreign agent. Clearly, it would be nice if the correspondent node could find out that the mobile node is actually on the same network and deliver the packet directly. In the more general case, the goal is to deliver packets as directly as possible from

correspondent node to mobile node without passing through a home agent. This is sometimes referred to as the *triangle routing problem* since the path from correspondent to mobile node via home agent takes two sides of a triangle, rather than the third side that is the direct path.

The basic idea behind the solution to triangle routing is to let the correspondent node know the care-of address of the mobile node. The correspondent node can then create its own tunnel to the foreign agent. This is treated as an optimization of the process just described. If the sender has been equipped with the necessary software to learn the care-of address and create its own tunnel, then the route can be optimized; if not, packets just follow the suboptimal route.

When a home agent sees a packet destined for one of the mobile nodes that it supports, it can deduce that the sender is not using the optimal route. Therefore, it sends a “binding update” message back to the source, in addition to forwarding the data packet to the foreign agent. The source, if capable, uses this binding update to create an entry in a *binding cache*, which consists of a list of mappings from mobile node addresses to care-of addresses. The next time this source has a data packet to send to that mobile node, it will find the binding in the cache and can tunnel the packet directly to the foreign agent.

There is an obvious problem with this scheme, which is that the binding cache may become out-of-date if the mobile host moves to a new network. If an out-of-date cache entry is used, the foreign agent will receive tunneled packets for a mobile node that is no longer registered on its network. In this case, it sends a *binding warning* message back to the sender to tell it to stop using this cache entry. This scheme works only in the case where the foreign agent is not the mobile node itself, however. For this reason, cache entries need to be deleted after some period of time; the exact amount is specified in the binding update message.

As noted above, mobile routing provides some interesting security challenges, which are clearer now that we have seen how Mobile IP works. For example, an attacker wishing to intercept the packets destined to some other node in an internetwork could contact the home agent for that node and announce itself as the new foreign agent for the node. Thus, it is clear that some authentication mechanisms are required. We discuss such mechanisms in Chapter 8.

### Mobility in IPv6

There are a handful of significant differences between mobility support in IPv4 and IPv6. Most importantly, it was possible to build mobility support into the standards for IPv6 pretty much from the beginning, thus alleviating a number of incremental deployment problems. (It may be more correct to say that IPv6 is one big incremental deployment problem, which, if solved, will deliver mobility support as part of the package.)

Since all IPv6-capable hosts can acquire an address whenever they are attached to a foreign network (using several mechanisms defined as part of the core v6 specifications), Mobile IPv6 does away with the foreign agent and includes the necessary capabilities to act as a foreign agent in every host.

One other interesting aspect of IPv6 that comes into play with Mobile IP is its inclusion of a flexible set of extension headers, as described in Section 4.1.3. This is used in the optimized routing scenario described above. Rather than *tunneling* a packet to the mobile node at its care-of address, an IPv6 node can send an IP packet to the care-of address with the home address contained in a *routing header*. This header is ignored by all the intermediate nodes, but it enables the mobile node to treat the packet as if it were sent to the home address, thus enabling it to continue presenting higher layer protocols with the illusion that its IP address is fixed. Using an extension header rather than a tunnel is more efficient from the perspective of both bandwidth consumption and processing.

Finally, we note that many open issues remain in mobile networking. Managing the power consumption of mobile devices is increasingly important, so that smaller devices with limited battery power can be built. There is also the problem of *ad hoc* mobile networks—enabling a group of mobile nodes to form a network in the absence of any fixed nodes—which has some special challenges (see the sidebar). A particularly challenging class of mobile networks, *sensor networks*, was mentioned previously. Sensors typically are small, inexpensive, and often battery powered, meaning that issues of very low power consumption and limited processing capability must also be considered. Furthermore, since wireless communications and mobility typically go hand in hand, the continual advances in wireless technologies keep on producing new challenges and opportunities for mobile networking.

### Mobile Ad Hoc Networks

For most of this section, we've been assuming that only the end nodes (hosts) are mobile. This is certainly a good description of the way most of us deal with networks today. Our laptops and phones move around, and connect to fixed infrastructure, such as cell towers and 802.11 access points, which connect over fixed links to the Internet's backbone. However, many modern routers are also quite small enough to be mobile, and there are environments where mobile routers would be useful, such as building networks among moving vehicles. Because routing protocols are dynamic, you might imagine that the occasional mobile router would not be a problem, and that is roughly correct. However, what if all or most of the nodes in a network where mobile? Taken to the logical extreme, you could have a network with no fixed infrastructure at all—just a collection of mobile nodes, some or all of which function as routers. Would standard routing protocols work in such an environment?

The environment where everything is mobile and there is no fixed infrastructure is often called a Mobile *Ad Hoc* Network (MANET, which is the name of an IETF working group tackling the problem space). To understand why special solutions might be needed for the mobile *ad hoc* environment, consider the fact that, unlike a fixed network, the neighbors of any given *ad hoc* router are likely to change very frequently as the nodes move. Since any change in neighbor relationships typically requires a routing protocol message to be sent and a new routing table to be calculated, it's easy to see that there may be concerns about using a protocol not optimized for this environment. Compounding this issue is the fact that communication is likely to be wireless, which consumes power, and many mobile nodes are likely to run off power-constrained batteries. Link bandwidths are also likely to be constrained. Hence, reducing the overhead caused by sending routing protocol messages, and reflooding them to all of one's neighbors, is a key concern for *ad hoc* routing.

At the time of writing, several approaches to optimizing routing for mobile *ad hoc* environments have been developed. These are broadly characterized as reactive and proactive approaches. Optimized Link State Routing (OLSR) is the dominant proactive approach, and its name gives a good sense of what it is; it resembles a conventional link-state protocol (like OSPF, [Section 3.3.3](#)), with a number of optimizations to reduce the amount of flooding of routing messages. Reactive protocols include *Ad Hoc On-Demand Distance Vector* (AODV) and *Dynamic MANET On Demand* (DYMO), both of which are based on distance vector protocols as described in [Section 3.3.2](#). These approaches seek to reduce the amount of routing protocol overhead by only building routes as needed, such as when a given node has traffic for a particular destination. There is a rich solution space in which tradeoffs can be made, and this space continues to be explored.

## 4.5 SUMMARY

The main theme of this chapter was dealing with the continued growth of the Internet. The Internet keeps on attaching more users, and each user sends more traffic as applications such as video streaming become more bandwidth intensive. Thus, while the Internet has proved to be a hugely scalable system, new scaling issues continue to demand solutions. In addition to scaling, the Internet also needs to evolve to support new capabilities and services.

The major scaling issues today are the efficient use of address space and the growth of routing tables as the Internet grows. The hierarchical IP address format, with its network and host parts, gives us one level of hierarchy to manage scale. Routing areas provide another level of hierarchy. Autonomous systems allow us to partition the routing problem into two parts, interdomain and intradomain routing, each of which is much smaller than the total routing problem would be. BGP, the interdomain routing protocol of the Internet, has been remarkably successful in dealing with the growth of the Internet.

In spite of the many steps taken to scale IPv4, it is clear that a new, longer address format will soon be needed. This requires a new IP datagram format and a new version of the protocol. Originally known as Next Generation IP (IPng), this new protocol is now known as IPv6, and it provides a 128-bit address with (mostly) CIDR-like addressing and routing. While many new capabilities have been claimed for IPv6, its main advantage remains its ability to support an extremely large number of addressable devices.

Finally, the Internet also needs to evolve in function as well as size. In that regard, we looked at three enhancements to the original IP datagram model. The first, multicast, enables efficient delivery of the same data to groups of receivers. As with unicast, many of the challenges in multicast relate to scaling, and a number of different protocols and multicast modes have been developed to optimize scaling and routing in different environments. The second enhancement, MPLS, brings some of the aspects of virtual circuit networks to IP and has been widely used to extend the capabilities of IP. Applications of MPLS range from traffic engineering to the support of virtual private networks over the Internet. And, finally, mobility support, which was far from the minds of the original designers of IP, is increasingly important as more networked devices, both hosts and routers, become mobile.

Roughly 20 years have elapsed since the shortage of IPv4 address space became serious enough to warrant proposals for a new version of IP. The original IPv6 specification is now more than 15 years old. IPv6-capable host operating systems are now widely available and the major router vendors offer varying degrees of support for IPv6 in their products. Yet, the deployment of IPv6 in the Internet can only be described as embryonic. It is worth wondering when deployment is likely to begin in earnest and what will cause it.

One reason why IPv6 has *not* been needed sooner is because of the extensive use of Network Address Translation (NAT), described earlier in this chapter. As providers viewed IPv4 addresses as a scarce resource, they handed out fewer of them to their customers,

### WHAT'S NEXT: DEPLOYMENT OF IPv6

or charged for the number of addresses used; customers responded by hiding many of their devices behind a NAT box and a single IPv4 address. For example, it is generally the case that home networks with more than one IP-capable device have some sort of NAT in the network to conserve addresses. So one factor that might drive IPv6 deployment would be applications that don't work well with NAT. While client-server applications work reasonably well when the client's address is hidden behind a NAT box, peer-to-peer applications fare less well. Examples of applications that would work better without NAT and would therefore benefit from more liberal address allocation policies are multiplayer gaming and IP telephony. Yet, even these applications have found ways to deal with NAT, and NAT traversal technologies are now widely available.

Obtaining blocks of IPv4 addresses has been getting more difficult for years, and this is particularly noticeable in countries outside of the United States. As the difficulty increases, the incentive for providers to start offering IPv6 addresses to their customers also rises. At the same time, for existing providers, offering IPv6 is a substantial additional cost, because they don't get to stop supporting IPv4 when they start to offer IPv6. This means, for example, that the size of a provider's routing tables can only increase initially, because they need to carry all the existing IPv4 prefixes plus new IPv6 prefixes.

At the moment, IPv6 deployment has been led by research networks. A few service providers are starting to offer it, especially outside the United States (often with some incentive from national governments). Commercial routers and host operating systems support IPv6 to varying degrees. It seems certain that IPv6 deployment will continue to accelerate, but it also seems likely that the overwhelming majority of hosts and networks will be IPv4-only for several more years at least.

## ■ FURTHER READING

Our first selection, an RFC by Bradner and Mankin, gives an informative overview on how the rapidly growing Internet has stressed the scalability of the original architecture, ultimately resulting IPv6. The paper by Paxson describes a study of how routers behave in the Internet. Even though it is more than 15 years old, it continues to be highly cited and is a good example of how researchers study the dynamic behavior of the Internet. The final paper discusses multicast, presenting the approach to multicast originally used on the MBone.

- Bradner, S., and A. Mankin. The recommendation for the next generation IP protocol. *Request for Comments* 1752, January 1995.
- Paxson, V. End-to-end routing behavior in the Internet. *SIGCOMM '96*, pages 25–38, August 1996.
- Deering, S., and D. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems* 8(2):85–110, May 1990.

Some interesting experimental studies of the behavior of Internet routing are presented in Labovitz et al. [LAAJ00]. Another useful paper on the stability of BGP is by Gao and Rexford [GR01].

A collection of RFCs related to IPv6 can be found in Bradner and Mankin [BM95], and the most recent IPv6 spec is by Deering and Hinden [DH98]. There are dozens of other IPv6-related RFCs.

Protocol Independent Multicast (PIM) is described in Deering et al. [DEF<sup>+</sup>96] and Fenner et al. [FHHK06]; PIM-SSM is described in [Bha03]. [Wil00] and [HC99] are both very readable introductions to multicast with interesting historical details.

Multiprotocol Label Switching and the related protocols that fed its development are described in Chandranmenon et al. [CV95], Rekhter et al. [RDR<sup>+</sup>97], and Davie et al. [DR00]. The latter reference describes many applications of MPLS such as traffic engineering, fast recovery from network failures, and virtual private networks. [RR06] provides the specification of MPLS/BGP VPNs, a form of layer 3 VPN that can be provided over MPLS networks.

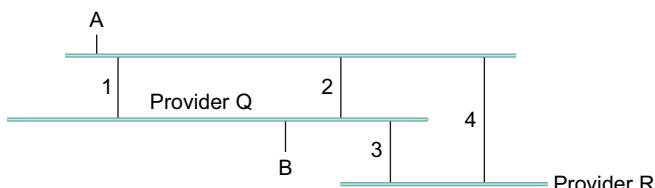
Mobile IP is defined in Perkins [Per02] for IPv4 and in Johnson et al. [JPA04] for IPv6. Basagni et al. [BCGS04] provide a good background of the problems and research in mobile *ad hoc* networking, while one of the primary MANET routing protocols is described by Clausen et al. [CHCB01].

Finally, we recommend the following live references:

- <http://www.isoc.org/internet/history/>: A collection of links related to Internet history, including some articles written by the original researchers who built the Internet.
- <http://bgp.potaroo.net/>: Lots of data about the growth of the routing tables in the Internet, including IPv6 deployment.

## EXERCISES

1. Consider the network shown in Figure 4.28, in which horizontal lines represent transit providers and numbered vertical lines are interprovider links.
- (a) How many routes to P could provider Q's BGP speakers receive?
- (b) Suppose Q and P adopt the policy that outbound traffic is routed to the closest link to the destination's provider, thus minimizing their own cost. What paths will traffic from host A to host B and from host B to host A take?



■ FIGURE 4.28 Network for Exercise 1.

- (c) What could Q do to have the B → A traffic use the closer link 1?
- (d) What could Q do to have the B → A traffic pass through R?
2. Give an example of an arrangement of routers grouped into autonomous systems so that the path with the fewest hops from a point A to another point B crosses the same AS twice. Explain what BGP would do with this situation.
3. Let  $A$  be the number of autonomous systems on the Internet, and let  $D$  (for diameter) be the maximum AS path length.
- Give a connectivity model for which  $D$  is of order  $\log A$  and another for which  $D$  is of order  $\sqrt{A}$ .
  - Assuming each AS number is 2 bytes and each network number is 4 bytes, give an estimate for the amount of data a BGP speaker must receive to keep track of the AS path to every network. Express your answer in terms of  $A$ ,  $D$ , and the number of networks  $N$ .
4. Propose a plausible addressing plan for IPv6 that runs out of bits. Specifically, provide a diagram such as Figure 4.11, perhaps with additional ID fields, that adds up to more than 128 bits, together with plausible justifications for the size of each field. You may assume fields are divided on byte boundaries and that the InterfaceID is 64 bits. (Hint: Consider fields that would approach maximum allocation only under unusual circumstances.) Can you do this if the InterfaceID is 48 bits?
5. Suppose P, Q, and R are network service providers with respective CIDR address allocations C1.0.0.0/8, C2.0.0.0/8, and C3.0.0.0/8. Each provider's customers initially receive address allocations that are a subset of the provider's. P has the following customers:

PA, with allocation C1.A3.0.0/16

PB, with allocation C1.B0.0.0/12.

Q has the following customers:

QA, with allocation C2.0A.10.0/20

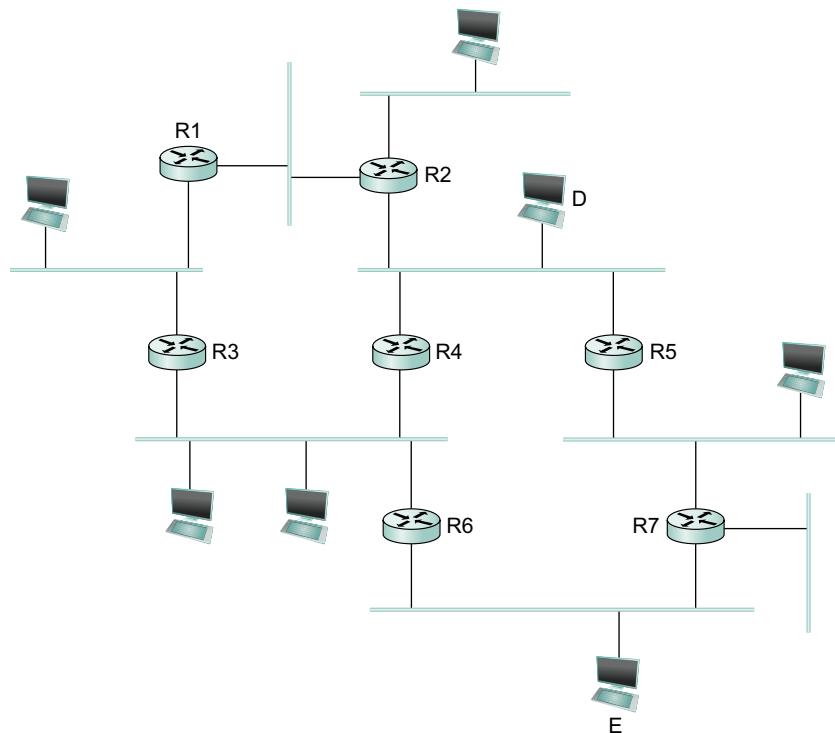
QB, with allocation C2.0B.0.0/16.

Assume there are no other providers or customers.

- Give routing tables for P, Q, and R assuming each provider connects to both of the others.

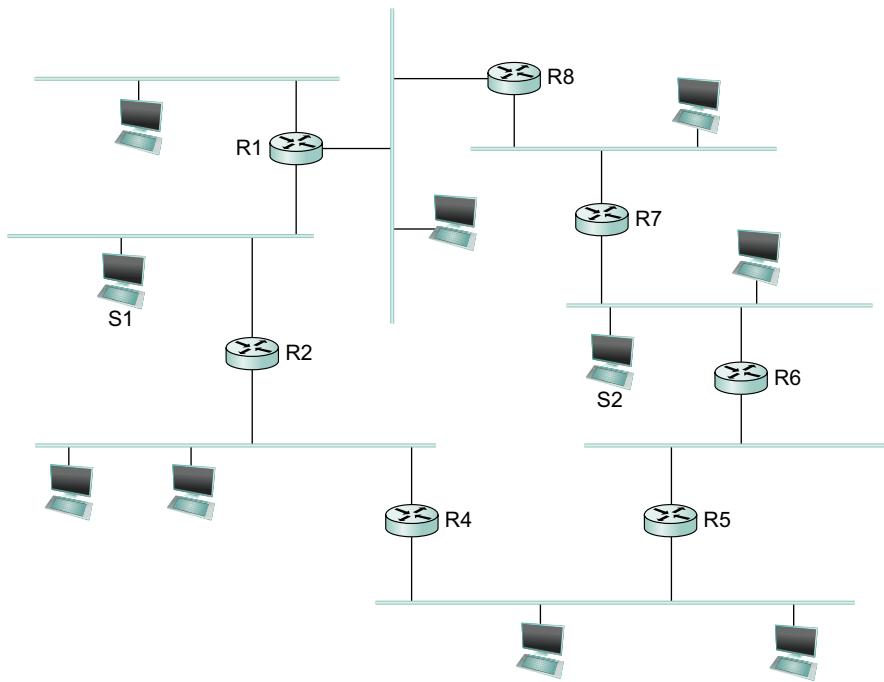
- (b) Now assume P is connected to Q and Q is connected to R, but P and R are not directly connected. Give tables for P and R.
- (c) Suppose customer PA acquires a direct link to Q, and QA acquires a direct link to P, in addition to existing links. Give tables for P and Q, ignoring R.
6. In the previous problem, assume each provider connects to both others. Suppose customer PA switches to provider Q and customer QB switches to provider R. Use the CIDR longest-match rule to give routing tables for all three providers that allow PA and QB to switch without renumbering.
7. Suppose most of the Internet used some form of geographical addressing, but that a large international organization has a single IP network address and routes its internal traffic over its own links.
- (a) Explain the routing inefficiency for the organization's inbound traffic inherent in this situation.
  - (b) Explain how the organization might solve this problem for outbound traffic.
  - (c) For your method above to work for inbound traffic, what would have to happen?
  - (d) Suppose the large organization now changes its addressing to separate geographical addresses for each office. What will its internal routing structure have to look like if internal traffic is still to be routed internally?
8. The telephone system uses geographical addressing. Why do you think this wasn't adopted as a matter of course by the Internet?
9. Suppose a small ISP X pays a larger ISP A to connect him to the rest of the Internet and also pays another ISP B to provide a fall-back connection to the Internet in the event that he loses connectivity via ISP A. If ISP X learns of a path to some prefix via ISP A, should he advertise that path to ISP B? Why or why not?
10. Suppose a site A is *multihomed*, in that it has two Internet connections from two different providers, P and Q. Provider-based addressing as in Exercise 5 is used, and A takes its address assignment from P. Q has a CIDR longest-match routing entry for A.

- (a) Describe what inbound traffic might flow on the A–Q connection. Consider cases where Q does and does not advertise A to the world using BGP.
  - (b) What is the minimum advertising of its route to A that Q must do in order for all inbound traffic to reach A via Q if the P–A link breaks?
  - (c) What problems must be overcome if A is to use both links for its outbound traffic?
- ★ 11. Suppose a network N within a larger organization A acquires its own direct connection to an Internet Service Provider, in addition to an existing connection via A. Let R1 be the router connecting N to its own provider, and let R2 be the router connecting N to the rest of A.
- (a) Assuming N remains a subnet of A, how should R1 and R2 be configured? What limitations would still exist with N's use of its separate connection? Would A be prevented from using N's connection? Specify your configuration in terms of what R1 and R2 should advertise, and with what paths. Assume a BGP-like mechanism is available.
  - (b) Now suppose N gets its own network number; how does this change your answer in (a)?
  - (c) Describe a router configuration that would allow A to use N's link when its own link is down.
12. How do routers determine that an incoming IP packet is to be multicast? Give answers for both IPv4 and IPv6.
13. Suppose a multicast group is intended to be private to a particular routing domain. Can an IP multicast address be assigned to the group without consulting with other domains with no risk of conflicts?
14. Under what conditions could a non-router host on an Ethernet receive a IP multicast packet for a multicast group it has not joined?
- ✓ 15. Consider the example internet shown in Figure 4.29, in which sources D and E send packets to multicast group G. All hosts except D and E are members of G. Show the shortest-path multicast trees for each source.



■ FIGURE 4.29 Example internet for Exercise 15.

16. Consider the example internet shown in Figure 4.30 in which sources S1 and S2 send packets to multicast group G. All hosts except S1 and S2 are members of G. Show the shortest-path multicast trees for each source.
17. Suppose host A is sending to a multicast group; the recipients are leaf nodes of a tree rooted at A with depth  $N$  and with each nonleaf node having  $k$  children; there are thus  $k^N$  recipients.
  - (a) How many individual link transmissions are involved if A sends a multicast message to all recipients?
  - (b) How many individual link transmissions are involved if A sends unicast messages to each individual recipient?
  - (c) Suppose A sends to all recipients, but some messages are lost and retransmission is necessary. Unicast retransmissions to what fraction of the recipients is equivalent, in terms of individual link transmissions, to a multicast retransmission to all recipients?



■ FIGURE 4.30 Example Network for Exercise 16.

18. The existing Internet depends in many respects on participants being good “network citizens”—cooperating above and beyond adherence to standard protocols.
  - (a) In the PIM-SM scheme, who determines when to create a source-specific tree? How might this be a problem?
  - (b) In the PIM-SSM scheme, who determines when to create a source-specific tree? Why is this presumably not a problem?
19. (a) Draw an example internetwork where the BIDIR-PIM route from a source’s router to a group member’s router is longer than the PIM-SM source-specific route.  
(b) Draw an example where they are the same.
20. Determine whether or not the following IPv6 address notations are correct:
  - (a) ::0F53:6382:AB00:67DB:BB27:7332
  - (b) 7803:42F2::88EC:D4BA:B75D:11CD
  - (c) ::4BA8:95CC::DB97:4EAB

- (d) 74DC::02BA
- (e) ::00FF:128.112.92.116

21. MPLS labels are usually 20 bits long. Explain why this provides enough labels when MPLS is used for destination-based forwarding.
22. MPLS has sometimes been claimed to improve router performance. Explain why this might be true, and suggest reasons why in practice this may not be the case.
23. Assume that it takes 32 bits to carry each MPLS label that is added to a packet when the “shim” header of [Figure 4.20\(b\)](#) is used.
  - (a) How many additional bytes are needed to tunnel a packet using the MPLS techniques described in [Section 4.3.3](#)?
  - (b) How many additional bytes are needed, at a minimum, to tunnel a packet using an additional IP header as described in [Section 3.2.9](#)?
  - (c) Calculate the efficiency of bandwidth usage for each of the two tunneling approaches when the average packet size is 300 bytes. Repeat for 64-byte packets. Bandwidth efficiency is defined as (payload bytes carried) ÷ (total bytes carried).
24. RFC 791 describes the Internet Protocol and includes two options for source routing. Describe three disadvantages of using IP source route options compared to using MPLS for explicit routing. (Hint: The IP header including options may be at most 15 words long.)
25. DHCP allows a computer to acquire a new IP address whenever it moves to a new subnet. Why is this not always enough to address the communications needs of mobile hosts?
26. What is the main downside of requiring traffic destined to a mobile node to be sent first to its home agent?
27. Mobile IP allows a home agent to tell a correspondent node a new care-of address for a mobile node. How might such a mechanism be used to steal traffic? How could it be used to launch a flood of attack traffic at another node?

This page intentionally left blank

A SYSTEMS APPROACH

# End-to-End Protocols



*Victory is the beautiful, bright coloured flower. Transport is the stem without which it could never have blossomed.*

—Winston Churchill

The previous three chapters have described various technologies that can be used to connect together a collection of computers, ranging from simple Ethernets and wireless networks to global-scale internetworks. The next problem is to turn this host-to-host packet delivery service into a process-to-process communication channel. This is the role played by the *transport* level of the network architecture, which, because it supports communication between application programs running in end nodes, is sometimes called the *end-to-end* protocol.

## PROBLEM: GETTING PROCESSES TO COMMUNICATE

Two forces shape the end-to-end protocol. From above, the application-level processes that use its services have certain requirements. The following list itemizes some of the common properties that a transport protocol can be expected to provide:

- Guarantees message delivery
- Delivers messages in the same order they are sent

- Delivers at most one copy of each message
- Supports arbitrarily large messages
- Supports synchronization between the sender and the receiver
- Allows the receiver to apply flow control to the sender
- Supports multiple application processes on each host

Note that this list does not include all the functionality that application processes might want from the network. For example, it does not include security features like authentication or encryption, which are typically provided by protocols that sit above the transport level.

From below, the underlying network upon which the transport protocol operates has certain limitations in the level of service it can provide. Some of the more typical limitations of the network are that it may

- Drop messages
- Reorder messages
- Deliver duplicate copies of a given message
- Limit messages to some finite size
- Deliver messages after an arbitrarily long delay

Such a network is said to provide a *best-effort* level of service, as exemplified by the Internet.

The challenge, therefore, is to develop algorithms that turn the less-than-desirable properties of the underlying network into the high level of service required by application programs. Different transport protocols employ different combinations of these algorithms. This chapter looks at these algorithms in the context of four representative services—a simple asynchronous demultiplexing service, a reliable byte-stream service, a request/reply service, and a service for real-time applications.

In the case of the demultiplexing and byte-stream services, we use the Internet's User Datagram Protocol (UDP) and Transmission Control Protocol (TCP), respectively, to illustrate how these services are provided in practice. In the case of a request/reply service, we discuss the role it plays in a Remote Procedure Call (RPC) service and what features that entails. This discussion is capped off with a description of two widely used RPC protocols: SunRPC and DCE-RPC.

Finally, real-time applications make particular demands on the transport protocol, such as the need to carry timing information that allows audio or video samples to be played back at the appropriate point in time. We look at the requirements

placed by applications on such a protocol and the most widely used example, the Real-Time Transport Protocol (RTP).

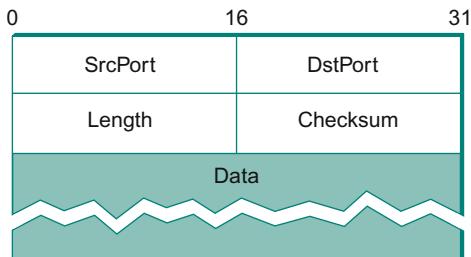
## 5.1 SIMPLE DEMULTIPLEXER (UDP)

The simplest possible transport protocol is one that extends the host-to-host delivery service of the underlying network into a process-to-process communication service. There are likely to be many processes running on any given host, so the protocol needs to add a level of demultiplexing, thereby allowing multiple application processes on each host to share the network. Aside from this requirement, the transport protocol adds no other functionality to the best-effort service provided by the underlying network. The Internet's User Datagram Protocol is an example of such a transport protocol.

The only interesting issue in such a protocol is the form of the address used to identify the target process. Although it is possible for processes to *directly* identify each other with an OS-assigned process id (pid), such an approach is only practical in a closed distributed system in which a single OS runs on all hosts and assigns each process a unique id. A more common approach, and the one used by UDP, is for processes to *indirectly* identify each other using an abstract locator, usually called a *port*. The basic idea is for a source process to send a message to a port and for the destination process to receive the message from a port.

The header for an end-to-end protocol that implements this demultiplexing function typically contains an identifier (port) for both the sender (source) and the receiver (destination) of the message. For example, the UDP header is given in Figure 5.1. Notice that the UDP port field is only 16 bits long. This means that there are up to 64K possible ports, clearly not enough to identify all the processes on all the hosts in the Internet. Fortunately, ports are not interpreted across the entire Internet, but only on a single host. That is, a process is really identified by a port on some particular host—a  $\langle \text{port}, \text{host} \rangle$  pair. In fact, this pair constitutes the demultiplexing key for the UDP protocol.

The next issue is how a process learns the port for the process to which it wants to send a message. Typically, a client process initiates a message

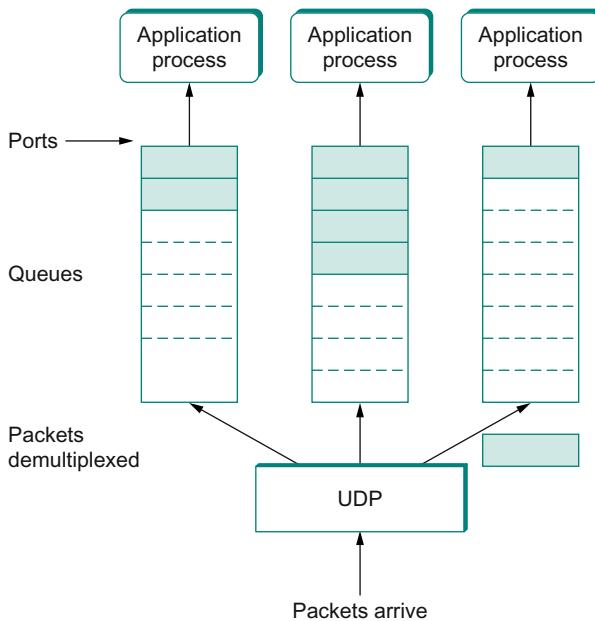


■ FIGURE 5.1 Format for UDP header.

exchange with a server process. Once a client has contacted a server, the server knows the client's port (from the *SrcPrt* field contained in the message header) and can reply to it. The real problem, therefore, is how the client learns the server's port in the first place. A common approach is for the server to accept messages at a *well-known port*. That is, each server receives its messages at some fixed port that is widely published, much like the emergency telephone service available in the United States at the well-known phone number 911. In the Internet, for example, the Domain Name Server (DNS) receives messages at well-known port 53 on each host, the mail service listens for messages at port 25, and the Unix talk program accepts messages at well-known port 517, and so on. This mapping is published periodically in an RFC and is available on most Unix systems in file /etc/services. Sometimes a well-known port is just the starting point for communication: The client and server use the well-known port to agree on some other port that they will use for subsequent communication, leaving the well-known port free for other clients.

An alternative strategy is to generalize this idea, so that there is only a single well-known port—the one at which the *port mapper* service accepts messages. A client would send a message to the port mapper's well-known port asking for the port it should use to talk to the “whatever” service, and the port mapper returns the appropriate port. This strategy makes it easy to change the port associated with different services over time and for each host to use a different port for the same service.

As just mentioned, a port is purely an abstraction. Exactly how it is implemented differs from system to system, or more precisely, from OS to OS. For example, the socket API described in Chapter 1 is an example implementation of ports. Typically, a port is implemented by a message queue, as illustrated in Figure 5.2. When a message arrives, the protocol (e.g., UDP) appends the message to the end of the queue. Should the



■ FIGURE 5.2 UDP message queue.

queue be full, the message is discarded. There is no flow-control mechanism in UDP to tell the sender to slow down. When an application process wants to receive a message, one is removed from the front of the queue. If the queue is empty, the process blocks until a message becomes available.

Finally, although UDP does not implement flow control or reliable/ordered delivery, it does provide one more function aside from demultiplexing messages to some application process—it also ensures the correctness of the message by the use of a checksum. (The UDP checksum is optional in IPv4 but is mandatory in IPv6.) The basic UDP checksum algorithm is the same one used for IP, as defined in Section 2.4.2—that is, it adds up a set of 16-bit words using ones complement arithmetic and takes the ones complement of the result. But the input data that is used for the checksum is a little counterintuitive.

The UDP checksum takes as input the UDP header, the contents of the message body, and something called the *pseudoheader*. The pseudoheader consists of three fields from the IP header—protocol number, source IP address, and destination IP address—plus the UDP length field. (Yes, the UDP length field is included twice in the checksum calculation.) The motivation behind having the pseudoheader is to verify that this

message has been delivered between the correct two endpoints. For example, if the destination IP address was modified while the packet was in transit, causing the packet to be misdelivered, this fact would be detected by the UDP checksum.



LAB 10:  
TCP

## 5.2 RELIABLE BYTE STREAM (TCP)

In contrast to a simple demultiplexing protocol like UDP, a more sophisticated transport protocol is one that offers a reliable, connection-oriented, byte-stream service. Such a service has proven useful to a wide assortment of applications because it frees the application from having to worry about missing or reordered data. The Internet's Transmission Control Protocol is probably the most widely used protocol of this type; it is also the most carefully tuned. It is for these two reasons that this section studies TCP in detail, although we identify and discuss alternative design choices at the end of the section.

In terms of the properties of transport protocols given in the problem statement at the start of this chapter, TCP guarantees the reliable, in-order delivery of a stream of bytes. It is a full-duplex protocol, meaning that each TCP connection supports a pair of byte streams, one flowing in each direction. It also includes a flow-control mechanism for each of these byte streams that allows the receiver to limit how much data the sender can transmit at a given time. Finally, like UDP, TCP supports a demultiplexing mechanism that allows multiple application programs on any given host to simultaneously carry on a conversation with their peers. In addition to the above features, TCP also implements a highly tuned (and still evolving) congestion-control mechanism. The idea of this mechanism is to throttle how fast TCP sends data, not for the sake of keeping the sender from over-running the receiver, but so as to keep the sender from overloading the network. A description of TCP's congestion-control mechanism is postponed until [Chapter 6](#), where we discuss it in the larger context of how network resources are fairly allocated.

Since many people confuse congestion control and flow control, we restate the difference. *Flow control* involves preventing senders from over-running the capacity of receivers. *Congestion control* involves preventing too much data from being injected into the network, thereby causing switches or links to become overloaded. Thus, flow control is an end-to-end issue, while congestion control is concerned with how hosts and networks interact.

### 5.2.1 End-to-End Issues

At the heart of TCP is the sliding window algorithm. Even though this is the same basic algorithm we saw in Section 2.5.2, because TCP runs over the Internet rather than a point-to-point link, there are many important differences. This subsection identifies these differences and explains how they complicate TCP. The following subsections then describe how TCP addresses these and other complications.

First, whereas the sliding window algorithm presented in Section 2.5.2 runs over a single physical link that always connects the same two computers, TCP supports logical connections between processes that are running on any two computers in the Internet. This means that TCP needs an explicit connection establishment phase during which the two sides of the connection agree to exchange data with each other. This difference is analogous to having to dial up the other party, rather than having a dedicated phone line. TCP also has an explicit connection teardown phase. One of the things that happens during connection establishment is that the two parties establish some shared state to enable the sliding window algorithm to begin. Connection teardown is needed so each host knows it is OK to free this state.

Second, whereas a single physical link that always connects the same two computers has a fixed round-trip time (RTT), TCP connections are likely to have widely different round-trip times. For example, a TCP connection between a host in San Francisco and a host in Boston, which are separated by several thousand kilometers, might have an RTT of 100 ms, while a TCP connection between two hosts in the same room, only a few meters apart, might have an RTT of only 1 ms. The same TCP protocol must be able to support both of these connections. To make matters worse, the TCP connection between hosts in San Francisco and Boston might have an RTT of 100 ms at 3 a.m., but an RTT of 500 ms at 3 p.m. Variations in the RTT are even possible during a single TCP connection that lasts only a few minutes. What this means to the sliding window algorithm is that the timeout mechanism that triggers retransmissions must be adaptive. (Certainly, the timeout for a point-to-point link must be a settable parameter, but it is not necessary to adapt this timer for a particular pair of nodes.)

A third difference is that packets may be reordered as they cross the Internet, but this is not possible on a point-to-point link where the first packet put into one end of the link must be the first to appear at the

other end. Packets that are slightly out of order do not cause a problem since the sliding window algorithm can reorder packets correctly using the sequence number. The real issue is how far out of order packets can get or, said another way, how late a packet can arrive at the destination. In the worst case, a packet can be delayed in the Internet until the IP time to live (TTL) field expires, at which time the packet is discarded (and hence there is no danger of it arriving late). Knowing that IP throws packets away after their TTL expires, TCP assumes that each packet has a maximum lifetime. The exact lifetime, known as the *maximum segment lifetime* (MSL), is an engineering choice. The current recommended setting is 120 seconds. Keep in mind that IP does not directly enforce this 120-second value; it is simply a conservative estimate that TCP makes of how long a packet might live in the Internet. The implication is significant—TCP has to be prepared for very old packets to suddenly show up at the receiver, potentially confusing the sliding window algorithm.

Fourth, the computers connected to a point-to-point link are generally engineered to support the link. For example, if a link's delay  $\times$  bandwidth product is computed to be 8 KB—meaning that a window size is selected to allow up to 8 KB of data to be unacknowledged at a given time—then it is likely that the computers at either end of the link have the ability to buffer up to 8 KB of data. Designing the system otherwise would be silly. On the other hand, almost any kind of computer can be connected to the Internet, making the amount of resources dedicated to any one TCP connection highly variable, especially considering that any one host can potentially support hundreds of TCP connections at the same time. This means that TCP must include a mechanism that each side uses to “learn” what resources (e.g., how much buffer space) the other side is able to apply to the connection. This is the flow control issue.

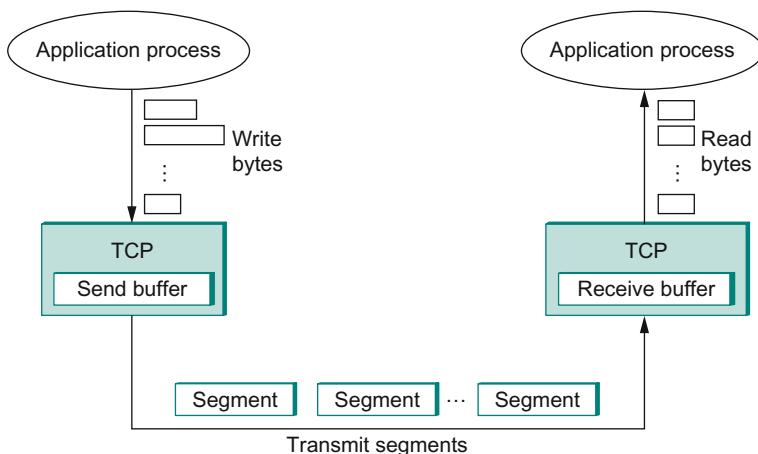
Fifth, because the transmitting side of a directly connected link cannot send any faster than the bandwidth of the link allows, and only one host is pumping data into the link, it is not possible to unknowingly congest the link. Said another way, the load on the link is visible in the form of a queue of packets at the sender. In contrast, the sending side of a TCP connection has no idea what links will be traversed to reach the destination. For example, the sending machine might be directly connected to a relatively fast Ethernet—and capable of sending data at a rate of 100 Mbps—but somewhere out in the middle of the network, a 1.5-Mbps T1 link must be traversed. And, to make matters worse, data being generated by many different sources might be trying to traverse this same slow link. This leads

to the problem of network congestion. Discussion of this topic is delayed until Chapter 6.

We conclude this discussion of end-to-end issues by comparing TCP's approach to providing a reliable/ordered delivery service with the approach used by X.25 networks. In TCP, the underlying IP network is assumed to be unreliable and to deliver messages out of order; TCP uses the sliding window algorithm on an end-to-end basis to provide reliable/ordered delivery. In contrast, X.25 networks use the sliding window protocol within the network, on a hop-by-hop basis. The assumption behind this approach is that if messages are delivered reliably and in order between each pair of nodes along the path between the source host and the destination host, then the end-to-end service also guarantees reliable/ordered delivery.

The problem with this latter approach is that a sequence of hop-by-hop guarantees does not necessarily add up to an end-to-end guarantee. First, if a heterogeneous link (say, an Ethernet) is added to one end of the path, then there is no guarantee that this hop will preserve the same service as the other hops. Second, just because the sliding window protocol guarantees that messages are delivered correctly from node A to node B, and then from node B to node C, it does not guarantee that node B behaves perfectly. For example, network nodes have been known to introduce errors into messages while transferring them from an input buffer to an output buffer. They have also been known to accidentally reorder messages. As a consequence of these small windows of vulnerability, it is still necessary to provide true end-to-end checks to guarantee reliable/ordered service, even though the lower levels of the system also implement that functionality.

This discussion serves to illustrate one of the most important principles in system design—the *end-to-end argument*. In a nutshell, the end-to-end argument says that a function (in our example, providing reliable/ordered delivery) should not be provided in the lower levels of the system unless it can be completely and correctly implemented at that level. Therefore, this rule argues in favor of the TCP/IP approach. This rule is not absolute, however. It does allow for functions to be incompletely provided at a low level as a performance optimization. This is why it is perfectly consistent with the end-to-end argument to perform error detection (e.g., CRC) on a hop-by-hop basis; detecting and retransmitting a single corrupt packet across one hop is preferable to having to retransmit an entire file end-to-end.



■ FIGURE 5.3 How TCP manages a byte stream.

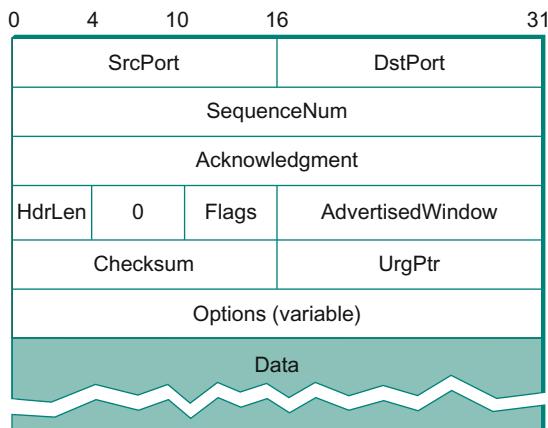
### 5.2.2 Segment Format

TCP is a byte-oriented protocol, which means that the sender writes bytes into a TCP connection and the receiver reads bytes out of the TCP connection. Although “byte stream” describes the service TCP offers to application processes, TCP does not, itself, transmit individual bytes over the Internet. Instead, TCP on the source host buffers enough bytes from the sending process to fill a reasonably sized packet and then sends this packet to its peer on the destination host. TCP on the destination host then empties the contents of the packet into a receive buffer, and the receiving process reads from this buffer at its leisure. This situation is illustrated in Figure 5.3, which, for simplicity, shows data flowing in only one direction. Remember that, in general, a single TCP connection supports byte streams flowing in both directions.

The packets exchanged between TCP peers in Figure 5.3 are called *segments*, since each one carries a segment of the byte stream. Each TCP segment contains the header schematically depicted in Figure 5.4. The relevance of most of these fields will become apparent throughout this section. For now, we simply introduce them.

The SrcPort and DstPort fields identify the source and destination ports, respectively, just as in UDP. These two fields, plus the source and destination IP addresses, combine to uniquely identify each TCP connection. That is, TCP’s demux key is given by the 4-tuple

$$\langle \text{SrcPort}, \text{SrcIPAddr}, \text{DstPort}, \text{DstIPAddr} \rangle$$

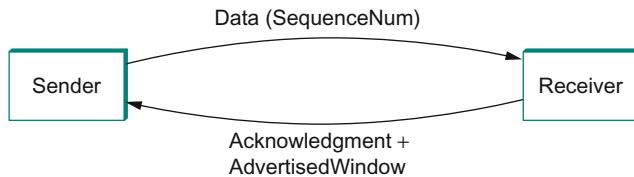


■ FIGURE 5.4 TCP header format.

Note that because TCP connections come and go, it is possible for a connection between a particular pair of ports to be established, used to send and receive data, and closed, and then at a later time for the same pair of ports to be involved in a second connection. We sometimes refer to this situation as two different *incarnations* of the same connection.

The Acknowledgment, SequenceNum, and AdvertisedWindow fields are all involved in TCP's sliding window algorithm. Because TCP is a byte-oriented protocol, each byte of data has a sequence number. The SequenceNum field contains the sequence number for the first byte of data carried in that segment, and the Acknowledgment and AdvertisedWindow fields carry information about the flow of data going in the other direction. To simplify our discussion, we ignore the fact that data can flow in both directions, and we concentrate on data that has a particular SequenceNum flowing in one direction and Acknowledgment and AdvertisedWindow values flowing in the opposite direction, as illustrated in Figure 5.5. The use of these three fields is described more fully in Section 5.2.4.

The 6-bit Flags field is used to relay control information between TCP peers. The possible flags include SYN, FIN, RESET, PUSH, URG, and ACK. The SYN and FIN flags are used when establishing and terminating a TCP connection, respectively. Their use is described in Section 5.2.3. The ACK flag is set any time the Acknowledgment field is valid, implying that the receiver should pay attention to it. The URG flag signifies that this segment contains urgent data. When this flag is set, the UrgPtr field indicates



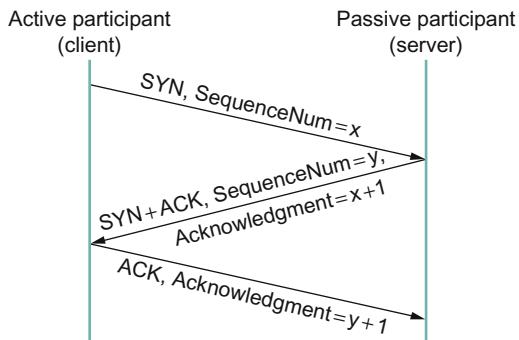
■ FIGURE 5.5 Simplified illustration (showing only one direction) of the TCP process, with data flow in one direction and ACKs in the other.

where the nonurgent data contained in this segment begins. The urgent data is contained at the front of the segment body, up to and including a value of UrgPtr bytes into the segment. The PUSH flag signifies that the sender invoked the push operation, which indicates to the receiving side of TCP that it should notify the receiving process of this fact. We discuss these last two features more in Section 5.2.7. Finally, the RESET flag signifies that the receiver has become confused—for example, because it received a segment it did not expect to receive—and so wants to abort the connection.

Finally, the Checksum field is used in exactly the same way as for UDP—it is computed over the TCP header, the TCP data, and the pseudoheader, which is made up of the source address, destination address, and length fields from the IP header. The checksum is required for TCP in both IPv4 and IPv6. Also, since the TCP header is of variable length (options can be attached after the mandatory fields), a HdrLen field is included that gives the length of the header in 32-bit words. This field is also known as the Offset field, since it measures the offset from the start of the packet to the start of the data.

### 5.2.3 Connection Establishment and Termination

A TCP connection begins with a client (caller) doing an active open to a server (callee). Assuming that the server had earlier done a passive open, the two sides engage in an exchange of messages to establish the connection. (Recall from Chapter 1 that a party wanting to initiate a connection performs an active open, while a party willing to accept a connection does a passive open.) Only after this connection establishment phase is over do the two sides begin sending data. Likewise, as soon as a participant is done sending data, it closes one direction of the connection, which causes TCP to initiate a round of connection termination messages. Notice that,



■ FIGURE 5.6 Timeline for three-way handshake algorithm.

while connection setup is an asymmetric activity (one side does a passive open and the other side does an active open), connection teardown is symmetric (each side has to close the connection independently).<sup>1</sup> Therefore, it is possible for one side to have done a close, meaning that it can no longer send data, but for the other side to keep the other half of the bidirectional connection open and to continue sending data.

### Three-Way Handshake

The algorithm used by TCP to establish and terminate a connection is called a *three-way handshake*. We first describe the basic algorithm and then show how it is used by TCP. The three-way handshake involves the exchange of three messages between the client and the server, as illustrated by the timeline given in Figure 5.6.

The idea is that two parties want to agree on a set of parameters, which, in the case of opening a TCP connection, are the starting sequence numbers the two sides plan to use for their respective byte streams. In general, the parameters might be any facts that each side wants the other to know about. First, the client (the active participant) sends a segment to the server (the passive participant) stating the initial sequence number it plans to use (Flags = SYN, SequenceNum =  $x$ ). The server then responds with a single segment that both acknowledges the client's sequence number (Flags = ACK, Ack =  $x + 1$ ) and states its own beginning sequence number (Flags = SYN, SequenceNum =  $y$ ). That is, both the SYN and

<sup>1</sup>To be more precise, connection setup can be symmetric, with both sides trying to open the connection at the same time, but the common case is for one side to do an active open and the other side to do a passive open.

ACK bits are set in the Flags field of this second message. Finally, the client responds with a third segment that acknowledges the server's sequence number (Flags = ACK, Ack =  $y + 1$ ). The reason why each side acknowledges a sequence number that is one larger than the one sent is that the Acknowledgment field actually identifies the "next sequence number expected," thereby implicitly acknowledging all earlier sequence numbers. Although not shown in this timeline, a timer is scheduled for each of the first two segments, and if the expected response is not received the segment is retransmitted.

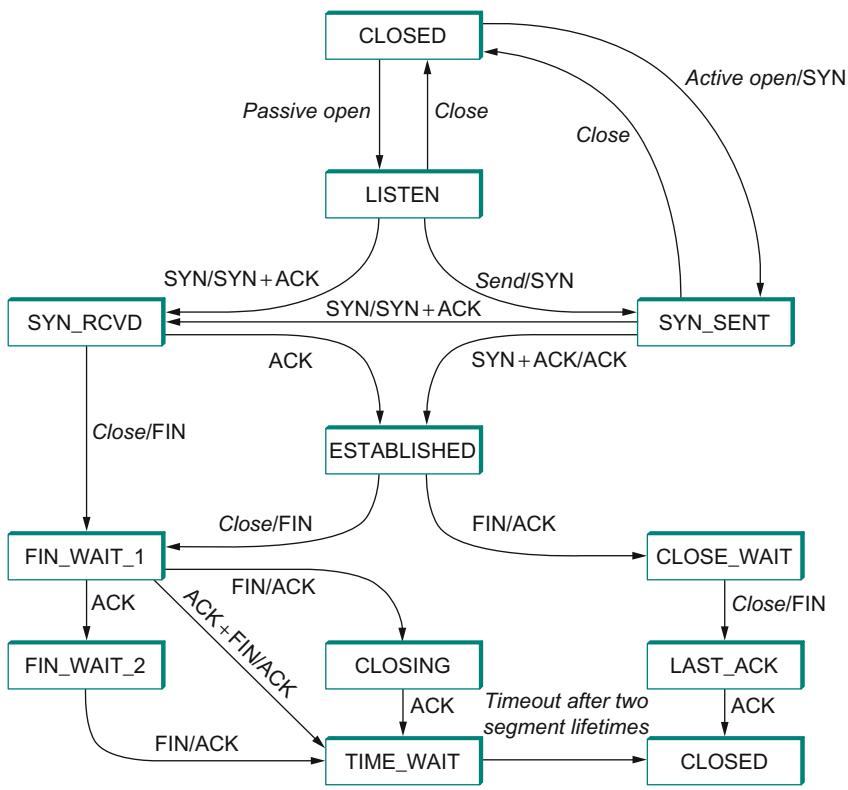
You may be asking yourself why the client and server have to exchange starting sequence numbers with each other at connection setup time. It would be simpler if each side simply started at some "well-known" sequence number, such as 0. In fact, the TCP specification requires that each side of a connection select an initial starting sequence number at random. The reason for this is to protect against two incarnations of the same connection reusing the same sequence numbers too soon—that is, while there is still a chance that a segment from an earlier incarnation of a connection might interfere with a later incarnation of the connection.

#### *State-Transition Diagram*

TCP is complex enough that its specification includes a state-transition diagram. A copy of this diagram is given in Figure 5.7. This diagram shows only the states involved in opening a connection (everything above ESTABLISHED) and in closing a connection (everything below ESTABLISHED). Everything that goes on while a connection is open—that is, the operation of the sliding window algorithm—is hidden in the ESTABLISHED state.

TCP's state-transition diagram is fairly easy to understand. Each circle denotes a state that one end of a TCP connection can find itself in. All connections start in the CLOSED state. As the connection progresses, the connection moves from state to state according to the arcs. Each arc is labeled with a tag of the form *event/action*. Thus, if a connection is in the LISTEN state and a SYN segment arrives (i.e., a segment with the SYN flag set), the connection makes a transition to the SYN\_RCVD state and takes the action of replying with an ACK + SYN segment.

Notice that two kinds of events trigger a state transition: (1) a segment arrives from the peer (e.g., the event on the arc from LISTEN to



■ FIGURE 5.7 TCP state-transition diagram.

SYN\\_RCVD), or (2) the local application process invokes an operation on TCP (e.g., the *active open* event on the arc from CLOSED to SYN\\_SENT). In other words, TCP's state-transition diagram effectively defines the *semantics* of both its peer-to-peer interface and its service interface, as defined in Section 1.3.1. The *syntax* of these two interfaces is given by the segment format (as illustrated in Figure 5.4) and by some application programming interface (an example of which is given in Section 1.4.1), respectively.

Now let's trace the typical transitions taken through the diagram in Figure 5.7. Keep in mind that at each end of the connection, TCP makes different transitions from state to state. When opening a connection, the server first invokes a passive open operation on TCP, which causes TCP to move to the LISTEN state. At some later time, the client does an active

open, which causes its end of the connection to send a SYN segment to the server and to move to the SYN\_SENT state. When the SYN segment arrives at the server, it moves to the SYN\_RECV state and responds with a SYN + ACK segment. The arrival of this segment causes the client to move to the ESTABLISHED state and to send an ACK back to the server. When this ACK arrives, the server finally moves to the ESTABLISHED state. In other words, we have just traced the three-way handshake.

There are three things to notice about the connection establishment half of the state-transition diagram. First, if the client's ACK to the server is lost, corresponding to the third leg of the three-way handshake, then the connection still functions correctly. This is because the client side is already in the ESTABLISHED state, so the local application process can start sending data to the other end. Each of these data segments will have the ACK flag set, and the correct value in the Acknowledgment field, so the server will move to the ESTABLISHED state when the first data segment arrives. This is actually an important point about TCP—every segment reports what sequence number the sender is expecting to see next, even if this repeats the same sequence number contained in one or more previous segments.

The second thing to notice about the state-transition diagram is that there is a funny transition out of the LISTEN state whenever the local process invokes a *send* operation on TCP. That is, it is possible for a passive participant to identify both ends of the connection (i.e., itself and the remote participant that it is willing to have connect to it), and then for it to change its mind about waiting for the other side and instead actively establish the connection. To the best of our knowledge, this is a feature of TCP that no application process actually takes advantage of.

The final thing to notice about the diagram is the arcs that are not shown. Specifically, most of the states that involve sending a segment to the other side also schedule a timeout that eventually causes the segment to be present if the expected response does not happen. These retransmissions are not depicted in the state-transition diagram. If after several tries the expected response does not arrive, TCP gives up and returns to the CLOSED state.

Turning our attention now to the process of terminating a connection, the important thing to keep in mind is that the application process on both sides of the connection must independently close its half of the connection. If only one side closes the connection, then this

means it has no more data to send, but it is still available to receive data from the other side. This complicates the state-transition diagram because it must account for the possibility that the two sides invoke the *close* operator at the same time, as well as the possibility that first one side invokes *close* and then, at some later time, the other side invokes *close*. Thus, on any one side there are three combinations of transitions that get a connection from the ESTABLISHED state to the CLOSED state:

- This side closes first: ESTABLISHED → FIN\_WAIT\_1 → FIN\_WAIT\_2 → TIME\_WAIT → CLOSED.
- The other side closes first: ESTABLISHED → CLOSE\_WAIT → LAST\_ACK → CLOSED.
- Both sides close at the same time: ESTABLISHED → FIN\_WAIT\_1 → CLOSING → TIME\_WAIT → CLOSED.

There is actually a fourth, although rare, sequence of transitions that leads to the CLOSED state; it follows the arc from FIN\_WAIT\_1 to TIME\_WAIT. We leave it as an exercise for you to figure out what combination of circumstances leads to this fourth possibility.

The main thing to recognize about connection teardown is that a connection in the TIME\_WAIT state cannot move to the CLOSED state until it has waited for two times the maximum amount of time an IP datagram might live in the Internet (i.e., 120 seconds). The reason for this is that, while the local side of the connection has sent an ACK in response to the other side's FIN segment, it does not know that the ACK was successfully delivered. As a consequence, the other side might retransmit its FIN segment, and this second FIN segment might be delayed in the network. If the connection were allowed to move directly to the CLOSED state, then another pair of application processes might come along and open the same connection (i.e., use the same pair of port numbers), and the delayed FIN segment from the earlier incarnation of the connection would immediately initiate the termination of the later incarnation of that connection.

### 5.2.4 Sliding Window Revisited

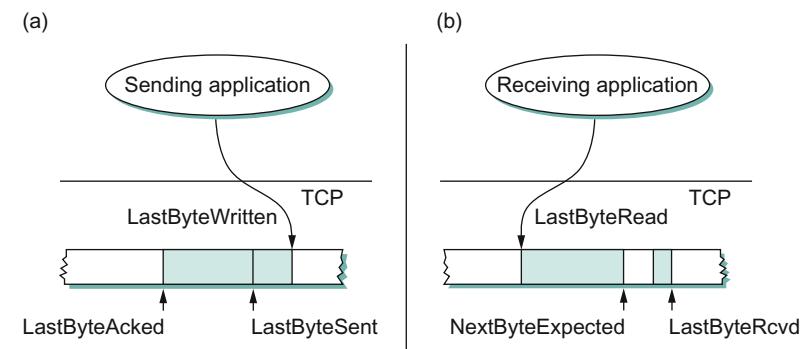
We are now ready to discuss TCP's variant of the sliding window algorithm, which serves several purposes: (1) it guarantees the reliable delivery of data, (2) it ensures that data is delivered in order, and (3) it enforces

flow control between the sender and the receiver. TCP's use of the sliding window algorithm is the same as we saw in Section 2.5.2 in the case of the first two of these three functions. Where TCP differs from the earlier algorithm is that it folds the flow-control function in as well. In particular, rather than having a fixed-size sliding window, the receiver *advertises* a window size to the sender. This is done using the `AdvertisedWindow` field in the TCP header. The sender is then limited to having no more than a value of `AdvertisedWindow` bytes of unacknowledged data at any given time. The receiver selects a suitable value for `AdvertisedWindow` based on the amount of memory allocated to the connection for the purpose of buffering data. The idea is to keep the sender from over-running the receiver's buffer. We discuss this at greater length below.

#### *Reliable and Ordered Delivery*

To see how the sending and receiving sides of TCP interact with each other to implement reliable and ordered delivery, consider the situation illustrated in Figure 5.8. TCP on the sending side maintains a send buffer. This buffer is used to store data that has been sent but not yet acknowledged, as well as data that has been written by the sending application but not transmitted. On the receiving side, TCP maintains a receive buffer. This buffer holds data that arrives out of order, as well as data that is in the correct order (i.e., there are no missing bytes earlier in the stream) but that the application process has not yet had the chance to read.

To make the following discussion simpler to follow, we initially ignore the fact that both the buffers and the sequence numbers are of some finite



■ FIGURE 5.8 Relationship between TCP send buffer (a) and receive buffer (b).

size and hence will eventually wrap around. Also, we do not distinguish between a pointer into a buffer where a particular byte of data is stored and the sequence number for that byte.

Looking first at the sending side, three pointers are maintained into the send buffer, each with an obvious meaning: `LastByteAcked`, `LastByteSent`, and `LastByteWritten`. Clearly,

$$\text{LastByteAcked} \leq \text{LastByteSent}$$

since the receiver cannot have acknowledged a byte that has not yet been sent, and

$$\text{LastByteSent} \leq \text{LastByteWritten}$$

since TCP cannot send a byte that the application process has not yet written. Also note that none of the bytes to the left of `LastByteAcked` need to be saved in the buffer because they have already been acknowledged, and none of the bytes to the right of `LastByteWritten` need to be buffered because they have not yet been generated.

A similar set of pointers (sequence numbers) are maintained on the receiving side: `LastByteRead`, `NextByteExpected`, and `LastByteRcvd`. The inequalities are a little less intuitive, however, because of the problem of out-of-order delivery. The first relationship

$$\text{LastByteRead} < \text{NextByteExpected}$$

is true because a byte cannot be read by the application until it is received *and* all preceding bytes have also been received. `NextByteExpected` points to the byte immediately after the latest byte to meet this criterion. Second,

$$\text{NextByteExpected} \leq \text{LastByteRcvd} + 1$$

since, if data has arrived in order, `NextByteExpected` points to the byte after `LastByteRcvd`, whereas if data has arrived out of order, then `NextByteExpected` points to the start of the first gap in the data, as in Figure 5.8. Note that bytes to the left of `LastByteRead` need not be buffered because they have already been read by the local application process, and bytes to the right of `LastByteRcvd` need not be buffered because they have not yet arrived.

#### Flow Control

Most of the above discussion is similar to that found in Section 2.5.2; the only real difference is that this time we elaborated on the fact that the

sending and receiving application processes are filling and emptying their local buffer, respectively. (The earlier discussion glossed over the fact that data arriving from an upstream node was filling the send buffer and data being transmitted to a downstream node was emptying the receive buffer.)

You should make sure you understand this much before proceeding because now comes the point where the two algorithms differ more significantly. In what follows, we reintroduce the fact that both buffers are of some finite size, denoted `MaxSendBuffer` and `MaxRcvBuffer`, although we don't worry about the details of how they are implemented. In other words, we are only interested in the number of bytes being buffered, not in where those bytes are actually stored.

Recall that in a sliding window protocol, the size of the window sets the amount of data that can be sent without waiting for acknowledgment from the receiver. Thus, the receiver throttles the sender by advertising a window that is no larger than the amount of data that it can buffer. Observe that TCP on the receive side must keep

$$\text{LastByteRcvd} - \text{LastByteRead} \leq \text{MaxRcvBuffer}$$

to avoid overflowing its buffer. It therefore advertises a window size of

$$\begin{aligned}\text{AdvertisedWindow} &= \text{MaxRcvBuffer} - ((\text{NextByteExpected} - 1) \\ &\quad - \text{LastByteRead})\end{aligned}$$

which represents the amount of free space remaining in its buffer. As data arrives, the receiver acknowledges it as long as all the preceding bytes have also arrived. In addition, `LastByteRcvd` moves to the right (is incremented), meaning that the advertised window potentially shrinks. Whether or not it shrinks depends on how fast the local application process is consuming data. If the local process is reading data just as fast as it arrives (causing `LastByteRead` to be incremented at the same rate as `LastByteRcvd`), then the advertised window stays open (i.e.,  $\text{AdvertisedWindow} = \text{MaxRcvBuffer}$ ). If, however, the receiving process falls behind, perhaps because it performs a very expensive operation on each byte of data that it reads, then the advertised window grows smaller with every segment that arrives, until it eventually goes to 0.

TCP on the send side must then adhere to the advertised window it gets from the receiver. This means that at any given time, it must ensure that

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{AdvertisedWindow}$$

Said another way, the sender computes an *effective* window that limits how much data it can send:

$$\text{EffectiveWindow} = \text{AdvertisedWindow} - (\text{LastByteSent} - \text{LastByteAcked})$$

Clearly, EffectiveWindow must be greater than 0 before the source can send more data. It is possible, therefore, that a segment arrives acknowledging  $x$  bytes, thereby allowing the sender to increment LastByteAcked by  $x$ , but because the receiving process was not reading any data, the advertised window is now  $x$  bytes smaller than the time before. In such a situation, the sender would be able to free buffer space, but not to send any more data.

All the while this is going on, the send side must also make sure that the local application process does not overflow the send buffer—that is, that

$$\text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSendBuffer}$$

If the sending process tries to write  $y$  bytes to TCP, but

$$(\text{LastByteWritten} - \text{LastByteAcked}) + y > \text{MaxSendBuffer}$$

then TCP blocks the sending process and does not allow it to generate more data.

It is now possible to understand how a slow receiving process ultimately stops a fast sending process. First, the receive buffer fills up, which means the advertised window shrinks to 0. An advertised window of 0 means that the sending side cannot transmit any data, even though data it has previously sent has been successfully acknowledged. Finally, not being able to transmit any data means that the send buffer fills up, which ultimately causes TCP to block the sending process. As soon as the receiving process starts to read data again, the receive-side TCP is able to open its window back up, which allows the send-side TCP to transmit data out of its buffer. When this data is eventually acknowledged, LastByteAcked is incremented, the buffer space holding this acknowledged data becomes free, and the sending process is unblocked and allowed to proceed.

There is only one remaining detail that must be resolved—how does the sending side know that the advertised window is no longer 0? As mentioned above, TCP *always* sends a segment in response to a received data segment, and this response contains the latest values for the Acknowledge and AdvertisedWindow fields, even if these values have not changed since

the last time they were sent. The problem is this. Once the receive side has advertised a window size of 0, the sender is not permitted to send any more data, which means it has no way to discover that the advertised window is no longer 0 at some time in the future. TCP on the receive side does not spontaneously send nondata segments; it only sends them in response to an arriving data segment.

TCP deals with this situation as follows. Whenever the other side advertises a window size of 0, the sending side persists in sending a segment with 1 byte of data every so often. It knows that this data will probably not be accepted, but it tries anyway, because each of these 1-byte segments triggers a response that contains the current advertised window. Eventually, one of these 1-byte probes triggers a response that reports a nonzero advertised window.



Note that the reason the sending side periodically sends this probe segment is that TCP is designed to make the receive side as simple as possible—it simply responds to segments from the sender, and it never initiates any activity on its own. This is an example of a well-recognized (although not universally applied) protocol design rule, which, for lack of a better name, we call the *smart sender/dumb receiver* rule. Recall that we saw another example of this rule when we discussed the use of NAKs in Section 2.5.2.

#### *Protecting against Wraparound*

This subsection and the next consider the size of the SequenceNum and AdvertisedWindow fields and the implications of their sizes on TCP's correctness and performance. TCP's SequenceNum field is 32 bits long, and its AdvertisedWindow field is 16 bits long, meaning that TCP has easily satisfied the requirement of the sliding window algorithm that the sequence number space be twice as big as the window size:  $2^{32} \gg 2 \times 2^{16}$ . However, this requirement is not the interesting thing about these two fields. Consider each field in turn.

The relevance of the 32-bit sequence number space is that the sequence number used on a given connection might wrap around—a byte with sequence number  $x$  could be sent at one time, and then at a later time a second byte with the same sequence number  $x$  might be sent. Once again, we assume that packets cannot survive in the Internet for longer than the recommended MSL. Thus, we currently need to make sure that the sequence number does not wrap around within a 120-second period

**Table 5.1 Time Until 32-Bit Sequence Number Space Wraps Around**

Bandwidth	Time until Wraparound
T1 (1.5 Mbps)	6.4 hours
Ethernet (10 Mbps)	57 minutes
T3 (45 Mbps)	13 minutes
Fast Ethernet (100 Mbps)	6 minutes
OC-3 (155 Mbps)	4 minutes
OC-12 (622 Mbps)	55 seconds
OC-48 (2.5 Gbps)	14 seconds

of time. Whether or not this happens depends on how fast data can be transmitted over the Internet—that is, how fast the 32-bit sequence number space can be consumed. (This discussion assumes that we are trying to consume the sequence number space as fast as possible, but of course we will be if we are doing our job of keeping the pipe full.) **Table 5.1** shows how long it takes for the sequence number to wrap around on networks with various bandwidths.

As you can see, the 32-bit sequence number space is adequate for most situations encountered on today's networks, but given that OC-192 links exist in the Internet backbone, and that most servers now come with gigabit Ethernet (or 10 Gbps) interfaces, it is getting close to the point where 32 bits is too small. Fortunately, the IETF has already worked out an extension to TCP that effectively extends the sequence number space to protect against the sequence number wrapping around. This and related extensions are described in [Section 5.2.8](#).

#### *Keeping the Pipe Full*

The relevance of the 16-bit AdvertisedWindow field is that it must be big enough to allow the sender to keep the pipe full. Clearly, the receiver is free to not open the window as large as the AdvertisedWindow field allows; we are interested in the situation in which the receiver has enough buffer space to handle as much data as the largest possible AdvertisedWindow allows.

In this case, it is not just the network bandwidth but the  $\text{delay} \times \text{bandwidth}$  product that dictates how big the AdvertisedWindow field needs to

**Table 5.2 Required Window Size for 100-ms RTT**

Bandwidth	Delay × Bandwidth Product
T1 (1.5 Mbps)	18 KB
Ethernet (10 Mbps)	122 KB
T3 (45 Mbps)	549 KB
Fast Ethernet (100 Mbps)	1.2 MB
OC-3 (155 Mbps)	1.8 MB
OC-12 (622 Mbps)	7.4 MB
OC-48 (2.5 Gbps)	29.6 MB

be—the window needs to be opened far enough to allow a full delay  $\times$  bandwidth product's worth of data to be transmitted. Assuming an RTT of 100 ms (a typical number for a cross-country connection in the United States), Table 5.2 gives the delay  $\times$  bandwidth product for several network technologies.

As you can see, TCP's `AdvertisedWindow` field is in even worse shape than its `SequenceNum` field—it is not big enough to handle even a T3 connection across the continental United States, since a 16-bit field allows us to advertise a window of only 64 KB. The very same TCP extension mentioned above (see Section 5.2.8) provides a mechanism for effectively increasing the size of the advertised window.

### 5.2.5 Triggering Transmission

We next consider a surprisingly subtle issue: how TCP decides to transmit a segment. As described earlier, TCP supports a byte-stream abstraction; that is, application programs write bytes into the stream, and it is up to TCP to decide that it has enough bytes to send a segment. What factors govern this decision?

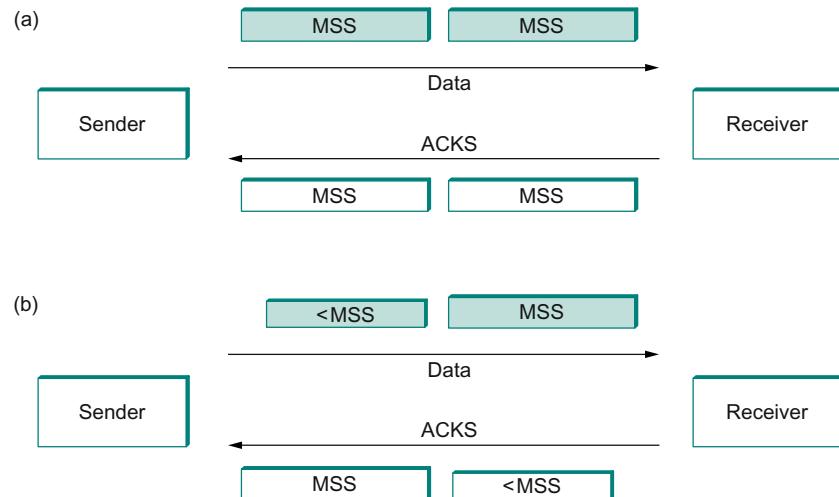
If we ignore the possibility of flow control—that is, we assume the window is wide open, as would be the case when a connection first starts—then TCP has three mechanisms to trigger the transmission of a segment. First, TCP maintains a variable, typically called the *maximum segment size (MSS)*, and it sends a segment as soon as it has collected MSS bytes from the sending process. MSS is usually set to the size of

the largest segment TCP can send without causing the local IP to fragment. That is, MSS is set to the maximum transmission unit (MTU) of the directly connected network, minus the size of the TCP and IP headers. The second thing that triggers TCP to transmit a segment is that the sending process has explicitly asked it to do so. Specifically, TCP supports a *push* operation, and the sending process invokes this operation to effectively flush the buffer of unsent bytes. The final trigger for transmitting a segment is that a timer fires; the resulting segment contains as many bytes as are currently buffered for transmission. However, as we will soon see, this “timer” isn’t exactly what you expect.

### Silly Window Syndrome

Of course, we can’t just ignore flow control, which plays an obvious role in throttling the sender. If the sender has MSS bytes of data to send and the window is open at least that much, then the sender transmits a full segment. Suppose, however, that the sender is accumulating bytes to send, but the window is currently closed. Now suppose an ACK arrives that effectively opens the window enough for the sender to transmit, say,  $\text{MSS}/2$  bytes. Should the sender transmit a half-full segment or wait for the window to open to a full MSS? The original specification was silent on this point, and early implementations of TCP decided to go ahead and transmit a half-full segment. After all, there is no telling how long it will be before the window opens further.

It turns out that the strategy of aggressively taking advantage of any available window leads to a situation now known as the *silly window syndrome*. Figure 5.9 helps visualize what happens. If you think of a TCP stream as a conveyer belt with “full” containers (data segments) going in one direction and empty containers (ACKs) going in the reverse direction, then MSS-sized segments correspond to large containers and 1-byte segments correspond to very small containers. As long as the sender is sending MSS-sized segments and the receiver ACKs at least one MSS of data at a time, everything is good (Figure 5.9(a)). But, what if the receiver has to reduce the window, so that at some time the sender can’t send a full MSS of data? If the sender aggressively fills a smaller-than-MSS empty container as soon as it arrives, then the receiver will ACK that smaller number of bytes, and hence the small container introduced into the system remains in the system indefinitely. That is, it is immediately filled and emptied at each end and is never coalesced with adjacent containers to



**FIGURE 5.9** Silly window syndrome. (a) As long as the sender sends MSS-sized segments and the receiver ACKs one MSS at a time, the system works smoothly. (b) As soon as the sender sends less than one MSS, or the receiver ACKs less than one MSS, a small “container” enters the system and continues to circulate.

create larger containers, as in Figure 5.9(b). This scenario was discovered when early implementations of TCP regularly found themselves filling the network with tiny segments.

Note that the silly window syndrome is only a problem when either the sender transmits a small segment or the receiver opens the window a small amount. If neither of these happens, then the small container is never introduced into the stream. It's not possible to outlaw sending small segments; for example, the application might do a *push* after sending a single byte. It is possible, however, to keep the receiver from introducing a small container (i.e., a small open window). The rule is that after advertising a zero window the receiver must wait for space equal to an MSS before it advertises an open window.

Since we can't eliminate the possibility of a small container being introduced into the stream, we also need mechanisms to coalesce them. The receiver can do this by delaying ACKs—sending one combined ACK rather than multiple smaller ones—but this is only a partial solution because the receiver has no way of knowing how long it is safe to delay waiting either for another segment to arrive or for the application to read more data (thus opening the window). The ultimate solution falls to the

sender, which brings us back to our original issue: When does the TCP sender decide to transmit a segment?

#### Nagle's Algorithm

Returning to the TCP sender, if there is data to send but the window is open less than MSS, then we may want to wait some amount of time before sending the available data, but the question is how long? If we wait too long, then we hurt interactive applications like Telnet. If we don't wait long enough, then we risk sending a bunch of tiny packets and falling into the silly window syndrome. The answer is to introduce a timer and to transmit when the timer expires.

While we could use a clock-based timer—for example, one that fires every 100 ms—Nagle introduced an elegant *self-clocking* solution. The idea is that as long as TCP has any data in flight, the sender will eventually receive an ACK. This ACK can be treated like a timer firing, triggering the transmission of more data. Nagle's algorithm provides a simple, unified rule for deciding when to transmit:

When the application produces data to send

```
if both the available data and the window ≥ MSS
    send a full segment
else
    if there is unACKed data in flight
        buffer the new data until an ACK arrives
    else
        send all the new data now
```

In other words, it's always OK to send a full segment if the window allows. It's also all right to immediately send a small amount of data if there are currently no segments in transit, but if there is anything in flight the sender must wait for an ACK before transmitting the next segment. Thus, an interactive application like Telnet that continually writes one byte at a time will send data at a rate of one segment per RTT. Some segments will contain a single byte, while others will contain as many bytes as the user was able to type in one round-trip time. Because some applications cannot afford such a delay for each write it does to a TCP connection, the socket interface allows the application to turn off Nagle's algorithm by setting the TCP\_NODELAY option. Setting this option means that data is transmitted as soon as possible.

### 5.2.6 Adaptive Retransmission

Because TCP guarantees the reliable delivery of data, it retransmits each segment if an ACK is not received in a certain period of time. TCP sets this timeout as a function of the RTT it expects between the two ends of the connection. Unfortunately, given the range of possible RTTs between any pair of hosts in the Internet, as well as the variation in RTT between the same two hosts over time, choosing an appropriate timeout value is not that easy. To address this problem, TCP uses an adaptive retransmission mechanism. We now describe this mechanism and how it has evolved over time as the Internet community has gained more experience using TCP.

#### *Original Algorithm*

We begin with a simple algorithm for computing a timeout value between a pair of hosts. This is the algorithm that was originally described in the TCP specification—and the following description presents it in those terms—but it could be used by any end-to-end protocol.

The idea is to keep a running average of the RTT and then to compute the timeout as a function of this RTT. Specifically, every time TCP sends a data segment, it records the time. When an ACK for that segment arrives, TCP reads the time again, and then takes the difference between these two times as a *SampleRTT*. TCP then computes an *EstimatedRTT* as a weighted average between the previous estimate and this new sample. That is,

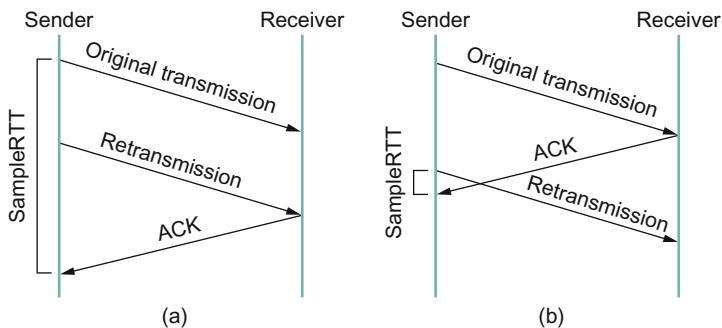
$$\text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + (1 - \alpha) \times \text{SampleRTT}$$

The parameter  $\alpha$  is selected to *smooth* the EstimatedRTT. A small  $\alpha$  tracks changes in the RTT but is perhaps too heavily influenced by temporary fluctuations. On the other hand, a large  $\alpha$  is more stable but perhaps not quick enough to adapt to real changes. The original TCP specification recommended a setting of  $\alpha$  between 0.8 and 0.9. TCP then uses EstimatedRTT to compute the timeout in a rather conservative way:

$$\text{TimeOut} = 2 \times \text{EstimatedRTT}$$

#### *Karn/Partridge Algorithm*

After several years of use on the Internet, a rather obvious flaw was discovered in this simple algorithm. The problem was that an ACK does not really acknowledge a transmission; it actually acknowledges the receipt of data. In other words, whenever a segment is retransmitted and then



■ FIGURE 5.10 Associating the ACK with (a) original transmission versus (b) retransmission.

an ACK arrives at the sender, it is impossible to determine if this ACK should be associated with the first or the second transmission of the segment for the purpose of measuring the sample RTT. It is necessary to know which transmission to associate it with so as to compute an accurate SampleRTT. As illustrated in Figure 5.10, if you assume that the ACK is for the original transmission but it was really for the second, then the SampleRTT is too large (a); if you assume that the ACK is for the second transmission but it was actually for the first, then the SampleRTT is too small (b).

The solution, which was proposed in 1987, is surprisingly simple. Whenever TCP retransmits a segment, it stops taking samples of the RTT; it only measures SampleRTT for segments that have been sent only once. This solution is known as the Karn/Partridge algorithm, after its inventors. Their proposed fix also includes a second small change to TCP's timeout mechanism. Each time TCP retransmits, it sets the next timeout to be twice the last timeout, rather than basing it on the last EstimatedRTT. That is, Karn and Partridge proposed that TCP use exponential backoff, similar to what the Ethernet does. The motivation for using exponential backoff is simple: Congestion is the most likely cause of lost segments, meaning that the TCP source should not react too aggressively to a timeout. In fact, the more times the connection times out, the more cautious the source should become. We will see this idea again, embodied in a much more sophisticated mechanism, in Chapter 6.

#### *Jacobson/Karels Algorithm*

The Karn/Partridge algorithm was introduced at a time when the Internet was suffering from high levels of network congestion. Their approach

was designed to fix some of the causes of that congestion, but, although it was an improvement, the congestion was not eliminated. The following year (1988), two other researchers—Jacobson and Karels—proposed a more drastic change to TCP to battle congestion. The bulk of that proposed change is described in [Chapter 6](#). Here, we focus on the aspect of that proposal that is related to deciding when to time out and retransmit a segment.

As an aside, it should be clear how the timeout mechanism is related to congestion—if you time out too soon, you may unnecessarily retransmit a segment, which only adds to the load on the network. As we will see in [Chapter 6](#), the other reason for needing an accurate timeout value is that a timeout is taken to imply congestion, which triggers a congestion-control mechanism. Finally, note that there is nothing about the Jacobson/Karels timeout computation that is specific to TCP. It could be used by any end-to-end protocol.

The main problem with the original computation is that it does not take the variance of the sample RTTs into account. Intuitively, if the variation among samples is small, then the EstimatedRTT can be better trusted and there is no reason for multiplying this estimate by 2 to compute the timeout. On the other hand, a large variance in the samples suggests that the timeout value should not be too tightly coupled to the EstimatedRTT.

In the new approach, the sender measures a new SampleRTT as before. It then folds this new sample into the timeout calculation as follows:

$$\text{Difference} = \text{SampleRTT} - \text{EstimatedRTT}$$

$$\text{EstimatedRTT} = \text{EstimatedRTT} + (\delta \times \text{Difference})$$

$$\text{Deviation} = \text{Deviation} + \delta(|\text{Difference}| - \text{Deviation})$$

where  $\delta$  is a fraction between 0 and 1. That is, we calculate both the mean RTT and the variation in that mean.

TCP then computes the timeout value as a function of both Estimated-RTT and Deviation as follows:

$$\text{TimeOut} = \mu \times \text{EstimatedRTT} + \phi \times \text{Deviation}$$

where based on experience,  $\mu$  is typically set to 1 and  $\phi$  is set to 4. Thus, when the variance is small, TimeOut is close to EstimatedRTT; a large variance causes the Deviation term to dominate the calculation.

### Implementation

There are two items of note regarding the implementation of timeouts in TCP. The first is that it is possible to implement the calculation for EstimatedRTT and Deviation without using floating-point arithmetic. Instead, the whole calculation is scaled by  $2^n$ , with  $\delta$  selected to be  $1/2^n$ . This allows us to do integer arithmetic, implementing multiplication and division using shifts, thereby achieving higher performance. The resulting calculation is given by the following code fragment, where  $n = 3$  (i.e.,  $\delta = 1/8$ ). Note that EstimatedRTT and Deviation are stored in their scaled-up forms, while the value of SampleRTT at the start of the code and of TimeOut at the end are real, unscaled values. If you find the code hard to follow, you might want to try plugging some real numbers into it and verifying that it gives the same results as the equations above.

```
{  
    SampleRTT -= (EstimatedRTT >> 3);  
    EstimatedRTT += SampleRTT;  
    if (SampleRTT < 0)  
        SampleRTT = -SampleRTT;  
    SampleRTT -= (Deviation >> 3);  
    Deviation += SampleRTT;  
    TimeOut = (EstimatedRTT >> 3) + (Deviation >> 1);  
}
```

The second point of note is that the Jacobson/Karels algorithm is only as good as the clock used to read the current time. On typical Unix implementations at the time, the clock granularity was as large as 500 ms, which is significantly larger than the average cross-country RTT of somewhere between 100 and 200 ms. To make matters worse, the Unix implementation of TCP only checked to see if a timeout should happen every time this 500-ms clock ticked and would only take a sample of the round-trip time once per RTT. The combination of these two factors could mean that a timeout would happen 1 second after the segment was transmitted. Once again, the extensions to TCP include a mechanism that makes this RTT calculation a bit more precise.

All of the retransmission algorithms we have discussed are based on acknowledgment timeouts, which indicate that a segment has probably been lost. Note that a timeout does not, however, tell the sender whether any segments it sent after the lost segment were successfully received.

This is because TCP acknowledgments are cumulative; they identify only the last segment that was received without any preceding gaps. The reception of segments that occur after a gap grows more frequent as faster networks lead to larger windows. If ACKs also told the sender which subsequent segments, if any, had been received, then the sender could be more intelligent about which segments it retransmits, draw better conclusions about the state of congestion, and make better RTT estimates. A TCP extension supporting this is described in Section 5.2.8.

### 5.2.7 Record Boundaries

Since TCP is a byte-stream protocol, the number of bytes written by the sender are not necessarily the same as the number of bytes read by the receiver. For example, the application might write 8 bytes, then 2 bytes, then 20 bytes to a TCP connection, while on the receiving side the application reads 5 bytes at a time inside a loop that iterates 6 times. TCP does not interject record boundaries between the 8th and 9th bytes, nor between the 10th and 11th bytes. This is in contrast to a message-oriented protocol, such as UDP, in which the message that is sent is exactly the same length as the message that is received.

Even though TCP is a byte-stream protocol, it has two different features that can be used by the sender to insert record boundaries into this byte stream, thereby informing the receiver how to break the stream of bytes into records. (Being able to mark record boundaries is useful, for example, in many database applications.) Both of these features were originally included in TCP for completely different reasons; they have only come to be used for this purpose over time.

The first mechanism is the urgent data feature, as implemented by the URG flag and the UrgPtr field in the TCP header. Originally, the urgent data mechanism was designed to allow the sending application to send *out-of-band* data to its peer. By “out of band” we mean data that is separate from the normal flow of data (e.g., a command to interrupt an operation already under way). This out-of-band data was identified in the segment using the UrgPtr field and was to be delivered to the receiving process as soon as it arrived, even if that meant delivering it before data with an earlier sequence number. Over time, however, this feature has not been used, so instead of signifying “urgent” data, it has come to be used to signify “special” data, such as a record marker. This use has developed because, as with the push operation, TCP on the receiving side must

inform the application that urgent data has arrived. That is, the urgent data in itself is not important. It is the fact that the sending process can effectively send a signal to the receiver that is important.

The second mechanism for inserting end-of-record markers into a byte is the *push* operation. Originally, this mechanism was designed to allow the sending process to tell TCP that it should send (flush) whatever bytes it had collected to its peer. The *push* operation can be used to implement record boundaries because the specification says that TCP must send whatever data it has buffered at the source when the application says push, and, optionally, TCP at the destination notifies the application whenever an incoming segment has the PUSH flag set. If the receiving side supports this option (the socket interface does not), then the push operation can be used to break the TCP stream into records.

Of course, the application program is always free to insert record boundaries without any assistance from TCP. For example, it can send a field that indicates the length of a record that is to follow, or it can insert its own record boundary markers into the data stream.

### 5.2.8 TCP Extensions

We have mentioned at four different points in this section that there are now extensions to TCP that help to mitigate some problem that TCP is facing as the underlying network gets faster. These extensions are designed to have as small an impact on TCP as possible. In particular, they are realized as options that can be added to the TCP header. (We glossed over this point earlier, but the reason why the TCP header has a HdrLen field is that the header can be of variable length; the variable part of the TCP header contains the options that have been added.) The significance of adding these extensions as options rather than changing the core of the TCP header is that hosts can still communicate using TCP even if they do not implement the options. Hosts that do implement the optional extensions, however, can take advantage of them. The two sides agree that they will use the options during TCP's connection establishment phase.

The first extension helps to improve TCP's timeout mechanism. Instead of measuring the RTT using a coarse-grained event, TCP can read the actual system clock when it is about to send a segment, and put this time—think of it as a 32-bit *timestamp*—in the segment's header. The receiver then echoes this timestamp back to the sender in its acknowledgement, and the sender subtracts this timestamp from the current time to

measure the RTT. In essence, the timestamp option provides a convenient place for TCP to store the record of when a segment was transmitted; it stores the time in the segment itself. Note that the endpoints in the connection do not need synchronized clocks, since the timestamp is written and read at the same end of the connection.

The second extension addresses the problem of TCP's 32-bit SequenceNum field wrapping around too soon on a high-speed network. Rather than define a new 64-bit sequence number field, TCP uses the 32-bit timestamp just described to effectively extend the sequence number space. In other words, TCP decides whether to accept or reject a segment based on a 64-bit identifier that has the SequenceNum field in the low-order 32 bits and the timestamp in the high-order 32 bits. Since the timestamp is always increasing, it serves to distinguish between two different incarnations of the same sequence number. Note that the timestamp is being used in this setting only to protect against wraparound; it is not treated as part of the sequence number for the purpose of ordering or acknowledging data.

The third extension allows TCP to advertise a larger window, thereby allowing it to fill larger delay  $\times$  bandwidth pipes that are made possible by high-speed networks. This extension involves an option that defines a *scaling factor* for the advertised window. That is, rather than interpreting the number that appears in the AdvertisedWindow field as indicating how many bytes the sender is allowed to have unacknowledged, this option allows the two sides of TCP to agree that the AdvertisedWindow field counts larger chunks (e.g., how many 16-byte units of data the sender can have unacknowledged). In other words, the window scaling option specifies how many bits each side should left-shift the AdvertisedWindow field before using its contents to compute an effective window.

The fourth extension allows TCP to augment its cumulative acknowledgment with selective acknowledgments of any additional segments that have been received but aren't contiguous with all previously received segments. This is the *selective acknowledgment*, or SACK, option. When the SACK option is used, the receiver continues to acknowledge segments normally—the meaning of the Acknowledge field does not change—but it also uses optional fields in the header to acknowledge any additional blocks of received data. This allows the sender to retransmit just the segments that are missing according to the selective acknowledgment.

Without SACK, there are only two reasonable strategies for a sender. The pessimistic strategy responds to a timeout by retransmitting not just

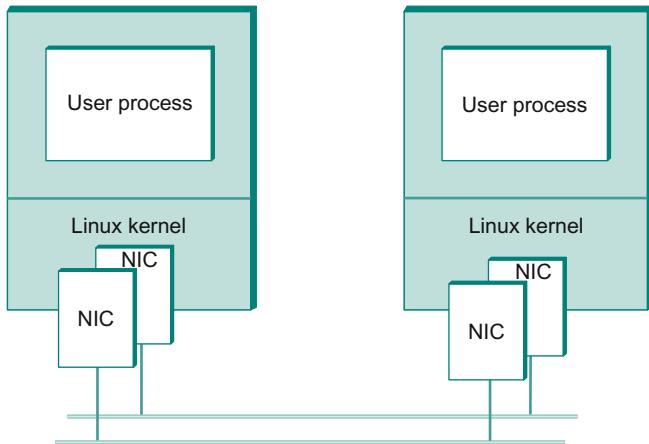
the segment that timed out, but any segments transmitted subsequently. In effect, the pessimistic strategy assumes the worst: that all those segments were lost. The disadvantage of the pessimistic strategy is that it may unnecessarily retransmit segments that were successfully received the first time. The other strategy is the optimistic strategy, which responds to a timeout by retransmitting only the segment that timed out. In effect, the optimistic approach assumes the rosiest scenario: that only the one segment has been lost. The disadvantage of the optimistic strategy is that it is very slow, unnecessarily, when a series of consecutive segments has been lost, as might happen when there is congestion. It is slow because each segment's loss is not discovered until the sender receives an ACK for its retransmission of the previous segment. So it consumes one RTT per segment until it has retransmitted all the segments in the lost series. With the SACK option, a better strategy is available to the sender: retransmit just the segments that fill the gaps between the segments that have been selectively acknowledged.

These extensions, by the way, are not the full story. We'll see some more extensions in the next chapter when we look at how TCP handles congestion. The Internet Assigned Numbers Authority (IANA) keeps track of all the options that are defined for TCP (and for many other Internet protocols). About 30 TCP options were defined at the time of writing (quite a few are experimental or obsolete, however). See the references at the end of the chapter for a link to IANA's protocol number registry.

### 5.2.9 Performance

Recall that [Chapter 1](#) introduced the two quantitative metrics by which network performance is evaluated: latency and throughput. As mentioned in that discussion, these metrics are influenced not only by the underlying hardware (e.g., propagation delay and link bandwidth) but also by software overheads. Now that we have a complete software-based protocol graph available to us that includes alternative transport protocols, we can discuss how to meaningfully measure its performance. The importance of such measurements is that they represent the performance seen by application programs.

We begin, as any report of experimental results should, by describing our experimental method. This includes the apparatus used in the experiments; in this case, each workstation has a pair of dual CPU 2.4-GHz Xeon processors running Linux. In order to enable speeds above 1 Gbps, a pair of Ethernet adaptors (labeled NIC, for network interface card) are used on



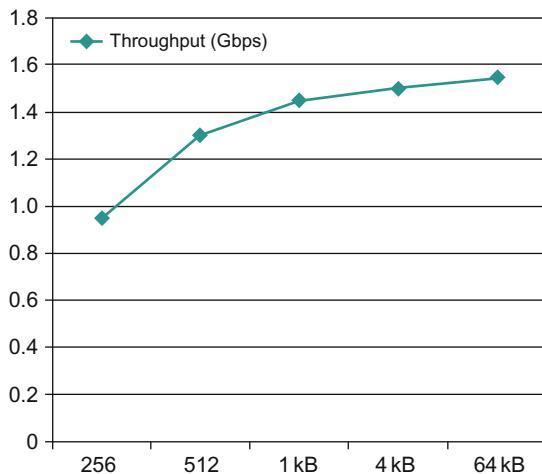
■ FIGURE 5.11 Measured system: Two Linux workstations and a pair of Gbps Ethernet links.

each machine. The Ethernet spans a single machine room so propagation is not an issue, making this a measure of processor/software overheads. A test program running on top of the socket interface simply tries to transfer data as quickly as possible from one machine to the other. Figure 5.11 illustrates the setup.

You may notice that this experimental setup is not especially bleeding edge in terms of the hardware or link speed. The point of this section is not to show how fast a particular protocol can run, but to illustrate the general methodology for measuring and reporting protocol performance.

The throughput test is performed for a variety of message sizes using a standard benchmarking tool called TTCP. The results of the throughput test are given in Figure 5.12. The key thing to notice in this graph is that throughput improves as the messages get larger. This makes sense—each message involves a certain amount of overhead, so a larger message means that this overhead is amortized over more bytes. The throughput curve flattens off above 1 KB, at which point the per-message overhead becomes insignificant when compared to the large number of bytes that the protocol stack has to process.

It's worth noting that the maximum throughput is less than 2 Gbps, the available link speed in this setup. Further testing and analysis of results would be needed to figure out where the bottleneck is (or if there is more than one). For example, looking at CPU load might give an indication



■ FIGURE 5.12 Measured throughput using TCP, for various message sizes.

of whether the CPU is the bottleneck or whether memory bandwidth, adaptor performance, or some other issue is to blame.

We also note that the network in this test is basically “perfect.” It has almost no delay or loss, so the only factors affecting performance are the TCP implementation and the workstation hardware and software. By contrast, most of the time we deal with networks that are far from perfect, notably our bandwidth-constrained, last-mile links and loss-prone wireless links. Before we can fully appreciate how these links affect TCP performance, we need to understand how TCP deals with *congestion*, which is the topic of Section 6.3.

At various times in the history of networking, the steadily increasing speed of network links has threatened to run ahead of what could be delivered to applications. For example, a large research effort was begun in the United States in 1989 to build “gigabit networks,” where the goal was not only to build links and switches that could run at 1Gbps or higher but also to deliver that throughput all the way to a single application process. There were some real problems (e.g., network adaptors, workstation architectures, and operating systems all had to be designed with network-to-application throughput in mind) and also some perceived problems that turned out to be not so serious. High on the list of such problems was the concern that existing transport protocols, TCP in particular, might not be up to the challenge of gigabit operation.

As it turns out, TCP has done well keeping up with the increasing demands of high-speed networks and applications. One of the most important factors was the introduction of window scaling to deal with larger bandwidth-delay products. However, there is often a big difference between the theoretical performance of TCP and what is achieved in practice. Relatively simple problems like copying the data more times than necessary as it passes from network adaptor to application can drive down performance, as can insufficient buffer memory when the bandwidth-delay product is large. And the dynamics of TCP are complex enough (as will become even more apparent in the next chapter) that subtle interactions among network behavior, application behavior, and the TCP protocol itself can dramatically alter performance.

For our purposes, it's worth noting that TCP continues to perform very well as network speeds increase, and when it runs up against some limit (normally related to congestion, increasing bandwidth-delay products, or both), researchers rush in to find solutions. We've seen some of those in this chapter, and we'll see some more in the next.

### 5.2.10 Alternative Design Choices

Although TCP has proven to be a robust protocol that satisfies the needs of a wide range of applications, the design space for transport protocols is quite large. TCP is by no means the only valid point in that design space. We conclude our discussion of TCP by considering alternative design choices. While we offer an explanation for why TCP's designers made the choices they did, we observe that other protocols exist that have made other choices, and more such protocols may appear in the future.

First, we have suggested from the very first chapter of this book that there are at least two interesting classes of transport protocols: stream-oriented protocols like TCP and request/reply protocols like RPC. In other words, we have implicitly divided the design space in half and placed TCP squarely in the stream-oriented half of the world. We could further divide the stream-oriented protocols into two groups—reliable and unreliable—with the former containing TCP and the latter being more suitable for interactive video applications that would rather drop a frame than incur the delay associated with a retransmission.

This exercise in building a transport protocol taxonomy is interesting and could be continued in greater and greater detail, but the world isn't as black and white as we might like. Consider the suitability of TCP as

a transport protocol for request/reply applications, for example. TCP is a full-duplex protocol, so it would be easy to open a TCP connection between the client and server, send the request message in one direction, and send the reply message in the other direction. There are two complications, however. The first is that TCP is a *byte*-oriented protocol rather than a *message*-oriented protocol, and request/reply applications always deal with messages. (We explore the issue of bytes versus messages in greater detail in a moment.) The second complication is that in those situations where both the request message and the reply message fit in a single network packet, a well-designed request/reply protocol needs only two packets to implement the exchange, whereas TCP would need at least nine: three to establish the connection, two for the message exchange, and four to tear down the connection. Of course, if the request or reply messages are large enough to require multiple network packets (e.g., it might take 100 packets to send a 100,000-byte reply message), then the overhead of setting up and tearing down the connection is inconsequential. In other words, it isn't always the case that a particular protocol cannot support a certain functionality; it's sometimes the case that one design is more efficient than another under particular circumstances.

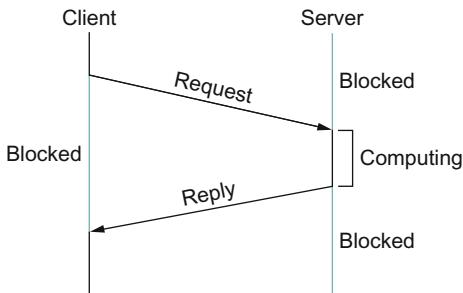
Second, as just suggested, you might question why TCP chose to provide a reliable *byte*-stream service rather than a reliable *message*-stream service; messages would be the natural choice for a database application that wants to exchange records. There are two answers to this question. The first is that a message-oriented protocol must, by definition, establish an upper bound on message sizes. After all, an infinitely long message is a byte stream. For any message size that a protocol selects, there will be applications that want to send larger messages, rendering the transport protocol useless and forcing the application to implement its own transport-like services. The second reason is that, while message-oriented protocols are definitely more appropriate for applications that want to send records to each other, you can easily insert record boundaries into a byte stream to implement this functionality, as described in Section 5.2.7.

A third decision made in the design of TCP is that it delivers bytes *in order* to the application. This means that it may hold onto bytes that were received out of order from the network, awaiting some missing bytes to fill a hole. This is enormously helpful for many applications but turns out to be quite unhelpful if the application is capable of processing data out of order. As a simple example, a Web page containing multiple embedded

images doesn't need all the images to be delivered in order before starting to display the page. In fact, there is a class of applications that would prefer to handle out-of-order data at the application layer, in return for getting data sooner when packets are dropped or misordered within the network. The desire to support such applications led to the creation of another IETF standard transport protocol known as the *Stream Control Transmission Protocol* (SCTP). SCTP provides a partially ordered delivery service, rather than the strictly ordered service of TCP. (SCTP also makes some other design decisions that differ from TCP, including message orientation and support of multiple IP addresses for a single session. See the Further Reading section for more details.)

Fourth, TCP chose to implement explicit setup/teardown phases, but this is not required. In the case of connection setup, it would certainly be possible to send all necessary connection parameters along with the first data message. TCP elected to take a more conservative approach that gives the receiver the opportunity to reject the connection before any data arrives. In the case of teardown, we could quietly close a connection that has been inactive for a long period of time, but this would complicate applications like Telnet that want to keep a connection alive for weeks at a time; such applications would be forced to send out-of-band “keep alive” messages to keep the connection state at the other end from disappearing.

Finally, TCP is a window-based protocol, but this is not the only possibility. The alternative is a *rate-based* design, in which the receiver tells the sender the rate—expressed in either bytes or packets per second—at which it is willing to accept incoming data. For example, the receiver might inform the sender that it can accommodate 100 packets a second. There is an interesting duality between windows and rate, since the number of packets (bytes) in the window, divided by the RTT, is exactly the rate. For example, a window size of 10 packets and a 100-ms RTT implies that the sender is allowed to transmit at a rate of 100 packets a second. It is by increasing or decreasing the advertised window size that the receiver is effectively raising or lowering the rate at which the sender can transmit. In TCP, this information is fed back to the sender in the `AdvertisedWindow` field of the ACK for every segment. One of the key issues in a rate-based protocol is how often the desired rate—which may change over time—is relayed back to the source: Is it for every packet, once per RTT, or only when the rate changes? While we have just now considered window versus rate in the context of flow control, it is an even more hotly



■ FIGURE 5.13 Timeline for RPC.

contested issue in the context of congestion control, which we will discuss in Chapter 6.

### 5.3 REMOTE PROCEDURE CALL

As discussed in Chapter 1, one common pattern of communication used by application programs is the request/reply paradigm, also called *message transaction*: A client sends a request message to a server, and the server responds with a reply message, with the client blocking (suspending execution) to wait for the reply. Figure 5.13 illustrates the basic interaction between the client and server in such a message transaction.

A transport protocol that supports the request/reply paradigm is much more than a UDP message going in one direction followed by a UDP message going in the other direction. It needs to deal with correctly identifying processes on remote hosts and correlating requests with responses. It may also need to overcome some or all of the limitations of the underlying network outlined in the problem statement at the beginning of this chapter. While TCP overcomes these limitations by providing a reliable byte-stream service, it doesn't match the request/reply paradigm very well either—going to the trouble to establish a TCP connection just to exchange a pair of messages seems like overkill. This section describes a third category of transport protocol, called *Remote Procedure Call* (RPC), that more closely matches the needs of an application involved in a request/reply message exchange.

#### 5.3.1 RPC Fundamentals

RPC is actually more than just a protocol—it is a popular mechanism for structuring distributed systems. RPC is popular because it is based on

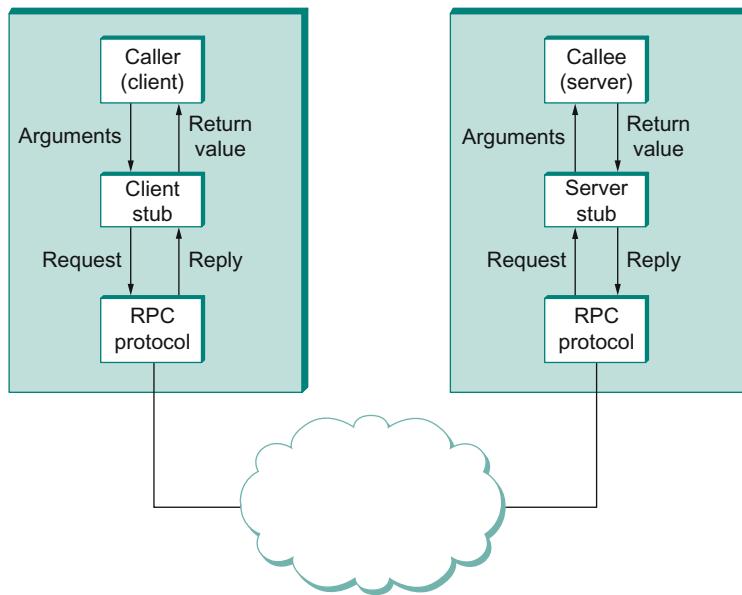
the semantics of a local procedure call—the application program makes a call into a procedure without regard for whether it is local or remote and blocks until the call returns. An application developer can be largely unaware of whether the procedure is local or remote, simplifying his task considerably. When the procedures being called are actually methods of remote objects in an object-oriented language, RPC is known as *remote method invocation* (RMI). While the RPC concept is simple, there are two main problems that make it more complicated than local procedure calls:

- The network between the calling process and the called process has much more complex properties than the backplane of a computer. For example, it is likely to limit message sizes and has a tendency to lose and reorder messages.
- The computers on which the calling and called processes run may have significantly different architectures and data representation formats.

Thus, a complete RPC mechanism actually involves two major components:

1. A protocol that manages the messages sent between the client and the server processes and that deals with the potentially undesirable properties of the underlying network
2. Programming language and compiler support to package the arguments into a request message on the client machine and then to translate this message back into the arguments on the server machine, and likewise with the return value (this piece of the RPC mechanism is usually called a *stub compiler*)

Figure 5.14 schematically depicts what happens when a client invokes a remote procedure. First, the client calls a local stub for the procedure, passing it the arguments required by the procedure. This stub hides the fact that the procedure is remote by translating the arguments into a request message and then invoking an RPC protocol to send the request message to the server machine. At the server, the RPC protocol delivers the request message to the server stub (sometimes called a *skeleton*), which translates it into the arguments to the procedure and then calls the local procedure. After the server procedure completes, it returns the answer to the server stub, which packages this return value in a reply message that it hands off to the RPC protocol for transmission back to



■ FIGURE 5.14 Complete RPC mechanism.

the client. The RPC protocol on the client passes this message up to the client stub, which translates it into a return value that it returns to the client program.

This section considers just the protocol-related aspects of an RPC mechanism. That is, it ignores the stubs and focuses instead on the RPC protocol, sometimes referred to as a request/reply protocol, that transmits messages between client and server. The transformation of arguments into messages and *vice versa* is covered in Chapter 7.

The term *RPC* refers to a type of protocol rather than a specific standard like TCP, so specific RPC protocols vary in the functions they perform. And, unlike TCP, which is the dominant reliable byte-stream protocol, there is no one dominant RPC protocol. Thus, in this section we will talk more about alternative design choices than previously.

#### *Identifiers in RPC*

Two functions that must be performed by any RPC protocol are:

- Provide a name space for uniquely identifying the procedure to be called.
- Match each reply message to the corresponding request message.

### What Layer Is RPC?

Once again, the “What layer is this?” issue raises its ugly head. To many people, especially those who adhere to a strictly layerist view of protocol architecture, RPC is implemented on top of a transport protocol (usually UDP) and so cannot itself (by definition) be a transport protocol. It is certainly valid, however, to argue that the Internet should have an RPC protocol, since RPC offers a process-to-process service that is fundamentally different from that offered by TCP and UDP. The usual response to such a suggestion, however, is that the Internet architecture does not prohibit network designers from implementing their own RPC protocol on top of UDP. Whichever side of the issue of whether the Internet should have an official RPC protocol you support, the important point is that the way you implement RPC in the Internet architecture says nothing about whether RPC should be considered a transport protocol or not.

Interestingly, there are other people who believe that RPC is the most interesting protocol in the world and that TCP/IP is just what you do when you want to go “off site.” This is the predominant view of the operating systems community, which has built countless OS kernels for distributed systems that contain exactly one protocol—you guessed it, RPC—running on top of a network device driver.

Our position is that any protocol that offers process-to-process service, as opposed to node-to-node or host-to-host service, qualifies as a transport protocol. Thus, RPC is a transport protocol and, in fact, can be implemented on top of other protocols that are themselves valid transport protocols.

The first problem has some similarities to the problem of identifying nodes in a network, something that we saw in previous chapters (IP addresses, for example, in [Chapter 4](#)). One of the design choices when identifying things is whether to make this name space flat or hierarchical. A flat name space would simply assign a unique, unstructured identifier (e.g., an integer) to each procedure, and this number would be carried in a single field in an RPC request message. This would require some kind of central coordination to avoid assigning the same procedure number to two different procedures. Alternatively, the protocol could implement a hierarchical name space, analogous to that used for file pathnames, which requires only that a file’s “basename” be unique within its directory. This approach potentially simplifies the job of ensuring uniqueness of procedure names. A hierarchical name space for RPC could be implemented by defining a set of fields in the request message format,

one for each level of naming in, say, a two- or three-level hierarchical name space.

The key to matching a reply message to the corresponding request is to uniquely identify request-replies pairs using a message ID field. A reply message had its message ID field set to the same value as the request message. When the client RPC module receives the reply, it uses the message ID to search for the corresponding outstanding request. To make the RPC transaction appear like a local procedure call to the caller, the caller is blocked (e.g., by using a semaphore) until the reply message is received. When the reply is received, the blocked caller is identified based on the request number in the reply, the remote procedure's return value is obtained from the reply, and the caller is unblocked so that it can return with that return value.

One of the recurrent challenges in RPC is dealing with unexpected responses, and we see this with message IDs. For example, consider the following pathological (but realistic) situation. A client machine sends a request message with a message ID of 0, then crashes and reboots, and then sends an unrelated request message, also with a message ID of 0. The server may not have been aware that the client crashed and rebooted and, upon seeing a request message with a message ID of 0, acknowledges it and discards it as a duplicate. The client never gets a response to the request.

One way to eliminate this problem is to use a *boot ID*. A machine's boot ID is a number that is incremented each time the machine reboots; this number is read from nonvolatile storage (e.g., a disk or flash drive), incremented, and written back to the storage device during the machine's start-up procedure. This number is then put in every message sent by that host. If a message is received with an old message ID but a new boot ID, it is recognized as a new message. In effect, the message ID and boot ID combine to form a unique ID for each transaction.

#### *Overcoming Network Limitations*

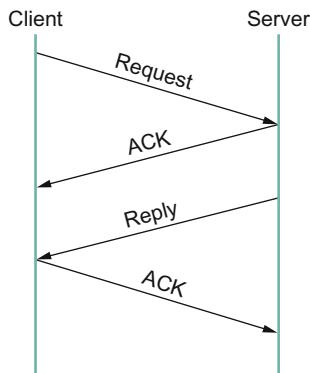
RPC protocols often perform additional functions to deal with the fact that networks are not perfect channels. Two such functions are:

- Provide reliable message delivery
- Support large message sizes through fragmentation and reassembly

An RPC protocol might implement reliability because the underlying protocols (e.g., UDP/IP) do not provide it, or perhaps to recover more quickly or efficiently from failures that otherwise would eventually be repaired by underlying protocols. An RPC protocol can implement reliability using acknowledgments and timeouts, similarly to TCP. The basic algorithm is straightforward, as illustrated by the timeline given in Figure 5.15. The client sends a request message and the server acknowledges it. Then, after executing the procedure, the server sends a reply message and the client acknowledges the reply.

Either a message carrying data (a request message or a reply message) or the ACK sent to acknowledge that message may be lost in the network. To account for this possibility, both client and server save a copy of each message they send until an ACK for it has arrived. Each side also sets a RETRANSMIT timer and resends the message should this timer expire. Both sides reset this timer and try again some agreed-upon number of times before giving up and freeing the message.

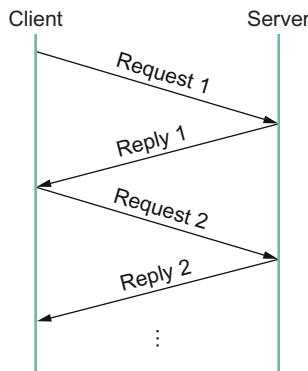
If an RPC client receives a reply message, clearly the corresponding request message must have been received by the server. Hence, the reply message itself is an *implicit acknowledgment*, and any additional acknowledgment from the server is not logically necessary. Similarly, a request message could implicitly acknowledge the preceding reply message—assuming the protocol makes request-reply transactions sequential, so that one transaction must complete before the next begins. Unfortunately, this sequentiality would severely limit RPC performance.



■ FIGURE 5.15 Simple timeline for a reliable RPC protocol.

A way out of this predicament is for the RPC protocol to implement a *channel* abstraction. Within a given channel, request/reply transactions are sequential—there can be only one transaction active on a given channel at any given time—but there can be multiple channels. Each message includes a channel ID field to indicate which channel the message belongs to. A request message in a given channel would implicitly acknowledge the previous reply in that channel, if it hadn't already been acknowledged. An application program can open multiple channels to a server if it wants to have more than one request/reply transaction between them at the same time (the application would need multiple threads). As illustrated in Figure 5.16, the reply message serves to acknowledge the request message, and a subsequent request acknowledges the preceding reply. Note that we saw a very similar approach—called *concurrent logical channels*—in Section 2.5.3 as a way to improve on the performance of a stop-and-wait reliability mechanism.

Another complication that RPC must address is that the server may take an arbitrarily long time to produce the result, and, worse yet, it may crash before generating the reply. Keep in mind that we are talking about the period of time after the server has acknowledged the request but before it has sent the reply. To help the client distinguish between a slow server and a dead server, the RPC's client side can periodically send an “Are you alive?” message to the server, and the server side responds with an ACK. Alternatively, the server could send “I am still alive” messages to the client without the client having first solicited them. The



■ FIGURE 5.16 Timeline for a reliable RPC protocol using implicit acknowledgment.

client-initiated approach is more scalable because it puts more of the per-client burden (managing the timeout timer) on the clients.

RPC reliability may include the property known as *at-most-once semantics*. This means that for every request message that the client sends, at most one copy of that message is delivered to the server. Each time the client calls a remote procedure, that procedure is invoked at most one time on the server machine. We say “at most once” rather than “exactly once” because it is always possible that either the network or the server machine has failed, making it impossible to deliver even one copy of the request message.

To implement at-most-once semantics, RPC on the server side must recognize duplicate requests (and ignore them), even if it has already successfully replied to the original request. Hence, it must maintain some state information that identifies past requests. One approach is to identify requests using sequence numbers, so a server need only remember the most recent sequence number. Unfortunately, this would limit an RPC to one outstanding request (to a given server) at a time, since one request must be completed before the request with the next sequence number can be transmitted. Once again, channels provide a solution. The server could recognize duplicate requests by remembering the current sequence number for each channel, without limiting the client to one request at a time.

As obvious as at-most-once sounds, not all RPC protocols support this behavior. Some support a semantics that is facetiously called *zero-or-more* semantics; that is, each invocation on a client results in the remote procedure being invoked zero or more times. It is not difficult to understand how this would cause problems for a remote procedure that changed some local state variable (e.g., incremented a counter) or that had some externally visible side effect (e.g., launched a missile) each time it was invoked. On the other hand, if the remote procedure being invoked is *idempotent*—multiple invocations have the same effect as just one—then the RPC mechanism need not support at-most-once semantics; a simpler (possibly faster) implementation will suffice.

As was the case with reliability, the two reasons why an RPC protocol might implement message fragmentation and reassembly are that it is not provided by the underlying protocol stack or that it can be implemented more efficiently by the RPC protocol. Consider the case where RPC is implemented on top of UDP/IP and relies on IP for fragmentation

and reassembly. If even one fragment of a message fails to arrive within a certain amount of time, IP discards the fragments that did arrive and the message is effectively lost. Eventually, the RPC protocol (assuming it implements reliability) would time out and retransmit the message. In contrast, consider an RPC protocol that implements its own fragmentation and reassembly and aggressively ACKs or NACKs (negatively acknowledges) individual fragments. Lost fragments would be more quickly detected and retransmitted, and only the lost fragments would be retransmitted, not the whole message.

#### *Synchronous versus Asynchronous Protocols*

One way to characterize a protocol is by whether it is *synchronous* or *asynchronous*. The precise meaning of these terms depends on where in the protocol hierarchy you use them. At the transport layer, it is most accurate to think of them as defining the extremes of a spectrum rather than as two mutually exclusive alternatives. The key attribute of any point along the spectrum is how much the sending process knows after the operation to send a message returns. In other words, if we assume that an application program invokes a `send` operation on a transport protocol, then exactly what does the application know about the success of the operation when the `send` operation returns?

At the *asynchronous* end of the spectrum, the application knows absolutely nothing when `send` returns. Not only does it not know if the message was received by its peer, but it doesn't even know for sure that the message has successfully left the local machine. At the *synchronous* end of the spectrum, the `send` operation typically returns a reply message. That is, the application not only knows that the message it sent was received by its peer, but it also knows that the peer has returned an answer. Thus, synchronous protocols implement the request/reply abstraction, while asynchronous protocols are used if the sender wants to be able to transmit many messages without having to wait for a response. Using this definition, RPC protocols are obviously synchronous protocols.

Although we have not discussed them in this chapter, there are interesting points between these two extremes. For example, the transport protocol might implement `send` so that it blocks (does not return) until the message has been successfully received at the remote machine, but returns before the sender's peer on that machine has actually processed and responded to it. This is sometimes called a *reliable datagram protocol*.

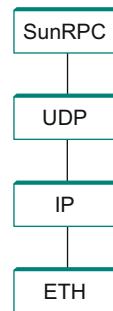
### 5.3.2 RPC Implementations (SunRPC, DCE)

We now turn our discussion to some example implementations of RPC protocols. These will serve to highlight some of the different design decisions that protocol designers have made. Our first example is SunRPC, a widely used RPC protocol also known as Open Network Computing RPC (ONC RPC). Our second example, which we will refer to as DCE-RPC, is part of the Distributed Computing Environment (DCE). DCE is a set of standards and software for building distributed systems that was defined by the Open Software Foundation (OSF), a consortium of computer companies that originally included IBM, Digital Equipment Corporation, and Hewlett-Packard; today, OSF goes by the name The Open Group. These two examples represent interesting alternative design choices in the RPC solution space.

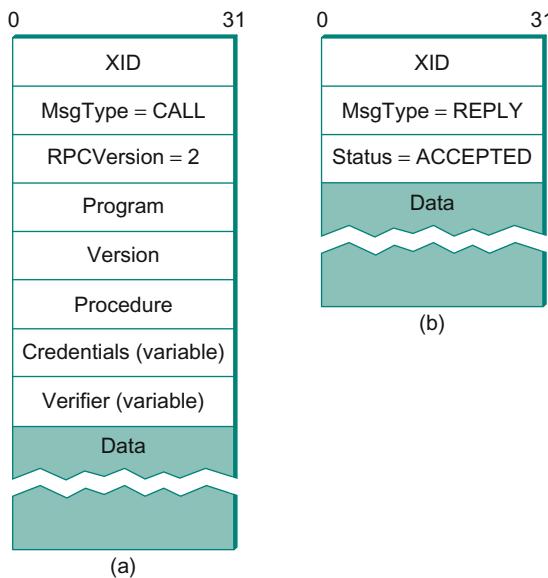
#### *SunRPC*

SunRPC became a *de facto* standard thanks to its wide distribution with Sun workstations and the central role it plays in Sun's popular Network File System (NFS). The IETF subsequently adopted it as a standard Internet protocol under the name ONC RPC.

SunRPC can be implemented over several different transport protocols. Figure 5.17 illustrates the protocol graph when SunRPC is implemented on UDP. As we noted earlier in this section, a strict layerist might frown on the idea of running a transport protocol over a transport protocol, or argue that RPC must be something other than a transport protocol since it appears “above” the transport layer. Pragmatically, the design decision to run RPC over an existing transport layer makes quite a lot of sense, as will be apparent in the following discussion.



■ FIGURE 5.17 Protocol graph for SunRPC on top of UDP.



■ FIGURE 5.18 SunRPC header formats: (a) request; (b) reply.

SunRPC uses two-tier identifiers to identify remote procedures: a 32-bit program number and a 32-bit procedure number. (There is also a 32-bit version number, but we ignore that in the following discussion.) For example, the NFS server has been assigned program number x00100003, and within this program getattr is procedure 1, setattr is procedure 2, read is procedure 6, write is procedure 8, and so on. The program number and procedure number are transmitted in the SunRPC request message's header, whose fields are shown in Figure 5.18. The server—which may support several program numbers—is responsible for calling the specified procedure of the specified program. A SunRPC request really represents a request to call the specified program and procedure on the particular machine to which the request was sent, even though the same program number may be implemented on other machines in the same network. Thus, the address of the server's machine (e.g., an IP address) is an implicit third tier of the RPC address.

Different program numbers may belong to different servers on the same machine. These different servers have different transport layer demux keys (e.g., UDP ports), most of which are not well-known numbers but instead are assigned dynamically. These demux keys are called

*transport selectors.* How can a SunRPC client that wants to talk to a particular program determine which transport selector to use to reach the corresponding server? The solution is to assign a well-known address to *just one* program on the remote machine and let that program handle the task of telling clients which transport selector to use to reach any other program on the machine. The original version of this SunRPC program is called the *Port Mapper*, and it supports only UDP and TCP as underlying protocols. Its program number is x00100000, and its well-known port is 111. RPCBIND, which evolved from the Port Mapper, supports arbitrary underlying transport protocols. As each SunRPC server starts, it calls an RPCBIND registration procedure, on the server's own home machine, to register its transport selector and the program numbers that it supports. A remote client can then call an RPCBIND lookup procedure to look up the transport selector for a particular program number.

To make this more concrete, consider an example using the Port Mapper with UDP. To send a request message to NFS's *read* procedure, a client first sends a request message to the Port Mapper at well-known UDP port 111, asking that procedure 3 be invoked to map program number x00100003 to the UDP port where the NFS program currently resides.<sup>2</sup> The client then sends a SunRPC request message with program number x00100003 and procedure number 6 to this UDP port, and the SunRPC module listening at that port calls the NFS *read* procedure. The client also caches the program-to-port number mapping so that it need not go back to the Port Mapper each time it wants to talk to the NFS program.

To match up a reply message with the corresponding request, so that the result of the RPC can be returned to the correct caller, both request and reply message headers include a XID (transaction ID) field, as in Figure 5.18. A XID is a unique transaction ID used only by one request and the corresponding reply. After the server has successfully replied to a given request, it does not remember the XID. Because of this, SunRPC cannot guarantee at-most-once semantics.

The details of SunRPC's semantics depend on the underlying transport protocol. It does not implement its own reliability, so it is only reliable if the underlying transport is reliable. (Of course, any application that runs over SunRPC may also choose to implement its own reliability

---

<sup>2</sup>In practice, NFS is such an important program that it has been given its own well-known UDP port, but for the purposes of illustration we're pretending that's not the case.

mechanisms above the level of SunRPC.) The ability to send request and reply messages that are larger than the network MTU is also dependent on the underlying transport. In other words, SunRPC does not make any attempt to improve on the underlying transport when it comes to reliability and message size. Since SunRPC can run over many different transport protocols, this gives it considerable flexibility without complicating the design of the RPC protocol itself.

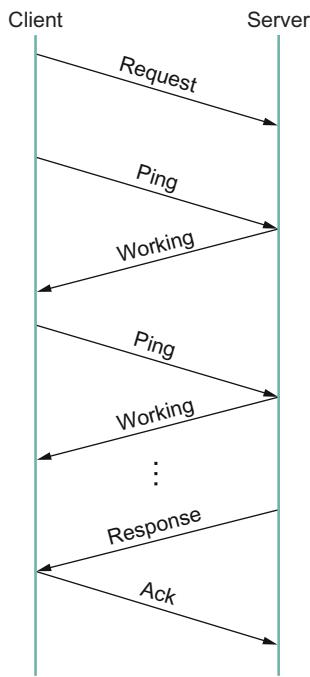
Returning to the SunRPC header format of [Figure 5.18](#), the request message contains variable-length **Credentials** and **Verifier** fields, both of which are used by the client to authenticate itself to the server—that is, to give evidence that the client has the right to invoke the server. How a client authenticates itself to a server is a general issue that must be addressed by any protocol that wants to provide a reasonable level of security. This topic is discussed in more detail in the next chapter.

#### DCE-RPC

DCE-RPC is the RPC protocol at the core of the DCE system and was the basis of the RPC mechanism underlying Microsoft’s DCOM and ActiveX. It can be used with the Network Data Representation (NDR) stub compiler described in [Chapter 7](#), but it also serves as the underlying RPC protocol for the Common Object Request Broker Architecture (CORBA), which is an industry-wide standard for building distributed, object-oriented systems.

DCE-RPC, like SunRPC, can be implemented on top of several transport protocols including UDP and TCP. It is also similar to SunRPC in that it defines a two-level addressing scheme: the transport protocol demultiplexes to the correct server, DCE-RPC dispatches to a particular procedure exported by that server, and clients consult an “endpoint mapping service” (similar to SunRPC’s Port Mapper) to learn how to reach a particular server. Unlike SunRPC, however, DCE-RPC implements at-most-once call semantics. (In truth, DCE-RPC supports multiple call semantics, including an idempotent semantics similar to SunRPC’s, but at-most-once is the default behavior.) There are some other differences between the two approaches, which we will highlight in the following paragraphs.

[Figure 5.19](#) gives a timeline for the typical exchange of messages, where each message is labeled by its DCE-RPC type. The client sends a Request message, the server eventually replies with a Response message, and the



■ FIGURE 5.19 Typical DCE-RPC message exchange.

client acknowledges (Ack) the response. Instead of the server acknowledging the request messages, however, the client periodically sends a Ping message to the server, which responds with a Working message to indicate that the remote procedure is still in progress. If the server's reply is received reasonably quickly, no Pings are sent. Although not shown in the figure, other message types are also supported. For example, the client can send a Quit message to the server, asking it to abort an earlier call that is still in progress; the server responds with a Quack (quit acknowledgment) message. Also, the server can respond to a Request message with a Reject message (indicating that a call has been rejected), and it can respond to a Ping message with a Nocall message (indicating that the server has never heard of the caller).

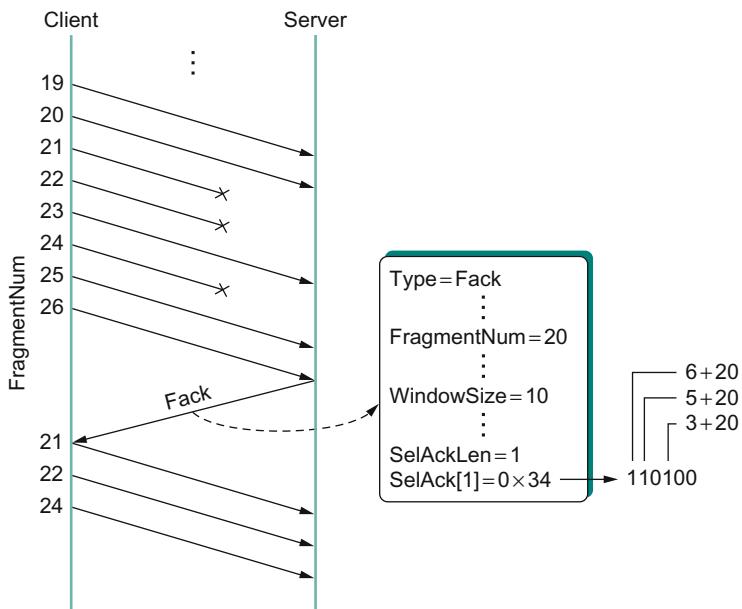
Each request/reply transaction in DCE-RPC takes place in the context of an *activity*. An activity is a logical request/reply channel between a pair of participants. At any given time, there can be only one message transaction active on a given channel. Like the concurrent logical channel approach described above and in Section 2.5.3, the application programs have to

open multiple channels if they want to have more than one request/reply transaction between them at the same time. The activity to which a message belongs is identified by the message's ActivityId field. A Sequence-Num field then distinguishes between calls made as part of the same activity; it serves the same purpose as SunRPC's XID (transaction id) field. Unlike SunRPC, DCE-RPC keeps track of the last sequence number used as part of a particular activity, so as to ensure at-most-once semantics. To distinguish between replies sent before and after a server machine reboots, DCE-RPC uses a ServerBoot field to hold the machine's boot ID.

Another design choice made in DCE-RPC that differs from SunRPC is the support of fragmentation and reassembly in the RPC protocol. As noted above, even if an underlying protocol such as IP provides fragmentation/reassembly, a more sophisticated algorithm implemented as part of RPC can result in quicker recovery and reduced bandwidth consumption when fragments are lost. The FragmentNum field uniquely identifies each fragment that makes up a given request or reply message. Each DCE-RPC fragment is assigned a unique fragment number (0, 1, 2, 3, and so on). Both the client and server implement a selective acknowledgment mechanism, which works as follows. (We describe the mechanism in terms of a client sending a fragmented request message to the server; the same mechanism applies when a server sends a fragment response to the client.)

First, each fragment that makes up the request message contains both a unique FragmentNum and a flag indicating whether this packet is a fragment of a call (`frag`) or the last fragment of a call (`last_frag`); request messages that fit in a single packet carry a `no_frag` flag. The server knows it has received the complete request message when it has the `last_frag` packet and there are no gaps in the fragment numbers. Second, in response to each arriving fragment, the server sends a Fack (fragment acknowledgment) message to the client. This acknowledgment identifies the highest fragment number that the server has successfully received. In other words, the acknowledgment is cumulative, much like in TCP. In addition, however, the server selectively acknowledges any higher fragment numbers it has received out of order. It does so with a bit vector that identifies these out-of-order fragments relative to the highest in-order fragment it has received. Finally, the client responds by retransmitting the missing fragments.

Figure 5.20 illustrates how this all works. Suppose the server has successfully received fragments up through number 20, plus fragments 23,



■ FIGURE 5.20 Fragmentation with selective acknowledgments.

25, and 26. The server responds with a Fack that identifies fragment 20 as the highest in-order fragment, plus a bit-vector (SelAck) with the third ( $23 = 20 + 3$ ), fifth ( $25 = 20 + 5$ ), and sixth ( $26 = 20 + 6$ ) bits turned on. So as to support an (almost) arbitrarily long bit vector, the size of the vector (measured in 32-bit words) is given in the SelAckLen field.

Given DCE-RPC's support for very large messages—the FragmentNum field is 16 bits long, meaning it can support 64K fragments—it is not appropriate for the protocol to blast all the fragments that make up a message as fast as it can since doing so might overrun the receiver. Instead, DCE-RPC implements a flow-control algorithm that is very similar to TCP's. Specifically, each Fack message not only acknowledges received fragments but also informs the sender of how many fragments it may now send. This is the purpose of the WindowSize field in Figure 5.20, which serves exactly the same purpose as TCP's AdvertisedWindow field except it counts fragments rather than bytes. DCE-RPC also implements a congestion-control mechanism that is similar to TCP's, which we will see in Chapter 6. Given the complexity of congestion control, it is perhaps not surprising that some RPC protocols avoid it by avoiding fragmentation.

In summary, designers have quite a range of options open to them when designing an RPC protocol. SunRPC takes the more minimalist approach and adds relatively little to the underlying transport beyond the essentials of locating the right procedure and identifying messages. DCE-RPC adds more functionality, with the possibility of improved performance in some environments at the cost of greater complexity.

## 5.4 TRANSPORT FOR REAL-TIME APPLICATIONS (RTP)

In the early days of packet switching, most applications were concerned with the movement of data: accessing remote computing resources, transferring files, sending email, etc. However, at least as early as 1981, experiments were under way to carry real-time traffic, such as digitized voice samples, over packet networks. We call an application “real-time” when it has strong requirements for the timely delivery of information. Internet telephony, or Voice over IP (VoIP), is a classic example of a real-time application, because you can’t easily carry on a conversation with someone if it takes more than a fraction of a second to get a response. As we will see shortly, real-time applications place some specific demands on the transport protocol that are not well met by the protocols discussed so far in this chapter.

Multimedia applications—those that involve video, audio, and data—are sometimes divided into two classes: *interactive* applications and *streaming* applications. An early and at one time popular example of the interactive class is vat, a multiparty audioconferencing tool that is often used over networks supporting IP multicast. The control panel for a typical vat conference is shown in Figure 5.21. Internet telephony is also a class of interactive application and probably the most widely used one at the time of writing. Internet-based multimedia conferencing applications, as mentioned in Chapter 1 and illustrated in Figure 1.1, provide another example. Modern instant messaging applications also use real-time audio and video. These are the sort of applications with the most stringent real-time requirements.

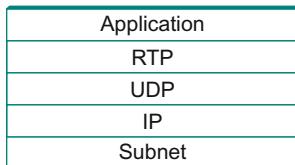
Streaming applications typically deliver audio or video streams from a server to a client and are typified by such commercial products as RealAudio®. Streaming video, typified by YouTube, has become one of the dominant forms of traffic on the Internet. Because streaming applications lack human-to-human interaction, they place somewhat less



■ FIGURE 5.21 User interface of a `vat` audioconference.

stringent real-time requirements on the underlying protocols. Timeliness is still important, however—for example, you want a video to start playing soon after pushing “play,” and once it starts to play, late packets will either cause it to stall or create some sort of visual degradation. So, while streaming applications are not strictly real time, they still have enough in common with interactive multimedia applications to warrant consideration of a common protocol for both types of application.

It should by now be apparent that designers of a transport protocol for real-time and multimedia applications face a real challenge in defining the requirements broadly enough to meet the needs of very different applications. They must also pay attention to the interactions among different applications, such as the synchronization of audio and video streams. We will see below how these concerns affected the design of the primary real-time transport protocol in use today, RTP.



■ FIGURE 5.22 Protocol stack for multimedia applications using RTP.

Much of RTP actually derives from protocol functionality that was originally embedded in the application itself. When the vat application was first developed, it ran over UDP, and the designers figured out which features were needed to handle the real-time nature of voice communication. Later, they realized that these features could be useful to many other applications and defined a protocol with those features, which became RTP. RTP can run over many lower-layer protocols, but still commonly runs over UDP. That leads to the protocol stack shown in Figure 5.22. Note that we are therefore running a transport protocol over a transport protocol. There is no rule against that,<sup>3</sup> and in fact it makes a lot of sense, since UDP provides such a minimal level of functionality, and the basic demultiplexing based on port numbers happens to be just what RTP needs as a starting point. So, rather than recreate port numbers in RTP, RTP outsources the demultiplexing function to UDP.

### 5.4.1 Requirements

The most basic requirement for a general-purpose multimedia protocol is that it allow similar applications to interoperate with each other. For example, it should be possible for two independently implemented audioconferencing applications to talk to each other. This immediately suggests that the applications had better use the same method of encoding and compressing voice; otherwise, the data sent by one party will be incomprehensible to the receiving party. Since there are quite a few different coding schemes for voice, each with its own trade-offs among quality, bandwidth requirements, and computational cost, it would probably be a bad idea to decree that only one such scheme can be used. Instead, our protocol should provide a way that a sender can tell a receiver which coding scheme it wants to use, and possibly negotiate until a scheme that is available to both parties is identified.

<sup>3</sup>But it has caused some confusion as to whether RTP is really a transport protocol.

Just as with audio, there are many different video coding schemes. Thus, we see that the first common function that RTP can provide is the ability to communicate that choice of coding scheme. Note that this also serves to identify the type of application (e.g., audio or video); once we know what coding algorithm is being used, we know what type of data is being encoded as well.

Another important requirement is to enable the recipient of a data stream to determine the timing relationship among the received data. Recall from Section 6.5 that real-time applications need to place received data into a *playback buffer* to smooth out the jitter that may have been introduced into the data stream during transmission across the network. Thus, some sort of timestamping of the data will be necessary to enable the receiver to play it back at the appropriate time.

Related to the timing of a single media stream is the issue of synchronization of multiple media in a conference. The obvious example of this would be to synchronize an audio and video stream that are originating from the same sender. As we will see below, this is a slightly more complex problem than playback time determination for a single stream.

Another important function to be provided is an indication of packet loss. Note that an application with tight latency bounds generally cannot use a reliable transport like TCP because retransmission of data to correct for loss would probably cause the packet to arrive too late to be useful. Thus, the application must be able to deal with missing packets, and the first step in dealing with them is noticing that they are in fact missing. As an example, a video application using MPEG encoding may take different actions when a packet is lost, depending on whether the packet came from an I frame, a B frame, or a P frame.

Packet loss is also a potential indicator of congestion. Since multimedia applications generally do not run over TCP, they also miss out on the congestion avoidance features of TCP (described in Section 6.3). Yet, many multimedia applications are capable of responding to congestion—for example, by changing the parameters of the coding algorithm to reduce the bandwidth consumed. Clearly, to make this work, the receiver needs to notify the sender that losses are occurring so that the sender can adjust its coding parameters.

Another common function across multimedia applications is the concept of frame boundary indication. A frame in this context is application

specific. For example, it may be helpful to notify a video application that a certain set of packets correspond to a single frame. In an audio application it is helpful to mark the beginning of a “talkspurt,” which is a collection of sounds or words followed by silence. The receiver can then identify the silences between talkspurts and use them as opportunities to move the playback point. This follows the observation that slight shortening or lengthening of the spaces between words are not perceptible to users, whereas shortening or lengthening the words themselves is both perceptible and annoying.

A final function that we might want to put into the protocol is some way of identifying senders that is more user-friendly than an IP address. As illustrated in Figure 5.21, audio and video conferencing applications can display strings such as Joe User (user@domain.com) on their control panels, and thus the application protocol should support the association of such a string with a data stream.

In addition to the functionality that is required from our protocol, we note an additional requirement: It should make reasonably efficient use of bandwidth. Put another way, we don’t want to introduce a lot of extra bits that need to be sent with every packet in the form of a long header. The reason for this is that audio packets, which are one of the most common types of multimedia data, tend to be small, so as to reduce the time it takes to fill them with samples. Long audio packets would mean high latency due to packetization, which has a negative effect on the perceived quality of conversations. (This was one of the factors in choosing the length of ATM cells.) Since the data packets themselves are short, a large header would mean that a relatively large amount of link bandwidth would be used by headers, thus reducing the available capacity for “useful” data. We will see several aspects of the design of RTP that have been influenced by the necessity of keeping the header short.

You could argue whether every single feature just described *really* needs to be in a real-time transport protocol, and you could probably find some more that could be added. The key idea here is to make life easier for application developers by giving them a useful set of abstractions and building blocks for their applications. For example, by putting a timestamping mechanism into RTP, we save every developer of a real-time application from inventing his own. We also increase the chances that two different real-time applications might interoperate.

### 5.4.2 RTP Design

Now that we have seen the rather long list of requirements for our transport protocol for multimedia, we turn to the details of the protocol that has been specified to meet those requirements. This protocol, RTP, was developed in the IETF and is in widespread use. The RTP standard actually defines a pair of protocols, RTP and the Real-time Transport Control Protocol (RTCP). The former is used for the exchange of multimedia data, while the latter is used to periodically send control information associated with a certain data flow. When running over UDP, the RTP data stream and the associated RTCP control stream use consecutive transport-layer ports. The RTP data uses an even port number and the RTCP control information uses the next higher (odd) port number.

Because RTP is designed to support a wide variety of applications, it provides a flexible mechanism by which new applications can be developed without repeatedly revising the RTP protocol itself. For each class of application (e.g., audio), RTP defines a *profile* and one or more *formats*. The profile provides a range of information that ensures a common understanding of the fields in the RTP header for that application class, as will be apparent when we examine the header in detail. The format specification explains how the data that follows the RTP header is to be interpreted. For example, the RTP header might just be followed by a sequence of bytes, each of which represents a single audio sample taken a defined interval after the previous one. Alternatively, the format of the data might be much more complex; an MPEG-encoded video stream, for example, would need to have a good deal of structure to represent all the different types of information.



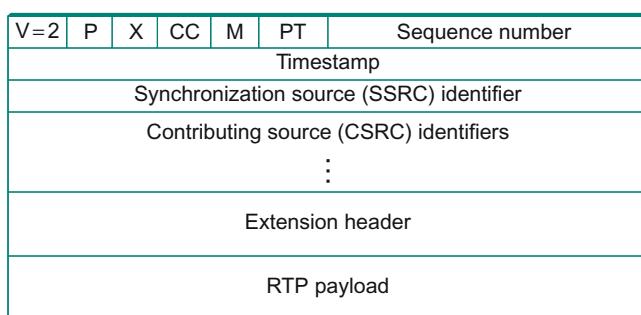
The design of RTP embodies an architectural principle known as *Application Level Framing* (ALF). This principle was put forward by Clark and Tennenhouse in 1990 as a new way to design protocols for emerging multimedia applications. They recognized that these new applications were unlikely to be well served by existing protocols such as TCP, and that furthermore they might not be well served by any sort of “one-size-fits-all” protocol. At the heart of this principle is the belief that an application understands its own needs best. For example, an MPEG video application knows how best to recover from lost frames and how to react differently if an I frame or a B frame is lost. The same application also understands best how to segment the data for transmission—for example, it’s better to send the data from different frames in different datagrams, so that a lost packet only corrupts a single frame, not two. It is for this reason that RTP leaves so many of the protocol details to the profile and format documents that are specific to an application.

### Header Format

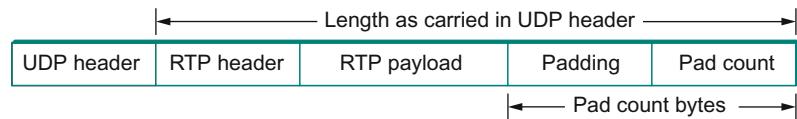
Figure 5.23 shows the header format used by RTP. The first 12 bytes are always present, whereas the contributing source identifiers are only used in certain circumstances. After this header there may be optional header extensions, as described below. Finally, the header is followed by the RTP payload, the format of which is determined by the application. The intention of this header is that it contain only the fields that are likely to be used by many different applications, since anything that is very specific to a single application would be more efficiently carried in the RTP payload for that application only.

The first two bits are a version identifier, which contains the value 2 in the RTP version deployed at the time of writing. You might think that the designers of the protocol were rather bold to think that 2 bits would be enough to contain all future versions of RTP, but recall that bits are at a premium in the RTP header. Furthermore, the use of profiles for different applications makes it less likely that many revisions to the base RTP protocol would be needed. In any case, if it turns out that another version of RTP is needed beyond version 2, it would be possible to consider a change to the header format so that more than one future version would be possible. For example, a new RTP header with the value 3 in the version field could have a “subversion” field somewhere else in the header.

The next bit is the *padding* (P) bit, which is set in circumstances in which the RTP payload has been padded for some reason. RTP data might be padded to fill up a block of a certain size as required by an encryption algorithm, for example. In such a case, the complete length of the RTP header, data, and padding would be conveyed by the lower-layer protocol



■ FIGURE 5.23 RTP header format.



■ FIGURE 5.24 Padding of an RTP packet.

header (e.g., the UDP header), and the last byte of the padding would contain a count of how many bytes should be ignored. This is illustrated in Figure 5.24. Note that this approach to padding removes any need for a length field in the RTP header (thus serving the goal of keeping the header short); in the common case of no padding, the length is deduced from the lower-layer protocol.

The *extension* (X) bit is used to indicate the presence of an extension header, which would be defined for a specific application and follow the main header. Such headers are rarely used, since it is generally possible to define a payload-specific header as part of the payload format definition for a particular application.

The X bit is followed by a 4-bit field that counts the number of *contributing sources*, if any are included in the header. Contributing sources are discussed below.

We noted above the frequent need for some sort of frame indication; this is provided by the marker bit, which has a profile-specific use. For a voice application, it could be set at the beginning of a talkspurt, for example. The 7-bit payload type field follows; it indicates what type of multimedia data is carried in this packet. One possible use of this field would be to enable an application to switch from one coding scheme to another based on information about resource availability in the network or feedback on application quality. The exact usage of the payload type is also determined by the application profile.

Note that the payload type is generally not used as a demultiplexing key to direct data to different applications (or to different streams within a single application, such as the audio and video stream for a videoconference). This is because such demultiplexing is typically provided at a lower layer (e.g., by UDP, as described in Section 5.1). Thus, two media streams using RTP would typically use different UDP port numbers.

The sequence number is used to enable the receiver of an RTP stream to detect missing and misordered packets. The sender simply increments the value by one for each transmitted packet. Note that RTP does not do

anything when it detects a lost packet, in contrast to TCP, which both corrects for the loss (by retransmission) and interprets the loss as a congestion indication (which may cause it to reduce its window size). Rather, it is left to the application to decide what to do when a packet is lost because this decision is likely to be highly application dependent. For example, a video application might decide that the best thing to do when a packet is lost is to replay the last frame that was correctly received. Some applications might also decide to modify their coding algorithms to reduce bandwidth needs in response to loss, but this is not a function of RTP. It would not be sensible for RTP to decide that the sending rate should be reduced, as this might make the application useless.

The function of the timestamp field is to enable the receiver to play back samples at the appropriate intervals and to enable different media streams to be synchronized. Because different applications may require different granularities of timing, RTP itself does not specify the units in which time is measured. Instead, the timestamp is just a counter of “ticks,” where the time between ticks is dependent on the encoding in use. For example, an audio application that samples data once every 125 μs could use that value as its clock resolution. The clock granularity is one of the details that is specified in the RTP profile or payload format for an application.

The timestamp value in the packet is a number representing the time at which the *first* sample in the packet was generated. The timestamp is not a reflection of the time of day; only the differences between timestamps are relevant. For example, if the sampling interval is 125 μs and the first sample in packet  $n + 1$  was generated 10 ms after the first sample in packet  $n$ , then the number of sampling instants between these two samples is

$$\begin{aligned} \text{TimeBetweenPackets} \div \text{TimePerSample} &= (10 \times 10^{-3}) \div (125 \times 10^{-6}) \\ &= 80 \end{aligned}$$

Assuming the clock granularity is the same as the sampling interval, then the timestamp in packet  $n + 1$  would be greater than that in packet  $n$  by 80. Note that fewer than 80 samples might have been sent due to compression techniques such as silence detection, and yet the timestamp allows the receiver to play back the samples with the correct temporal relationship.

The synchronization source (SSRC) is a 32-bit number that uniquely identifies a single source of an RTP stream. In a given multimedia conference, each sender picks a random SSRC and is expected to resolve conflicts in the unlikely event that two sources pick the same value. By

making the source identifier something other than the network or transport address of the source, RTP ensures independence from the lower-layer protocol. It also enables a single node with multiple sources (e.g., several cameras) to distinguish those sources. When a single node generates different media streams (e.g., audio and video), it is not required to use the same SSRC in each stream, as there are mechanisms in RTCP (described below) to allow intermedia synchronization.

The contributing source (CSRC) is used only when a number of RTP streams pass through a mixer. A mixer can be used to reduce the bandwidth requirements for a conference by receiving data from many sources and sending it as a single stream. For example, the audio streams from several concurrent speakers could be decoded and recoded as a single audio stream. In this case, the mixer lists itself as the synchronization source but also lists the contributing sources—the SSRC values of the speakers who contributed to the packet in question.

### 5.4.3 Control Protocol

RTCP provides a control stream that is associated with a data stream for a multimedia application. This control stream provides three main functions:

1. Feedback on the performance of the application and the network
2. A way to correlate and synchronize different media streams that have come from the same sender
3. A way to convey the identity of a sender for display on a user interface (e.g., the vat interface shown in [Figure 5.21](#))

The first function may be useful for detecting and responding to congestion. Some applications are able to operate at different rates and may use performance data to decide to use a more aggressive compression scheme to reduce congestion, for example, or to send a higher-quality stream when there is little congestion. Performance feedback can also be useful in diagnosing network problems.

You might think that the second function is already provided by the synchronization source ID (SSRC) of RTP, but in fact it is not. As already noted, multiple cameras from a single node might have different SSRC values. Furthermore, there is no requirement that an audio and video stream from the same node use the same SSRC. Because collisions of SSRC values may occur, it may be necessary to change the SSRC value of

a stream. To deal with this problem, RTCP uses the concept of a *canonical name* (CNAME) that is assigned to a sender, which is then associated with the various SSRC values that might be used by that sender using RTCP mechanisms.

Simply correlating two streams is only part of the problem of intermedia synchronization. Because different streams may have completely different clocks (with different granularities and even different amounts of inaccuracy, or drift), there needs to be a way to accurately synchronize streams with each other. RTCP addresses this problem by conveying timing information that correlates actual time of day with the clock-rate-dependent timestamps that are carried in RTP data packets.

RTCP defines a number of different packet types, including

- Sender reports, which enable active senders to a session to report transmission and reception statistics
- Receiver reports, which receivers who are not senders use to report reception statistics
- Source descriptions, which carry CNAMEs and other sender description information
- Application-specific control packets

These different RTCP packet types are sent over the lower-layer protocol, which, as we have noted, is typically UDP. Several RTCP packets can be packed into a single PDU of the lower-level protocol. It is required that at least two RTCP packets are sent in every lower-level PDU: One of these is a report packet; the other is a source description packet. Other packets may be included up to the size limits imposed by the lower-layer protocols.

Before looking further at the contents of an RTCP packet, we note that there is a potential problem with every member of a multicast group sending periodic control traffic. Unless we take some steps to limit it, this control traffic has the potential to be a significant consumer of bandwidth. In an audioconference, for example, no more than two or three senders are likely to send audio data at any instant, since there is no point in everyone talking at once. But there is no such social limit on everyone sending control traffic, and this could be a severe problem in a conference with thousands of participants. To deal with this problem, RTCP has a set of mechanisms by which the participants scale back their reporting frequency as the number of participants increases. These rules

are somewhat complex, but the basic goal is this: Limit the total amount of RTCP traffic to a small percentage (typically 5%) of the RTP data traffic. To accomplish this goal, the participants should know how much data bandwidth is likely to be in use (e.g., the amount to send three audio streams) and the number of participants. They learn the former from means outside RTP (known as *session management*, discussed at the end of this section), and they learn the latter from the RTCP reports of other participants. Because RTCP reports might be sent at a very low rate, it might only be possible to get an approximate count of the current number of recipients, but that is typically sufficient. Also, it is recommended to allocate more RTCP bandwidth to active senders, on the assumption that most participants would like to see reports from them—for example, to find out who is speaking.

Once a participant has determined how much bandwidth it can consume with RTCP traffic, it sets about sending periodic reports at the appropriate rate. Sender reports and receiver reports differ only in that the former include some extra information about the sender. Both types of reports contain information about the data that was received from all sources in the most recent reporting period.

The extra information in a sender report consists of

- A timestamp containing the actual time of day when this report was generated
- The RTP timestamp corresponding to the time when the report was generated
- Cumulative counts of the packets and bytes sent by this sender since it began transmission

Note that the first two quantities can be used to enable synchronization of different media streams from the same source, even if those streams use different clock granularities in their RTP data streams, since it gives the key to convert time of day to the RTP timestamps.

Both sender and receiver reports contain one block of data per source that has been heard from since the last report. Each block contains the following statistics for the source in question:

- Its SSRC
- The fraction of data packets from this source that were lost since the last report was sent (calculated by comparing the number of

- packets received with the number of packets expected; this last value can be determined from the RTP sequence numbers)
- Total number of packets lost from this source since the first time it was heard from
- Highest sequence number received from this source (extended to 32 bits to account for wrapping of the sequence number)
- Estimated interarrival jitter for the source (calculated by comparing the interarrival spacing of received packets with the expected spacing at transmission time)
- Last actual timestamp received via RTCP for this source
- Delay since last sender report received via RTCP for this source

As you might imagine, the recipients of this information can learn all sorts of things about the state of the session. In particular, they can see if other recipients are getting much better quality from some sender than they are, which might be an indication that a resource reservation needs to be made, or that there is a problem in the network that needs to be attended to. In addition, if a sender notices that many receivers are experiencing high loss of its packets, it might decide that it should reduce its sending rate or use a coding scheme that is more resilient to loss.

The final aspect of RTCP that we will consider is the source description packet. Such a packet contains, at a minimum, the SSRC of the sender and the sender's CNAME. The canonical name is derived in such a way that all applications that generate media streams that might need to be synchronized (e.g., separately generated audio and video streams from the same user) will choose the same CNAME even though they might choose different SSRC values. This enables a receiver to identify the media stream that came from the same sender. The most common format of the CNAME is `user@host`, where `host` is the fully qualified domain name of the sending machine. Thus, an application launched by the user whose user name is `jdoe` running on the machine `cicada.cs.princeton.edu` would use the string `jdoe@cicada.cs.princeton.edu` as its CNAME. The large and variable number of bytes used in this representation would make it a bad choice for the format of an SSRC, since the SSRC is sent with every data packet and must be processed in real time. Allowing CNAMEs to be bound to SSRC values in periodic RTCP messages enables a compact and efficient format for the SSRC.

Other items may be included in the source description packet, such as the real name and email address of the user. These are used in user interface displays and to contact participants, but are less essential to the operation of RTP than the CNAME.

Like TCP, RTP and RTCP are a fairly complex pair of protocols. This complexity comes in large part from the desire to make life easier for application designers. Because there is an infinite number of possible applications, the challenge in designing a transport protocol is to make it general enough to meet the widely varying needs of many different applications without making the protocol itself impossible to implement. RTP has proven very successful in this regard, forming the basis for the majority of real-time multimedia communications over the Internet today.

## 5.5 SUMMARY

This chapter has described four very different end-to-end protocols. The first protocol we considered is a simple demultiplexer, as typified by UDP. All such a protocol does is dispatch messages to the appropriate application process based on a port number. It does not enhance the best-effort service model of the underlying network in any way—it simply offers an unreliable, connectionless datagram service to application programs.

The second type is a reliable byte-stream protocol, and the specific example of this type that we looked at is TCP. The challenges faced with such a protocol are to recover from messages that may be lost by the network, to deliver messages in the same order in which they are sent, and to allow the receiver to do flow control on the sender. TCP uses the basic sliding window algorithm, enhanced with an advertised window, to implement this functionality. The other item of note for this protocol is the importance of an accurate timeout/retransmission mechanism. Interestingly, even though TCP is a single protocol, we saw that it employs at least five different algorithms—sliding window, Nagle, three-way handshake, Karn/Partridge, and Jacobson/Karels—all of which can be of value to any end-to-end protocol.

The third type of transport protocol we looked at is request/reply protocols that form the basis for RPC. Such protocols must dispatch requests to the correct remote procedures and match replies to the corresponding

requests. They may additionally provide reliability, such as at-most-once semantics, or support large message sizes by message fragmentation and reassembly.

Finally, we looked at transport protocols for the class of applications that involve multimedia data (such as audio and video) and that have a requirement for real-time delivery. Such a transport protocol needs to provide help with recovering the timing information of a single media stream and with synchronizing multiple media streams. It also needs to provide information to upper layers (e.g., the application layer) about lost data (since there will normally not be enough time to retransmit lost packets) so that appropriate application-specific recovery and congestion avoidance methods can be employed. The protocol that has been developed to meet these needs is RTP, which includes a companion control protocol called RTCP.

**W**hat should be clear after reading this chapter is that transport protocol design is a tricky business. As we have seen, getting a transport protocol right in the first place is hard enough, but changing circumstances make matters more complicated. Some changes are not too hard to predict—networks keep getting faster, for example—but sometimes a new class of applications appears and changes the requirements for transport-level services. The challenge is finding ways to adapt to these changes. Sometimes existing protocols can be tweaked to deal with new circumstances—TCP options have evolved over time to achieve this—at other times it may be necessary to create a new transport protocol, as with RTP and

### WHAT'S NEXT: TRANSPORT PROTOCOL DIVERSITY

SCTP. It is an open question as to when we should reuse an existing protocol versus create a new one.

In addition to making small changes to an existing protocol to accommodate changing network environments or new application needs, sometimes we might be able to tweak the application to get it to work with an existing protocol. For example, an application that needs RPC semantics would probably do best with an RPC protocol, but with a bit of work and some loss of efficiency it can use TCP. We see this happening a lot today when web-based applications need some sort of RPC. TCP is ubiquitous on the

Web and known to work in all sorts of challenging environments, such as through NAT devices and firewalls. Consequently, it turns out to be simpler sometimes to build RPC-like mechanisms that run on top of TCP (and HTTP in many cases) even if that's not strictly the most efficient approach.

Similarly, it seems that RTP should be the protocol of choice for streaming of video, and it continues to be widely used, but there is also quite a trend toward the use of TCP (and HTTP) to deliver streaming video content. This approach is increasingly popular for delivery of entertainment video such as TV shows and movies for web-browser-based viewing. It is too early to tell how this will end up, but it is not hard to imagine a future in which TCP is the dominant transport protocol for video delivery.

Looking back on the “hourglass” architecture of the Internet that we discussed in Chapter 1, it’s clear that some diversity of transport protocols was expected by the Internet’s designers. But, at this point in the Internet’s history, it almost looks like TCP might itself become part of the waist of the hourglass, providing the ubiquitous service on which a majority of applications will depend.

---

## ■ FURTHER READING

There is no doubt that TCP is a complex protocol and that in fact it has subtleties not illuminated in this chapter; therefore, the recommended reading list for this chapter includes the original TCP specification. Our motivation for including this specification is not so much to fill in the missing details as to expose you to what an honest-to-goodness protocol specification looks like. The next paper, by Birrell and Nelson, is the seminal paper on RPC. Third, the paper by Clark and Tennenhouse on protocol architecture introduced the concept of *Application Layer Framing* which inspired the design of RTP; this paper provides considerable insight into the challenges of designing protocols as application needs change.

- USC-ISI. Transmission Control Protocol. *Request for Comments* 793, September 1981.
- Birrell, A., and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems* 2(1):39–59, February 1984.
- Clark, D., and D. Tennenhouse. Architectural considerations for a new generation of protocols. *Proceedings of the SIGCOMM ’90 Symposium*, pages 200–208, September 1990.

Beyond the protocol specification, the most complete description of TCP, including its implementation in Unix, can be found in Stevens [Ste94],

[SW95]. Also, the third volume of Comer and Stevens' TCP/IP series of books describes how to write client/server applications on top of TCP and UDP, using the Posix socket interface [CS00], the Windows socket interface [CS97], and the BSD socket interface [CS96]. SCTP, a reliable transport protocol that stakes out a different point in the design space than TCP, is described in a helpful overview [OY02] and specified in [Ste07].

Several papers evaluate the performance of different transport protocols at a very detailed level. For example, the article by Clark et al. [CJRS89] measures the processing overheads of TCP, a paper by Mosberger et al. [MPBO96] explores the limitations of protocol processing overheads, and Thekkath and Levy [TL93] and Schroeder and Burrows [SB89] examine RPC's performance in great detail.

The original TCP timeout calculation was described in the TCP specification (see above), while the Karn/Partridge algorithm was described in [KP91] and the Jacobson/Karels algorithm was proposed in [Jac88]. The TCP extensions are defined by Jacobson et al. [JBB92], while O'Malley and Peterson [OP91] argue that extending TCP in this way is not the right approach to solving the problem.

Several distributed operating systems have defined their own RPC protocols. Notable examples include the V system, described by Cheriton and Zwaenepoel [CZ85]; Sprite, described by Ousterhout et al. [OCD<sup>+</sup>88]; and Amoeba, described by Mullender [Mul90].

RTP is described in RFC 3550 [SCFJ03], and there are numerous other RFCs (such as RFC 3551 [SC03]) that describe the profiles of various applications that use RTP. McCanne and Jacobson [MJ95] describe vic, one of the early video applications to use RTP.

Finally, the following live reference provides lots of information related to transport and other protocols:

- <http://www.iana.org/protocols/>: Information about all the options, constants, port numbers, etc., that have been defined for various Internet protocols. You can find here, among other things, the lists of TCP header flags, TCP options, and well-known port numbers for protocols that run over TCP and UDP.

## EXERCISES

1. If a UDP datagram is sent from host A, port P to host B, port Q, but at host B there is no process listening to port Q, then B is to

send back an ICMP Port Unreachable message to A. Like all ICMP messages, this is addressed to A as a whole, not to port P on A.

- (a) Give an example of when an application might want to receive such ICMP messages.
  - (b) Find out what an application has to do, on the operating system of your choice, to receive such messages.
  - (c) Why might it not be a good idea to send such messages directly back to the originating port P on A?
2. Consider a simple UDP-based protocol for requesting files (based somewhat loosely on the Trivial File Transport Protocol, or TFTP). The client sends an initial file request, and the server answers (if the file can be sent) with the first data packet. Client and server then continue with a stop-and-wait transmission mechanism.
- (a) Describe a scenario by which a client might request one file but get another; you may allow the client application to exit abruptly and be restarted with the same port.
  - (b) Propose a change in the protocol that will make this situation much less likely.
3. Design a simple UDP-based protocol for retrieving files from a server. No authentication is to be provided. Stop-and-wait transmission of the data may be used. Your protocol should address the following issues:
- (a) Duplication of the first packet should not duplicate the “connection.”
  - (b) Loss of the final ACK should not necessarily leave the server in doubt as to whether the transfer succeeded.
  - (c) A late-arriving packet from a past connection shouldn’t be interpretable as part of a current connection.
4. This chapter explains three sequences of state transitions during TCP connection teardown. There is a fourth possible sequence, which traverses an additional arc (not shown in Figure 5.7) from FIN\_WAIT\_1 to TIME\_WAIT and labelled FIN + ACK/ACK. Explain the circumstances that result in this fourth teardown sequence.
5. When closing a TCP connection, why is the two-segment-lifetime timeout not necessary on the transition from LAST\_ACK to CLOSED?

6. A sender on a TCP connection that receives a 0 advertised window periodically probes the receiver to discover when the window becomes nonzero. Why would the receiver need an extra timer if it were responsible for reporting that its advertised window had become nonzero (i.e., if the sender did not probe)?
7. Read the man page (or Windows equivalent) for the Unix/Windows utility netstat. Use netstat to see the state of the local TCP connections. Find out how long closing connections spend in TIME\_WAIT.
8. The sequence number field in the TCP header is 32 bits long, which is big enough to cover over 4 billion bytes of data. Even if this many bytes were never transferred over a single connection, why might the sequence number still wrap around from  $2^{32} - 1$  to 0?
9. You are hired to design a reliable byte-stream protocol that uses a sliding window (like TCP). This protocol will run over a 1-Gbps network. The RTT of the network is 100 ms, and the maximum segment lifetime is 30 seconds.
  - (a) How many bits would you include in the AdvertisedWindow and SequenceNum fields of your protocol header?
  - (b) How would you determine the numbers given above, and which values might be less certain?
10. You are hired to design a reliable byte-stream protocol that uses a sliding window (like TCP). This protocol will run over a 1-Gbps network. The RTT of the network is 140 ms, and the maximum segment lifetime is 60 seconds. How many bits would you include in the AdvertisedWindow and SequenceNum fields of your protocol header?
11. Suppose a host wants to establish the reliability of a link by sending packets and measuring the percentage that is received; routers, for example, do this. Explain the difficulty doing this over a TCP connection.
12. Suppose TCP operates over a 1-Gbps link.
  - (a) Assuming TCP could utilize the full bandwidth continuously, how long would it take the sequence numbers to wrap around completely?

- (b) Suppose an added 32-bit timestamp field increments 1000 times during the wraparound time you found above. How long would it take for the timestamp to wrap around?

- ✓ 13. Suppose TCP operates over a 40-Gbps STS-768 link.
- (a) Assuming TCP could utilize the full bandwidth continuously, how long would it take the sequence numbers to wrap around completely?
  - (b) Suppose an added 32-bit timestamp field which increments 1000 times during the wraparound time you found above. How long would it take for the timestamp to wrap around?
14. If host A receives two SYN packets from the same port from remote host B, the second may be either a retransmission of the original or, if B has crashed and rebooted, an entirely new connection request.
- (a) Describe the difference as seen by host A between these two cases.
  - (b) Give an algorithmic description of what the TCP layer needs to do upon receiving a SYN packet. Consider the duplicate/ new cases above and the possibility that nothing is listening to the destination port.
15. Suppose  $x$  and  $y$  are two TCP sequence numbers. Write a function to determine whether  $x$  comes before  $y$  (in the notation of *Request for Comments* 793, “ $x = < y$ ”) or after  $y$ ; your solution should work even when sequence numbers wrap around.
16. Suppose an idle TCP connection exists between sockets A and B. A third party has eavesdropped and knows the current sequence number at both ends.
- (a) Suppose the third party sends A a forged packet ostensibly from B and with 100 bytes of new data. What happens? (Hint: Look up in *Request for Comments* 793 what TCP does when it receives an ACK that is not an “acceptable ACK.”)
  - (b) Suppose the third party sends each end such a forged 100-byte data packet ostensibly from the other end. What happens now? What would happen if A later sent 200 bytes of data to B?
17. Suppose party A connects to the Internet via a wireless network using DHCP to assign IP addresses. A opens several Telnet

connections (using TCP) and is then disconnected from the wireless network. Party B then connects and is assigned the same IP address that A had had. Assuming B were able to guess to what host(s) A had been connected, describe a sequence of probes that could enable B to obtain sufficient state information to continue with A's connections.

18. Diagnostic programs are commonly available that record the first 100 bytes, say, of every TCP connection to a certain  $\langle \text{host}, \text{port} \rangle$ . Outline what must be done with each received TCP packet, P, in order to determine if it contains data that belongs to the first 100 bytes of a connection to host HOST, port PORT. Assume the IP header is P.IPHEAD, the TCP header is P.TCPHEAD, and header fields are as named in Figures 3.16 and 5.4. (Hint: To get initial sequence numbers (ISNs) you will have to examine every packet with the SYN bit set. Ignore the fact that sequence numbers will eventually be reused.)
19. If a packet arrives at host A with B's source address, it could just as easily have been forged by any third host C. If, however, A accepts a TCP connection from B, then during the three-way handshake A sent  $\text{ISN}_A$  to B's address and received an acknowledgment of it. If C is not located so as to be able to eavesdrop on  $\text{ISN}_A$ , then it might seem that C could not have forged B's response.

However, the algorithm for choosing  $\text{ISN}_A$  does give other unrelated hosts a fair chance of guessing it. Specifically, A selects  $\text{ISN}_A$  based on a clock value at the time of connection. *Request for Comments* 793 specifies that this clock value be incremented every  $4 \mu\text{s}$ ; common Berkeley implementations once simplified this to incrementing by 250,000 (or 256,000) once per second.

- (a) Given this simplified increment-once-per-second implementation, explain how an arbitrary host C could masquerade as B in at least the opening of a TCP connection. You may assume that B does not respond to SYN + ACK packets A is tricked into sending to it.
- (b) Assuming real RTTs can be estimated to within 40 ms, about how many tries would you expect it to take to implement the strategy of part (a) with the unsimplified “increment every  $4 \mu\text{s}$ ” TCP implementation?

20. The Nagle algorithm, built into most TCP implementations, requires the sender to hold a partial segment's worth of data (even if PUSHed) until either a full segment accumulates or the most recent outstanding ACK arrives.
  - (a) Suppose the letters abcdefghi are sent, one per second, over a TCP connection with an RTT of 4.1 seconds. Draw a timeline indicating when each packet is sent and what it contains.
  - (b) If the above were typed over a full-duplex Telnet connection, what would the user see?
  - (c) Suppose that mouse position changes are being sent over the connection. Assuming that multiple position changes are sent each RTT, how would a user perceive the mouse motion with and without the Nagle algorithm?
21. Suppose a client C repeatedly connects via TCP to a given port on a server S, and that each time it is C that initiates the close.
  - (a) How many TCP connections a second can C make here before it ties up all its available ports in TIME\_WAIT state? Assume client ephemeral ports are in the range of 1024 to 5119, and that TIME\_WAIT lasts 60 seconds.
  - (b) Berkeley-derived TCP implementations typically allow a socket in TIME\_WAIT state to be reopened before TIME\_WAIT expires, if the highest sequence number used by the old incarnation of the connection is less than the ISN used by the new incarnation. This solves the problem of old data being accepted as new; however, TIME\_WAIT also serves the purpose of handling late final FINs. What would such an implementation have to do to address this and still achieve strict compliance with the TCP requirement that a FIN sent anytime before or during a connection's TIME\_WAIT receive the same response?
22. Explain why TIME\_WAIT is a somewhat more serious problem if the server initiates the close than if the client does. Describe a situation in which this might reasonably happen.
23. What is the justification for the exponential increase in timeout value proposed by Karn and Partridge? Why, specifically, might a linear (or slower) increase be less desirable?

- ★ 24. The Jacobson/Karels algorithm sets TimeOut to be 4 mean deviations above the mean. Assume that individual packet round-trip times follow a statistical normal distribution, for which 4 mean deviations are  $\pi$  standard deviations. Using statistical tables, for example, what is the probability that a packet will take more than TimeOut time to arrive?
25. Suppose a TCP connection, with window size 1, loses every other packet. Those that do arrive have RTT = 1 second. What happens? What happens to TimeOut? Do this for two cases:
- After a packet is eventually received, we pick up where we left off, resuming with EstimatedRTT initialized to its pre-timeout value, and TimeOut double that.
  - After a packet is eventually received, we resume with TimeOut initialized to the last exponentially backed-off value used for the timeout interval.
- In the following four exercises, the calculations involved are straightforward with a spreadsheet.
26. Suppose, in TCP's adaptive retransmission mechanism, that EstimatedRTT is 4.0 seconds at some point and subsequent measured RTT's all are 1.0 second. How long does it take before the TimeOut value, as calculated by the Jacobson/Karels algorithm, falls below 4.0 seconds? Assume a plausible initial value of Deviation; how sensitive is your answer to this choice? Use  $\delta = 1/8$ .
- ✓ 27. Suppose, in TCP's adaptive retransmission mechanism, that EstimatedRTT is 90 at some point and subsequent measured RTTs are all 200. How long does it take before the TimeOut value, as calculated by the Jacobson/Karels algorithm, falls below 300? Assume initial Deviation value of 25; use  $\delta = 1/8$ .
28. Suppose TCP's measured RTT is 1.0 second except that every  $N$ th RTT is 4.0 seconds. What is the largest  $N$ , approximately, that doesn't result in timeouts in the steady state (i.e., for which the Jacobson/Karels TimeOut remains greater than 4.0 seconds)? Use  $\delta = 1/8$ .
29. Suppose that TCP is measuring RTTs of 1.0 second, with a mean deviation of 0.1 second. Suddenly the RTT jumps to 5.0 seconds,

with no deviation. Compare the behaviors of the original and Jacobson/Karels algorithms for computing TimeOut. Specifically, how many timeouts are encountered with each algorithm? What is the largest TimeOut calculated? Use  $\delta = 1/8$ .

30. Suppose that, when a TCP segment is sent more than once, we take SampleRTT to be the time between the original transmission and the ACK, as in Figure 5.10(a). Show that if a connection with a 1-packet window loses every other packet (i.e., each packet is transmitted twice), then EstimatedRTT increases to infinity. Assume TimeOut = EstimatedRTT; both algorithms presented in the text always set TimeOut even larger. (Hint: EstimatedRTT = EstimatedRTT +  $\beta \times (\text{SampleRTT} - \text{EstimatedRTT})$ .)
31. Suppose that, when a TCP segment is sent more than once, we take SampleRTT to be the time between the most recent transmission and the ACK, as in Figure 5.10(b). Assume, for definiteness, that TimeOut =  $2 \times \text{EstimatedRTT}$ . Sketch a scenario in which no packets are lost but EstimatedRTT converges to a third of the true RTT, and give a diagram illustrating the final steady state. (Hint: Begin with a sudden jump in the true RTT to just over the established TimeOut.)
32. Consult *Request for Comments* 793 to find out how TCP is supposed to respond if a FIN or an RST arrives with a sequence number other than NextByteExpected. Consider both when the sequence number is within the receive window and when it is not.
33. One of the purposes of TIME\_WAIT is to handle the case of a data packet from a first incarnation of a connection arriving very late and being accepted as data for the second incarnation.
  - (a) Explain why, for this to happen (in the absence of TIME\_WAIT), the hosts involved would have to exchange several packets in sequence *after* the delayed packet was sent but before it was delivered.
  - (b) Propose a network scenario that might account for such a late delivery.
34. Propose an extension to TCP by which one end of a connection can hand off its end to a third host; that is, if A were connected to B, and A handed off its connection to C, then afterwards C would be connected to B and A would not. Specify the new states and

transitions needed in the TCP state-transition diagram and any new packet types involved. You may assume all parties will understand this new option. What state should A go into immediately after the handoff?

35. TCP's simultaneous open feature is seldom used.
  - (a) Propose a change to TCP in which this is disallowed. Indicate what changes would be made in the state diagram (and if necessary in the undiagrammed event responses).
  - (b) Could TCP reasonably disallow simultaneous close?
  - (c) Propose a change to TCP in which simultaneous SYNs exchanged by two hosts lead to two separate connections. Indicate what state diagram changes this entails and what header changes become necessary. Note that this now means that more than one connection can exist over a given pair of ⟨host, port⟩s. (You might also look up the first “Discussion” item on page 87 of *Request for Comments* 1122.)
36. TCP is a very symmetric protocol, but the client/server model is not. Consider an asymmetric TCP-like protocol in which only the server side is assigned a port number visible to the application layers. Client-side sockets would simply be abstractions that can be connected to server ports.
  - (a) Propose header data and connection semantics to support this. What will you use to replace the client port number?
  - (b) What form does TIME\_WAIT now take? How would this be seen through the programming interface? Assume that a client socket could now be reconnected arbitrarily many times to a given server port, resources permitting.
  - (c) Look up the rsh/rlogin protocol. How would the above break this?
37. The following exercise is concerned with the TCP state FIN\_WAIT\_2 (see Figure 5.7).
  - (a) Describe how a client might leave a suitable server in state FIN\_WAIT\_2 indefinitely. What feature of the server's protocol is necessary here for this scenario?
  - (b) Try this with some appropriate existing server. Either write a stub client or use an existing Telnet client capable of connecting to an arbitrary port. Use the netstat utility to verify that the server is in FIN\_WAIT\_2 state.

**38.** *Request for Comments* 1122 states (of TCP):

A host MAY implement a “half-duplex” TCP close sequence, so that an application that has called CLOSE cannot continue to read data from the connection. If such a host issues a CLOSE call while received data is still pending in TCP, or if new data is received after CLOSE is called, its TCP SHOULD send an RST to show that data was lost.

Sketch a scenario involving the above in which data sent by (*not to!*) the closing host is lost. You may assume that the remote host, upon receiving an RST, discards all received data still unread in buffers.

- 39.** When TCP sends a  $\langle \text{SYN}, \text{SequenceNum} = x \rangle$  or  $\langle \text{FIN}, \text{SequenceNum} = x \rangle$ , the consequent ACK has Acknowledgment =  $x + 1$ ; that is, SYNs and FINs each take up one unit in sequence number space. Is this necessary? If so, give an example of an ambiguity that would arise if the corresponding Acknowledgment were  $x$  instead of  $x + 1$ ; if not, explain why.
- 40.** Find out the generic format for TCP header options from *Request for Comments* 793.
  - (a)** Outline a strategy that would expand the space available for options beyond the current limit of 44 bytes.
  - (b)** Suggest an extension to TCP allowing the sender of an option a way of specifying what the receiver should do if the option is not understood. List several such receiver actions that might be useful, and try to give an example application of each.
- 41.** The TCP header does not have a boot ID field. Why isn’t there a problem with one end of a TCP connection crashing and rebooting, then sending a message with an ID it had previously used?
- 42.** Suppose we were to implement remote file system mounting using an unreliable RPC protocol that offers zero-or-more semantics. If a message reply is received, this improves to at-least-once semantics. We define  $\text{read}(n)$  to return the specified  $n$ th block, rather than the next block in sequence; this way, reading once is the same as reading twice and at-least-once semantics is thus the same as exactly once.

- (a) For what other file system operations is there no difference between at-least-once and exactly once semantics? Consider open, create, write, seek, opendir, readdir, mkdir, delete (*aka* unlink), and rmdir.
- (b) For the remaining operations, which can have their semantics altered to achieve equivalence of at-least-once and exactly once? What file system operations are irreconcilable with at-least-once semantics?
- (c) Suppose the semantics of the rmdir system call are now that the given directory is removed if it exists, and nothing is done otherwise. How could you write a program to delete directories that distinguishes between these two cases?
43. The RPC-based NFS remote file system is sometimes considered to have slower than expected write performance. In NFS, a server's RPC reply to a client write request means that the data is physically written to the server's disk, not just placed in a queue.
- (a) Explain the bottleneck we might expect, even with infinite bandwidth, if the client sends all its write requests through a single logical channel, and explain why using a pool of channels could help. Hint: You will need to know a little about disk controllers.
- (b) Suppose the server's reply means only that the data has been placed in the disk queue. Explain how this could lead to data loss that wouldn't occur with a local disk. Note that a system crash immediately after data was enqueued doesn't count, because that would cause data loss on a local disk as well.
- (c) An alternative would be for the server to respond immediately to acknowledge the write request and to send its own separate request later to confirm the physical write. Propose different RPC semantics to achieve the same effect, but with a single logical request and reply.
44. Consider a client and server using an RPC mechanism that includes a channel abstraction and boot IDs.
- (a) Give a scenario involving server reboot in which an RPC request is sent twice by the client and is executed twice by the server, with only a single ACK.
- (b) How might the client become aware this had happened? Would the client be sure it had happened?

45. Suppose an RPC request is of the form: “Increment the value of field X of disk block N by 10%.” Specify a mechanism to be used by the executing server to guarantee that an arriving request is executed exactly once, even if the server crashes while in the middle of the operation. Assume that individual disk block writes are either complete or else the block is unchanged. You may also assume that some designated “undo log” blocks are available. Your mechanism should include how the RPC server is to behave at restart.
46. Consider a SunRPC client sending a request to a server.
- (a) Under what circumstances can the client be sure its request has executed exactly once?
  - (b) Suppose we wished to add at-most-once semantics to SunRPC. What changes would have to be made? Explain why adding one or more fields to the existing headers would not be sufficient.
47. Suppose TCP were to be used as the underlying transport in an RPC protocol; each TCP connection is to carry a sequential stream of requests and replies. What analog, if any, would TCP have for:
- (a) Channel ID
  - (b) Message ID
  - (c) Boot ID
  - (d) A message type for requests
  - (e) A message type for replies
  - (f) A message type for acknowledgments
  - (g) A message type for are-you-alive? messages
- Which of these would the overlying RPC protocol have to provide? Would some analog of implicit acknowledgments exist?
48. Write a test program that uses the socket interface to send messages between a pair of Unix workstations connected by some LAN (e.g., Ethernet, 802.11). Use this test program to perform the following experiments:
- (a) Measure the round-trip latency of TCP and UDP for different message sizes (e.g., 1 byte, 100 bytes, 200 bytes, . . . , 1000 bytes).

- (b) Measure the throughput of TCP and UDP for 1-KB, 2-KB, 3-KB, ..., 32-KB messages. Plot the measured throughput as a function of message size.
- (c) Measure the throughput of TCP by sending 1 MB of data from one host to another. Do this in a loop that sends a message of some size—for example, 1024 iterations of a loop that sends 1-KB messages. Repeat the experiment with different message sizes and plot the results.
49. Try to find situations where an RTP application might reasonably do the following:
- Send multiple packets at essentially the same time that need different timestamps.
  - Send packets at different times that need the same timestamp.
- Argue, in consequence, that RTP timestamps must, in at least some cases, be provided (at least indirectly) by the application. (Hint: Think of cases where the sending rate and playback rate might not match.)
50. Having the RTP timestamp clock count time in units of one frame time or one voice sample time would be the minimum resolution to ensure accurate playback, but the time unit is usually considerably smaller; what is the purpose of this?
51. Suppose we want returning RTCP reports from receivers to amount to no more than 5% of the outgoing primary RTP stream. If each report is 84 bytes, the RTP traffic is 320 kbps, and there are 1000 recipients, how often do individual receivers get to report? What if there are 10,000 recipients?
52. RFC 3550 specifies that the time interval between receiver RTCP reports include a randomization factor to avoid having all the receivers sending at the same time. If all the receivers sent in the same 5% subinterval of their reply time interval, the arriving upstream RTCP traffic would rival the downstream RTP traffic.
- (a) Video receivers might reasonably wait to send their reports until the higher-priority task of processing and displaying one frame is completed; this might mean their RTCP transmissions were synchronized on frame boundaries. Is this likely to be a serious concern?

- (b) With 10 receivers, what is the probability of their all sending in one particular 5% subinterval?
  - (c) With 10 receivers, what is the probability half will send in one particular 5% subinterval? Multiply this by 20 for an estimate of the probability half will all send in the same arbitrary 5% subinterval. (Hint: How many ways can we choose 5 receivers out of 10?)
53. What might a server actually do with the packet-loss-rate data and jitter data in receiver reports?
54. Propose a mechanism for deciding when to report an RTP packet as lost. How does your mechanism compare with the TCP adaptive retransmission mechanisms of [Section 5.2.6](#)?