

KUROSE  
•  
ROSS

# Computer Networking

A TOP-DOWN APPROACH

SEVENTH EDITION

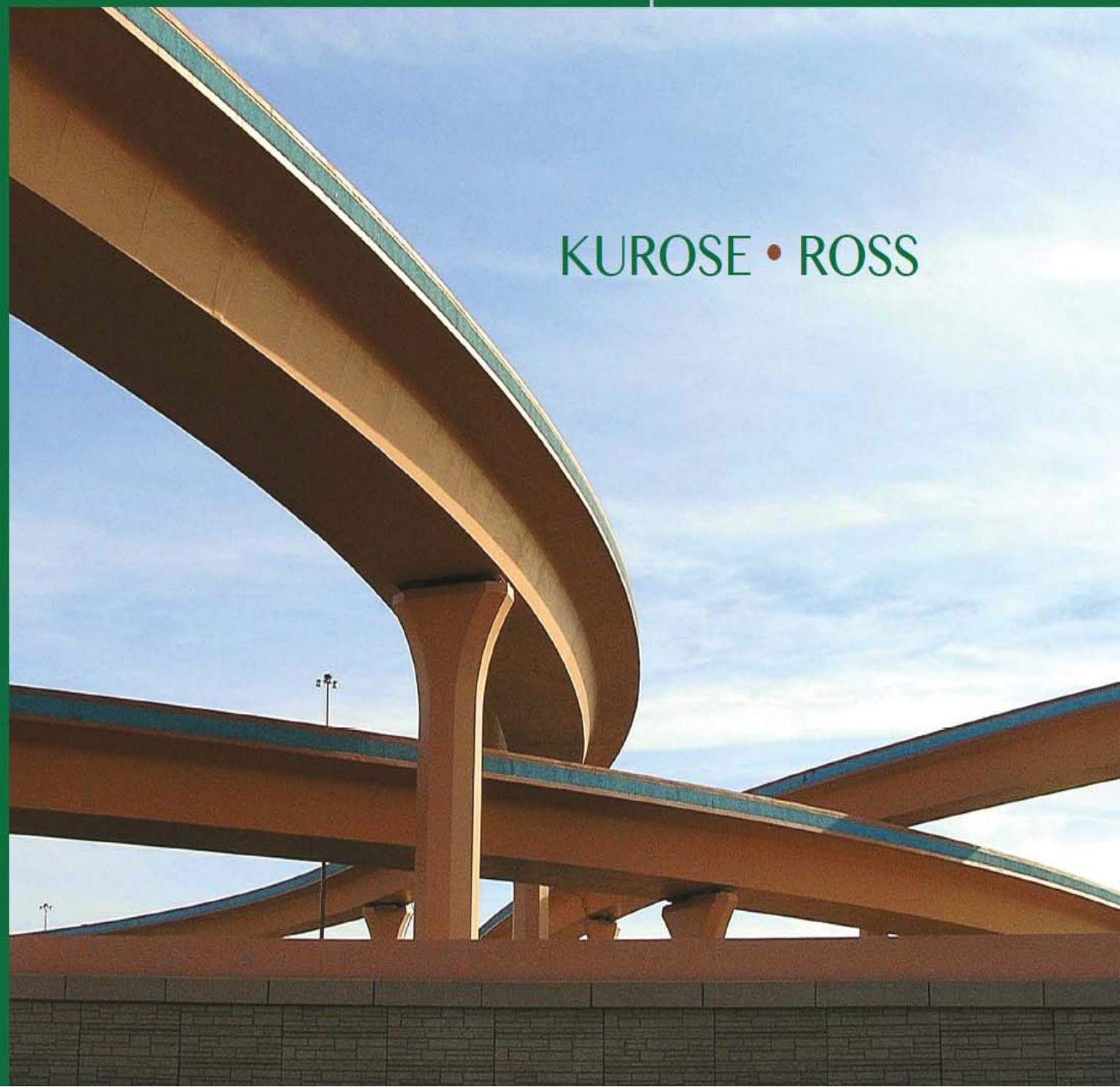
PEARSON

# Computer Networking

A TOP-DOWN APPROACH

SEVENTH EDITION

KUROSE • ROSS



# Computer Networking

*A Top-Down Approach*

Seventh Edition

**James F. Kurose**

University of Massachusetts, Amherst

**Keith W. Ross**

NYU and NYU Shanghai

**PEARSON**

Boston Columbus Indianapolis New York San Francisco Hoboken Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montréal Toronto Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

**Vice President, Editorial Director, ECS:** Marcia Horton

**Acquisitions Editor:** Matt Goldstein

**Editorial Assistant:** Kristy Alaura

**Vice President of Marketing:** Christy Lesko

**Director of Field Marketing:** Tim Galligan

**Product Marketing Manager:** Bram Van Kempen

**Field Marketing Manager:** Demetrius Hall

**Marketing Assistant:** Jon Bryant

**Director of Product Management:** Erin Gregg

**Team Lead, Program and Project Management:** Scott Disanno

**Program Manager:** Joanne Manning and Carole Snyder

**Project Manager:** Katrina Ostler, Ostler Editorial, Inc.

**Senior Specialist, Program Planning and Support:** Maura Zaldivar-Garcia

**Cover Designer:** Joyce Wells

**Manager, Rights and Permissions:** Ben Ferrini

**Project Manager, Rights and Permissions:** Jenny Hoffman, Aptara Corporation

**Inventory Manager:** Ann Lam

**Cover Image:** Marc Gutierrez/Getty Images

**Media Project Manager:** Steve Wright

**Composition:** Cenveo Publishing Services

**Printer/Binder:** Edwards Brothers Malloy

**Cover and Insert Printer:** Phoenix Color/ Hagerstown

Credits and acknowledgments borrowed from other sources and reproduced, with permission, in this textbook appear on appropriate page within text.

Copyright © 2017, 2013, 2010 **Pearson Education, Inc.** All rights reserved. Manufactured in the United States of America. This publication is protected by Copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit [www.pearsoned.com/permissions/](http://www.pearsoned.com/permissions/). Many of the designations by manufacturers and seller to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed in initial caps or all caps.

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

Names: Kurose, James F. | Ross, Keith W., 1956-

Title: Computer networking: a top-down approach / James F. Kurose, University of Massachusetts, Amherst, Keith W. Ross, NYU and NYU Shanghai.

Description: Seventh edition. | Hoboken, New Jersey: Pearson, [2017] | Includes bibliographical references and index.

Identifiers: LCCN 2016004976 | ISBN 9780133594140 | ISBN 0133594149

Subjects: LCSH: Internet. | Computer networks.

Classification: LCC TK5105.875.I57 K88 2017 | DDC 004.6-dc23

PEARSON

ISBN-10: 0-13-359414-9

ISBN-13: 978-0-13-359414-0

## About the Authors

### *Jim Kurose*

Jim Kurose is a Distinguished University Professor of Computer Science at the University of Massachusetts, Amherst. He is currently on leave from the University of Massachusetts, serving as an Assistant Director at the US National Science Foundation, where he leads the Directorate of Computer and Information Science and Engineering.

Dr. Kurose has received a number of recognitions for his educational activities including Outstanding Teacher Awards from the National Technological University (eight times), the University of Massachusetts, and the Northeast Association of Graduate Schools. He received the IEEE Taylor Booth Education Medal and was recognized for his leadership of Massachusetts' Commonwealth Information Technology Initiative. He has won several conference best paper awards and received the IEEE Infocom Achievement Award and the ACM Sigcomm Test of Time Award.



Dr. Kurose is a former Editor-in-Chief of *IEEE Transactions on Communications* and of *IEEE/ACM Transactions on Networking*. He has served as Technical Program co-Chair for *IEEE Infocom*, *ACM SIGCOMM*, *ACM Internet Measurement Conference*, and *ACM SIGMETRICS*. He is a Fellow of the IEEE and the ACM. His research interests include network protocols and architecture, network measurement, multimedia communication, and modeling and performance evaluation. He holds a PhD in Computer Science from Columbia University.

### *Keith Ross*

Keith Ross is the Dean of Engineering and Computer Science at NYU Shanghai and the Leonard J. Shustek Chair Professor in the Computer Science and Engineering Department at NYU. Previously he was at University of Pennsylvania (13 years), Eurecom Institute (5 years) and Polytechnic University (10 years). *He received a B.S.E.E from Tufts University, a M.S.E.E. from Columbia University, and a Ph.D. in Computer and Control Engineering from The University of Michigan. Keith Ross is also the co-founder and original CEO of Wimba, which develops online multimedia applications for e-learning and was acquired by Blackboard in 2010.*



Professor Ross's research interests are in privacy, social networks, peer-to-peer networking, Internet measurement, content distribution networks, and stochastic modeling. He is an ACM Fellow, an IEEE Fellow, recipient of the Infocom 2009 Best Paper Award, and recipient of 2011 and 2008 Best Paper Awards for Multimedia Communications (awarded by IEEE Communications Society). He has served on numerous journal editorial boards and conference program committees, including *IEEE/ACM Transactions on Networking*, *ACM SIGCOMM*, *ACM CoNext*, and *ACM Internet Measurement Conference*. He also has served as an advisor to the Federal Trade Commission on P2P file sharing.

To Julie and our three precious ones—Chris, Charlie, and Nina

**JFK**

A big THANKS to my professors, colleagues, and students all over the world.

**KWR**

## Preface

Welcome to the seventh edition of *Computer Networking: A Top-Down Approach*. Since the publication of the first edition 16 years ago, our book has been adopted for use at many hundreds of colleges and universities, translated into 14 languages, and used by over one hundred thousand students and practitioners worldwide. We've heard from many of these readers and have been overwhelmed by the positive response.

## What's New in the Seventh Edition?

We think one important reason for this success has been that our book continues to offer a fresh and timely approach to computer networking instruction. We've made changes in this seventh edition, but we've also kept unchanged what we believe (and the instructors and students who have used our book have confirmed) to be the most important aspects of this book: its top-down approach, its focus on the Internet and a modern treatment of computer networking, its attention to both principles and practice, and its accessible style and approach toward learning about computer networking. Nevertheless, the seventh edition has been revised and updated substantially.

Long-time readers of our book will notice that for the first time since this text was published, we've changed the organization of the chapters themselves. The network layer, which had been previously covered in a single chapter, is now covered in [Chapter 4](#) (which focuses on the so-called “data plane” component of the network layer) and [Chapter 5](#) (which focuses on the network layer’s “control plane”). This expanded coverage of the network layer reflects the swift rise in importance of software-defined networking (SDN), arguably the most important and exciting advance in networking in decades. Although a relatively recent innovation, SDN has been rapidly adopted in practice—so much so that it's already hard to imagine an introduction to modern computer networking that doesn't cover SDN. The topic of network management, previously covered in [Chapter 9](#), has now been folded into the new [Chapter 5](#). As always, we've also updated many other sections of the text to reflect recent changes in the dynamic field of networking since the sixth edition. As always, material that has been retired from the printed text can always be found on this book's Companion Website. The most important updates are the following:

- [Chapter 1](#) has been updated to reflect the ever-growing reach and use of the Internet.
- [Chapter 2](#), which covers the application layer, has been significantly updated. We've removed the material on the FTP protocol and distributed hash tables to make room for a new section on application-level video streaming and content distribution networks, together with Netflix and YouTube case studies. The socket programming sections have been updated from Python 2 to Python 3.
- [Chapter 3](#), which covers the transport layer, has been modestly updated. The material on asynchronous transport mode (ATM) networks has been replaced by more modern material on the Internet's explicit congestion notification (ECN), which teaches the same principles.
- [Chapter 4](#) covers the “data plane” component of the network layer—the *per-router* forwarding function that determine how a packet arriving on one of a router's input links is forwarded to one of that router's output links. We updated the material on traditional Internet forwarding found in all previous editions, and added material on packet scheduling. We've also added a new section on generalized forwarding, as practiced in SDN. There are also numerous updates throughout the chapter. Material on multicast and broadcast communication has been removed to make way for the new material.
- In [Chapter 5](#), we cover the control plane functions of the network layer—the *network-wide* logic that controls how a datagram is routed along an end-to-end path of routers from the source host to the destination host. As in previous editions, we cover routing algorithms, as well as routing protocols (with an updated treatment of BGP) used in today's Internet. We've added a significant new section on the SDN control plane, where routing and other functions are implemented in so-called SDN controllers.
- [Chapter 6](#), which now covers the link layer, has an updated treatment of Ethernet, and of data center networking.
- [Chapter 7](#), which covers wireless and mobile networking, contains updated material on 802.11 (so-called “WiFi) networks and cellular networks, including 4G and LTE.
- [Chapter 8](#), which covers network security and was extensively updated in the sixth edition, has only

modest updates in this seventh edition.

- **Chapter 9**, on multimedia networking, is now slightly “thinner” than in the sixth edition, as material on video streaming and content distribution networks has been moved to **Chapter 2**, and material on packet scheduling has been incorporated into **Chapter 4**.
- Significant new material involving end-of-chapter problems has been added. As with all previous editions, homework problems have been revised, added, and removed.

As always, our aim in creating this new edition of our book is to continue to provide a focused and modern treatment of computer networking, emphasizing both principles and practice.

## Audience

This textbook is for a first course on computer networking. It can be used in both computer science and electrical engineering departments. In terms of programming languages, the book assumes only that the student has experience with C, C++, Java, or Python (and even then only in a few places). Although this book is more precise and analytical than many other introductory computer networking texts, it rarely uses any mathematical concepts that are not taught in high school. We have made a deliberate effort to avoid using any advanced calculus, probability, or stochastic process concepts (although we’ve included some homework problems for students with this advanced background). The book is therefore appropriate for undergraduate courses and for first-year graduate courses. It should also be useful to practitioners in the telecommunications industry.

## *What Is Unique About This Textbook?*

The subject of computer networking is enormously complex, involving many concepts, protocols, and technologies that are woven together in an intricate manner. To cope with this scope and complexity, many computer networking texts are often organized around the “layers” of a network architecture. With a layered organization, students can see through the complexity of computer networking—they learn about the distinct concepts and protocols in one part of the architecture while seeing the big picture of how all parts fit together. From a pedagogical perspective, our personal experience has been that such a layered approach indeed works well. Nevertheless, we have found that the traditional approach of teaching—bottom up; that is, from the physical layer towards the application layer—is not the best approach for a modern course on computer networking.

## A Top-Down Approach

Our book broke new ground 16 years ago by treating networking in a top-down manner—that is, by beginning at the application layer and working its way down toward the physical layer. The feedback we received from teachers and students alike have confirmed that this top-down approach has many advantages and does indeed work well pedagogically. First, it places emphasis on the application layer (a “high growth area” in networking). Indeed, many of the recent revolutions in computer networking—including the Web, peer-to-peer file sharing, and media streaming—have taken place at the application layer. An early emphasis on application-layer issues differs from the approaches taken in most other texts, which have only a small amount of material on network applications, their requirements, application-layer paradigms (e.g., client-server and peer-to-peer), and application programming interfaces. Second, our experience as instructors (and that of many instructors who have used this text) has been that teaching networking applications near the beginning of the course is a powerful motivational tool. Students are thrilled to learn about how networking

applications work—applications such as e-mail and the Web, which most students use on a daily basis. Once a student understands the applications, the student can then understand the network services needed to support these applications. The student can then, in turn, examine the various ways in which such services might be provided and implemented in the lower layers. Covering applications early thus provides motivation for the remainder of the text.

Third, a top-down approach enables instructors to introduce network application development at an early stage. Students not only see how popular applications and protocols work, but also learn how easy it is to create their own network applications and application-level protocols. With the top-down approach, students get early exposure to the notions of socket programming, service models, and protocols—important concepts that resurface in all subsequent layers. By providing socket programming examples in Python, we highlight the central ideas without confusing students with complex code. Undergraduates in electrical engineering and computer science should not have difficulty following the Python code.

### An Internet Focus

Although we dropped the phrase “Featuring the Internet” from the title of this book with the fourth edition, this doesn’t mean that we dropped our focus on the Internet. Indeed, nothing could be further from the case! Instead, since the Internet has become so pervasive, we felt that any networking textbook must have a significant focus on the Internet, and thus this phrase was somewhat unnecessary. We continue to use the Internet’s architecture and protocols as primary vehicles for studying fundamental computer networking concepts. Of course, we also include concepts and protocols from other network architectures. But the spotlight is clearly on the Internet, a fact reflected in our organizing the book around the Internet’s five-layer architecture: the application, transport, network, link, and physical layers.

Another benefit of spotlighting the Internet is that most computer science and electrical engineering students are eager to learn about the Internet and its protocols. They know that the Internet has been a revolutionary and disruptive technology and can see that it is profoundly changing our world. Given the enormous relevance of the Internet, students are naturally curious about what is “under the hood.” Thus, it is easy for an instructor to get students excited about basic principles when using the Internet as the guiding focus.

### Teaching Networking Principles

Two of the unique features of the book—its top-down approach and its focus on the Internet—have appeared in the titles of our book. If we could have squeezed a *third* phrase into the subtitle, it would have contained the word *principles*. The field of networking is now mature enough that a number of fundamentally important issues can be identified. For example, in the transport layer, the fundamental issues include reliable communication over an unreliable network layer, connection establishment/ teardown and handshaking, congestion and flow control, and multiplexing. Three fundamentally important network-layer issues are determining “good” paths between two routers, interconnecting a large number of heterogeneous networks, and managing the complexity of a modern network. In the link layer, a fundamental problem is sharing a multiple access channel. In network security, techniques for providing confidentiality, authentication, and message integrity are all based on cryptographic fundamentals. This text identifies fundamental networking issues and studies approaches towards addressing these issues. The student learning these principles will gain knowledge with a long “shelf life”—long after today’s network standards and protocols have become obsolete, the principles they embody will remain important and relevant. We believe that the combination of using the Internet to get the student’s foot in the door and then emphasizing fundamental issues and solution approaches will allow the student to

quickly understand just about any networking technology.

## The Website

Each new copy of this textbook includes twelve months of access to a Companion Website for all book readers at <http://www.pearsonhighered.com/cs-resources/>, which includes:

- **Interactive learning material.** The book's Companion Website contains VideoNotes—video presentations of important topics throughout the book done by the authors, as well as walkthroughs of solutions to problems similar to those at the end of the chapter. We've seeded the Web site with VideoNotes and online problems for **Chapters 1 through 5** and will continue to actively add and update this material over time. As in earlier editions, the Web site contains the interactive Java applets that animate many key networking concepts. The site also has interactive quizzes that permit students to check their basic understanding of the subject matter. Professors can integrate these interactive features into their lectures or use them as mini labs.
- **Additional technical material.** As we have added new material in each edition of our book, we've had to remove coverage of some existing topics to keep the book at manageable length. For example, to make room for the new material in this edition, we've removed material on FTP, distributed hash tables, and multicasting. Material that appeared in earlier editions of the text is still of interest, and thus can be found on the book's Web site.
- **Programming assignments.** The Web site also provides a number of detailed programming assignments, which include building a multithreaded Web server, building an e-mail client with a GUI interface, programming the sender and receiver sides of a reliable data transport protocol, programming a distributed routing algorithm, and more.
- **Wireshark labs.** One's understanding of network protocols can be greatly deepened by seeing them in action. The Web site provides numerous Wireshark assignments that enable students to actually observe the sequence of messages exchanged between two protocol entities. The Web site includes separate Wireshark labs on HTTP, DNS, TCP, UDP, IP, ICMP, Ethernet, ARP, WiFi, SSL, and on tracing all protocols involved in satisfying a request to fetch a Web page. We'll continue to add new labs over time.

In addition to the Companion Website, the authors maintain a public Web site, [http://gaia.cs.umass.edu/kurose\\_ross/interactive](http://gaia.cs.umass.edu/kurose_ross/interactive), containing interactive exercises that create (and present solutions for) problems similar to selected end-of-chapter problems. Since students can generate (and view solutions for) an unlimited number of similar problem instances, they can work until the material is truly mastered.

## Pedagogical Features

We have each been teaching computer networking for more than 30 years. Together, we bring more than 60 years of teaching experience to this text, during which time we have taught many thousands of students. We have also been active researchers in computer networking during this time. (In fact, Jim and Keith first met each other as master's students in a computer networking course taught by Mischa Schwartz in 1979 at Columbia University.) We think all this gives us a good perspective on where networking has been and where it is likely to go in the future. Nevertheless, we have resisted temptations to bias the material in this book towards our own pet research projects. We figure you can visit our personal Web sites if you are interested in our research. Thus, this book is about modern computer networking—it is about contemporary protocols and technologies as well as the underlying principles behind these protocols and technologies. We also believe

that learning (and teaching!) about networking can be fun. A sense of humor, use of analogies, and real-world examples in this book will hopefully make this material more fun.

### *Supplements for Instructors*

We provide a complete supplements package to aid instructors in teaching this course. This material can be accessed from Pearson's Instructor Resource Center (<http://www.pearsonhighered.com/irc>). Visit the Instructor Resource Center for information about accessing these instructor's supplements.

- **PowerPoint® slides.** We provide PowerPoint slides for all nine chapters. The slides have been completely updated with this seventh edition. The slides cover each chapter in detail. They use graphics and animations (rather than relying only on monotonous text bullets) to make the slides interesting and visually appealing. We provide the original PowerPoint slides so you can customize them to best suit your own teaching needs. Some of these slides have been contributed by other instructors who have taught from our book.
- **Homework solutions.** We provide a solutions manual for the homework problems in the text, programming assignments, and Wireshark labs. As noted earlier, we've introduced many new homework problems in the first six chapters of the book.

### *Chapter Dependencies*

The first chapter of this text presents a self-contained overview of computer networking. Introducing many key concepts and terminology, this chapter sets the stage for the rest of the book. All of the other chapters directly depend on this first chapter. After completing **Chapter 1**, we recommend instructors cover **Chapters 2 through 6** in sequence, following our top-down philosophy. Each of these five chapters leverages material from the preceding chapters. After completing the first six chapters, the instructor has quite a bit of flexibility. There are no interdependencies among the last three chapters, so they can be taught in any order. However, each of the last three chapters depends on the material in the first six chapters. Many instructors first teach the first six chapters and then teach one of the last three chapters for “dessert.”

### *One Final Note: We'd Love to Hear from You*

We encourage students and instructors to e-mail us with any comments they might have about our book. It's been wonderful for us to hear from so many instructors and students from around the world about our first five editions. We've incorporated many of these suggestions into later editions of the book. We also encourage instructors to send us new homework problems (and solutions) that would complement the current homework problems. We'll post these on the instructor-only portion of the Web site. We also encourage instructors and students to create new Java applets that illustrate the concepts and protocols in this book. If you have an applet that you think would be appropriate for this text, please submit it to us. If the applet (including notation and terminology) is appropriate, we'll be happy to include it on the text's Web site, with an appropriate reference to the applet's authors.

So, as the saying goes, “Keep those cards and letters coming!” Seriously, please *do* continue to send us interesting URLs, point out typos, disagree with any of our claims, and tell us what works and what doesn't work. Tell us what you think should or shouldn't be included in the next edition. Send your e-mail to [kurose@cs.umass.edu](mailto:kurose@cs.umass.edu) and [keithross@nyu.edu](mailto:keithross@nyu.edu).

## Acknowledgments

Since we began writing this book in 1996, many people have given us invaluable help and have been influential in shaping our thoughts on how to best organize and teach a networking course. We want to say A BIG THANKS to everyone who has helped us from the earliest first drafts of this book, up to this seventh edition. We are also very thankful to the many hundreds of readers from around the world—students, faculty, practitioners—who have sent us thoughts and comments on earlier editions of the book and suggestions for future editions of the book. Special thanks go out to:

Al Aho (Columbia University)

Hisham Al-Mubaid (University of Houston-Clear Lake)

Pratima Akkunoor (Arizona State University)

Paul Amer (University of Delaware)

Shamiul Azom (Arizona State University)

Lichun Bao (University of California at Irvine)

Paul Barford (University of Wisconsin)

Bobby Bhattacharjee (University of Maryland)

Steven Bellovin (Columbia University)

Pravin Bhagwat (Wibhu)

Supratik Bhattacharyya (previously at Sprint)

Ernst Biersack (Eurécom Institute)

Shahid Bokhari (University of Engineering & Technology, Lahore)

Jean Bolot (Technicolor Research)

Daniel Brushteyn (former University of Pennsylvania student)

Ken Calvert (University of Kentucky)

Evandro Cantu (Federal University of Santa Catarina)

Jeff Case (SNMP Research International)

Jeff Chaltas (Sprint)

Vinton Cerf (Google)

Byung Kyu Choi (Michigan Technological University)

Bram Cohen (BitTorrent, Inc.)

Constantine Coutras (Pace University)

John Daigle (University of Mississippi)

Edmundo A. de Souza e Silva (Federal University of Rio de Janeiro)

Philippe Decuetos (Eurécom Institute)

Christophe Diot (Technicolor Research)

Prithula Dhunghel (Akamai)

Deborah Estrin (University of California, Los Angeles)

Michalis Faloutsos (University of California at Riverside)

Wu-chi Feng (Oregon Graduate Institute)

Sally Floyd (ICIR, University of California at Berkeley)

Paul Francis (Max Planck Institute)

David Fullager (Netflix)

Lixin Gao (University of Massachusetts)

JJ Garcia-Luna-Aceves (University of California at Santa Cruz)

Mario Gerla (University of California at Los Angeles)

David Goodman (NYU-Poly)

Yang Guo (Alcatel/Lucent Bell Labs)

Tim Griffin (Cambridge University)

Max Hailperin (Gustavus Adolphus College)

Bruce Harvey (Florida A&M University, Florida State University)

Carl Hauser (Washington State University)

Rachelle Heller (George Washington University)

Phillipp Hoschka (INRIA/W3C)

Wen Hsin (Park University)

Albert Huang (former University of Pennsylvania student)

Cheng Huang (Microsoft Research)

Esther A. Hughes (Virginia Commonwealth University)

Van Jacobson (Xerox PARC)

Pinak Jain (former NYU-Poly student)

Jobin James (University of California at Riverside)

Sugih Jamin (University of Michigan)

Shivkumar Kalyanaraman (IBM Research, India)

Jussi Kangasharju (University of Helsinki)

Sneha Kasera (University of Utah)

Parviz Kermani (formerly of IBM Research)

Hyojin Kim (former University of Pennsylvania student)

Leonard Kleinrock (University of California at Los Angeles)

David Kotz (Dartmouth College)

Beshan Kulapala (Arizona State University)

Rakesh Kumar (Bloomberg)

Miguel A. Labrador (University of South Florida)

Simon Lam (University of Texas)

Steve Lai (Ohio State University)

Tom LaPorta (Penn State University)

Tim-Berners Lee (World Wide Web Consortium)

Arnaud Legout (INRIA)

Lee Leitner (Drexel University)

Brian Levine (University of Massachusetts)

Chunchun Li (former NYU-Poly student)

Yong Liu (NYU-Poly)

William Liang (former University of Pennsylvania student)

Willis Marti (Texas A&M University)

Nick McKeown (Stanford University)

Josh McKinzie (Park University)

Deep Medhi (University of Missouri, Kansas City)

Bob Metcalfe (International Data Group)

Sue Moon (KAIST)

Jenni Moyer (Comcast)

Erich Nahum (IBM Research)

Christos Papadopoulos (Colorado State University)

Craig Partridge (BBN Technologies)

Radia Perlman (Intel)

Jitendra Padhye (Microsoft Research)

Vern Paxson (University of California at Berkeley)

Kevin Phillips (Sprint)

George Polyzos (Athens University of Economics and Business)

Sriram Rajagopalan (Arizona State University)

Ramachandran Ramjee (Microsoft Research)

Ken Reek (Rochester Institute of Technology)

Martin Reisslein (Arizona State University)

Jennifer Rexford (Princeton University)

Leon Reznik (Rochester Institute of Technology)

Pablo Rodriguez (Telefonica)

Sumit Roy (University of Washington)

Dan Rubenstein (Columbia University)

Avi Rubin (Johns Hopkins University)

Douglas Salane (John Jay College)

Despina Saporilla (Cisco Systems)

John Schanz (Comcast)

Henning Schulzrinne (Columbia University)

Mischa Schwartz (Columbia University)

Ardash Sethi (University of Delaware)

Harish Sethu (Drexel University)

K. Sam Shanmugan (University of Kansas)

Prashant Shenoy (University of Massachusetts)

Clay Shields (Georgetown University)

Subin Shrestra (University of Pennsylvania)

Bojie Shu (former NYU-Poly student)

Mihail L. Sichitiu (NC State University)

Peter Steenkiste (Carnegie Mellon University)

Tatsuya Suda (University of California at Irvine)

Kin Sun Tam (State University of New York at Albany)

Don Towsley (University of Massachusetts)

David Turner (California State University, San Bernardino)

Nitin Vaidya (University of Illinois)

Michele Weigle (Clemson University)

David Wetherall (University of Washington)

Ira Winston (University of Pennsylvania)

Di Wu (Sun Yat-sen University)

Shirley Wynn (NYU-Poly)

Raj Yavatkar (Intel)

Yechiam Yemini (Columbia University)

Dian Yu (NYU Shanghai)

Ming Yu (State University of New York at Binghamton)

Ellen Zegura (Georgia Institute of Technology)

Honggang Zhang (Suffolk University)

Hui Zhang (Carnegie Mellon University)

Lixia Zhang (University of California at Los Angeles)

Meng Zhang (former NYU-Poly student)

Shuchun Zhang (former University of Pennsylvania student)

Xiaodong Zhang (Ohio State University)

ZhiLi Zhang (University of Minnesota)

Phil Zimmermann (independent consultant)

Mike Zink (University of Massachusetts)

Cliff C. Zou (University of Central Florida)

We also want to thank the entire Pearson team—in particular, Matt Goldstein and Joanne Manning—who have done an absolutely outstanding job on this seventh edition (and who have put up with two very finicky authors who seem congenitally unable to meet deadlines!). Thanks also to our artists, Janet Theurer and Patrice Rossi Calkin, for their work on the beautiful figures in this and earlier editions of our book, and to Katie Ostler and her team at Cenveo for their wonderful production work on this edition. Finally, a most special thanks go to our previous two editors at Addison-Wesley—Michael Hirsch and Susan Hartman. This book would not be what it is (and may well not have been at all) without their graceful management, constant encouragement, nearly infinite patience, good humor, and perseverance.

# Table of Contents

## Chapter 1 Computer Networks and the Internet 1

### 1.1 What Is the Internet? 2

#### 1.1.1 A Nuts-and-Bolts Description 2

#### 1.1.2 A Services Description 5

#### 1.1.3 What Is a Protocol? 7

### 1.2 The Network Edge 9

#### 1.2.1 Access Networks 12

#### 1.2.2 Physical Media 18

### 1.3 The Network Core 21

#### 1.3.1 Packet Switching 23

#### 1.3.2 Circuit Switching 27

#### 1.3.3 A Network of Networks 31

### 1.4 Delay, Loss, and Throughput in Packet-Switched Networks 35

#### 1.4.1 Overview of Delay in Packet-Switched Networks 35

#### 1.4.2 Queuing Delay and Packet Loss 39

#### 1.4.3 End-to-End Delay 41

#### 1.4.4 Throughput in Computer Networks 43

### 1.5 Protocol Layers and Their Service Models 47

#### 1.5.1 Layered Architecture 47

#### 1.5.2 Encapsulation 53

### 1.6 Networks Under Attack 55

### 1.7 History of Computer Networking and the Internet 59

#### 1.7.1 The Development of Packet Switching: 1961–1972 59

#### 1.7.2 Proprietary Networks and Internetworking: 1972–1980 60

#### 1.7.3 A Proliferation of Networks: 1980–1990 62

#### 1.7.4 The Internet Explosion: The 1990s 63

#### 1.7.5 The New Millennium 64

### 1.8 Summary 65

## **Homework Problems and Questions 67**

**Wireshark Lab 77**

**Interview: Leonard Kleinrock 79**

### **Chapter 2 Application Layer 83**

**2.1 Principles of Network Applications 84**

**2.1.1 Network Application Architectures 86**

**2.1.2 Processes Communicating 88**

**2.1.3 Transport Services Available to Applications 90**

**2.1.4 Transport Services Provided by the Internet 93**

**2.1.5 Application-Layer Protocols 96**

**2.1.6 Network Applications Covered in This Book 97**

**2.2 The Web and HTTP 98**

**2.2.1 Overview of HTTP 98**

**2.2.2 Non-Persistent and Persistent Connections 100**

**2.2.3 HTTP Message Format 103**

**2.2.4 User-Server Interaction: Cookies 108**

**2.2.5 Web Caching 110**

**2.3 Electronic Mail in the Internet 116**

**2.3.1 SMTP 118**

**2.3.2 Comparison with HTTP 121**

**2.3.3 Mail Message Formats 121**

**2.3.4 Mail Access Protocols 122**

**2.4 DNS—The Internet’s Directory Service 126**

**2.4.1 Services Provided by DNS 127**

**2.4.2 Overview of How DNS Works 129**

**2.4.3 DNS Records and Messages 135**

**2.5 Peer-to-Peer Applications 140**

**2.5.1 P2P File Distribution 140**

**2.6 Video Streaming and Content Distribution Networks 147**

**2.6.1 Internet Video 148**

**2.6.2 HTTP Streaming and DASH 148**

<b>2.6.3 Content Distribution Networks</b>	<b>149</b>
<b>2.6.4 Case Studies: Netflix, YouTube, and Kankan</b>	<b>153</b>
<b>2.7 Socket Programming: Creating Network Applications</b>	<b>157</b>
<b>2.7.1 Socket Programming with UDP</b>	<b>159</b>
<b>2.7.2 Socket Programming with TCP</b>	<b>164</b>
<b>2.8 Summary</b>	<b>170</b>
<b>Homework Problems and Questions</b>	<b>171</b>
<b>Socket Programming Assignments</b>	<b>180</b>
<b>Wireshark Labs: HTTP, DNS</b>	<b>182</b>
<b>Interview: Marc Andreessen</b>	<b>184</b>
<b>Chapter 3 Transport Layer</b>	<b>187</b>
<b>3.1 Introduction and Transport-Layer Services</b>	<b>188</b>
<b>3.1.1 Relationship Between Transport and Network Layers</b>	<b>188</b>
<b>3.1.2 Overview of the Transport Layer in the Internet</b>	<b>191</b>
<b>3.2 Multiplexing and Demultiplexing</b>	<b>193</b>
<b>3.3 Connectionless Transport: UDP</b>	<b>200</b>
<b>3.3.1 UDP Segment Structure</b>	<b>204</b>
<b>3.3.2 UDP Checksum</b>	<b>204</b>
<b>3.4 Principles of Reliable Data Transfer</b>	<b>206</b>
<b>3.4.1 Building a Reliable Data Transfer Protocol</b>	<b>208</b>
<b>3.4.2 Pipelined Reliable Data Transfer Protocols</b>	<b>217</b>
<b>3.4.3 Go-Back-N (GBN)</b>	<b>221</b>
<b>3.4.4 Selective Repeat (SR)</b>	<b>226</b>
<b>3.5 Connection-Oriented Transport: TCP</b>	<b>233</b>
<b>3.5.1 The TCP Connection</b>	<b>233</b>
<b>3.5.2 TCP Segment Structure</b>	<b>236</b>
<b>3.5.3 Round-Trip Time Estimation and Timeout</b>	<b>241</b>
<b>3.5.4 Reliable Data Transfer</b>	<b>244</b>
<b>3.5.5 Flow Control</b>	<b>252</b>
<b>3.5.6 TCP Connection Management</b>	<b>255</b>
<b>3.6 Principles of Congestion Control</b>	<b>261</b>

**3.6.1 The Causes and the Costs of Congestion 261**

**3.6.2 Approaches to Congestion Control 268**

**3.7 TCP Congestion Control 269**

**3.7.1 Fairness 279**

**3.7.2 Explicit Congestion Notification (ECN): Network-assisted Congestion Control 282**

**3.8 Summary 284**

**Homework Problems and Questions 286**

**Programming Assignments 301**

**Wireshark Labs: Exploring TCP, UDP 302**

**Interview: Van Jacobson 303**

**Chapter 4 The Network Layer: Data Plane 305**

**4.1 Overview of Network Layer 306**

**4.1.1 Forwarding and Routing: The Network Data and Control Planes 306**

**4.1.2 Network Service Models 311**

**4.2 What's Inside a Router? 313**

**4.2.1 Input Port Processing and Destination-Based Forwarding 316**

**4.2.2 Switching 319**

**4.2.3 Output Port Processing 321**

**4.2.4 Where Does Queuing Occur? 321**

**4.2.5 Packet Scheduling 325**

**4.3 The Internet Protocol (IP): IPv4, Addressing, IPv6, and More 329**

**4.3.1 IPv4 Datagram Format 330**

**4.3.2 IPv4 Datagram Fragmentation 332**

**4.3.3 IPv4 Addressing 334**

**4.3.4 Network Address Translation (NAT) 345**

**4.3.5 IPv6 348**

**4.4 Generalized Forwarding and SDN 354**

**4.4.1 Match 356**

**4.4.2 Action 358**

**4.4.3 OpenFlow Examples of Match-plus-action in Action 358**

**4.5 Summary 361**

**Homework Problems and Questions 361**

**Wireshark Lab 370**

**Interview: Vinton G. Cerf 371**

**Chapter 5 The Network Layer: Control Plane 373**

**5.1 Introduction 374**

**5.2 Routing Algorithms 376**

**5.2.1 The Link-State (LS) Routing Algorithm 379**

**5.2.2 The Distance-Vector (DV) Routing Algorithm 384**

**5.3 Intra-AS Routing in the Internet: OSPF 391**

**5.4 Routing Among the ISPs: BGP 395**

**5.4.1 The Role of BGP 395**

**5.4.2 Advertising BGP Route Information 396**

**5.4.3 Determining the Best Routes 398**

**5.4.4 IP-Anycast 402**

**5.4.5 Routing Policy 403**

**5.4.6 Putting the Pieces Together: Obtaining Internet Presence 406**

**5.5 The SDN Control Plane 407**

**5.5.1 The SDN Control Plane: SDN Controller and SDN Control Applications 410**

**5.5.2 OpenFlow Protocol 412**

**5.5.3 Data and Control Plane Interaction: An Example 414**

**5.5.4 SDN: Past and Future 415**

**5.6 ICMP: The Internet Control Message Protocol 419**

**5.7 Network Management and SNMP 421**

**5.7.1 The Network Management Framework 422**

**5.7.2 The Simple Network Management Protocol (SNMP) 424**

**5.8 Summary 426**

**Homework Problems and Questions 427**

**Socket Programming Assignment 433**

**Programming Assignment 434**

**Wireshark Lab 435**

**Interview: Jennifer Rexford 436**

## **Chapter 6 The Link Layer and LANs 439**

### **6.1 Introduction to the Link Layer 440**

#### **6.1.1 The Services Provided by the Link Layer 442**

#### **6.1.2 Where Is the Link Layer Implemented? 443**

### **6.2 Error-Detection and -Correction Techniques 444**

#### **6.2.1 Parity Checks 446**

#### **6.2.2 Checksumming Methods 448**

#### **6.2.3 Cyclic Redundancy Check (CRC) 449**

### **6.3 Multiple Access Links and Protocols 451**

#### **6.3.1 Channel Partitioning Protocols 453**

#### **6.3.2 Random Access Protocols 455**

#### **6.3.3 Taking-Turns Protocols 464**

#### **6.3.4 DOCSIS: The Link-Layer Protocol for Cable Internet Access 465**

### **6.4 Switched Local Area Networks 467**

#### **6.4.1 Link-Layer Addressing and ARP 468**

#### **6.4.2 Ethernet 474**

#### **6.4.3 Link-Layer Switches 481**

#### **6.4.4 Virtual Local Area Networks (VLANs) 487**

### **6.5 Link Virtualization: A Network as a Link Layer 491**

#### **6.5.1 Multiprotocol Label Switching (MPLS) 492**

### **6.6 Data Center Networking 495**

### **6.7 Retrospective: A Day in the Life of a Web Page Request 500**

#### **6.7.1 Getting Started: DHCP, UDP, IP, and Ethernet 500**

#### **6.7.2 Still Getting Started: DNS and ARP 502**

#### **6.7.3 Still Getting Started: Intra-Domain Routing to the DNS Server 503**

#### **6.7.4 Web Client-Server Interaction: TCP and HTTP 504**

### **6.8 Summary 506**

### **Homework Problems and Questions 507**

### **Wireshark Lab 515**

### **Interview: Simon S. Lam 516**

## **Chapter 7 Wireless and Mobile Networks 519**

**7.1 Introduction 520**

**7.2 Wireless Links and Network Characteristics 525**

**7.2.1 CDMA 528**

**7.3 WiFi: 802.11 Wireless LANs 532**

**7.3.1 The 802.11 Architecture 533**

**7.3.2 The 802.11 MAC Protocol 537**

**7.3.3 The IEEE 802.11 Frame 542**

**7.3.4 Mobility in the Same IP Subnet 546**

**7.3.5 Advanced Features in 802.11 547**

**7.3.6 Personal Area Networks: Bluetooth and Zigbee 548**

**7.4 Cellular Internet Access 551**

**7.4.1 An Overview of Cellular Network Architecture 551**

**7.4.2 3G Cellular Data Networks: Extending the Internet to Cellular Subscribers 554**

**7.4.3 On to 4G: LTE 557**

**7.5 Mobility Management: Principles 560**

**7.5.1 Addressing 562**

**7.5.2 Routing to a Mobile Node 564**

**7.6 Mobile IP 570**

**7.7 Managing Mobility in Cellular Networks 574**

**7.7.1 Routing Calls to a Mobile User 576**

**7.7.2 Handoffs in GSM 577**

**7.8 Wireless and Mobility: Impact on Higher-Layer Protocols 580**

**7.9 Summary 582**

**Homework Problems and Questions 583**

**Wireshark Lab 588**

**Interview: Deborah Estrin 589**

## **Chapter 8 Security in Computer Networks 593**

**8.1 What Is Network Security? 594**

**8.2 Principles of Cryptography 596**

**8.2.1 Symmetric Key Cryptography 598**

**8.2.2 Public Key Encryption 604**

## **8.3 Message Integrity and Digital Signatures 610**

**8.3.1 Cryptographic Hash Functions 611**

**8.3.2 Message Authentication Code 613**

**8.3.3 Digital Signatures 614**

## **8.4 End-Point Authentication 621**

**8.4.1 Authentication Protocol ap1.0 622**

**8.4.2 Authentication Protocol ap2.0 622**

**8.4.3 Authentication Protocol ap3.0 623**

**8.4.4 Authentication Protocol ap3.1 623**

**8.4.5 Authentication Protocol ap4.0 624**

## **8.5 Securing E-Mail 626**

**8.5.1 Secure E-Mail 627**

**8.5.2 PGP 630**

## **8.6 Securing TCP Connections: SSL 631**

**8.6.1 The Big Picture 632**

**8.6.2 A More Complete Picture 635**

## **8.7 Network-Layer Security: IPsec and Virtual Private Networks 637**

**8.7.1 IPsec and Virtual Private Networks (VPNs) 638**

**8.7.2 The AH and ESP Protocols 640**

**8.7.3 Security Associations 640**

**8.7.4 The IPsec Datagram 641**

**8.7.5 IKE: Key Management in IPsec 645**

## **8.8 Securing Wireless LANs 646**

**8.8.1 Wired Equivalent Privacy (WEP) 646**

**8.8.2 IEEE 802.11i 648**

## **8.9 Operational Security: Firewalls and Intrusion Detection Systems 651**

**8.9.1 Firewalls 651**

**8.9.2 Intrusion Detection Systems 659**

## **8.10 Summary 662**

### **Homework Problems and Questions 664**

### **Wireshark Lab 672**

**IPsec Lab** 672

**Interview: Steven M. Bellovin** 673

**Chapter 9 Multimedia Networking** 675

**9.1 Multimedia Networking Applications** 676

**9.1.1 Properties of Video** 676

**9.1.2 Properties of Audio** 677

**9.1.3 Types of Multimedia Network Applications** 679

**9.2 Streaming Stored Video** 681

**9.2.1 UDP Streaming** 683

**9.2.2 HTTP Streaming** 684

**9.3 Voice-over-IP** 688

**9.3.1 Limitations of the Best-Effort IP Service** 688

**9.3.2 Removing Jitter at the Receiver for Audio** 691

**9.3.3 Recovering from Packet Loss** 694

**9.3.4 Case Study: VoIP with Skype** 697

**9.4 Protocols for Real-Time Conversational Applications** 700

**9.4.1 RTP** 700

**9.4.2 SIP** 703

**9.5 Network Support for Multimedia** 709

**9.5.1 Dimensioning Best-Effort Networks** 711

**9.5.2 Providing Multiple Classes of Service** 712

**9.5.3 Diffserv** 719

**9.5.4 Per-Connection Quality-of-Service (QoS) Guarantees: Resource Reservation and Call Admission** 723

**9.6 Summary** 726

**Homework Problems and Questions** 727

**Programming Assignment** 735

**Interview: Henning Schulzrinne** 736

**References** 741

**Index** 783



---

# Chapter 1 Computer Networks and the Internet

---

Today's Internet is arguably the largest engineered system ever created by mankind, with hundreds of millions of connected computers, communication links, and switches; with billions of users who connect via laptops, tablets, and smartphones; and with an array of new Internet-connected "things" including game consoles, surveillance systems, watches, eye glasses, thermostats, body scales, and cars. Given that the Internet is so large and has so many diverse components and uses, is there any hope of understanding how it works? Are there guiding principles and structure that can provide a foundation for understanding such an amazingly large and complex system? And if so, is it possible that it actually could be both interesting *and* fun to learn about computer networks? Fortunately, the answer to all of these questions is a resounding YES! Indeed, it's our aim in this book to provide you with a modern introduction to the dynamic field of computer networking, giving you the principles and practical insights you'll need to understand not only today's networks, but tomorrow's as well.

This first chapter presents a broad overview of computer networking and the Internet. Our goal here is to paint a broad picture and set the context for the rest of this book, to see the forest through the trees. We'll cover a lot of ground in this introductory chapter and discuss a lot of the pieces of a computer network, without losing sight of the big picture.

We'll structure our overview of computer networks in this chapter as follows. After introducing some basic terminology and concepts, we'll first examine the basic hardware and software components that make up a network. We'll begin at the network's edge and look at the end systems and network applications running in the network. We'll then explore the core of a computer network, examining the links and the switches that transport data, as well as the access networks and physical media that connect end systems to the network core. We'll learn that the Internet is a network of networks, and we'll learn how these networks connect with each other.

After having completed this overview of the edge and core of a computer network, we'll take the broader and more abstract view in the second half of this chapter. We'll examine delay, loss, and throughput of data in a computer network and provide simple quantitative models for end-to-end throughput and delay: models that take into account transmission, propagation, and queuing delays. We'll then introduce some of the key architectural principles in computer networking, namely, protocol layering and service models. We'll also learn that computer networks are vulnerable to many different types of attacks; we'll survey

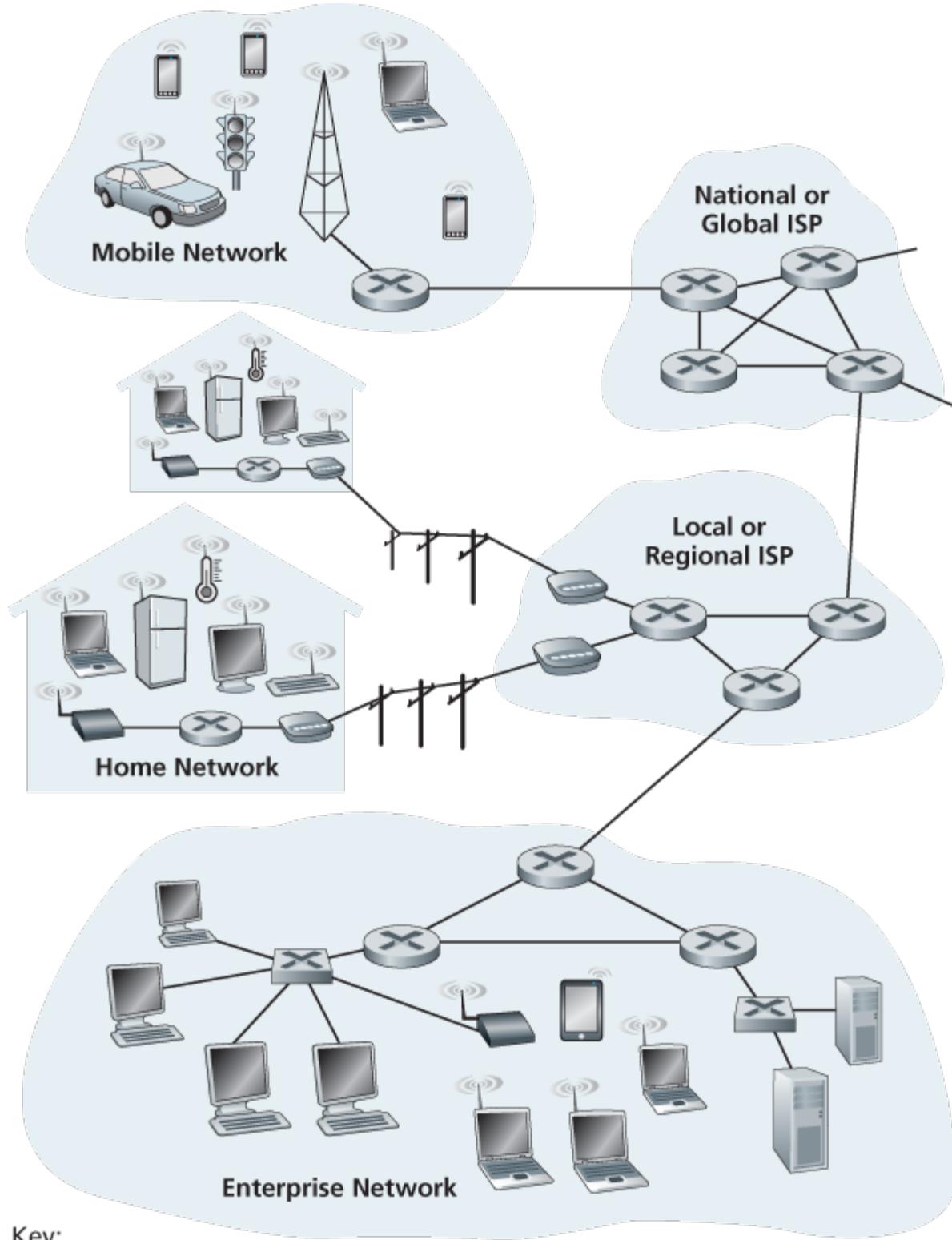
some of these attacks and consider how computer networks can be made more secure. Finally, we'll close this chapter with a brief history of computer networking.

## 1.1 What Is the Internet?

In this book, we'll use the public Internet, a specific computer network, as our principal vehicle for discussing computer networks and their protocols. But what *is* the Internet? There are a couple of ways to answer this question. First, we can describe the nuts and bolts of the Internet, that is, the basic hardware and software components that make up the Internet. Second, we can describe the Internet in terms of a networking infrastructure that provides services to distributed applications. Let's begin with the nuts-and-bolts description, using [Figure 1.1](#) to illustrate our discussion.

### 1.1.1 A Nuts-and-Bolts Description

The Internet is a computer network that interconnects billions of computing devices throughout the world. Not too long ago, these computing devices were primarily traditional desktop PCs, Linux workstations, and so-called servers that store and transmit information such as Web pages and e-mail messages. Increasingly, however, nontraditional Internet “things” such as laptops, smartphones, tablets, TVs, gaming consoles, thermostats, home security systems, home appliances, watches, eye glasses, cars, traffic control systems and more are being connected to the Internet. Indeed, the term *computer network* is beginning to sound a bit dated, given the many nontraditional devices that are being hooked up to the Internet. In Internet jargon, all of these devices are called **hosts** or **end systems**. By some estimates, in 2015 there were about 5 billion devices connected to the Internet, and the number will reach 25 billion by 2020 [\[Gartner 2014\]](#). It is estimated that in 2015 there were over 3.2 billion Internet users worldwide, approximately 40% of the world population [\[ITU 2015\]](#).



Key:

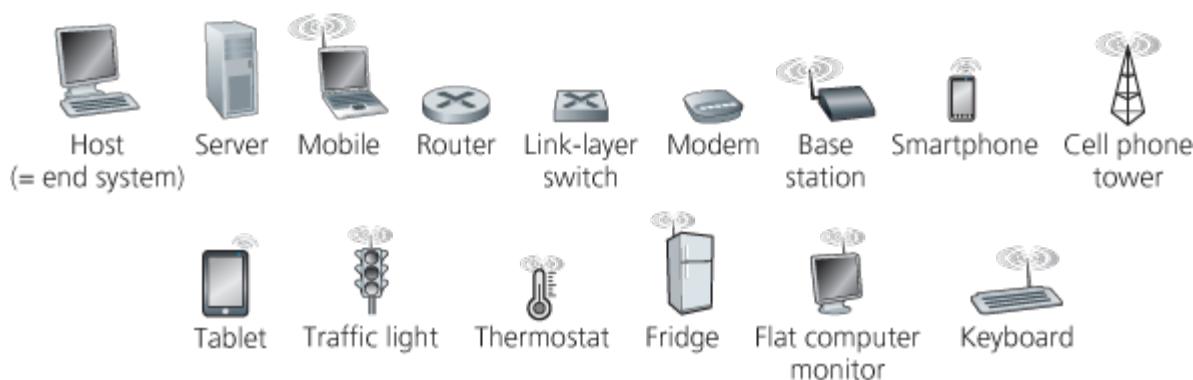


Figure 1.1 Some pieces of the Internet

End systems are connected together by a network of **communication links** and **packet switches**.

We'll see in **Section 1.2** that there are many types of communication links, which are made up of

different types of physical media, including coaxial cable, copper wire, optical fiber, and radio spectrum. Different links can transmit data at different rates, with the **transmission rate** of a link measured in bits/second. When one end system has data to send to another end system, the sending end system segments the data and adds header bytes to each segment. The resulting packages of information, known as **packets** in the jargon of computer networks, are then sent through the network to the destination end system, where they are reassembled into the original data.

A packet switch takes a packet arriving on one of its incoming communication links and forwards that packet on one of its outgoing communication links. Packet switches come in many shapes and flavors, but the two most prominent types in today's Internet are **routers** and **link-layer switches**. Both types of switches forward packets toward their ultimate destinations. Link-layer switches are typically used in access networks, while routers are typically used in the network core. The sequence of communication links and packet switches traversed by a packet from the sending end system to the receiving end system is known as a **route** or **path** through the network. Cisco predicts annual global IP traffic will pass the zettabyte ( $10^{21}$  bytes) threshold by the end of 2016, and will reach 2 zettabytes per year by 2019 [Cisco VNI 2015].

Packet-switched networks (which transport packets) are in many ways similar to transportation networks of highways, roads, and intersections (which transport vehicles). Consider, for example, a factory that needs to move a large amount of cargo to some destination warehouse located thousands of kilometers away. At the factory, the cargo is segmented and loaded into a fleet of trucks. Each of the trucks then independently travels through the network of highways, roads, and intersections to the destination warehouse. At the destination warehouse, the cargo is unloaded and grouped with the rest of the cargo arriving from the same shipment. Thus, in many ways, packets are analogous to trucks, communication links are analogous to highways and roads, packet switches are analogous to intersections, and end systems are analogous to buildings. Just as a truck takes a path through the transportation network, a packet takes a path through a computer network.

End systems access the Internet through **Internet Service Providers (ISPs)**, including residential ISPs such as local cable or telephone companies; corporate ISPs; university ISPs; ISPs that provide WiFi access in airports, hotels, coffee shops, and other public places; and cellular data ISPs, providing mobile access to our smartphones and other devices. Each ISP is in itself a network of packet switches and communication links. ISPs provide a variety of types of network access to the end systems, including residential broadband access such as cable modem or DSL, high-speed local area network access, and mobile wireless access. ISPs also provide Internet access to content providers, connecting Web sites and video servers directly to the Internet. The Internet is all about connecting end systems to each other, so the ISPs that provide access to end systems must also be interconnected. These lower-tier ISPs are interconnected through national and international upper-tier ISPs such as Level 3 Communications, AT&T, Sprint, and NTT. An upper-tier ISP consists of high-speed routers interconnected with high-speed fiber-optic links. Each ISP network, whether upper-tier or lower-tier, is

managed independently, runs the IP protocol (see below), and conforms to certain naming and address conventions. We'll examine ISPs and their interconnection more closely in [Section 1.3](#).

End systems, packet switches, and other pieces of the Internet run **protocols** that control the sending and receiving of information within the Internet. The **Transmission Control Protocol (TCP)** and the **Internet Protocol (IP)** are two of the most important protocols in the Internet. The IP protocol specifies the format of the packets that are sent and received among routers and end systems. The Internet's principal protocols are collectively known as **TCP/IP**. We'll begin looking into protocols in this introductory chapter. But that's just a start—much of this book is concerned with computer network protocols!

Given the importance of protocols to the Internet, it's important that everyone agree on what each and every protocol does, so that people can create systems and products that interoperate. This is where standards come into play. **Internet standards** are developed by the Internet Engineering Task Force (IETF) [\[IETF 2016\]](#). The IETF standards documents are called **requests for comments (RFCs)**. RFCs started out as general requests for comments (hence the name) to resolve network and protocol design problems that faced the precursor to the Internet [\[Allman 2011\]](#). RFCs tend to be quite technical and detailed. They define protocols such as TCP, IP, HTTP (for the Web), and SMTP (for e-mail). There are currently more than 7,000 RFCs. Other bodies also specify standards for network components, most notably for network links. The IEEE 802 LAN/MAN Standards Committee [\[IEEE 802 2016\]](#), for example, specifies the Ethernet and wireless WiFi standards.

### 1.1.2 A Services Description

Our discussion above has identified many of the pieces that make up the Internet. But we can also describe the Internet from an entirely different angle—namely, as *an infrastructure that provides services to applications*. In addition to traditional applications such as e-mail and Web surfing, Internet applications include mobile smartphone and tablet applications, including Internet messaging, mapping with real-time road-traffic information, music streaming from the cloud, movie and television streaming, online social networks, video conferencing, multi-person games, and location-based recommendation systems. The applications are said to be **distributed applications**, since they involve multiple end systems that exchange data with each other. Importantly, Internet applications run on end systems—they do not run in the packet switches in the network core. Although packet switches facilitate the exchange of data among end systems, they are not concerned with the application that is the source or sink of data.

Let's explore a little more what we mean by an infrastructure that provides services to applications. To this end, suppose you have an exciting new idea for a distributed Internet application, one that may greatly benefit humanity or one that may simply make you rich and famous. How might you go about

transforming this idea into an actual Internet application? Because applications run on end systems, you are going to need to write programs that run on the end systems. You might, for example, write your programs in Java, C, or Python. Now, because you are developing a distributed Internet application, the programs running on the different end systems will need to send data to each other. And here we get to a central issue—one that leads to the alternative way of describing the Internet as a platform for applications. How does one program running on one end system instruct the Internet to deliver data to another program running on another end system?

End systems attached to the Internet provide a **socket interface** that specifies how a program running on one end system asks the Internet infrastructure to deliver data to a specific destination program running on another end system. This Internet socket interface is a set of rules that the sending program must follow so that the Internet can deliver the data to the destination program. We'll discuss the Internet socket interface in detail in [Chapter 2](#). For now, let's draw upon a simple analogy, one that we will frequently use in this book. Suppose Alice wants to send a letter to Bob using the postal service. Alice, of course, can't just write the letter (the data) and drop the letter out her window. Instead, the postal service requires that Alice put the letter in an envelope; write Bob's full name, address, and zip code in the center of the envelope; seal the envelope; put a stamp in the upper-right-hand corner of the envelope; and finally, drop the envelope into an official postal service mailbox. Thus, the postal service has its own "postal service interface," or set of rules, that Alice must follow to have the postal service deliver her letter to Bob. In a similar manner, the Internet has a socket interface that the program sending data must follow to have the Internet deliver the data to the program that will receive the data.

The postal service, of course, provides more than one service to its customers. It provides express delivery, reception confirmation, ordinary use, and many more services. In a similar manner, the Internet provides multiple services to its applications. When you develop an Internet application, you too must choose one of the Internet's services for your application. We'll describe the Internet's services in [Chapter 2](#).

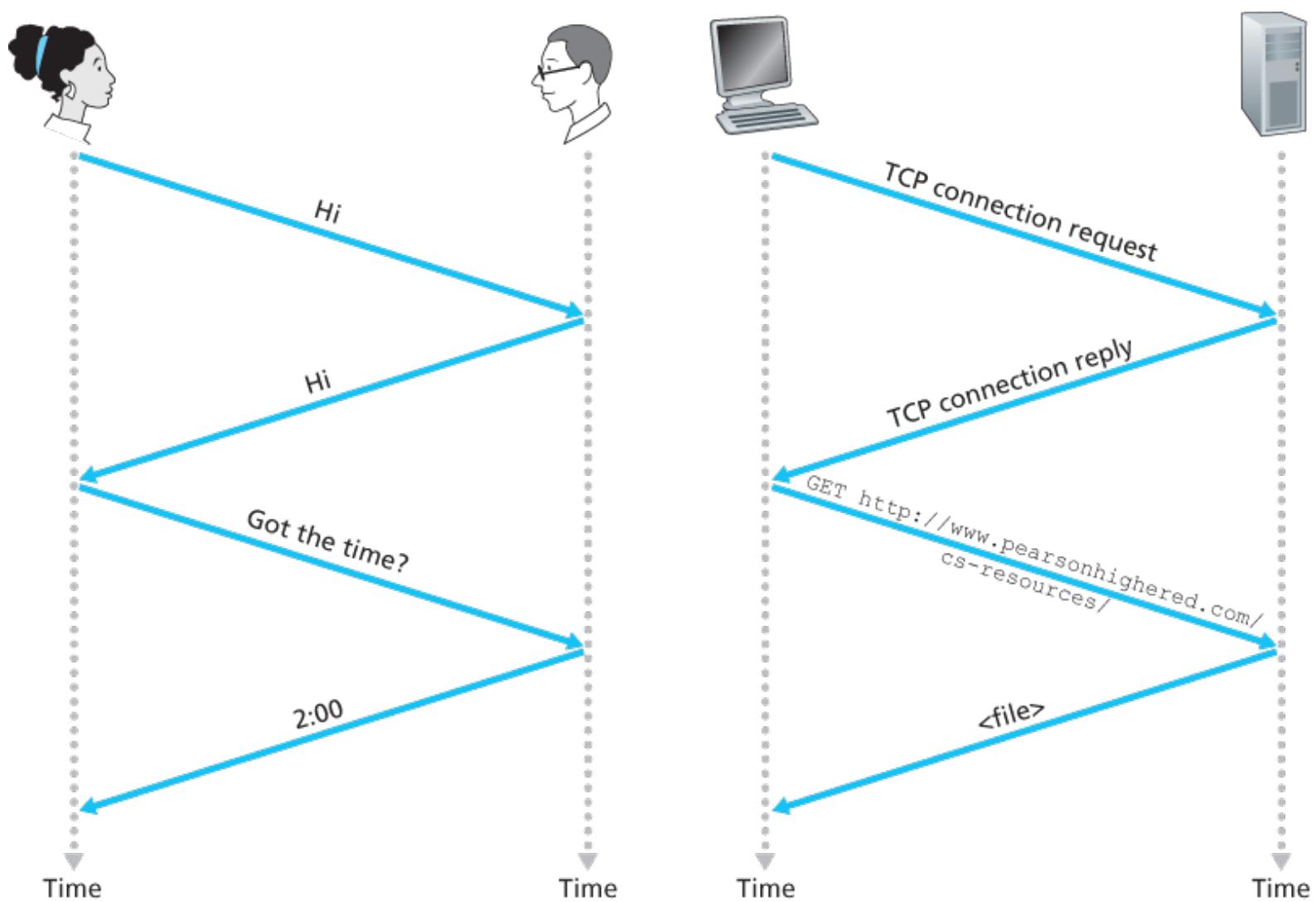
We have just given two descriptions of the Internet; one in terms of its hardware and software components, the other in terms of an infrastructure for providing services to distributed applications. But perhaps you are still confused as to what the Internet is. What are packet switching and TCP/IP? What are routers? What kinds of communication links are present in the Internet? What is a distributed application? How can a thermostat or body scale be attached to the Internet? If you feel a bit overwhelmed by all of this now, don't worry—the purpose of this book is to introduce you to both the nuts and bolts of the Internet and the principles that govern how and why it works. We'll explain these important terms and questions in the following sections and chapters.

### 1.1.3 What Is a Protocol?

Now that we've got a bit of a feel for what the Internet is, let's consider another important buzzword in computer networking: *protocol*. What is a protocol? What does a protocol do?

### A Human Analogy

It is probably easiest to understand the notion of a computer network protocol by first considering some human analogies, since we humans execute protocols all of the time. Consider what you do when you want to ask someone for the time of day. A typical exchange is shown in **Figure 1.2**. Human protocol (or good manners, at least) dictates that one first offer a greeting (the first “Hi” in **Figure 1.2**) to initiate communication with someone else. The typical response to a “Hi” is a returned “Hi” message. Implicitly, one then takes a cordial “Hi” response as an indication that one can proceed and ask for the time of day. A different response to the initial “Hi” (such as “Don’t bother me!” or “I don’t speak English,” or some unprintable reply) might



**Figure 1.2** A human protocol and a computer network protocol

indicate an unwillingness or inability to communicate. In this case, the human protocol would be not to ask for the time of day. Sometimes one gets no response at all to a question, in which case one typically gives up asking that person for the time. Note that in our human protocol, *there are specific messages*

*we send, and specific actions we take in response to the received reply messages or other events* (such as no reply within some given amount of time). Clearly, transmitted and received messages, and actions taken when these messages are sent or received or other events occur, play a central role in a human protocol. If people run different protocols (for example, if one person has manners but the other does not, or if one understands the concept of time and the other does not) the protocols do not interoperate and no useful work can be accomplished. The same is true in networking—it takes two (or more) communicating entities running the same protocol in order to accomplish a task.

Let's consider a second human analogy. Suppose you're in a college class (a computer networking class, for example!). The teacher is droning on about protocols and you're confused. The teacher stops to ask, "Are there any questions?" (a message that is transmitted to, and received by, all students who are not sleeping). You raise your hand (transmitting an implicit message to the teacher). Your teacher acknowledges you with a smile, saying "Yes . . ." (a transmitted message encouraging you to ask your question—teachers *love* to be asked questions), and you then ask your question (that is, transmit your message to your teacher). Your teacher hears your question (receives your question message) and answers (transmits a reply to you). Once again, we see that the transmission and receipt of messages, and a set of conventional actions taken when these messages are sent and received, are at the heart of this question-and-answer protocol.

### *Network Protocols*

A network protocol is similar to a human protocol, except that the entities exchanging messages and taking actions are hardware or software components of some device (for example, computer, smartphone, tablet, router, or other network-capable device). All activity in the Internet that involves two or more communicating remote entities is governed by a protocol. For example, hardware-implemented protocols in two physically connected computers control the flow of bits on the "wire" between the two network interface cards; congestion-control protocols in end systems control the rate at which packets are transmitted between sender and receiver; protocols in routers determine a packet's path from source to destination. Protocols are running everywhere in the Internet, and consequently much of this book is about computer network protocols.

As an example of a computer network protocol with which you are probably familiar, consider what happens when you make a request to a Web server, that is, when you type the URL of a Web page into your Web browser. The scenario is illustrated in the right half of **Figure 1.2**. First, your computer will send a connection request message to the Web server and wait for a reply. The Web server will eventually receive your connection request message and return a connection reply message. Knowing that it is now OK to request the Web document, your computer then sends the name of the Web page it wants to fetch from that Web server in a GET message. Finally, the Web server returns the Web page (file) to your computer.

Given the human and networking examples above, the exchange of messages and the actions taken when these messages are sent and received are the key defining elements of a protocol:

*A **protocol** defines the format and the order of messages exchanged between two or more communicating entities, as well as the actions taken on the transmission and/or receipt of a message or other event.*

The Internet, and computer networks in general, make extensive use of protocols. Different protocols are used to accomplish different communication tasks. As you read through this book, you will learn that some protocols are simple and straightforward, while others are complex and intellectually deep. Mastering the field of computer networking is equivalent to understanding the what, why, and how of networking protocols.

## 1.2 The Network Edge

In the previous section we presented a high-level overview of the Internet and networking protocols. We are now going to delve a bit more deeply into the components of a computer network (and the Internet, in particular). We begin in this section at the edge of a network and look at the components with which we are most familiar—namely, the computers, smartphones and other devices that we use on a daily basis. In the next section we'll move from the network edge to the network core and examine switching and routing in computer networks.

Recall from the previous section that in computer networking jargon, the computers and other devices connected to the Internet are often referred to as end systems. They are referred to as end systems because they sit at the edge of the Internet, as shown in [Figure 1.3](#). The Internet's end systems include desktop computers (e.g., desktop PCs, Macs, and Linux boxes), servers (e.g., Web and e-mail servers), and mobile devices (e.g., laptops, smartphones, and tablets). Furthermore, an increasing number of non-traditional “things” are being attached to the Internet as end systems (see the Case History feature).

End systems are also referred to as *hosts* because they host (that is, run) application programs such as a Web browser program, a Web server program, an e-mail client program, or an e-mail server program. Throughout this book we will use the

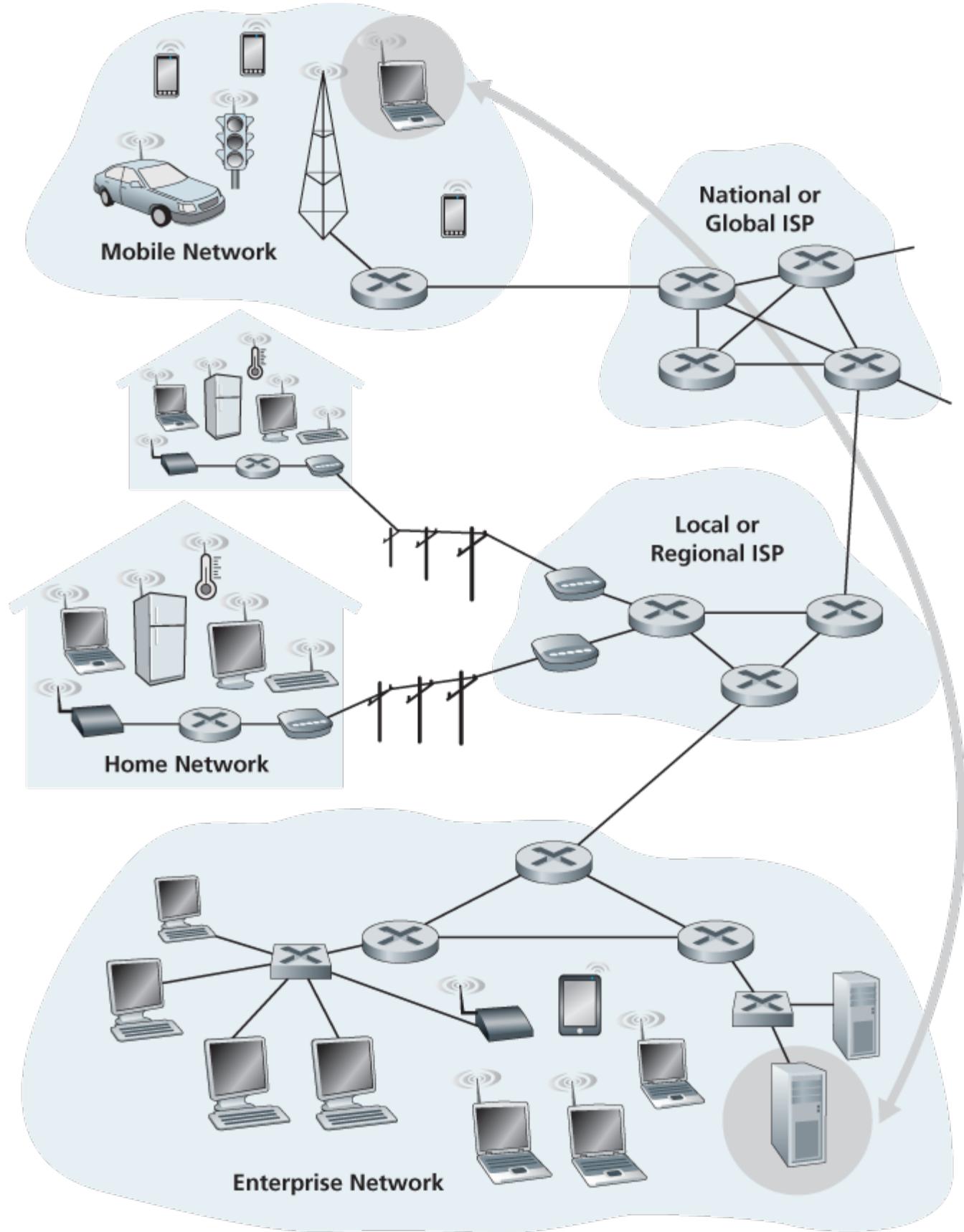


Figure 1.3 End-system interaction

#### CASE HISTORY

#### THE INTERNET OF THINGS

Can you imagine a world in which just about everything is wirelessly connected to the Internet? A world in which most people, cars, bicycles, eye glasses, watches, toys, hospital equipment, home sensors, classrooms, video surveillance systems, atmospheric sensors, store-shelf

products, and pets are connected? This world of the Internet of Things (IoT) may actually be just around the corner.

By some estimates, as of 2015 there are already 5 billion things connected to the Internet, and the number could reach 25 billion by 2020 [[Gartner 2014](#)]. These things include our smartphones, which already follow us around in our homes, offices, and cars, reporting our geo-locations and usage data to our ISPs and Internet applications. But in addition to our smartphones, a wide-variety of non-traditional “things” are already available as products. For example, there are Internet-connected wearables, including watches (from Apple and many others) and eye glasses. Internet-connected glasses can, for example, upload everything we see to the cloud, allowing us to share our visual experiences with people around the world in real-time. There are Internet-connected things already available for the smart home, including Internet-connected thermostats that can be controlled remotely from our smartphones, and Internet-connected body scales, enabling us to graphically review the progress of our diets from our smartphones. There are Internet-connected toys, including dolls that recognize and interpret a child’s speech and respond appropriately.

The IoT offers potentially revolutionary benefits to users. But at the same time there are also huge security and privacy risks. For example, attackers, via the Internet, might be able to hack into IoT devices or into the servers collecting data from IoT devices. For example, an attacker could hijack an Internet-connected doll and talk directly with a child; or an attacker could hack into a database that stores personal health and activity information collected from wearable devices. These security and privacy concerns could undermine the consumer confidence necessary for the technologies to meet their full potential and may result in less widespread adoption [[FTC 2015](#)].

terms hosts and end systems interchangeably; that is, *host = end system*. Hosts are sometimes further divided into two categories: **clients** and **servers**. Informally, clients tend to be desktop and mobile PCs, smartphones, and so on, whereas servers tend to be more powerful machines that store and distribute Web pages, stream video, relay e-mail, and so on. Today, most of the servers from which we receive search results, e-mail, Web pages, and videos reside in large **data centers**. For example, Google has 50-100 data centers, including about 15 large centers, each with more than 100,000 servers.

### 1.2.1 Access Networks

Having considered the applications and end systems at the “edge of the network,” let’s next consider the access network—the network that physically connects an end system to the first router (also known as the “edge router”) on a path from the end system to any other distant end system. [Figure 1.4](#) shows several types of access

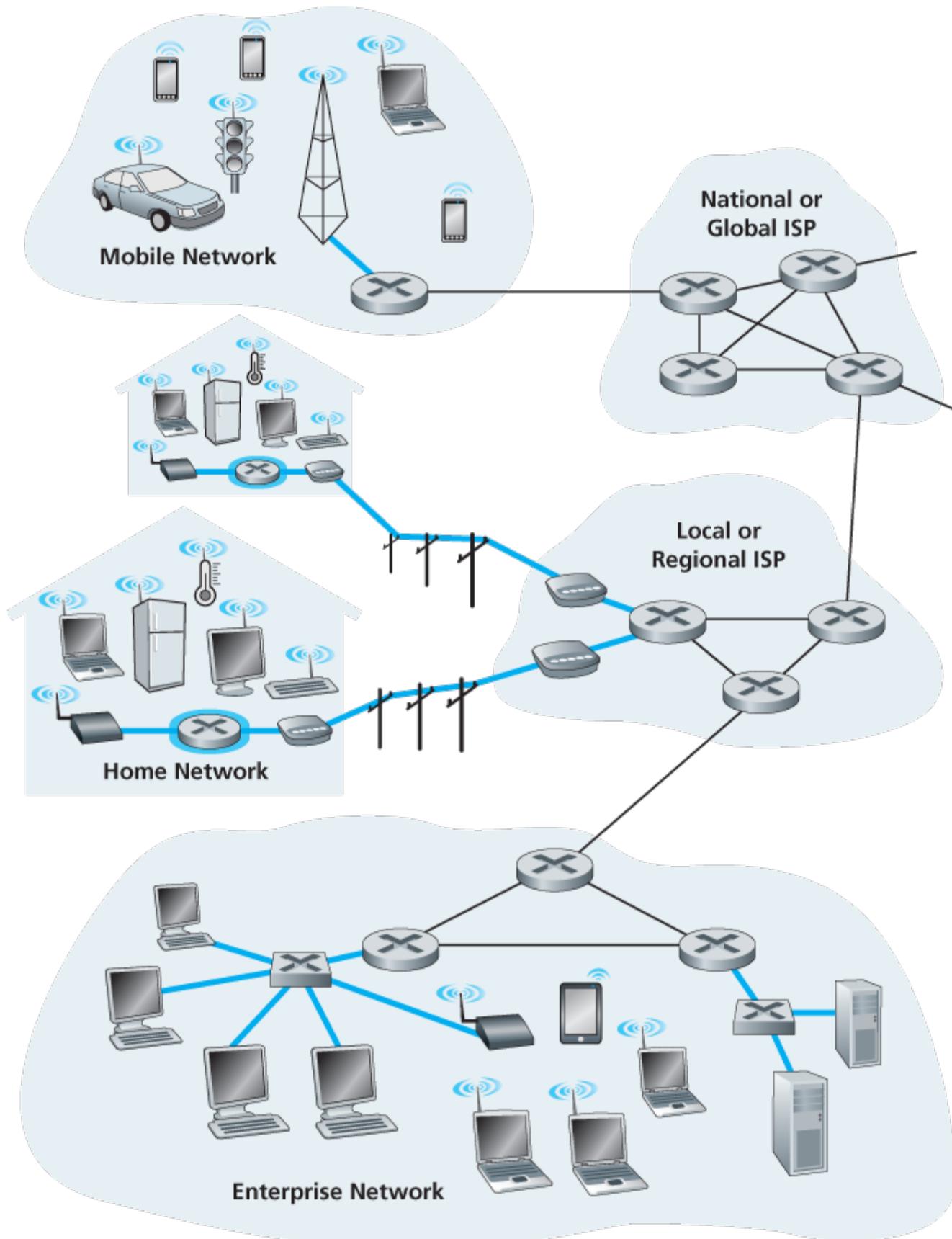


Figure 1.4 Access networks

networks with thick, shaded lines and the settings (home, enterprise, and wide-area mobile wireless) in which they are used.

*Home Access: DSL, Cable, FTTH, Dial-Up, and Satellite*

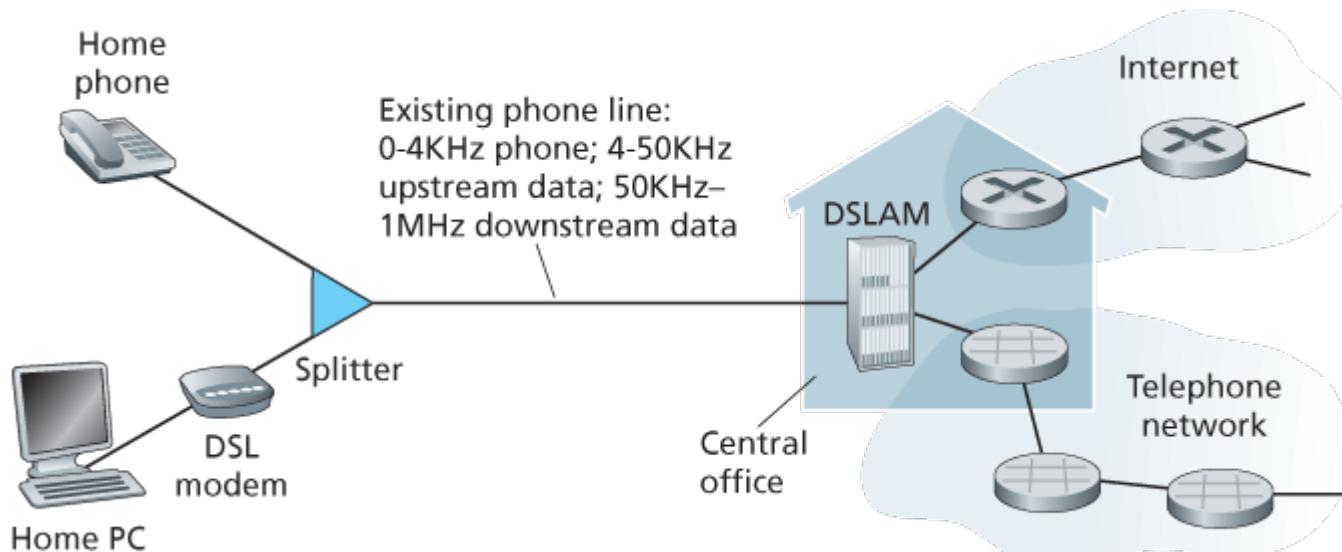
In developed countries as of 2014, more than 78 percent of the households have Internet access, with Korea, Netherlands, Finland, and Sweden leading the way with more than 80 percent of households having Internet access, almost all via a high-speed broadband connection [ITU 2015]. Given this widespread use of home access networks let's begin our overview of access networks by considering how homes connect to the Internet.

Today, the two most prevalent types of broadband residential access are **digital subscriber line (DSL)** and cable. A residence typically obtains DSL Internet access from the same local telephone company (telco) that provides its wired local phone access. Thus, when DSL is used, a customer's telco is also its ISP. As shown in **Figure 1.5**, each customer's DSL modem uses the existing telephone line (twisted-pair copper wire, which we'll discuss in **Section 1.2.2**) to exchange data with a digital subscriber line access multiplexer (DSLAM) located in the telco's local central office (CO). The home's DSL modem takes digital data and translates it to high-frequency tones for transmission over telephone wires to the CO; the analog signals from many such houses are translated back into digital format at the DSLAM.

The residential telephone line carries both data and traditional telephone signals simultaneously, which are encoded at different frequencies:

- A high-speed downstream channel, in the 50 kHz to 1 MHz band
- A medium-speed upstream channel, in the 4 kHz to 50 kHz band
- An ordinary two-way telephone channel, in the 0 to 4 kHz band

This approach makes the single DSL link appear as if there were three separate links, so that a telephone call and an Internet connection can share the DSL link at the same time.



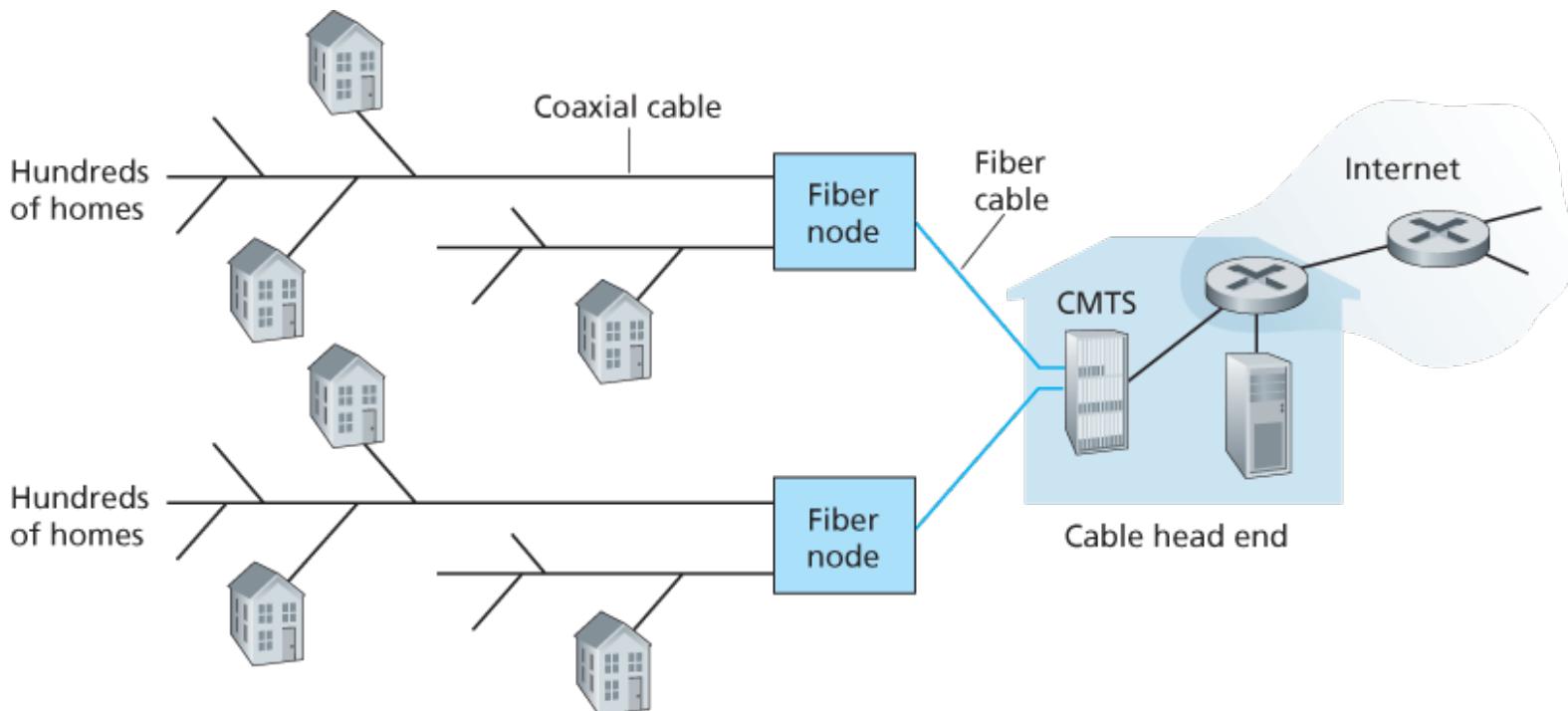
**Figure 1.5** DSL Internet access

(We'll describe this technique of frequency-division multiplexing in **Section 1.3.1**.) On the customer side, a splitter separates the data and telephone signals arriving to the home and forwards the data signal to

the DSL modem. On the telco side, in the CO, the DSLAM separates the data and phone signals and sends the data into the Internet. Hundreds or even thousands of households connect to a single DSLAM [Dischinger 2007].

The DSL standards define multiple transmission rates, including 12 Mbps downstream and 1.8 Mbps upstream [ITU 1999], and 55 Mbps downstream and 15 Mbps upstream [ITU 2006]. Because the downstream and upstream rates are different, the access is said to be asymmetric. The actual downstream and upstream transmission rates achieved may be less than the rates noted above, as the DSL provider may purposefully limit a residential rate when tiered service (different rates, available at different prices) are offered. The maximum rate is also limited by the distance between the home and the CO, the gauge of the twisted-pair line and the degree of electrical interference. Engineers have expressly designed DSL for short distances between the home and the CO; generally, if the residence is not located within 5 to 10 miles of the CO, the residence must resort to an alternative form of Internet access.

While DSL makes use of the telco's existing local telephone infrastructure, **cable Internet access** makes use of the cable television company's existing cable television infrastructure. A residence obtains cable Internet access from the same company that provides its cable television. As illustrated in **Figure 1.6**, fiber optics connect the cable head end to neighborhood-level junctions, from which traditional coaxial cable is then used to reach individual houses and apartments. Each neighborhood junction typically supports 500 to 5,000 homes. Because both fiber and coaxial cable are employed in this system, it is often referred to as hybrid fiber coax (HFC).



**Figure 1.6** A hybrid fiber-coaxial access network

Cable internet access requires special modems, called cable modems. As with a DSL modem, the cable

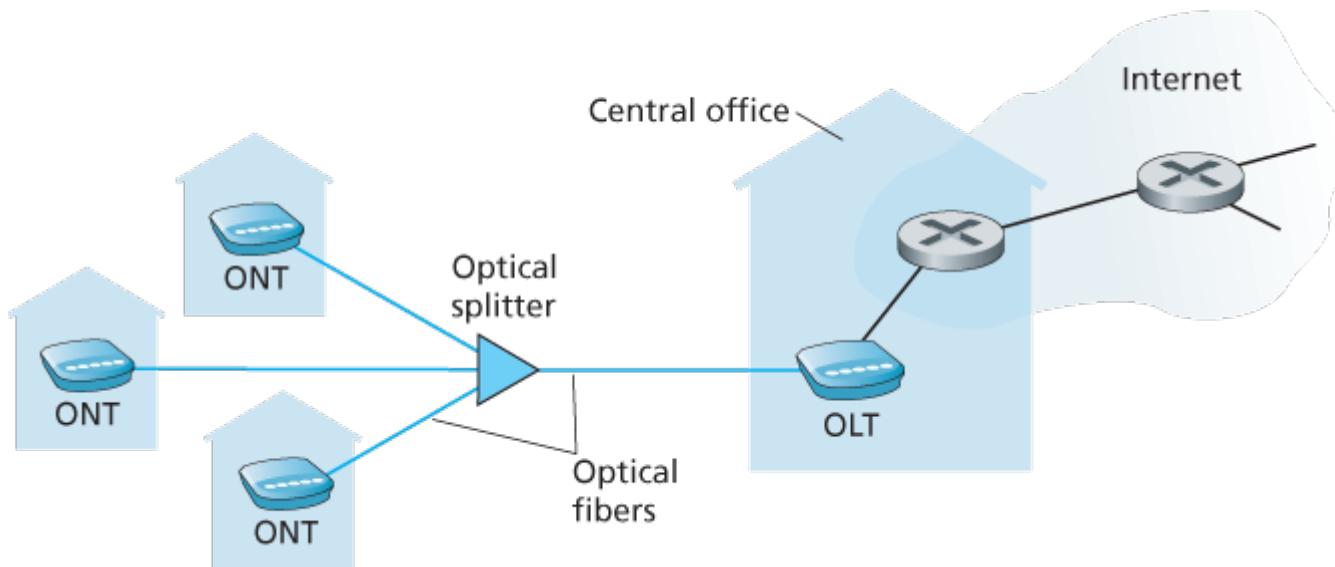
modem is typically an external device and connects to the home PC through an Ethernet port. (We will discuss Ethernet in great detail in [Chapter 6](#).) At the cable head end, the cable modem termination system (CMTS) serves a similar function as the DSL network's DSLAM—turning the analog signal sent from the cable modems in many downstream homes back into digital format. Cable modems divide the HFC network into two channels, a downstream and an upstream channel. As with DSL, access is typically asymmetric, with the downstream channel typically allocated a higher transmission rate than the upstream channel. The DOCSIS 2.0 standard defines downstream rates up to 42.8 Mbps and upstream rates of up to 30.7 Mbps. As in the case of DSL networks, the maximum achievable rate may not be realized due to lower contracted data rates or media impairments.

One important characteristic of cable Internet access is that it is a shared broadcast medium. In particular, every packet sent by the head end travels downstream on every link to every home and every packet sent by a home travels on the upstream channel to the head end. For this reason, if several users are simultaneously downloading a video file on the downstream channel, the actual rate at which each user receives its video file will be significantly lower than the aggregate cable downstream rate. On the other hand, if there are only a few active users and they are all Web surfing, then each of the users may actually receive Web pages at the full cable downstream rate, because the users will rarely request a Web page at exactly the same time. Because the upstream channel is also shared, a distributed multiple access protocol is needed to coordinate transmissions and avoid collisions. (We'll discuss this collision issue in some detail in [Chapter 6](#).)

Although DSL and cable networks currently represent more than 85 percent of residential broadband access in the United States, an up-and-coming technology that provides even higher speeds is [fiber to the home \(FTTH\)](#) [[FTTH Council 2016](#)]. As the name suggests, the FTTH concept is simple—provide an optical fiber path from the CO directly to the home. Many countries today—including the UAE, South Korea, Hong Kong, Japan, Singapore, Taiwan, Lithuania, and Sweden—now have household penetration rates exceeding 30% [[FTTH Council 2016](#)].

There are several competing technologies for optical distribution from the CO to the homes. The simplest optical distribution network is called direct fiber, with one fiber leaving the CO for each home. More commonly, each fiber leaving the central office is actually shared by many homes; it is not until the fiber gets relatively close to the homes that it is split into individual customer-specific fibers. There are two competing optical-distribution network architectures that perform this splitting: active optical networks (AONs) and passive optical networks (PONs). AON is essentially switched Ethernet, which is discussed in [Chapter 6](#).

Here, we briefly discuss PON, which is used in Verizon's FIOS service. [Figure 1.7](#) shows FTTH using the PON distribution architecture. Each home has an optical network terminator (ONT), which is connected by dedicated optical fiber to a neighborhood splitter. The splitter combines a number of homes (typically less



**Figure 1.7 FTTH Internet access**

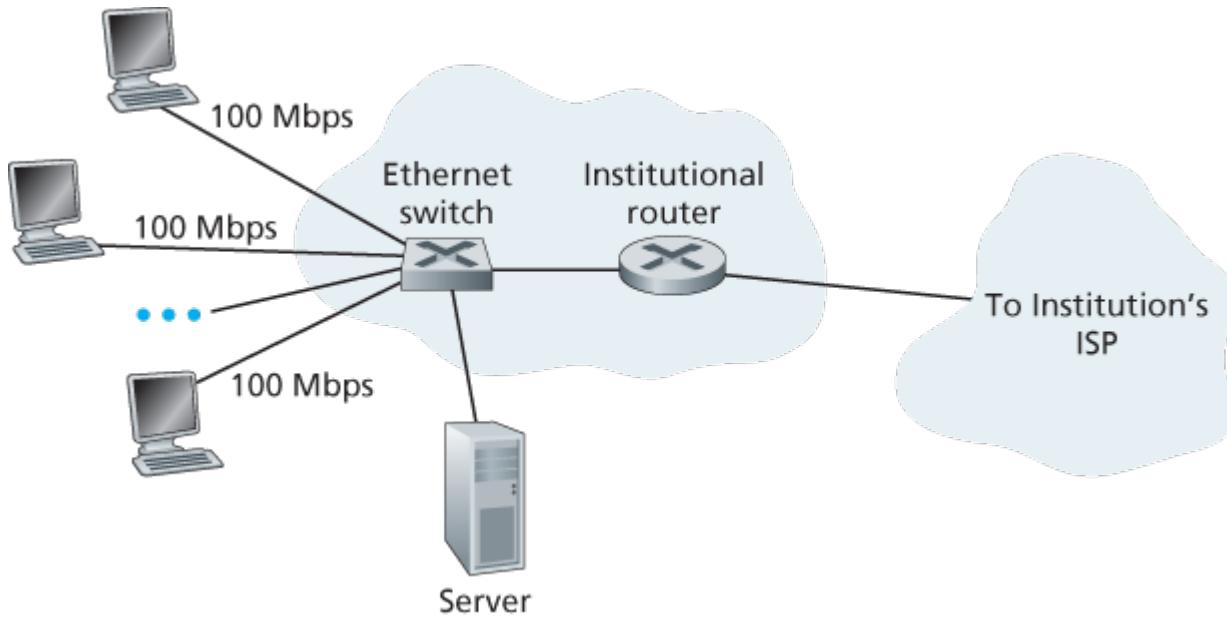
than 100) onto a single, shared optical fiber, which connects to an optical line terminator (OLT) in the telco's CO. The OLT, providing conversion between optical and electrical signals, connects to the Internet via a telco router. In the home, users connect a home router (typically a wireless router) to the ONT and access the Internet via this home router. In the PON architecture, all packets sent from OLT to the splitter are replicated at the splitter (similar to a cable head end).

FTTH can potentially provide Internet access rates in the gigabits per second range. However, most FTTH ISPs provide different rate offerings, with the higher rates naturally costing more money. The average downstream speed of US FTTH customers was approximately 20 Mbps in 2011 (compared with 13 Mbps for cable access networks and less than 5 Mbps for DSL) [FTTH Council 2011b].

Two other access network technologies are also used to provide Internet access to the home. In locations where DSL, cable, and FTTH are not available (e.g., in some rural settings), a satellite link can be used to connect a residence to the Internet at speeds of more than 1 Mbps; StarBand and HughesNet are two such satellite access providers. Dial-up access over traditional phone lines is based on the same model as DSL—a home modem connects over a phone line to a modem in the ISP. Compared with DSL and other broadband access networks, dial-up access is excruciatingly slow at 56 kbps.

#### *Access in the Enterprise (and the Home): Ethernet and WiFi*

On corporate and university campuses, and increasingly in home settings, a local area network (LAN) is used to connect an end system to the edge router. Although there are many types of LAN technologies, Ethernet is by far the most prevalent access technology in corporate, university, and home networks. As shown in [Figure 1.8](#), Ethernet users use twisted-pair copper wire to connect to an Ethernet switch, a technology discussed in detail in [Chapter 6](#). The Ethernet switch, or a network of such



**Figure 1.8 Ethernet Internet access**

interconnected switches, is then in turn connected into the larger Internet. With Ethernet access, users typically have 100 Mbps or 1 Gbps access to the Ethernet switch, whereas servers may have 1 Gbps or even 10 Gbps access.

Increasingly, however, people are accessing the Internet wirelessly from laptops, smartphones, tablets, and other “things” (see earlier sidebar on “[Internet of Things](#)”). In a wireless LAN setting, wireless users transmit/receive packets to/from an access point that is connected into the enterprise’s network (most likely using wired Ethernet), which in turn is connected to the wired Internet. A wireless LAN user must typically be within a few tens of meters of the access point. Wireless LAN access based on IEEE 802.11 technology, more colloquially known as WiFi, is now just about everywhere—universities, business offices, cafes, airports, homes, and even in airplanes. In many cities, one can stand on a street corner and be within range of ten or twenty base stations (for a browseable global map of 802.11 base stations that have been discovered and logged on a Web site by people who take great enjoyment in doing such things, see [\[wigle.net 2016\]](#)). As discussed in detail in [Chapter 7](#), 802.11 today provides a shared transmission rate of up to more than 100 Mbps.

Even though Ethernet and WiFi access networks were initially deployed in enterprise (corporate, university) settings, they have recently become relatively common components of home networks. Many homes combine broadband residential access (that is, cable modems or DSL) with these inexpensive wireless LAN technologies to create powerful home networks [\[Edwards 2011\]](#). [Figure 1.9](#) shows a typical home network. This home network consists of a roaming laptop as well as a wired PC; a base station (the wireless access point), which communicates with the wireless PC and other wireless devices in the home; a cable modem, providing broadband access to the Internet; and a router, which interconnects the base station and the stationary PC with the cable modem. This network allows household members to have broadband access to the Internet with one member roaming from the

kitchen to the backyard to the bedrooms.

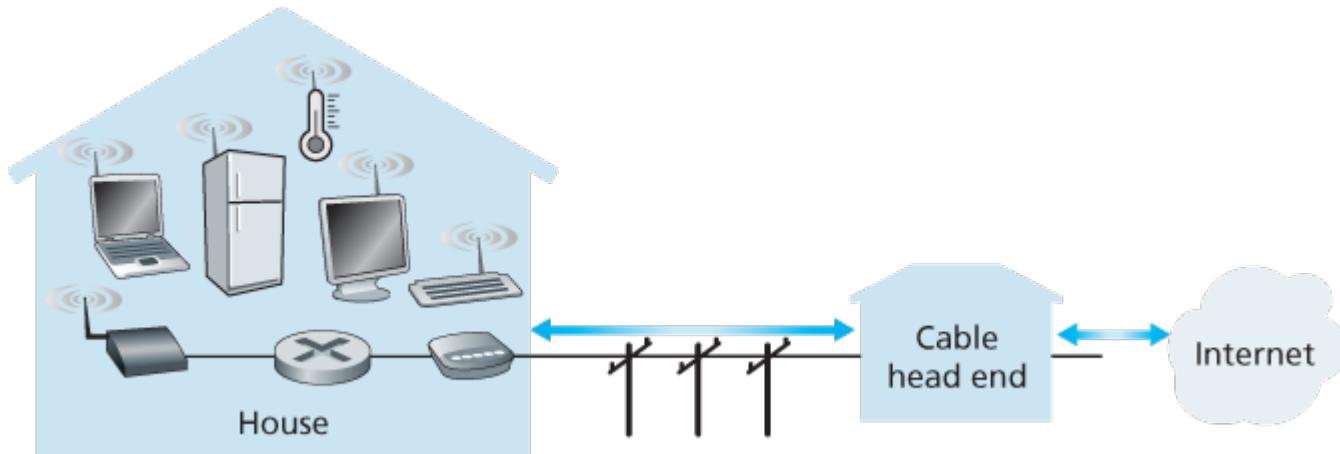


Figure 1.9 A typical home network

#### Wide-Area Wireless Access: 3G and LTE

Increasingly, devices such as iPhones and Android devices are being used to message, share photos in social networks, watch movies, and stream music while on the run. These devices employ the same wireless infrastructure used for cellular telephony to send/receive packets through a base station that is operated by the cellular network provider. Unlike WiFi, a user need only be within a few tens of kilometers (as opposed to a few tens of meters) of the base station.

Telecommunications companies have made enormous investments in so-called third-generation (3G) wireless, which provides packet-switched wide-area wireless Internet access at speeds in excess of 1 Mbps. But even higher-speed wide-area access technologies—a fourth-generation (4G) of wide-area wireless networks—are already being deployed. LTE (for “Long-Term Evolution”—a candidate for Bad Acronym of the Year Award) has its roots in 3G technology, and can achieve rates in excess of 10 Mbps. LTE downstream rates of many tens of Mbps have been reported in commercial deployments. We'll cover the basic principles of wireless networks and mobility, as well as WiFi, 3G, and LTE technologies (and more!) in [Chapter 7](#).

#### 1.2.2 Physical Media

In the previous subsection, we gave an overview of some of the most important network access technologies in the Internet. As we described these technologies, we also indicated the physical media used. For example, we said that HFC uses a combination of fiber cable and coaxial cable. We said that DSL and Ethernet use copper wire. And we said that mobile access networks use the radio spectrum. In this subsection we provide a brief overview of these and other transmission media that are commonly used in the Internet.

In order to define what is meant by a physical medium, let us reflect on the brief life of a bit. Consider a bit traveling from one end system, through a series of links and routers, to another end system. This poor bit gets kicked around and transmitted many, many times! The source end system first transmits the bit, and shortly thereafter the first router in the series receives the bit; the first router then transmits the bit, and shortly thereafter the second router receives the bit; and so on. Thus our bit, when traveling from source to destination, passes through a series of transmitter-receiver pairs. For each transmitter-receiver pair, the bit is sent by propagating electromagnetic waves or optical pulses across a **physical medium**. The physical medium can take many shapes and forms and does not have to be of the same type for each transmitter-receiver pair along the path. Examples of physical media include twisted-pair copper wire, coaxial cable, multimode fiber-optic cable, terrestrial radio spectrum, and satellite radio spectrum. Physical media fall into two categories: **guided media** and **unguided media**. With guided media, the waves are guided along a solid medium, such as a fiber-optic cable, a twisted-pair copper wire, or a coaxial cable. With unguided media, the waves propagate in the atmosphere and in outer space, such as in a wireless LAN or a digital satellite channel.

But before we get into the characteristics of the various media types, let us say a few words about their costs. The actual cost of the physical link (copper wire, fiber-optic cable, and so on) is often relatively minor compared with other networking costs. In particular, the labor cost associated with the installation of the physical link can be orders of magnitude higher than the cost of the material. For this reason, many builders install twisted pair, optical fiber, and coaxial cable in every room in a building. Even if only one medium is initially used, there is a good chance that another medium could be used in the near future, and so money is saved by not having to lay additional wires in the future.

### *Twisted-Pair Copper Wire*

The least expensive and most commonly used guided transmission medium is twisted-pair copper wire. For over a hundred years it has been used by telephone networks. In fact, more than 99 percent of the wired connections from the telephone handset to the local telephone switch use twisted-pair copper wire. Most of us have seen twisted pair in our homes (or those of our parents or grandparents!) and work environments. Twisted pair consists of two insulated copper wires, each about 1 mm thick, arranged in a regular spiral pattern. The wires are twisted together to reduce the electrical interference from similar pairs close by. Typically, a number of pairs are bundled together in a cable by wrapping the pairs in a protective shield. A wire pair constitutes a single communication link. **Unshielded twisted pair (UTP)** is commonly used for computer networks within a building, that is, for LANs. Data rates for LANs using twisted pair today range from 10 Mbps to 10 Gbps. The data rates that can be achieved depend on the thickness of the wire and the distance between transmitter and receiver.

When fiber-optic technology emerged in the 1980s, many people disparaged twisted pair because of its relatively low bit rates. Some people even felt that fiber-optic technology would completely replace twisted pair. But twisted pair did not give up so easily. Modern twisted-pair technology, such as category

6a cable, can achieve data rates of 10 Gbps for distances up to a hundred meters. In the end, twisted pair has emerged as the dominant solution for high-speed LAN networking.

As discussed earlier, twisted pair is also commonly used for residential Internet access. We saw that dial-up modem technology enables access at rates of up to 56 kbps over twisted pair. We also saw that DSL (digital subscriber line) technology has enabled residential users to access the Internet at tens of Mbps over twisted pair (when users live close to the ISP's central office).

### *Coaxial Cable*

Like twisted pair, coaxial cable consists of two copper conductors, but the two conductors are concentric rather than parallel. With this construction and special insulation and shielding, coaxial cable can achieve high data transmission rates. Coaxial cable is quite common in cable television systems. As we saw earlier, cable television systems have recently been coupled with cable modems to provide residential users with Internet access at rates of tens of Mbps. In cable television and cable Internet access, the transmitter shifts the digital signal to a specific frequency band, and the resulting analog signal is sent from the transmitter to one or more receivers. Coaxial cable can be used as a guided **shared medium**. Specifically, a number of end systems can be connected directly to the cable, with each of the end systems receiving whatever is sent by the other end systems.

### *Fiber Optics*

An optical fiber is a thin, flexible medium that conducts pulses of light, with each pulse representing a bit. A single optical fiber can support tremendous bit rates, up to tens or even hundreds of gigabits per second. They are immune to electromagnetic interference, have very low signal attenuation up to 100 kilometers, and are very hard to tap. These characteristics have made fiber optics the preferred long-haul guided transmission media, particularly for overseas links. Many of the long-distance telephone networks in the United States and elsewhere now use fiber optics exclusively. Fiber optics is also prevalent in the backbone of the Internet. However, the high cost of optical devices—such as transmitters, receivers, and switches—has hindered their deployment for short-haul transport, such as in a LAN or into the home in a residential access network. The Optical Carrier (OC) standard link speeds range from 51.8 Mbps to 39.8 Gbps; these specifications are often referred to as OC- $n$ , where the link speed equals  $n \times 51.8$  Mbps. Standards in use today include OC-1, OC-3, OC-12, OC-24, OC-48, OC-96, OC-192, OC-768. [\[Mukherjee 2006, Ramaswami 2010\]](#) provide coverage of various aspects of optical networking.

### *Terrestrial Radio Channels*

Radio channels carry signals in the electromagnetic spectrum. They are an attractive medium because they require no physical wire to be installed, can penetrate walls, provide connectivity to a mobile user,

and can potentially carry a signal for long distances. The characteristics of a radio channel depend significantly on the propagation environment and the distance over which a signal is to be carried. Environmental considerations determine path loss and shadow fading (which decrease the signal strength as the signal travels over a distance and around/through obstructing objects), multipath fading (due to signal reflection off of interfering objects), and interference (due to other transmissions and electromagnetic signals).

Terrestrial radio channels can be broadly classified into three groups: those that operate over very short distance (e.g., with one or two meters); those that operate in local areas, typically spanning from ten to a few hundred meters; and those that operate in the wide area, spanning tens of kilometers. Personal devices such as wireless headsets, keyboards, and medical devices operate over short distances; the wireless LAN technologies described in [Section 1.2.1](#) use local-area radio channels; the cellular access technologies use wide-area radio channels. We'll discuss radio channels in detail in [Chapter 7](#).

### *Satellite Radio Channels*

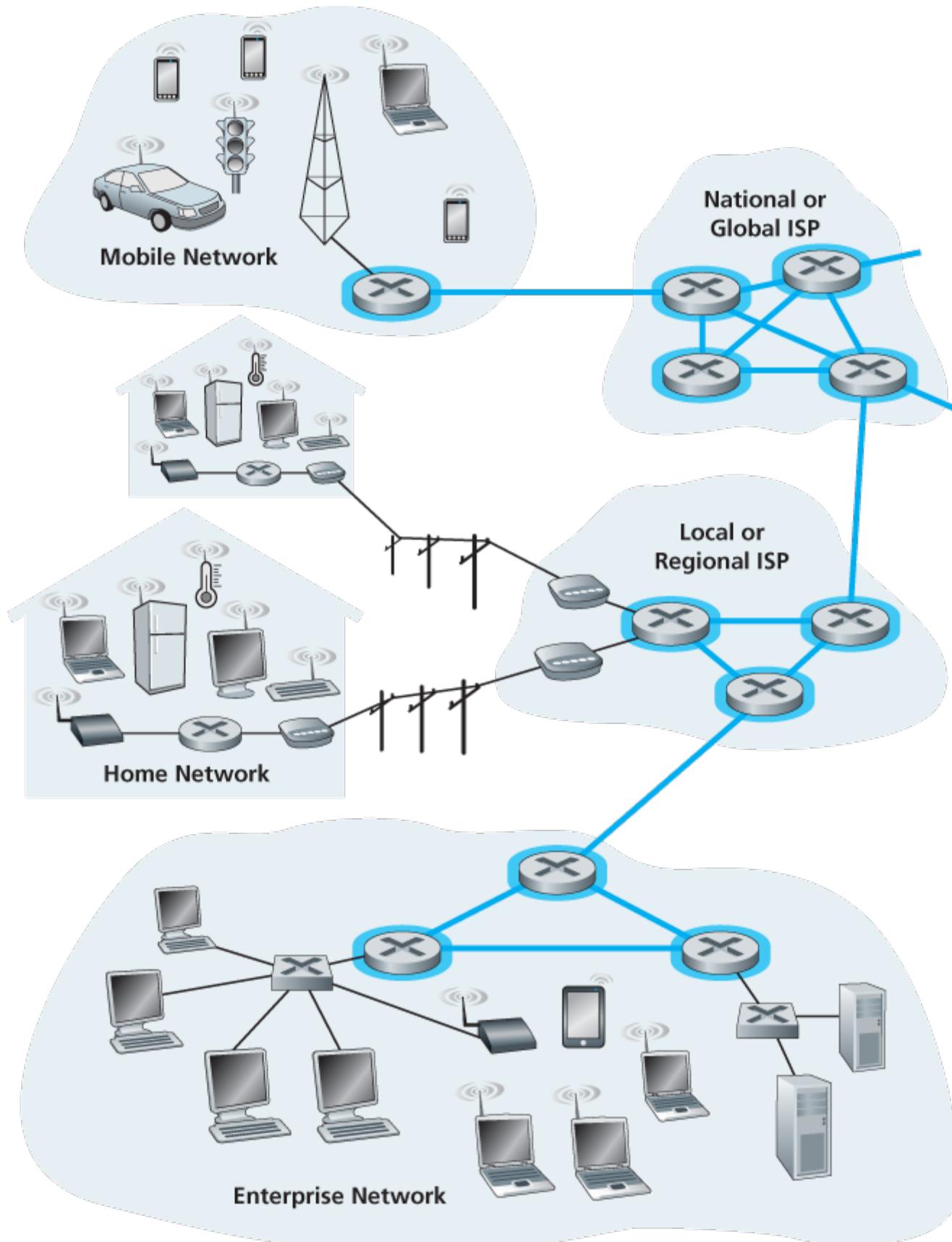
A communication satellite links two or more Earth-based microwave transmitter/ receivers, known as ground stations. The satellite receives transmissions on one frequency band, regenerates the signal using a repeater (discussed below), and transmits the signal on another frequency. Two types of satellites are used in communications: [\*\*geostationary satellites\*\*](#) and [\*\*low-earth orbiting \(LEO\) satellites\*\*](#) [Wiki Satellite 2016].

Geostationary satellites permanently remain above the same spot on Earth. This stationary presence is achieved by placing the satellite in orbit at 36,000 kilometers above Earth's surface. This huge distance from ground station through satellite back to ground station introduces a substantial signal propagation delay of 280 milliseconds. Nevertheless, satellite links, which can operate at speeds of hundreds of Mbps, are often used in areas without access to DSL or cable-based Internet access.

LEO satellites are placed much closer to Earth and do not remain permanently above one spot on Earth. They rotate around Earth (just as the Moon does) and may communicate with each other, as well as with ground stations. To provide continuous coverage to an area, many satellites need to be placed in orbit. There are currently many low-altitude communication systems in development. LEO satellite technology may be used for Internet access sometime in the future.

## 1.3 The Network Core

Having examined the Internet's edge, let us now delve more deeply inside the network core—the mesh of packet switches and links that interconnects the Internet's end systems. [Figure 1.10](#) highlights the network core with thick, shaded lines.



**Figure 1.10** The network core

### 1.3.1 Packet Switching

In a network application, end systems exchange **messages** with each other. Messages can contain anything the application designer wants. Messages may perform a control function (for example, the “Hi” messages in our handshaking example in [Figure 1.2](#)) or can contain data, such as an e-mail message, a JPEG image, or an MP3 audio file. To send a message from a source end system to a destination end system, the source breaks long messages into smaller chunks of data known as **packets**. Between source and destination, each packet travels through communication links and **packet switches** (for which there are two predominant types, **routers** and **link-layer switches**). Packets are transmitted over each communication link at a rate equal to the *full* transmission rate of the link. So, if a source end system or a packet switch is sending a packet of  $L$  bits over a link with transmission rate  $R$  bits/sec, then the time to transmit the packet is  $L/R$  seconds.

#### *Store-and-Forward Transmission*

Most packet switches use **store-and-forward transmission** at the inputs to the links. Store-and-forward transmission means that the packet switch must receive the entire packet before it can begin to transmit the first bit of the packet onto the outbound link. To explore store-and-forward transmission in more detail, consider a simple network consisting of two end systems connected by a single router, as shown in [Figure 1.11](#). A router will typically have many incident links, since its job is to switch an incoming packet onto an outgoing link; in this simple example, the router has the rather simple task of transferring a packet from one (input) link to the only other attached link. In this example, the source has three packets, each consisting of  $L$  bits, to send to the destination. At the snapshot of time shown in [Figure 1.11](#), the source has transmitted some of packet 1, and the front of packet 1 has already arrived at the router. Because the router employs store-and-forwarding, at this instant of time, the router cannot transmit the bits it has received; instead it must first buffer (i.e., “store”) the packet’s bits. Only after the router has received *all* of the packet’s bits can it begin to transmit (i.e., “forward”) the packet onto the outbound link. To gain some insight into store-and-forward transmission, let’s now calculate the amount of time that elapses from when the source begins to send the packet until the destination has received the entire packet. (Here we will ignore propagation delay—the time it takes for the bits to travel across the wire at near the speed of light—which will be discussed in [Section 1.4](#).) The source begins to transmit at time 0; at time  $L/R$  seconds, the source has transmitted the entire packet, and the entire packet has been received and stored at the router (since there is no propagation delay). At time  $L/R$  seconds, since the router has just received the entire packet, it can begin to transmit the packet onto the outbound link towards the destination; at time  $2L/R$ , the router has transmitted the entire packet, and the

entire packet has been received by the destination. Thus, the total delay is  $2L/R$ . If the

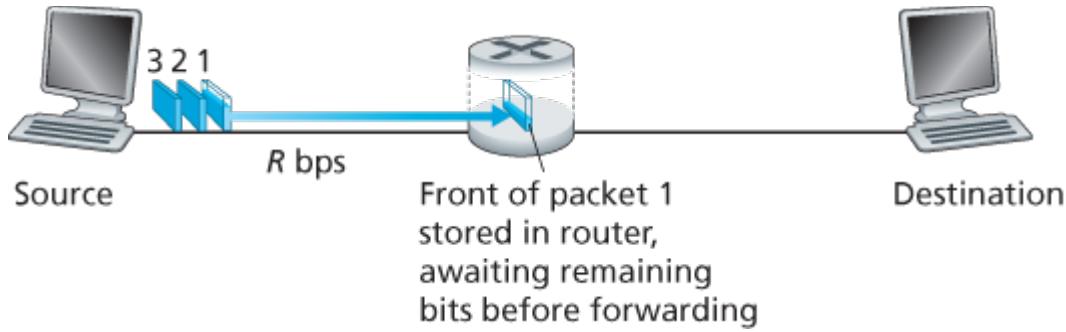


Figure 1.11 Store-and-forward packet switching

switch instead forwarded bits as soon as they arrive (without first receiving the entire packet), then the total delay would be  $L/R$  since bits are not held up at the router. But, as we will discuss in [Section 1.4](#), routers need to receive, store, and *process* the entire packet before forwarding.

Now let's calculate the amount of time that elapses from when the source begins to send the first packet until the destination has received all three packets. As before, at time  $L/R$ , the router begins to forward the first packet. But also at time  $L/R$  the source will begin to send the second packet, since it has just finished sending the entire first packet. Thus, at time  $2L/R$ , the destination has received the first packet and the router has received the second packet. Similarly, at time  $3L/R$ , the destination has received the first two packets and the router has received the third packet. Finally, at time  $4L/R$  the destination has received all three packets!

Let's now consider the general case of sending one packet from source to destination over a path consisting of  $N$  links each of rate  $R$  (thus, there are  $N-1$  routers between source and destination). Applying the same logic as above, we see that the end-to-end delay is:

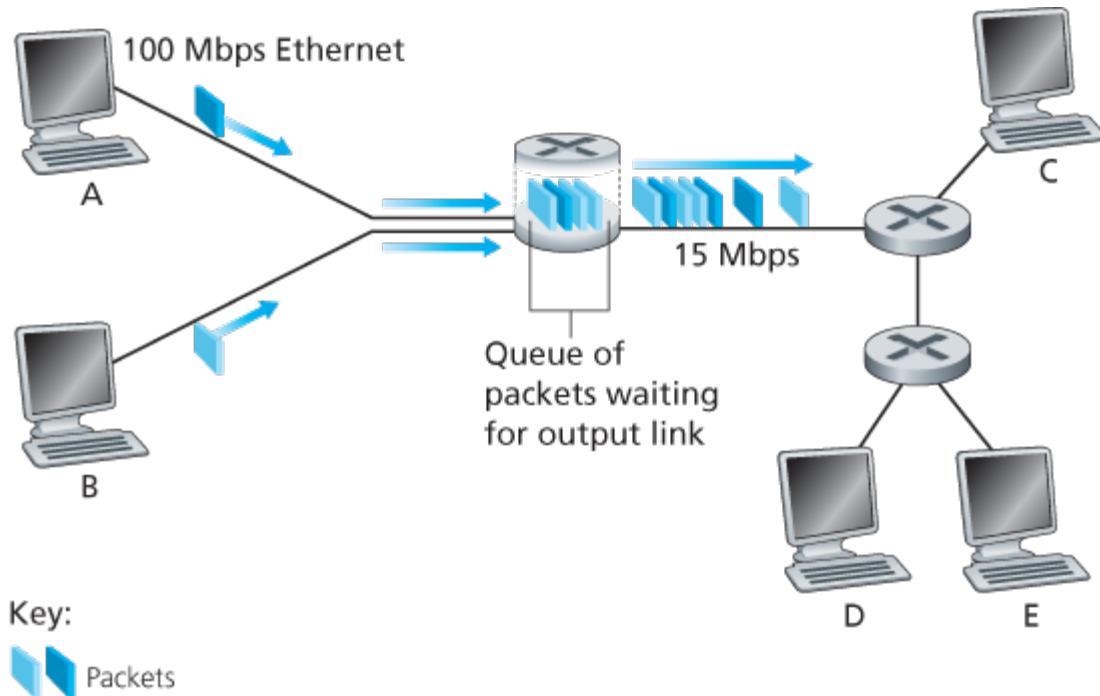
$$\text{end-to-end} = NLR \quad (1.1)$$

You may now want to try to determine what the delay would be for  $P$  packets sent over a series of  $N$  links.

### Queuing Delays and Packet Loss

Each packet switch has multiple links attached to it. For each attached link, the packet switch has an **output buffer** (also called an **output queue**), which stores packets that the router is about to send into that link. The output buffers play a key role in packet switching. If an arriving packet needs to be transmitted onto a link but finds the link busy with the transmission of another packet, the arriving packet must wait in the output buffer. Thus, in addition to the store-and-forward delays, packets suffer output buffer **queuing delays**. These delays are variable and depend on the level of congestion in the network.

Since the amount of buffer space is finite, an



**Figure 1.12** Packet switching

arriving packet may find that the buffer is completely full with other packets waiting for transmission. In this case, **packet loss** will occur—either the arriving packet or one of the already-queued packets will be dropped.

**Figure 1.12** illustrates a simple packet-switched network. As in **Figure 1.11**, packets are represented by three-dimensional slabs. The width of a slab represents the number of bits in the packet. In this figure, all packets have the same width and hence the same length. Suppose Hosts A and B are sending packets to Host E. Hosts A and B first send their packets along 100 Mbps Ethernet links to the first router. The router then directs these packets to the 15 Mbps link. If, during a short interval of time, the arrival rate of packets to the router (when converted to bits per second) exceeds 15 Mbps, congestion will occur at the router as packets queue in the link's output buffer before being transmitted onto the link. For example, if Host A and B each send a burst of five packets back-to-back at the same time, then most of these packets will spend some time waiting in the queue. The situation is, in fact, entirely analogous to many common-day situations—for example, when we wait in line for a bank teller or wait in front of a tollbooth. We'll examine this queuing delay in more detail in **Section 1.4**.

#### *Forwarding Tables and Routing Protocols*

Earlier, we said that a router takes a packet arriving on one of its attached communication links and forwards that packet onto another one of its attached communication links. But how does the router determine which link it should forward the packet onto? Packet forwarding is actually done in different ways in different types of computer networks. Here, we briefly describe how it is done in the Internet.

In the Internet, every end system has an address called an IP address. When a source end system wants to send a packet to a destination end system, the source includes the destination's IP address in the packet's header. As with postal addresses, this address has a hierarchical structure. When a packet arrives at a router in the network, the router examines a portion of the packet's destination address and forwards the packet to an adjacent router. More specifically, each router has a **forwarding table** that maps destination addresses (or portions of the destination addresses) to that router's outbound links. When a packet arrives at a router, the router examines the address and searches its forwarding table, using this destination address, to find the appropriate outbound link. The router then directs the packet to this outbound link.

The end-to-end routing process is analogous to a car driver who does not use maps but instead prefers to ask for directions. For example, suppose Joe is driving from Philadelphia to 156 Lakeside Drive in Orlando, Florida. Joe first drives to his neighborhood gas station and asks how to get to 156 Lakeside Drive in Orlando, Florida. The gas station attendant extracts the Florida portion of the address and tells Joe that he needs to get onto the interstate highway I-95 South, which has an entrance just next to the gas station. He also tells Joe that once he enters Florida, he should ask someone else there. Joe then takes I-95 South until he gets to Jacksonville, Florida, at which point he asks another gas station attendant for directions. The attendant extracts the Orlando portion of the address and tells Joe that he should continue on I-95 to Daytona Beach and then ask someone else. In Daytona Beach, another gas station attendant also extracts the Orlando portion of the address and tells Joe that he should take I-4 directly to Orlando. Joe takes I-4 and gets off at the Orlando exit. Joe goes to another gas station attendant, and this time the attendant extracts the Lakeside Drive portion of the address and tells Joe the road he must follow to get to Lakeside Drive. Once Joe reaches Lakeside Drive, he asks a kid on a bicycle how to get to his destination. The kid extracts the 156 portion of the address and points to the house. Joe finally reaches his ultimate destination. In the above analogy, the gas station attendants and kids on bicycles are analogous to routers.

We just learned that a router uses a packet's destination address to index a forwarding table and determine the appropriate outbound link. But this statement begs yet another question: How do forwarding tables get set? Are they configured by hand in each and every router, or does the Internet use a more automated procedure? This issue will be studied in depth in [Chapter 5](#). But to whet your appetite here, we'll note now that the Internet has a number of special **routing protocols** that are used to automatically set the forwarding tables. A routing protocol may, for example, determine the shortest path from each router to each destination and use the shortest path results to configure the forwarding tables in the routers.

How would you actually like to see the end-to-end route that packets take in the Internet? We now invite you to get your hands dirty by interacting with the Trace-route program. Simply visit the site [www.traceroute.org](http://www.traceroute.org), choose a source in a particular country, and trace the route from that source to your computer. (For a discussion of Traceroute, see [Section 1.4](#).)

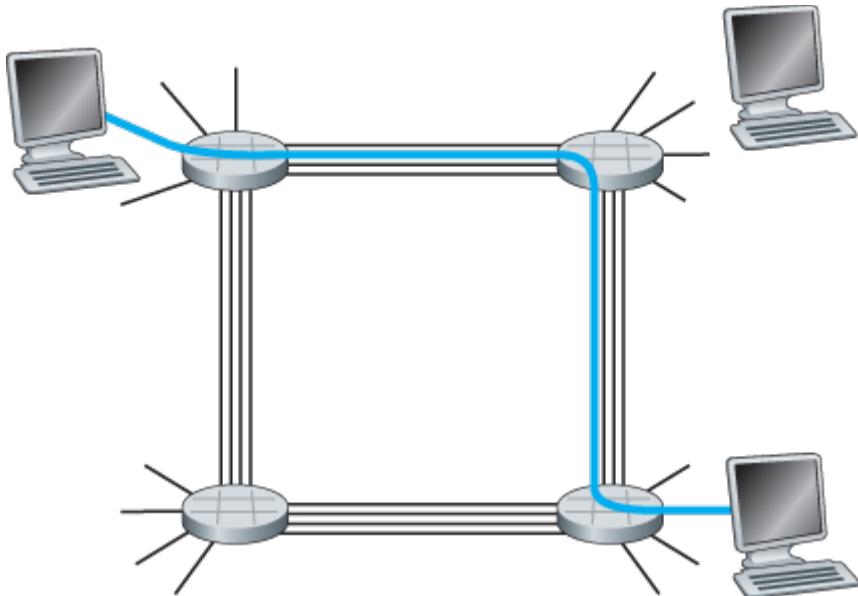
### 1.3.2 Circuit Switching

There are two fundamental approaches to moving data through a network of links and switches: **circuit switching** and **packet switching**. Having covered packet-switched networks in the previous subsection, we now turn our attention to circuit-switched networks.

In circuit-switched networks, the resources needed along a path (buffers, link transmission rate) to provide for communication between the end systems are *reserved* for the duration of the communication session between the end systems. In packet-switched networks, these resources are *not* reserved; a session's messages use the resources on demand and, as a consequence, may have to wait (that is, queue) for access to a communication link. As a simple analogy, consider two restaurants, one that requires reservations and another that neither requires reservations nor accepts them. For the restaurant that requires reservations, we have to go through the hassle of calling before we leave home. But when we arrive at the restaurant we can, in principle, immediately be seated and order our meal. For the restaurant that does not require reservations, we don't need to bother to reserve a table. But when we arrive at the restaurant, we may have to wait for a table before we can be seated.

Traditional telephone networks are examples of circuit-switched networks. Consider what happens when one person wants to send information (voice or facsimile) to another over a telephone network. Before the sender can send the information, the network must establish a connection between the sender and the receiver. This is a *bona fide* connection for which the switches on the path between the sender and receiver maintain connection state for that connection. In the jargon of telephony, this connection is called a **circuit**. When the network establishes the circuit, it also reserves a constant transmission rate in the network's links (representing a fraction of each link's transmission capacity) for the duration of the connection. Since a given transmission rate has been reserved for this sender-to-receiver connection, the sender can transfer the data to the receiver at the *guaranteed* constant rate.

**Figure 1.13** illustrates a circuit-switched network. In this network, the four circuit switches are interconnected by four links. Each of these links has four circuits, so that each link can support four simultaneous connections. The hosts (for example, PCs and workstations) are each directly connected to one of the switches. When two hosts want to communicate, the network establishes a dedicated **end-to-end connection** between the two hosts. Thus, in order for Host A to communicate with Host B, the network must first reserve one circuit on each of two links. In this example, the dedicated end-to-end connection uses the second circuit in the first link and the fourth circuit in the second link. Because each link has four circuits, for each link used by the end-to-end connection, the connection gets one fourth of the link's total transmission capacity for the duration of the connection. Thus, for example, if each link between adjacent switches has a transmission rate of 1 Mbps, then each end-to-end circuit-switch connection gets 250 kbps of dedicated transmission rate.



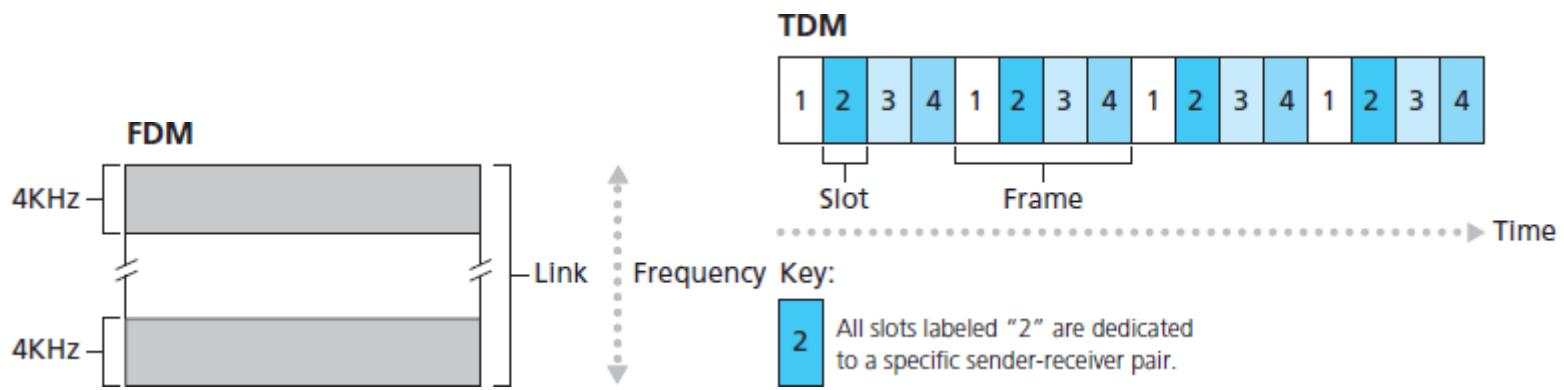
**Figure 1.13 A simple circuit-switched network consisting of four switches and four links**

In contrast, consider what happens when one host wants to send a packet to another host over a packet-switched network, such as the Internet. As with circuit switching, the packet is transmitted over a series of communication links. But different from circuit switching, the packet is sent into the network without reserving any link resources whatsoever. If one of the links is congested because other packets need to be transmitted over the link at the same time, then the packet will have to wait in a buffer at the sending side of the transmission link and suffer a delay. The Internet makes its best effort to deliver packets in a timely manner, but it does not make any guarantees.

#### *Multiplexing in Circuit-Switched Networks*

A circuit in a link is implemented with either **frequency-division multiplexing (FDM)** or **time-division multiplexing (TDM)**. With FDM, the frequency spectrum of a link is divided up among the connections established across the link. Specifically, the link dedicates a frequency band to each connection for the duration of the connection. In telephone networks, this frequency band typically has a width of 4 kHz (that is, 4,000 hertz or 4,000 cycles per second). The width of the band is called, not surprisingly, the **bandwidth**. FM radio stations also use FDM to share the frequency spectrum between 88 MHz and 108 MHz, with each station being allocated a specific frequency band.

For a TDM link, time is divided into frames of fixed duration, and each frame is divided into a fixed number of time slots. When the network establishes a connection across a link, the network dedicates one time slot in every frame to this connection. These slots are dedicated for the sole use of that connection, with one time slot available for use (in every frame) to transmit the connection's data.



**Figure 1.14**

With FDM, each circuit continuously gets a fraction of the bandwidth. With TDM, each circuit gets all of the bandwidth periodically during brief intervals of time (that is, during slots)

**Figure 1.14** illustrates FDM and TDM for a specific network link supporting up to four circuits. For FDM, the frequency domain is segmented into four bands, each of bandwidth 4 kHz. For TDM, the time domain is segmented into frames, with four time slots in each frame; each circuit is assigned the same dedicated slot in the revolving TDM frames. For TDM, the transmission rate of a circuit is equal to the frame rate multiplied by the number of bits in a slot. For example, if the link transmits 8,000 frames per second and each slot consists of 8 bits, then the transmission rate of each circuit is 64 kbps.

Proponents of packet switching have always argued that circuit switching is wasteful because the dedicated circuits are idle during **silent periods**. For example, when one person in a telephone call stops talking, the idle network resources (frequency bands or time slots in the links along the connection's route) cannot be used by other ongoing connections. As another example of how these resources can be underutilized, consider a radiologist who uses a circuit-switched network to remotely access a series of x-rays. The radiologist sets up a connection, requests an image, contemplates the image, and then requests a new image. Network resources are allocated to the connection but are not used (i.e., are wasted) during the radiologist's contemplation periods. Proponents of packet switching also enjoy pointing out that establishing end-to-end circuits and reserving end-to-end transmission capacity is complicated and requires complex signaling software to coordinate the operation of the switches along the end-to-end path.

Before we finish our discussion of circuit switching, let's work through a numerical example that should shed further insight on the topic. Let us consider how long it takes to send a file of 640,000 bits from Host A to Host B over a circuit-switched network. Suppose that all links in the network use TDM with 24 slots and have a bit rate of 1.536 Mbps. Also suppose that it takes 500 msec to establish an end-to-end circuit before Host A can begin to transmit the file. How long does it take to send the file? Each circuit has a transmission rate of  $(1.536 \text{ Mbps})/24=64 \text{ kbps}$ , so it takes  $(640,000 \text{ bits})/(64 \text{ kbps})=10 \text{ seconds}$  to transmit the file. To this 10 seconds we add the circuit establishment time, giving 10.5 seconds to send the file. Note that the transmission time is independent of the number of links: The transmission time would be 10 seconds if the end-to-end circuit passed through one link or a hundred links. (The actual

end-to-end delay also includes a propagation delay; see [Section 1.4](#).)

### Packet Switching Versus Circuit Switching

Having described circuit switching and packet switching, let us compare the two. Critics of packet switching have often argued that packet switching is not suitable for real-time services (for example, telephone calls and video conference calls) because of its variable and unpredictable end-to-end delays (due primarily to variable and unpredictable queuing delays). Proponents of packet switching argue that (1) it offers better sharing of transmission capacity than circuit switching and (2) it is simpler, more efficient, and less costly to implement than circuit switching. An interesting discussion of packet switching versus circuit switching is [\[Molinero-Fernandez 2002\]](#). Generally speaking, people who do not like to hassle with restaurant reservations prefer packet switching to circuit switching.

Why is packet switching more efficient? Let's look at a simple example. Suppose users share a 1 Mbps link. Also suppose that each user alternates between periods of activity, when a user generates data at a constant rate of 100 kbps, and periods of inactivity, when a user generates no data. Suppose further that a user is active only 10 percent of the time (and is idly drinking coffee during the remaining 90 percent of the time). With circuit switching, 100 kbps must be reserved for each user at all times. For example, with circuit-switched TDM, if a one-second frame is divided into 10 time slots of 100 ms each, then each user would be allocated one time slot per frame.

Thus, the circuit-switched link can support only  $10(=1 \text{ Mbps}/100 \text{ kbps})$  simultaneous users. With packet switching, the probability that a specific user is active is 0.1 (that is, 10 percent). If there are 35 users, the probability that there are 11 or more simultaneously active users is approximately 0.0004.

([Homework Problem P8](#) outlines how this probability is obtained.) When there are 10 or fewer simultaneously active users (which happens with probability 0.9996), the aggregate arrival rate of data is less than or equal to 1 Mbps, the output rate of the link. Thus, when there are 10 or fewer active users, users' packets flow through the link essentially without delay, as is the case with circuit switching. When there are more than 10 simultaneously active users, then the aggregate arrival rate of packets exceeds the output capacity of the link, and the output queue will begin to grow. (It continues to grow until the aggregate input rate falls back below 1 Mbps, at which point the queue will begin to diminish in length.) Because the probability of having more than 10 simultaneously active users is minuscule in this example, packet switching provides essentially the same performance as circuit switching, *but does so while allowing for more than three times the number of users*.

Let's now consider a second simple example. Suppose there are 10 users and that one user suddenly generates one thousand 1,000-bit packets, while other users remain quiescent and do not generate packets. Under TDM circuit switching with 10 slots per frame and each slot consisting of 1,000 bits, the active user can only use its one time slot per frame to transmit data, while the remaining nine time slots in each frame remain idle. It will be 10 seconds before all of the active user's one million bits of data has

been transmitted. In the case of packet switching, the active user can continuously send its packets at the full link rate of 1 Mbps, since there are no other users generating packets that need to be multiplexed with the active user's packets. In this case, all of the active user's data will be transmitted within 1 second.

The above examples illustrate two ways in which the performance of packet switching can be superior to that of circuit switching. They also highlight the crucial difference between the two forms of sharing a link's transmission rate among multiple data streams. Circuit switching pre-allocates use of the transmission link regardless of demand, with allocated but unneeded link time going unused. Packet switching on the other hand allocates link use *on demand*. Link transmission capacity will be shared on a packet-by-packet basis only among those users who have packets that need to be transmitted over the link.

Although packet switching and circuit switching are both prevalent in today's telecommunication networks, the trend has certainly been in the direction of packet switching. Even many of today's circuit-switched telephone networks are slowly migrating toward packet switching. In particular, telephone networks often use packet switching for the expensive overseas portion of a telephone call.

### 1.3.3 A Network of Networks

We saw earlier that end systems (PCs, smartphones, Web servers, mail servers, and so on) connect into the Internet via an access ISP. The access ISP can provide either wired or wireless connectivity, using an array of access technologies including DSL, cable, FTTH, Wi-Fi, and cellular. Note that the access ISP does not have to be a telco or a cable company; instead it can be, for example, a university (providing Internet access to students, staff, and faculty), or a company (providing access for its employees). But connecting end users and content providers into an access ISP is only a small piece of solving the puzzle of connecting the billions of end systems that make up the Internet. To complete this puzzle, the access ISPs themselves must be interconnected. This is done by creating a *network of networks*—understanding this phrase is the key to understanding the Internet.

Over the years, the network of networks that forms the Internet has evolved into a very complex structure. Much of this evolution is driven by economics and national policy, rather than by performance considerations. In order to understand today's Internet network structure, let's incrementally build a series of network structures, with each new structure being a better approximation of the complex Internet that we have today. Recall that the overarching goal is to interconnect the access ISPs so that all end systems can send packets to each other. One naive approach would be to have each access ISP *directly* connect with every other access ISP. Such a mesh design is, of course, much too costly for the access ISPs, as it would require each access ISP to have a separate communication link to each of the hundreds of thousands of other access ISPs all over the world.

Our first network structure, *Network Structure 1*, interconnects all of the access ISPs with a *single global transit ISP*. Our (imaginary) global transit ISP is a network of routers and communication links that not only spans the globe, but also has at least one router near each of the hundreds of thousands of access ISPs. Of course, it would be very costly for the global ISP to build such an extensive network. To be profitable, it would naturally charge each of the access ISPs for connectivity, with the pricing reflecting (but not necessarily directly proportional to) the amount of traffic an access ISP exchanges with the global ISP. Since the access ISP pays the global transit ISP, the access ISP is said to be a **customer** and the global transit ISP is said to be a **provider**.

Now if some company builds and operates a global transit ISP that is profitable, then it is natural for other companies to build their own global transit ISPs and compete with the original global transit ISP. This leads to *Network Structure 2*, which consists of the hundreds of thousands of access ISPs and *multiple* global transit ISPs. The access ISPs certainly prefer Network Structure 2 over Network Structure 1 since they can now choose among the competing global transit providers as a function of their pricing and services. Note, however, that the global transit ISPs themselves must interconnect: Otherwise access ISPs connected to one of the global transit providers would not be able to communicate with access ISPs connected to the other global transit providers.

Network Structure 2, just described, is a two-tier hierarchy with global transit providers residing at the top tier and access ISPs at the bottom tier. This assumes that global transit ISPs are not only capable of getting close to each and every access ISP, but also find it economically desirable to do so. In reality, although some ISPs do have impressive global coverage and do directly connect with many access ISPs, no ISP has presence in each and every city in the world. Instead, in any given region, there may be a **regional ISP** to which the access ISPs in the region connect. Each regional ISP then connects to **tier-1 ISPs**. Tier-1 ISPs are similar to our (imaginary) global transit ISP; but tier-1 ISPs, which actually do exist, do not have a presence in every city in the world. There are approximately a dozen tier-1 ISPs, including Level 3 Communications, AT&T, Sprint, and NTT. Interestingly, no group officially sanctions tier-1 status; as the saying goes—if you have to ask if you’re a member of a group, you’re probably not.

Returning to this network of networks, not only are there multiple competing tier-1 ISPs, there may be multiple competing regional ISPs in a region. In such a hierarchy, each access ISP pays the regional ISP to which it connects, and each regional ISP pays the tier-1 ISP to which it connects. (An access ISP can also connect directly to a tier-1 ISP, in which case it pays the tier-1 ISP). Thus, there is customer-provider relationship at each level of the hierarchy. Note that the tier-1 ISPs do not pay anyone as they are at the top of the hierarchy. To further complicate matters, in some regions, there may be a larger regional ISP (possibly spanning an entire country) to which the smaller regional ISPs in that region connect; the larger regional ISP then connects to a tier-1 ISP. For example, in China, there are access ISPs in each city, which connect to provincial ISPs, which in turn connect to national ISPs, which finally connect to tier-1 ISPs [Tian 2012]. We refer to this multi-tier hierarchy, which is still only a crude

approximation of today’s Internet, as *Network Structure 3*.

To build a network that more closely resembles today’s Internet, we must add points of presence (PoPs), multi-homing, peering, and Internet exchange points (IXPs) to the hierarchical Network Structure 3. PoPs exist in all levels of the hierarchy, except for the bottom (access ISP) level. A **PoP** is simply a group of one or more routers (at the same location) in the provider’s network where customer ISPs can connect into the provider ISP. For a customer network to connect to a provider’s PoP, it can lease a high-speed link from a third-party telecommunications provider to directly connect one of its routers to a router at the PoP. Any ISP (except for tier-1 ISPs) may choose to **multi-home**, that is, to connect to two or more provider ISPs. So, for example, an access ISP may multi-home with two regional ISPs, or it may multi-home with two regional ISPs and also with a tier-1 ISP. Similarly, a regional ISP may multi-home with multiple tier-1 ISPs. When an ISP multi-homes, it can continue to send and receive packets into the Internet even if one of its providers has a failure.

As we just learned, customer ISPs pay their provider ISPs to obtain global Internet interconnectivity. The amount that a customer ISP pays a provider ISP reflects the amount of traffic it exchanges with the provider. To reduce these costs, a pair of nearby ISPs at the same level of the hierarchy can **peer**, that is, they can directly connect their networks together so that all the traffic between them passes over the direct connection rather than through upstream intermediaries. When two ISPs peer, it is typically settlement-free, that is, neither ISP pays the other. As noted earlier, tier-1 ISPs also peer with one another, settlement-free. For a readable discussion of peering and customer-provider relationships, see [\[Van der Berg 2008\]](#). Along these same lines, a third-party company can create an **Internet Exchange Point (IXP)**, which is a meeting point where multiple ISPs can peer together. An IXP is typically in a stand-alone building with its own switches [\[Ager 2012\]](#). There are over 400 IXPs in the Internet today [\[IXP List 2016\]](#). We refer to this ecosystem—consisting of access ISPs, regional ISPs, tier-1 ISPs, PoPs, multi-homing, peering, and IXPs—as *Network Structure 4*.

We now finally arrive at *Network Structure 5*, which describes today’s Internet. Network Structure 5, illustrated in [Figure 1.15](#), builds on top of Network Structure 4 by adding **content-provider networks**. Google is currently one of the leading examples of such a content-provider network. As of this writing, it is estimated that Google has 50–100 data centers distributed across North America, Europe, Asia, South America, and Australia. Some of these data centers house over one hundred thousand servers, while other data centers are smaller, housing only hundreds of servers. The Google data centers are all interconnected via Google’s private TCP/IP network, which spans the entire globe but is nevertheless separate from the public Internet. Importantly, the Google private network only carries traffic to/from Google servers. As shown in [Figure 1.15](#), the Google private network attempts to “bypass” the upper tiers of the Internet by peering (settlement free) with lower-tier ISPs, either by directly connecting with them or by connecting with them at IXPs [\[Labovitz 2010\]](#). However, because many access ISPs can still only be reached by transiting through tier-1 networks, the Google network also connects to tier-1 ISPs, and pays those ISPs for the traffic it exchanges with them. By creating its own network, a content

provider not only reduces its payments to upper-tier ISPs, but also has greater control of how its services are ultimately delivered to end users. Google's network infrastructure is described in greater detail in [Section 2.6](#).

In summary, today's Internet—a network of networks—is complex, consisting of a dozen or so tier-1 ISPs and hundreds of thousands of lower-tier ISPs. The ISPs are diverse in their coverage, with some spanning multiple continents and oceans, and others limited to narrow geographic regions. The lower-tier ISPs connect to the higher-tier ISPs, and the higher-tier ISPs interconnect with one another. Users and content providers are customers of lower-tier ISPs, and lower-tier ISPs are customers of higher-tier ISPs. In recent years, major content providers have also created their own networks and connect directly into lower-tier ISPs where possible.

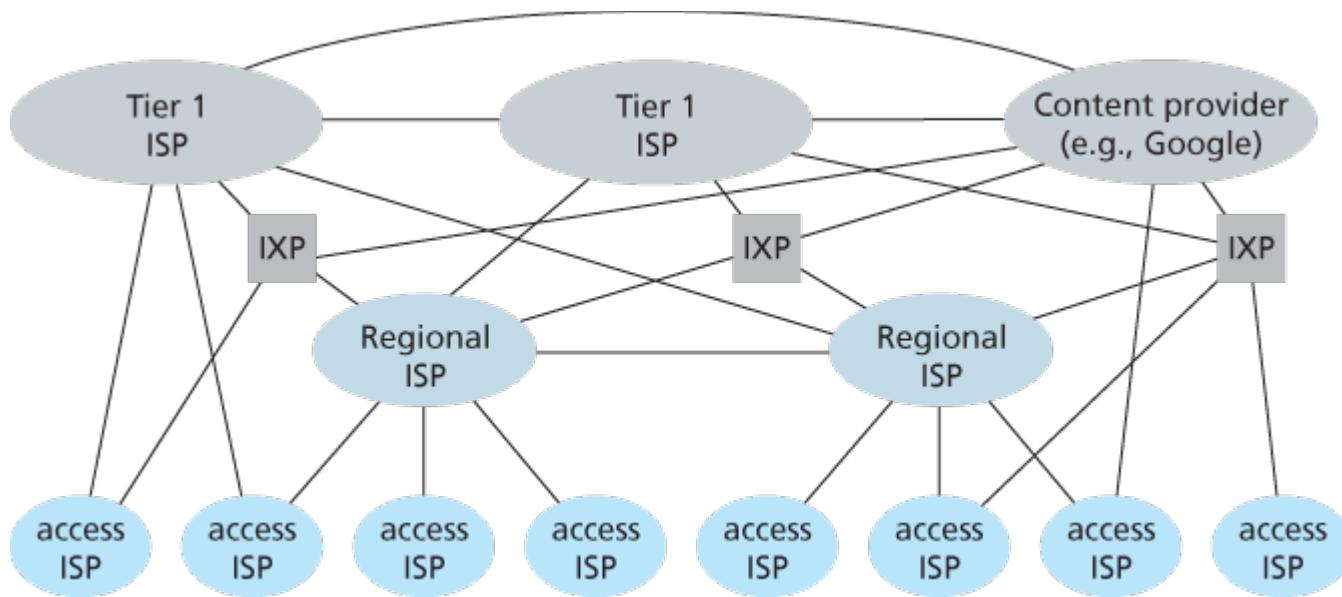


Figure 1.15 Interconnection of ISPs

## 1.4 Delay, Loss, and Throughput in Packet-Switched Networks

Back in [Section 1.1](#) we said that the Internet can be viewed as an infrastructure that provides services to distributed applications running on end systems. Ideally, we would like Internet services to be able to move as much data as we want between any two end systems, instantaneously, without any loss of data. Alas, this is a lofty goal, one that is unachievable in reality. Instead, computer networks necessarily constrain throughput (the amount of data per second that can be transferred) between end systems, introduce delays between end systems, and can actually lose packets. On one hand, it is unfortunate that the physical laws of reality introduce delay and loss as well as constrain throughput. On the other hand, because computer networks have these problems, there are many fascinating issues surrounding how to deal with the problems—more than enough issues to fill a course on computer networking and to motivate thousands of PhD theses! In this section, we'll begin to examine and quantify delay, loss, and throughput in computer networks.

### 1.4.1 Overview of Delay in Packet-Switched Networks

Recall that a packet starts in a host (the source), passes through a series of routers, and ends its journey in another host (the destination). As a packet travels from one node (host or router) to the subsequent node (host or router) along this path, the packet suffers from several types of delays at each node along the path. The most important of these delays are the [nodal processing delay](#), [queuing delay](#), [transmission delay](#), and [propagation delay](#); together, these delays accumulate to give a [total nodal delay](#). The performance of many Internet applications—such as search, Web browsing, e-mail, maps, instant messaging, and voice-over-IP—are greatly affected by network delays. In order to acquire a deep understanding of packet switching and computer networks, we must understand the nature and importance of these delays.

#### *Types of Delay*

Let's explore these delays in the context of [Figure 1.16](#). As part of its end-to-end route between source and destination, a packet is sent from the upstream node through router A to router B. Our goal is to characterize the nodal delay at router A. Note that router A has an outbound link leading to router B. This link is preceded by a queue (also known as a buffer). When the packet arrives at router A from the upstream node, router A examines the packet's header to determine the appropriate outbound link for the packet and then directs the packet to this link. In this example, the outbound link for the packet is the one that leads to router B. A packet can be transmitted on a link only if there is no other packet currently

being transmitted on the link and if there are no other packets preceding it in the queue; if the link is

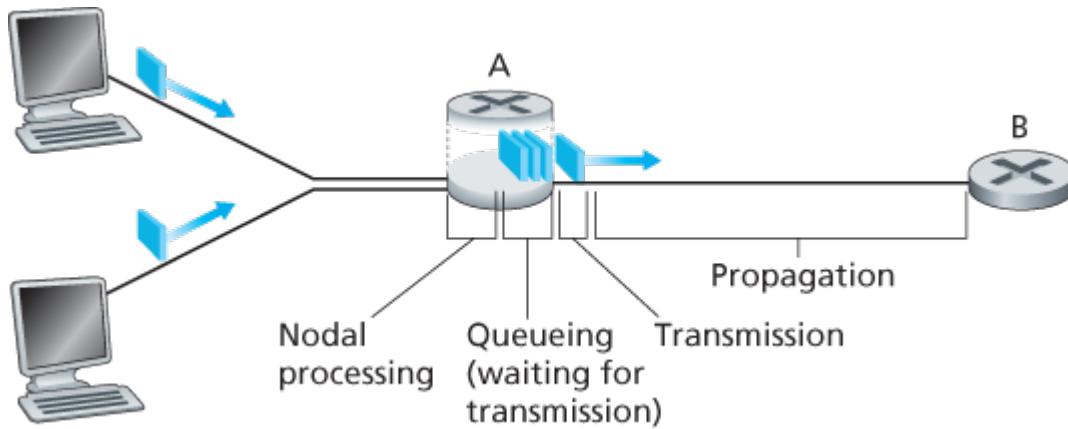


Figure 1.16 The nodal delay at router A

currently busy or if there are other packets already queued for the link, the newly arriving packet will then join the queue.

#### Processing Delay

The time required to examine the packet's header and determine where to direct the packet is part of the **processing delay**. The processing delay can also include other factors, such as the time needed to check for bit-level errors in the packet that occurred in transmitting the packet's bits from the upstream node to router A. Processing delays in high-speed routers are typically on the order of microseconds or less. After this nodal processing, the router directs the packet to the queue that precedes the link to router B. (In [Chapter 4](#) we'll study the details of how a router operates.)

#### Queuing Delay

At the queue, the packet experiences a **queuing delay** as it waits to be transmitted onto the link. The length of the queuing delay of a specific packet will depend on the number of earlier-arriving packets that are queued and waiting for transmission onto the link. If the queue is empty and no other packet is currently being transmitted, then our packet's queuing delay will be zero. On the other hand, if the traffic is heavy and many other packets are also waiting to be transmitted, the queuing delay will be long. We will see shortly that the number of packets that an arriving packet might expect to find is a function of the intensity and nature of the traffic arriving at the queue. Queuing delays can be on the order of microseconds to milliseconds in practice.

#### Transmission Delay

Assuming that packets are transmitted in a first-come-first-served manner, as is common in packet-switched networks, our packet can be transmitted only after all the packets that have arrived before it have been transmitted. Denote the length of the packet by  $L$  bits, and denote the transmission rate of

the link from router A to router B by  $R$  bits/sec. For example, for a 10 Mbps Ethernet link, the rate is  $R=10$  Mbps; for a 100 Mbps Ethernet link, the rate is  $R=100$  Mbps. The **transmission delay** is  $L/R$ . This is the amount of time required to push (that is, transmit) all of the packet's bits into the link. Transmission delays are typically on the order of microseconds to milliseconds in practice.

### Propagation Delay

Once a bit is pushed into the link, it needs to propagate to router B. The time required to propagate from the beginning of the link to router B is the **propagation delay**. The bit propagates at the propagation speed of the link. The propagation speed depends on the physical medium of the link (that is, fiber optics, twisted-pair copper wire, and so on) and is in the range of

2·108 meters/sec to 3·108 meters/sec

which is equal to, or a little less than, the speed of light. The propagation delay is the distance between two routers divided by the propagation speed. That is, the propagation delay is  $d/s$ , where  $d$  is the distance between router A and router B and  $s$  is the propagation speed of the link. Once the last bit of the packet propagates to node B, it and all the preceding bits of the packet are stored in router B. The whole process then continues with router B now performing the forwarding. In wide-area networks, propagation delays are on the order of milliseconds.

### Comparing Transmission and Propagation Delay



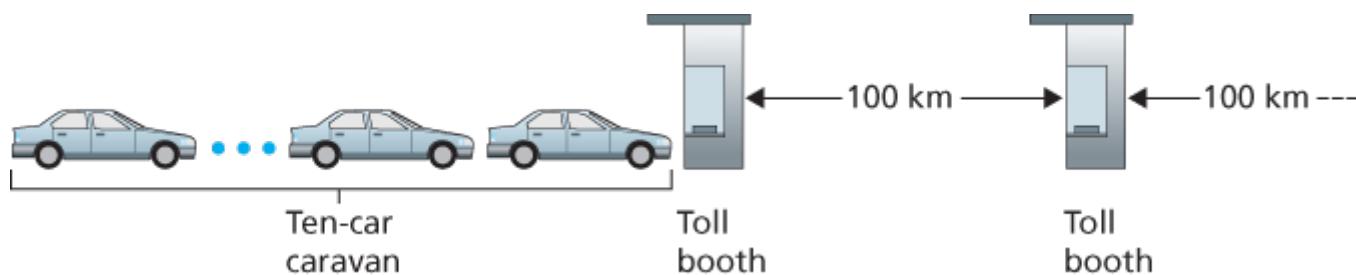
VideoNote

Exploring propagation delay and transmission delay

Newcomers to the field of computer networking sometimes have difficulty understanding the difference between transmission delay and propagation delay. The difference is subtle but important. The transmission delay is the amount of time required for the router to push out the packet; it is a function of the packet's length and the transmission rate of the link, but has nothing to do with the distance between the two routers. The propagation delay, on the other hand, is the time it takes a bit to propagate from one router to the next; it is a function of the distance between the two routers, but has nothing to do with the packet's length or the transmission rate of the link.

An analogy might clarify the notions of transmission and propagation delay. Consider a highway that has a tollbooth every 100 kilometers, as shown in [Figure 1.17](#). You can think of the highway segments

between tollbooths as links and the tollbooths as routers. Suppose that cars travel (that is, propagate) on the highway at a rate of 100 km/hour (that is, when a car leaves a tollbooth, it instantaneously accelerates to 100 km/hour and maintains that speed between tollbooths). Suppose next that 10 cars, traveling together as a caravan, follow each other in a fixed order. You can think of each car as a bit and the caravan as a packet. Also suppose that each



**Figure 1.17 Caravan analogy**

tollbooth services (that is, transmits) a car at a rate of one car per 12 seconds, and that it is late at night so that the caravan's cars are the only cars on the highway. Finally, suppose that whenever the first car of the caravan arrives at a tollbooth, it waits at the entrance until the other nine cars have arrived and lined up behind it. (Thus the entire caravan must be stored at the tollbooth before it can begin to be forwarded.) The time required for the tollbooth to push the entire caravan onto the highway is  $(10 \text{ cars})/(5 \text{ cars}/\text{minute})=2 \text{ minutes}$ . This time is analogous to the transmission delay in a router. The time required for a car to travel from the exit of one tollbooth to the next tollbooth is  $100 \text{ km}/(100 \text{ km}/\text{hour})=1 \text{ hour}$ . This time is analogous to propagation delay. Therefore, the time from when the caravan is stored in front of a tollbooth until the caravan is stored in front of the next tollbooth is the sum of transmission delay and propagation delay—in this example, 62 minutes.

Let's explore this analogy a bit more. What would happen if the tollbooth service time for a caravan were greater than the time for a car to travel between tollbooths? For example, suppose now that the cars travel at the rate of 1,000 km/hour and the tollbooth services cars at the rate of one car per minute. Then the traveling delay between two tollbooths is 6 minutes and the time to serve a caravan is 10 minutes. In this case, the first few cars in the caravan will arrive at the second tollbooth before the last cars in the caravan leave the first tollbooth. This situation also arises in packet-switched networks—the first bits in a packet can arrive at a router while many of the remaining bits in the packet are still waiting to be transmitted by the preceding router.

If a picture speaks a thousand words, then an animation must speak a million words. The Web site for this textbook provides an interactive Java applet that nicely illustrates and contrasts transmission delay and propagation delay. The reader is highly encouraged to visit that applet. [Smith 2009] also provides a very readable discussion of propagation, queueing, and transmission delays.

If we let  $d_{\text{proc}}$ ,  $d_{\text{queue}}$ ,  $d_{\text{trans}}$ , and  $d_{\text{prop}}$  denote the processing, queuing, transmission, and propagation

delays, then the total nodal delay is given by

$$d_{\text{nodal}} = d_{\text{proc}} + d_{\text{queue}} + d_{\text{trans}} + d_{\text{prop}}$$

The contribution of these delay components can vary significantly. For example,  $d_{\text{prop}}$  can be negligible (for example, a couple of microseconds) for a link connecting two routers on the same university campus; however,  $d_{\text{prop}}$  is hundreds of milliseconds for two routers interconnected by a geostationary satellite link, and can be the dominant term in  $d_{\text{nodal}}$ . Similarly,  $d_{\text{trans}}$  can range from negligible to significant. Its contribution is typically negligible for transmission rates of 10 Mbps and higher (for example, for LANs); however, it can be hundreds of milliseconds for large Internet packets sent over low-speed dial-up modem links. The processing delay,  $d_{\text{proc}}$ , is often negligible; however, it strongly influences a router's maximum throughput, which is the maximum rate at which a router can forward packets.

#### 1.4.2 Queuing Delay and Packet Loss

The most complicated and interesting component of nodal delay is the queuing delay,  $d_{\text{queue}}$ . In fact, queuing delay is so important and interesting in computer networking that thousands of papers and numerous books have been written about it [[Bertsekas 1991](#); [Daigle 1991](#); [Kleinrock 1975](#), [Kleinrock 1976](#); [Ross 1995](#)]. We give only a high-level, intuitive discussion of queuing delay here; the more curious reader may want to browse through some of the books (or even eventually write a PhD thesis on the subject!). Unlike the other three delays (namely,  $d_{\text{proc}}$ ,  $d_{\text{trans}}$ , and  $d_{\text{prop}}$ ), the queuing delay can vary from packet to packet. For example, if 10 packets arrive at an empty queue at the same time, the first packet transmitted will suffer no queuing delay, while the last packet transmitted will suffer a relatively large queuing delay (while it waits for the other nine packets to be transmitted). Therefore, when characterizing queuing delay, one typically uses statistical measures, such as average queuing delay, variance of queuing delay, and the probability that the queuing delay exceeds some specified value.

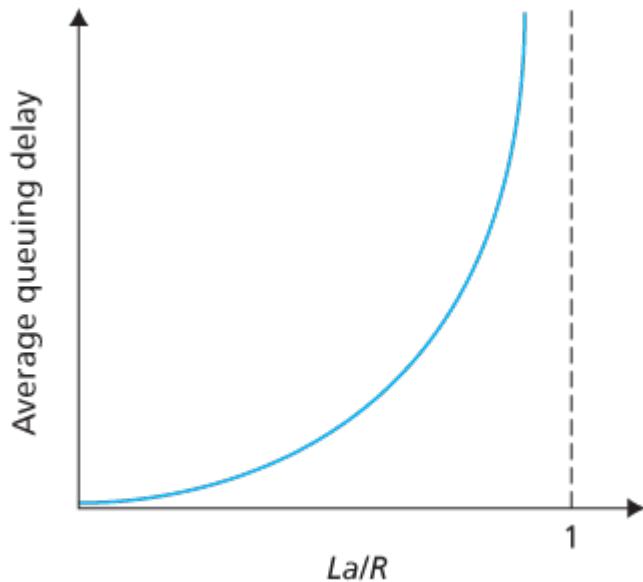
When is the queuing delay large and when is it insignificant? The answer to this question depends on the rate at which traffic arrives at the queue, the transmission rate of the link, and the nature of the arriving traffic, that is, whether the traffic arrives periodically or arrives in bursts. To gain some insight here, let  $a$  denote the average rate at which packets arrive at the queue ( $a$  is in units of packets/sec). Recall that  $R$  is the transmission rate; that is, it is the rate (in bits/sec) at which bits are pushed out of the queue. Also suppose, for simplicity, that all packets consist of  $L$  bits. Then the average rate at which bits arrive at the queue is  $La$  bits/sec. Finally, assume that the queue is very big, so that it can hold essentially an infinite number of bits. The ratio  $La/R$ , called the **traffic intensity**, often plays an important role in estimating the extent of the queuing delay. If  $La/R > 1$ , then the average rate at which bits arrive at the queue exceeds the rate at which the bits can be transmitted from the queue. In this

unfortunate situation, the queue will tend to increase without bound and the queuing delay will approach infinity! Therefore, one of the golden rules in traffic engineering is: *Design your system so that the traffic intensity is no greater than 1.*

Now consider the case  $La/R \leq 1$ . Here, the nature of the arriving traffic impacts the queuing delay. For example, if packets arrive periodically—that is, one packet arrives every  $L/R$  seconds—then every packet will arrive at an empty queue and there will be no queuing delay. On the other hand, if packets arrive in bursts but periodically, there can be a significant average queuing delay. For example, suppose  $N$  packets arrive simultaneously every  $(L/R)N$  seconds. Then the first packet transmitted has no queuing delay; the second packet transmitted has a queuing delay of  $L/R$  seconds; and more generally, the  $n$ th packet transmitted has a queuing delay of  $(n-1)L/R$  seconds. We leave it as an exercise for you to calculate the average queuing delay in this example.

The two examples of periodic arrivals described above are a bit academic. Typically, the arrival process to a queue is *random*; that is, the arrivals do not follow any pattern and the packets are spaced apart by random amounts of time. In this more realistic case, the quantity  $La/R$  is not usually sufficient to fully characterize the queuing delay statistics. Nonetheless, it is useful in gaining an intuitive understanding of the extent of the queuing delay. In particular, if the traffic intensity is close to zero, then packet arrivals are few and far between and it is unlikely that an arriving packet will find another packet in the queue. Hence, the average queuing delay will be close to zero. On the other hand, when the traffic intensity is close to 1, there will be intervals of time when the arrival rate exceeds the transmission capacity (due to variations in packet arrival rate), and a queue will form during these periods of time; when the arrival rate is less than the transmission capacity, the length of the queue will shrink. Nonetheless, as the traffic intensity approaches 1, the average queue length gets larger and larger. The qualitative dependence of average queuing delay on the traffic intensity is shown in [Figure 1.18](#).

One important aspect of [Figure 1.18](#) is the fact that as the traffic intensity approaches 1, the average queuing delay increases rapidly. A small percentage increase in the intensity will result in a much larger percentage-wise increase in delay. Perhaps you have experienced this phenomenon on the highway. If you regularly drive on a road that is typically congested, the fact that the road is typically



**Figure 1.18 Dependence of average queuing delay on traffic intensity**

congested means that its traffic intensity is close to 1. If some event causes an even slightly larger-than-usual amount of traffic, the delays you experience can be huge.

To really get a good feel for what queuing delays are about, you are encouraged once again to visit the textbook Web site, which provides an interactive Java applet for a queue. If you set the packet arrival rate high enough so that the traffic intensity exceeds 1, you will see the queue slowly build up over time.

### *Packet Loss*

In our discussions above, we have assumed that the queue is capable of holding an infinite number of packets. In reality a queue preceding a link has finite capacity, although the queuing capacity greatly depends on the router design and cost. Because the queue capacity is finite, packet delays do not really approach infinity as the traffic intensity approaches 1. Instead, a packet can arrive to find a full queue. With no place to store such a packet, a router will **drop** that packet; that is, the packet will be **lost**. This overflow at a queue can again be seen in the Java applet for a queue when the traffic intensity is greater than 1.

From an end-system viewpoint, a packet loss will look like a packet having been transmitted into the network core but never emerging from the network at the destination. The fraction of lost packets increases as the traffic intensity increases. Therefore, performance at a node is often measured not only in terms of delay, but also in terms of the probability of packet loss. As we'll discuss in the subsequent chapters, a lost packet may be retransmitted on an end-to-end basis in order to ensure that all data are eventually transferred from source to destination.

### 1.4.3 End-to-End Delay

Our discussion up to this point has focused on the nodal delay, that is, the delay at a single router. Let's now consider the total delay from source to destination. To get a handle on this concept, suppose there are  $N-1$  routers between the source host and the destination host. Let's also suppose for the moment that the network is uncongested (so that queuing delays are negligible), the processing delay at each router and at the source host is  $d_{\text{proc}}$ , the transmission rate out of each router and out of the source host is  $R$  bits/sec, and the propagation on each link is  $d_{\text{prop}}$ . The nodal delays accumulate and give an end-to-end delay,

$$\text{dend-end} = N(d_{\text{proc}} + d_{\text{trans}} + d_{\text{prop}}) \quad (1.2)$$

where, once again,  $d_{\text{trans}} = L/R$ , where  $L$  is the packet size. Note that **Equation 1.2** is a generalization of **Equation 1.1**, which did not take into account processing and propagation delays. We leave it to you to generalize **Equation 1.2** to the case of heterogeneous delays at the nodes and to the presence of an average queuing delay at each node.

### Traceroute



Using Traceroute to discover network paths and measure network delay

To get a hands-on feel for end-to-end delay in a computer network, we can make use of the Traceroute program. Traceroute is a simple program that can run in any Internet host. When the user specifies a destination hostname, the program in the source host sends multiple, special packets toward that destination. As these packets work their way toward the destination, they pass through a series of routers. When a router receives one of these special packets, it sends back to the source a short message that contains the name and address of the router.

More specifically, suppose there are  $N-1$  routers between the source and the destination. Then the source will send  $N$  special packets into the network, with each packet addressed to the ultimate destination. These  $N$  special packets are marked 1 through  $N$ , with the first packet marked 1 and the last packet marked  $N$ . When the  $n$ th router receives the  $n$ th packet marked  $n$ , the router does not forward the packet toward its destination, but instead sends a message back to the source. When the destination host receives the  $N$ th packet, it too returns a message back to the source. The source records the time that elapses between when it sends a packet and when it receives the corresponding

return message; it also records the name and address of the router (or the destination host) that returns the message. In this manner, the source can reconstruct the route taken by packets flowing from source to destination, and the source can determine the round-trip delays to all the intervening routers.

Traceroute actually repeats the experiment just described three times, so the source actually sends  $3 \cdot N$  packets to the destination. RFC 1393 describes Traceroute in detail.

Here is an example of the output of the Traceroute program, where the route was being traced from the source host [gaia.cs.umass.edu](#) (at the University of Massachusetts) to the host [cis.poly.edu](#) (at Polytechnic University in Brooklyn). The output has six columns: the first column is the  $n$  value described above, that is, the number of the router along the route; the second column is the name of the router; the third column is the address of the router (of the form xxx.xxx.xxx.xxx); the last three columns are the round-trip delays for three experiments. If the source receives fewer than three messages from any given router (due to packet loss in the network), Traceroute places an asterisk just after the router number and reports fewer than three round-trip times for that router.

```
1  cs-gw (128.119.240.254) 1.009 ms 0.899 ms 0.993 ms
2  128.119.3.154 (128.119.3.154) 0.931 ms 0.441 ms 0.651 ms
3  -border4-rt-gi-1-3.gw.umass.edu (128.119.2.194) 1.032 ms 0.484 ms
   0.451 ms
4  -acrl1-ge-2-1-0.Boston.cw.net (208.172.51.129) 10.006 ms 8.150 ms 8.460
   ms
5  -agr4-loopback.NewYork.cw.net (206.24.194.104) 12.272 ms 14.344 ms
   13.267 ms
6  -acr2-loopback.NewYork.cw.net (206.24.194.62) 13.225 ms 12.292 ms
   12.148 ms
7  -pos10-2.core2.NewYork1.Level3.net (209.244.160.133) 12.218 ms 11.823
   ms 11.793 ms
8  -gige9-1-52.hsipaccess1.NewYork1.Level3.net (64.159.17.39) 13.081 ms
   11.556 ms 13.297 ms
9  -p0-0.polyu.bbnplanet.net (4.25.109.122) 12.716 ms 13.052 ms 12.786 ms
10 cis.poly.edu (128.238.32.126) 14.080 ms 13.035 ms 12.802 ms
```

In the trace above there are nine routers between the source and the destination. Most of these routers have a name, and all of them have addresses. For example, the name of Router 3 is [border4-rt-gi-1-3.gw.umass.edu](#) and its address is [128.119.2.194](#). Looking at the data provided for this same router, we see that in the first of the three trials the round-trip delay between the source and the router was 1.03 msec. The round-trip delays for the subsequent two trials were 0.48 and 0.45 msec. These

round-trip delays include all of the delays just discussed, including transmission delays, propagation delays, router processing delays, and queuing delays. Because the queuing delay is varying with time, the round-trip delay of packet  $n$  sent to a router  $n$  can sometimes be longer than the round-trip delay of packet  $n+1$  sent to router  $n+1$ . Indeed, we observe this phenomenon in the above example: the delays to Router 6 are larger than the delays to Router 7!

Want to try out Traceroute for yourself? We *highly* recommend that you visit <http://www.traceroute.org>, which provides a Web interface to an extensive list of sources for route tracing. You choose a source and supply the hostname for any destination. The Traceroute program then does all the work. There are a number of free software programs that provide a graphical interface to Traceroute; one of our favorites is PingPlotter [\[PingPlotter 2016\]](#).

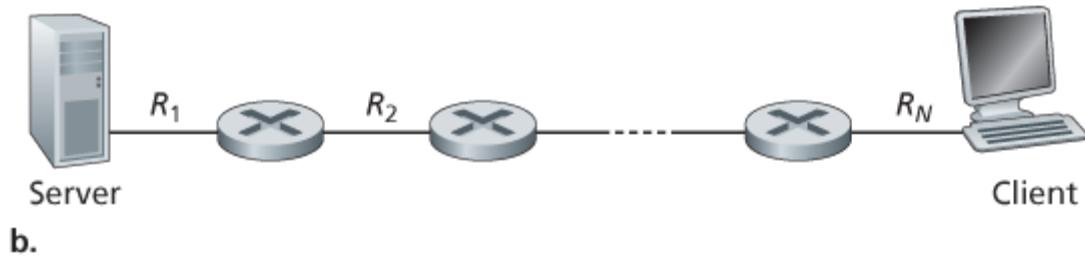
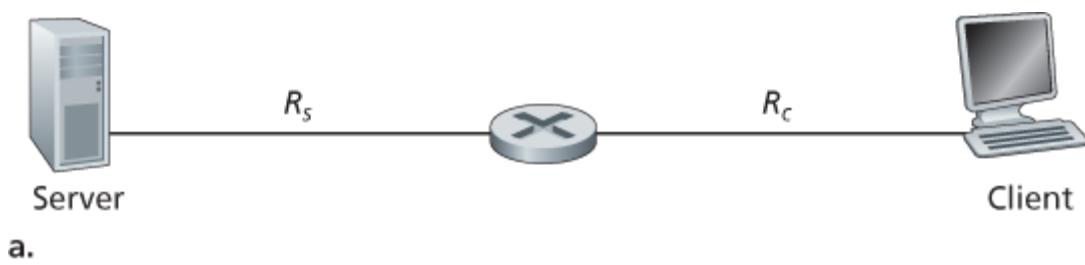
#### *End System, Application, and Other Delays*

In addition to processing, transmission, and propagation delays, there can be additional significant delays in the end systems. For example, an end system wanting to transmit a packet into a shared medium (e.g., as in a WiFi or cable modem scenario) may *purposefully* delay its transmission as part of its protocol for sharing the medium with other end systems; we'll consider such protocols in detail in [Chapter 6](#). Another important delay is media packetization delay, which is present in Voice-over-IP (VoIP) applications. In VoIP, the sending side must first fill a packet with encoded digitized speech before passing the packet to the Internet. This time to fill a packet—called the packetization delay—can be significant and can impact the user-perceived quality of a VoIP call. This issue will be further explored in a homework problem at the end of this chapter.

#### 1.4.4 Throughput in Computer Networks

In addition to delay and packet loss, another critical performance measure in computer networks is end-to-end throughput. To define throughput, consider transferring a large file from Host A to Host B across a computer network. This transfer might be, for example, a large video clip from one peer to another in a P2P file sharing system. The **instantaneous throughput** at any instant of time is the rate (in bits/sec) at which Host B is receiving the file. (Many applications, including many P2P file sharing systems, display the instantaneous throughput during downloads in the user interface—perhaps you have observed this before!) If the file consists of  $F$  bits and the transfer takes  $T$  seconds for Host B to receive all  $F$  bits, then the **average throughput** of the file transfer is  $F/T$  bits/sec. For some applications, such as Internet telephony, it is desirable to have a low delay and an instantaneous throughput consistently above some threshold (for example, over 24 kbps for some Internet telephony applications and over 256 kbps for some real-time video applications). For other applications, including those involving file transfers, delay is not critical, but it is desirable to have the highest possible throughput.

To gain further insight into the important concept of throughput, let's consider a few examples. [Figure 1.19\(a\)](#) shows two end systems, a server and a client, connected by two communication links and a router. Consider the throughput for a file transfer from the server to the client. Let  $R_s$  denote the rate of the link between the server and the router; and  $R_c$  denote the rate of the link between the router and the client. Suppose that the only bits being sent in the entire network are those from the server to the client. We now ask, in this ideal scenario, what is the server-to-client throughput? To answer this question, we may think of bits as *fluid* and communication links as *pipes*. Clearly, the server cannot pump bits through its link at a rate faster than  $R_s$  bps; and the router cannot forward bits at a rate faster than  $R_c$  bps. If  $R_s < R_c$ , then the bits pumped by the server will “flow” right through the router and arrive at the client at a rate of  $R_s$  bps, giving a throughput of  $R_s$  bps. If, on the other hand,  $R_c < R_s$ , then the router will not be able to forward bits as quickly as it receives them. In this case, bits will only leave the router at rate  $R_c$ , giving an end-to-end throughput of  $R_c$ . (Note also that if bits continue to arrive at the router at rate  $R_s$ , and continue to leave the router at  $R_c$ , the backlog of bits at the router waiting



**Figure 1.19 Throughput for a file transfer from server to client**

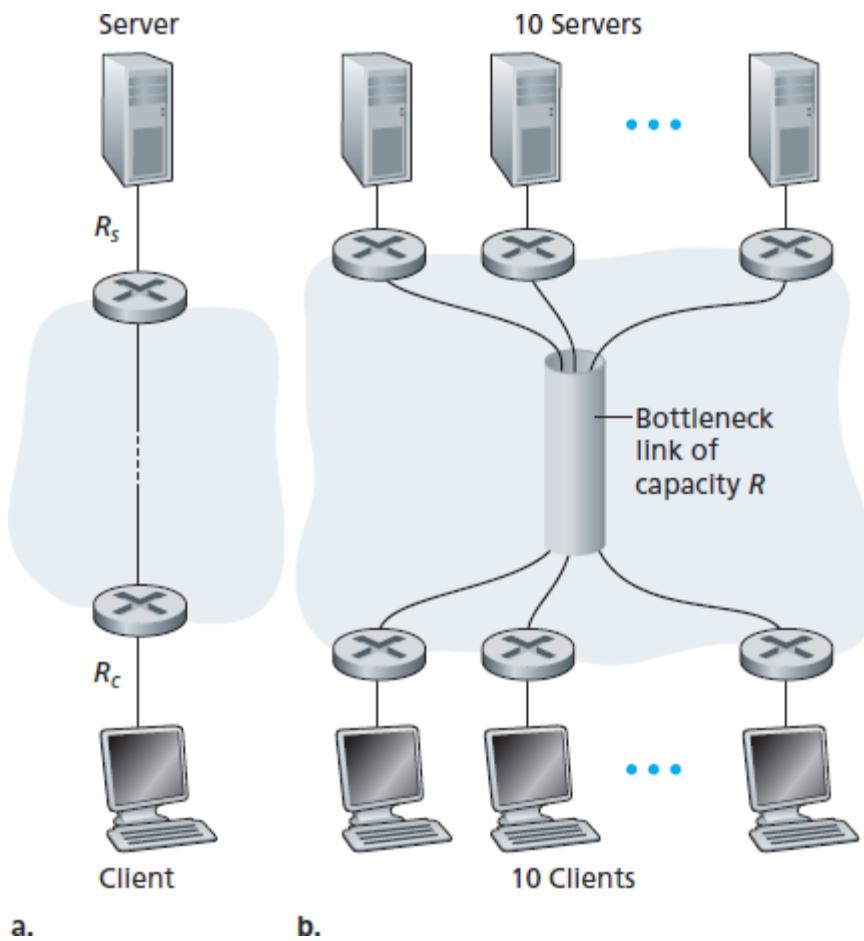
for transmission to the client will grow and grow—a most undesirable situation!) Thus, for this simple two-link network, the throughput is  $\min\{R_c, R_s\}$ , that is, it is the transmission rate of the **bottleneck link**. Having determined the throughput, we can now approximate the time it takes to transfer a large file of  $F$  bits from server to client as  $F/\min\{R_s, R_c\}$ . For a specific example, suppose you are downloading an MP3 file of  $F=32$  million bits, the server has a transmission rate of  $R_s=2$  Mbps, and you have an access link of  $R_c=1$  Mbps. The time needed to transfer the file is then 32 seconds. Of course, these expressions for throughput and transfer time are only approximations, as they do not account for store-and-forward and processing delays as well as protocol issues.

[Figure 1.19\(b\)](#) now shows a network with  $N$  links between the server and the client, with the transmission rates of the  $N$  links being  $R_1, R_2, \dots, R_N$ . Applying the same analysis as for the two-link network, we find that the throughput for a file transfer from server to client is  $\min\{R_1, R_2, \dots, R_N\}$ , which

is once again the transmission rate of the bottleneck link along the path between server and client.

Now consider another example motivated by today's Internet. **Figure 1.20(a)** shows two end systems, a server and a client, connected to a computer network. Consider the throughput for a file transfer from the server to the client. The server is connected to the network with an access link of rate  $R_s$  and the client is connected to the network with an access link of rate  $R_c$ . Now suppose that all the links in the core of the communication network have very high transmission rates, much higher than  $R_s$  and  $R_c$ . Indeed, today, the core of the Internet is over-provisioned with high speed links that experience little congestion. Also suppose that the only bits being sent in the entire network are those from the server to the client. Because the core of the computer network is like a wide pipe in this example, the rate at which bits can flow from source to destination is again the minimum of  $R_s$  and  $R_c$ , that is, throughput =  $\min\{R_s, R_c\}$ . Therefore, the constraining factor for throughput in today's Internet is typically the access network.

For a final example, consider **Figure 1.20(b)** in which there are 10 servers and 10 clients connected to the core of the computer network. In this example, there are 10 simultaneous downloads taking place, involving 10 client-server pairs. Suppose that these 10 downloads are the only traffic in the network at the current time. As shown in the figure, there is a link in the core that is traversed by all 10 downloads. Denote  $R$  for the transmission rate of this link  $R$ . Let's suppose that all server access links have the same rate  $R_s$ , all client access links have the same rate  $R_c$ , and the transmission rates of all the links in the core—except the one common link of rate  $R$ —are much larger than  $R_s$ ,  $R_c$ , and  $R$ . Now we ask, what are the throughputs of the downloads? Clearly, if the rate of the common link,  $R$ , is large—say a hundred times larger than both  $R_s$  and  $R_c$ —then the throughput for each download will once again be  $\min\{R_s, R_c\}$ . But what if the rate of the common link is of the same order as  $R_s$  and  $R_c$ ? What will the throughput be in this case? Let's take a look at a specific example. Suppose  $R_s=2$  Mbps,  $R_c=1$  Mbps,  $R=5$  Mbps, and the



**Figure 1.20 End-to-end throughput: (a) Client downloads a file from server; (b) 10 clients downloading with 10 servers**

common link divides its transmission rate equally among the 10 downloads. Then the bottleneck for each download is no longer in the access network, but is now instead the shared link in the core, which only provides each download with 500 kbps of throughput. Thus the end-to-end throughput for each download is now reduced to 500 kbps.

The examples in [Figure 1.19](#) and [Figure 1.20\(a\)](#) show that throughput depends on the transmission rates of the links over which the data flows. We saw that when there is no other intervening traffic, the throughput can simply be approximated as the minimum transmission rate along the path between source and destination. The example in [Figure 1.20\(b\)](#) shows that more generally the throughput depends not only on the transmission rates of the links along the path, but also on the intervening traffic. In particular, a link with a high transmission rate may nonetheless be the bottleneck link for a file transfer if many other data flows are also passing through that link. We will examine throughput in computer networks more closely in the homework problems and in the subsequent chapters.

## 1.5 Protocol Layers and Their Service Models

From our discussion thus far, it is apparent that the Internet is an *extremely* complicated system. We have seen that there are many pieces to the Internet: numerous applications and protocols, various types of end systems, packet switches, and various types of link-level media. Given this enormous complexity, is there any hope of organizing a network architecture, or at least our discussion of network architecture? Fortunately, the answer to both questions is yes.

### 1.5.1 Layered Architecture

Before attempting to organize our thoughts on Internet architecture, let's look for a human analogy. Actually, we deal with complex systems all the time in our everyday life. Imagine if someone asked you to describe, for example, the airline system. How would you find the structure to describe this complex system that has ticketing agents, baggage checkers, gate personnel, pilots, airplanes, air traffic control, and a worldwide system for routing airplanes? One way to describe this system might be to describe the series of actions you take (or others take for you) when you fly on an airline. You purchase your ticket, check your bags, go to the gate, and eventually get loaded onto the plane. The plane takes off and is routed to its destination. After your plane lands, you deplane at the gate and claim your bags. If the trip was bad, you complain about the flight to the ticket agent (getting nothing for your effort). This scenario is shown in **Figure 1.21**.

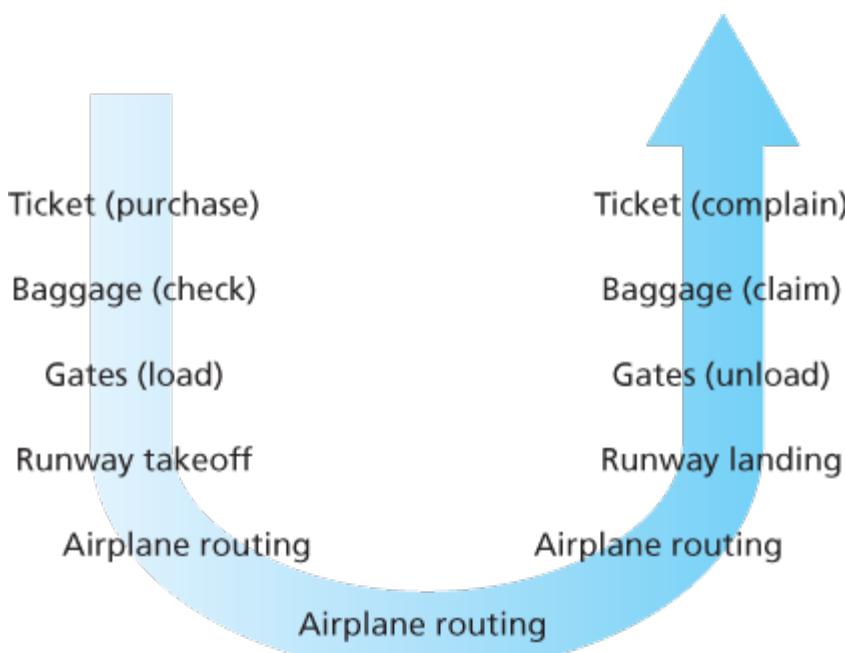
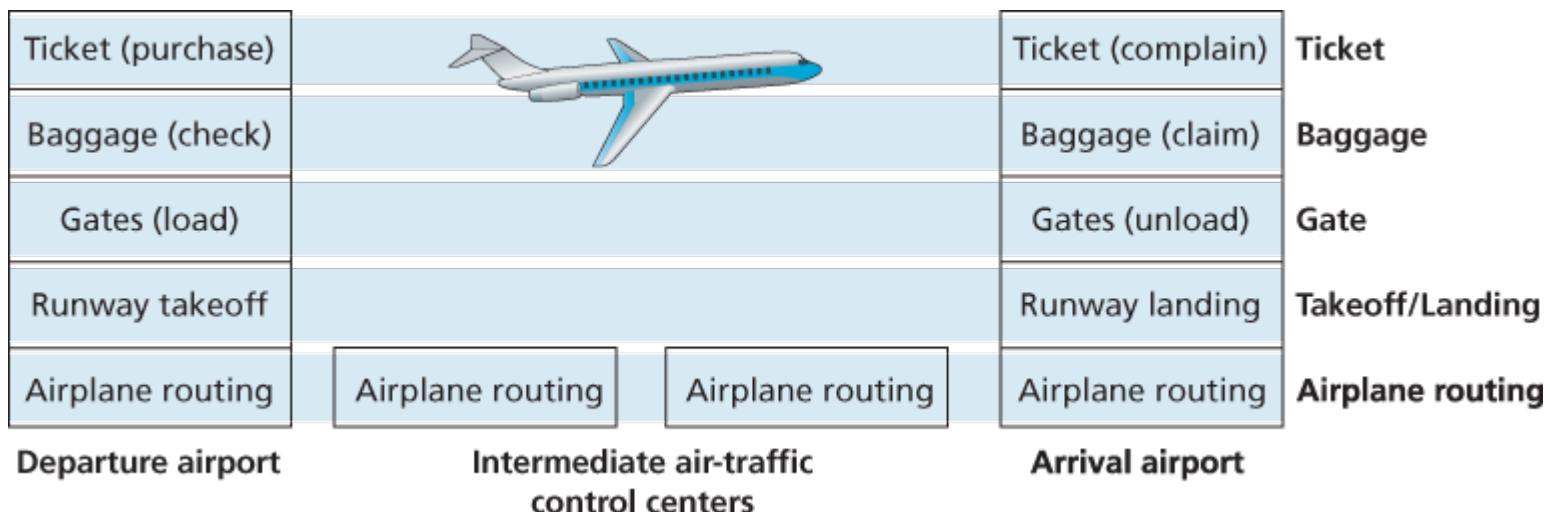


Figure 1.21 Taking an airplane trip: actions



**Figure 1.22** Horizontal layering of airline functionality

Already, we can see some analogies here with computer networking: You are being shipped from source to destination by the airline; a packet is shipped from source host to destination host in the Internet. But this is not quite the analogy we are after. We are looking for some *structure* in [Figure 1.21](#). Looking at [Figure 1.21](#), we note that there is a ticketing function at each end; there is also a baggage function for already-ticketed passengers, and a gate function for already-ticketed and already-baggage-checked passengers. For passengers who have made it through the gate (that is, passengers who are already ticketed, baggage-checked, and through the gate), there is a takeoff and landing function, and while in flight, there is an airplane-routing function. This suggests that we can look at the functionality in [Figure 1.21](#) in a *horizontal* manner, as shown in [Figure 1.22](#).

[Figure 1.22](#) has divided the airline functionality into layers, providing a framework in which we can discuss airline travel. Note that each layer, combined with the layers below it, implements some functionality, some *service*. At the ticketing layer and below, airline-counter-to-airline-counter transfer of a person is accomplished. At the baggage layer and below, baggage-check-to-baggage-claim transfer of a person and bags is accomplished. Note that the baggage layer provides this service only to an already-ticketed person. At the gate layer, departure-gate-to-arrival-gate transfer of a person and bags is accomplished. At the takeoff/landing layer, runway-to-runway transfer of people and their bags is accomplished. Each layer provides its service by (1) performing certain actions within that layer (for example, at the gate layer, loading and unloading people from an airplane) and by (2) using the services of the layer directly below it (for example, in the gate layer, using the runway-to-runway passenger transfer service of the takeoff/landing layer).

A layered architecture allows us to discuss a well-defined, specific part of a large and complex system. This simplification itself is of considerable value by providing modularity, making it much easier to change the implementation of the service provided by the layer. As long as the layer provides the same service to the layer above it, and uses the same services from the layer below it, the remainder of the system remains unchanged when a layer's implementation is changed. (Note that changing the

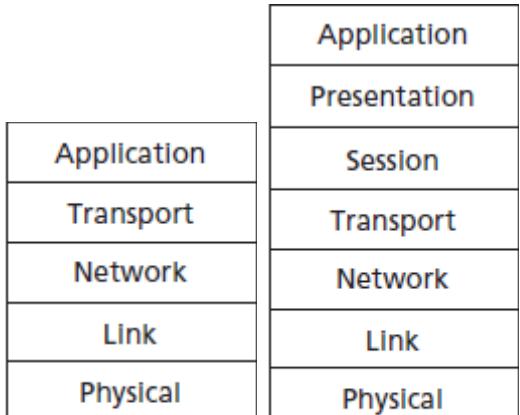
implementation of a service is very different from changing the service itself!) For example, if the gate functions were changed (for instance, to have people board and disembark by height), the remainder of the airline system would remain unchanged since the gate layer still provides the same function (loading and unloading people); it simply implements that function in a different manner after the change. For large and complex systems that are constantly being updated, the ability to change the implementation of a service without affecting other components of the system is another important advantage of layering.

### *Protocol Layering*

But enough about airlines. Let's now turn our attention to network protocols. To provide structure to the design of network protocols, network designers organize protocols—and the network hardware and software that implement the protocols—in **layers**. Each protocol belongs to one of the layers, just as each function in the airline architecture in [Figure 1.22](#) belonged to a layer. We are again interested in the **services** that a layer offers to the layer above—the so-called **service model** of a layer. Just as in the case of our airline example, each layer provides its service by (1) performing certain actions within that layer and by (2) using the services of the layer directly below it. For example, the services provided by layer  $n$  may include reliable delivery of messages from one edge of the network to the other. This might be implemented by using an unreliable edge-to-edge message delivery service of layer  $n-1$ , and adding layer  $n$  functionality to detect and retransmit lost messages.

A protocol layer can be implemented in software, in hardware, or in a combination of the two. Application-layer protocols—such as HTTP and SMTP—are almost always implemented in software in the end systems; so are transport-layer protocols. Because the physical layer and data link layers are responsible for handling communication over a specific link, they are typically implemented in a network interface card (for example, Ethernet or WiFi interface cards) associated with a given link. The network layer is often a mixed implementation of hardware and software. Also note that just as the functions in the layered airline architecture were distributed among the various airports and flight control centers that make up the system, so too is a layer  $n$  protocol *distributed* among the end systems, packet switches, and other components that make up the network. That is, there's often a piece of a layer  $n$  protocol in each of these network components.

Protocol layering has conceptual and structural advantages [[RFC 3439](#)]. As we have seen, layering provides a structured way to discuss system components. Modularity makes it easier to update system components. We mention, however, that some researchers and networking engineers are vehemently opposed to layering [[Wakeman 1992](#)]. One potential drawback of layering is that one layer may duplicate lower-layer functionality. For example, many protocol stacks provide error recovery



a. Five-layer Internet protocol stack      b. Seven-layer ISO OSI reference model

**Figure 1.23 The Internet protocol stack (a) and OSI reference model (b)**

on both a per-link basis and an end-to-end basis. A second potential drawback is that functionality at one layer may need information (for example, a timestamp value) that is present only in another layer; this violates the goal of separation of layers.

When taken together, the protocols of the various layers are called the **protocol stack**. The Internet protocol stack consists of five layers: the physical, link, network, transport, and application layers, as shown in **Figure 1.23(a)**. If you examine the Table of Contents, you will see that we have roughly organized this book using the layers of the Internet protocol stack. We take a **top-down approach**, first covering the application layer and then proceeding downward.

### *Application Layer*

The application layer is where network applications and their application-layer protocols reside. The Internet's application layer includes many protocols, such as the HTTP protocol (which provides for Web document request and transfer), SMTP (which provides for the transfer of e-mail messages), and FTP (which provides for the transfer of files between two end systems). We'll see that certain network functions, such as the translation of human-friendly names for Internet end systems like [www.ietf.org](http://www.ietf.org) to a 32-bit network address, are also done with the help of a specific application-layer protocol, namely, the domain name system (DNS). We'll see in **Chapter 2** that it is very easy to create and deploy our own new application-layer protocols.

An application-layer protocol is distributed over multiple end systems, with the application in one end system using the protocol to exchange packets of information with the application in another end system. We'll refer to this packet of information at the application layer as a **message**.

### *Transport Layer*

The Internet's transport layer transports application-layer messages between application endpoints. In the Internet there are two transport protocols, TCP and UDP, either of which can transport application-layer messages. TCP provides a connection-oriented service to its applications. This service includes guaranteed delivery of application-layer messages to the destination and flow control (that is, sender/receiver speed matching). TCP also breaks long messages into shorter segments and provides a congestion-control mechanism, so that a source throttles its transmission rate when the network is congested. The UDP protocol provides a connectionless service to its applications. This is a no-frills service that provides no reliability, no flow control, and no congestion control. In this book, we'll refer to a transport-layer packet as a **segment**.

### *Network Layer*

The Internet's network layer is responsible for moving network-layer packets known as **datagrams** from one host to another. The Internet transport-layer protocol (TCP or UDP) in a source host passes a transport-layer segment and a destination address to the network layer, just as you would give the postal service a letter with a destination address. The network layer then provides the service of delivering the segment to the transport layer in the destination host.

The Internet's network layer includes the celebrated IP protocol, which defines the fields in the datagram as well as how the end systems and routers act on these fields. There is only one IP protocol, and all Internet components that have a network layer must run the IP protocol. The Internet's network layer also contains routing protocols that determine the routes that datagrams take between sources and destinations. The Internet has many routing protocols. As we saw in [Section 1.3](#), the Internet is a network of networks, and within a network, the network administrator can run any routing protocol desired. Although the network layer contains both the IP protocol and numerous routing protocols, it is often simply referred to as the IP layer, reflecting the fact that IP is the glue that binds the Internet together.

### *Link Layer*

The Internet's network layer routes a datagram through a series of routers between the source and destination. To move a packet from one node (host or router) to the next node in the route, the network layer relies on the services of the link layer. In particular, at each node, the network layer passes the datagram down to the link layer, which delivers the datagram to the next node along the route. At this next node, the link layer passes the datagram up to the network layer.

The services provided by the link layer depend on the specific link-layer protocol that is employed over the link. For example, some link-layer protocols provide reliable delivery, from transmitting node, over one link, to receiving node. Note that this reliable delivery service is different from the reliable delivery service of TCP, which provides reliable delivery from one end system to another. Examples of link-layer

protocols include Ethernet, WiFi, and the cable access network's DOCSIS protocol. As datagrams typically need to traverse several links to travel from source to destination, a datagram may be handled by different link-layer protocols at different links along its route. For example, a datagram may be handled by Ethernet on one link and by PPP on the next link. The network layer will receive a different service from each of the different link-layer protocols. In this book, we'll refer to the link-layer packets as **frames**.

### *Physical Layer*

While the job of the link layer is to move entire frames from one network element to an adjacent network element, the job of the physical layer is to move the *individual bits* within the frame from one node to the next. The protocols in this layer are again link dependent and further depend on the actual transmission medium of the link (for example, twisted-pair copper wire, single-mode fiber optics). For example, Ethernet has many physical-layer protocols: one for twisted-pair copper wire, another for coaxial cable, another for fiber, and so on. In each case, a bit is moved across the link in a different way.

### *The OSI Model*

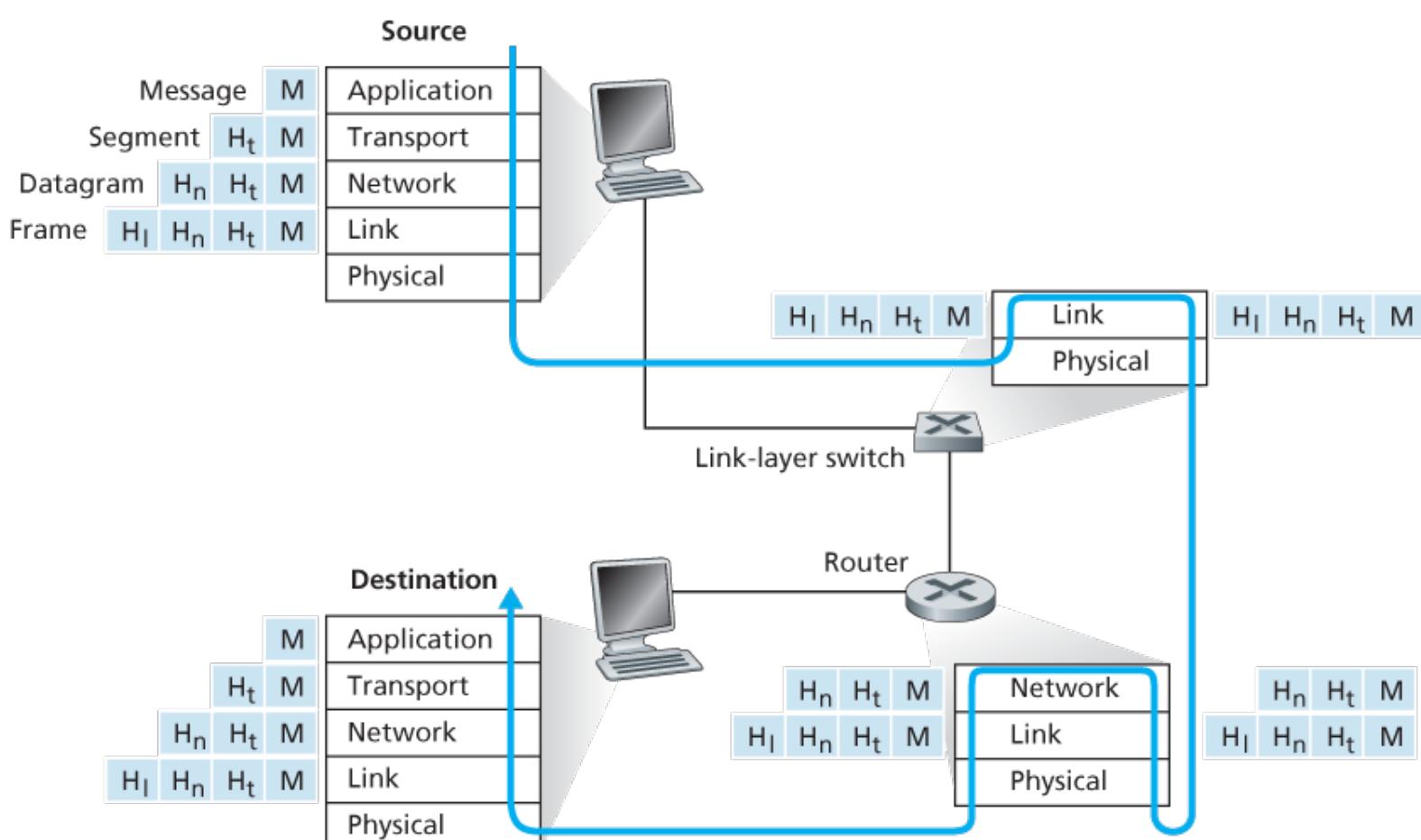
Having discussed the Internet protocol stack in detail, we should mention that it is not the only protocol stack around. In particular, back in the late 1970s, the International Organization for Standardization (ISO) proposed that computer networks be organized around seven layers, called the Open Systems Interconnection (OSI) model [ISO 2016]. The OSI model took shape when the protocols that were to become the Internet protocols were in their infancy, and were but one of many different protocol suites under development; in fact, the inventors of the original OSI model probably did not have the Internet in mind when creating it. Nevertheless, beginning in the late 1970s, many training and university courses picked up on the ISO mandate and organized courses around the seven-layer model. Because of its early impact on networking education, the seven-layer model continues to linger on in some networking textbooks and training courses.

The seven layers of the OSI reference model, shown in [Figure 1.23\(b\)](#), are: application layer, presentation layer, session layer, transport layer, network layer, data link layer, and physical layer. The functionality of five of these layers is roughly the same as their similarly named Internet counterparts. Thus, let's consider the two additional layers present in the OSI reference model—the presentation layer and the session layer. The role of the presentation layer is to provide services that allow communicating applications to interpret the meaning of data exchanged. These services include data compression and data encryption (which are self-explanatory) as well as data description (which frees the applications from having to worry about the internal format in which data are represented/stored—formats that may differ from one computer to another). The session layer provides for delimiting and synchronization of data exchange, including the means to build a checkpointing and recovery scheme.

The fact that the Internet lacks two layers found in the OSI reference model poses a couple of interesting questions: Are the services provided by these layers unimportant? What if an application *needs* one of these services? The Internet's answer to both of these questions is the same—it's up to the application developer. It's up to the application developer to decide if a service is important, and if the service *is* important, it's up to the application developer to build that functionality into the application.

### 1.5.2 Encapsulation

**Figure 1.24** shows the physical path that data takes down a sending end system's protocol stack, up and down the protocol stacks of an intervening link-layer switch



**Figure 1.24 Hosts, routers, and link-layer switches; each contains a different set of layers, reflecting their differences in functionality**

and router, and then up the protocol stack at the receiving end system. As we discuss later in this book, routers and link-layer switches are both packet switches. Similar to end systems, routers and link-layer switches organize their networking hardware and software into layers. But routers and link-layer switches do not implement *all* of the layers in the protocol stack; they typically implement only the bottom layers. As shown in **Figure 1.24**, link-layer switches implement layers 1 and 2; routers implement layers 1 through 3. This means, for example, that Internet routers are capable of implementing the IP protocol (a layer 3 protocol), while link-layer switches are not. We'll see later that

while link-layer switches do not recognize IP addresses, they are capable of recognizing layer 2 addresses, such as Ethernet addresses. Note that hosts implement all five layers; this is consistent with the view that the Internet architecture puts much of its complexity at the edges of the network.

**Figure 1.24** also illustrates the important concept of **encapsulation**. At the sending host, an **application-layer message** ( $M$  in **Figure 1.24**) is passed to the transport layer. In the simplest case, the transport layer takes the message and appends additional information (so-called transport-layer header information,  $H_t$  in **Figure 1.24**) that will be used by the receiver-side transport layer. The application-layer message and the transport-layer header information together constitute the **transport-layer segment**. The transport-layer segment thus encapsulates the application-layer message. The added information might include information allowing the receiver-side transport layer to deliver the message up to the appropriate application, and error-detection bits that allow the receiver to determine whether bits in the message have been changed in route. The transport layer then passes the segment to the network layer, which adds network-layer header information ( $H_n$  in **Figure 1.24**) such as source and destination end system addresses, creating a **network-layer datagram**. The datagram is then passed to the link layer, which (of course!) will add its own link-layer header information and create a **link-layer frame**. Thus, we see that at each layer, a packet has two types of fields: header fields and a **payload field**. The payload is typically a packet from the layer above.

A useful analogy here is the sending of an interoffice memo from one corporate branch office to another via the public postal service. Suppose Alice, who is in one branch office, wants to send a memo to Bob, who is in another branch office. The *memo* is analogous to the *application-layer message*. Alice puts the memo in an interoffice envelope with Bob's name and department written on the front of the envelope. The *interoffice envelope* is analogous to a *transport-layer segment*—it contains header information (Bob's name and department number) and it encapsulates the application-layer message (the memo). When the sending branch-office mailroom receives the interoffice envelope, it puts the interoffice envelope inside yet another envelope, which is suitable for sending through the public postal service. The sending mailroom also writes the postal address of the sending and receiving branch offices on the postal envelope. Here, the *postal envelope* is analogous to the *datagram*—it encapsulates the transport-layer segment (the interoffice envelope), which encapsulates the original message (the memo). The postal service delivers the postal envelope to the receiving branch-office mailroom. There, the process of de-encapsulation is begun. The mailroom extracts the interoffice memo and forwards it to Bob. Finally, Bob opens the envelope and removes the memo.

The process of encapsulation can be more complex than that described above. For example, a large message may be divided into multiple transport-layer segments (which might themselves each be divided into multiple network-layer datagrams). At the receiving end, such a segment must then be reconstructed from its constituent datagrams.

## 1.6 Networks Under Attack

The Internet has become mission critical for many institutions today, including large and small companies, universities, and government agencies. Many individuals also rely on the Internet for many of their professional, social, and personal activities. Billions of “things,” including wearables and home devices, are currently being connected to the Internet. But behind all this utility and excitement, there is a dark side, a side where “bad guys” attempt to wreak havoc in our daily lives by damaging our Internet-connected computers, violating our privacy, and rendering inoperable the Internet services on which we depend.

The field of network security is about how the bad guys can attack computer networks and about how we, soon-to-be experts in computer networking, can defend networks against those attacks, or better yet, design new architectures that are immune to such attacks in the first place. Given the frequency and variety of existing attacks as well as the threat of new and more destructive future attacks, network security has become a central topic in the field of computer networking. One of the features of this textbook is that it brings network security issues to the forefront.

Since we don’t yet have expertise in computer networking and Internet protocols, we’ll begin here by surveying some of today’s more prevalent security-related problems. This will whet our appetite for more substantial discussions in the upcoming chapters. So we begin here by simply asking, what can go wrong? How are computer networks vulnerable? What are some of the more prevalent types of attacks today?

### *The Bad Guys Can Put Malware into Your Host Via the Internet*

We attach devices to the Internet because we want to receive/send data from/to the Internet. This includes all kinds of good stuff, including Instagram posts, Internet search results, streaming music, video conference calls, streaming movies, and so on. But, unfortunately, along with all that good stuff comes malicious stuff—collectively known as **malware**—that can also enter and infect our devices. Once malware infects our device it can do all kinds of devious things, including deleting our files and installing spyware that collects our private information, such as social security numbers, passwords, and keystrokes, and then sends this (over the Internet, of course!) back to the bad guys. Our compromised host may also be enrolled in a network of thousands of similarly compromised devices, collectively known as a **botnet**, which the bad guys control and leverage for spam e-mail distribution or distributed denial-of-service attacks (soon to be discussed) against targeted hosts.

Much of the malware out there today is **self-replicating**: once it infects one host, from that host it seeks entry into other hosts over the Internet, and from the newly infected hosts, it seeks entry into yet more hosts. In this manner, self-replicating malware can spread exponentially fast. Malware can spread in the form of a virus or a worm. **Viruses** are malware that require some form of user interaction to infect the user's device. The classic example is an e-mail attachment containing malicious executable code. If a user receives and opens such an attachment, the user inadvertently runs the malware on the device. Typically, such e-mail viruses are self-replicating: once executed, the virus may send an identical message with an identical malicious attachment to, for example, every recipient in the user's address book. **Worms** are malware that can enter a device without any explicit user interaction. For example, a user may be running a vulnerable network application to which an attacker can send malware. In some cases, without any user intervention, the application may accept the malware from the Internet and run it, creating a worm. The worm in the newly infected device then scans the Internet, searching for other hosts running the same vulnerable network application. When it finds other vulnerable hosts, it sends a copy of itself to those hosts. Today, malware, is pervasive and costly to defend against. As you work through this textbook, we encourage you to think about the following question: What can computer network designers do to defend Internet-attached devices from malware attacks?

### *The Bad Guys Can Attack Servers and Network Infrastructure*

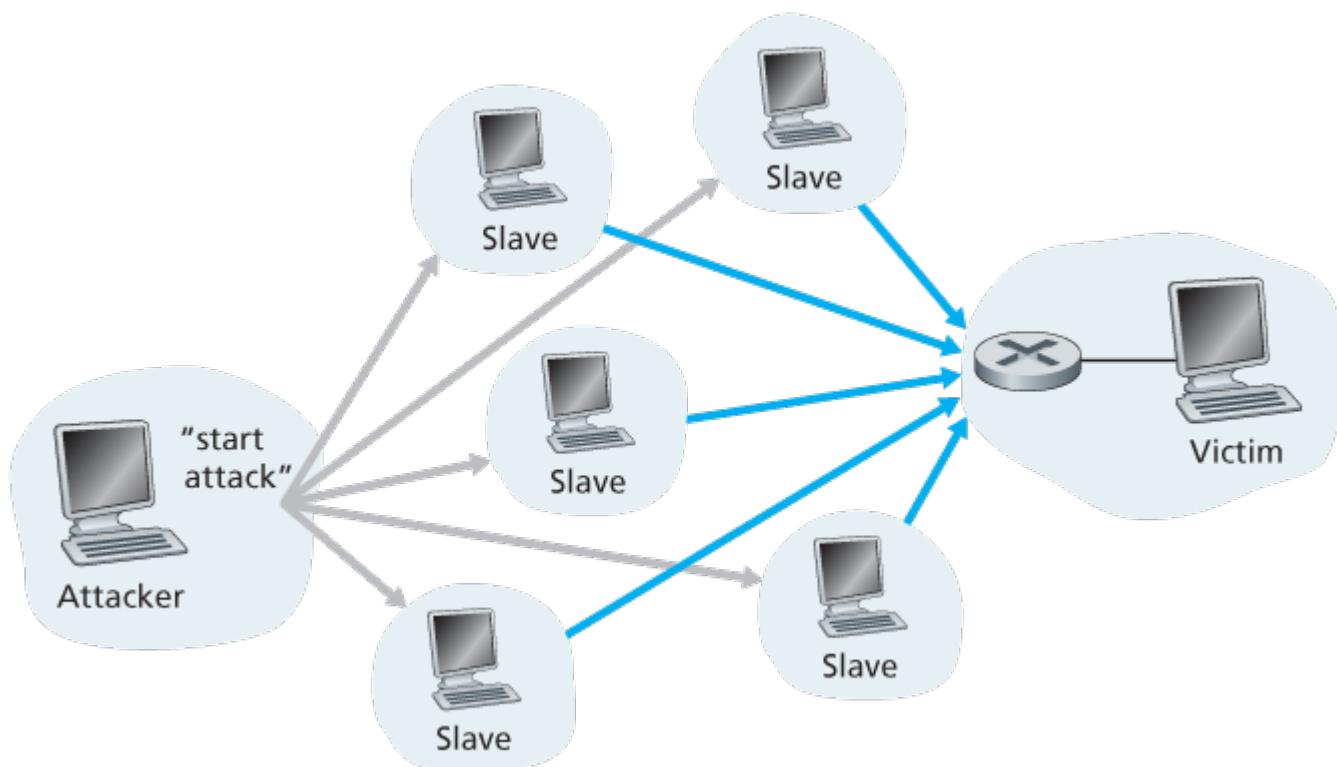
Another broad class of security threats are known as **denial-of-service (DoS) attacks**. As the name suggests, a DoS attack renders a network, host, or other piece of infrastructure unusable by legitimate users. Web servers, e-mail servers, DNS servers (discussed in [Chapter 2](#)), and institutional networks can all be subject to DoS attacks. Internet DoS attacks are extremely common, with thousands of DoS attacks occurring every year [\[Moore 2001\]](#). The site Digital Attack Map allows use to visualize the top daily DoS attacks worldwide [\[DAM 2016\]](#). Most Internet DoS attacks fall into one of three categories:

- **Vulnerability attack.** This involves sending a few well-crafted messages to a vulnerable application or operating system running on a targeted host. If the right sequence of packets is sent to a vulnerable application or operating system, the service can stop or, worse, the host can crash.
- **Bandwidth flooding.** The attacker sends a deluge of packets to the targeted host—so many packets that the target's access link becomes clogged, preventing legitimate packets from reaching the server.
- **Connection flooding.** The attacker establishes a large number of half-open or fully open TCP connections (TCP connections are discussed in [Chapter 3](#)) at the target host. The host can become so bogged down with these bogus connections that it stops accepting legitimate connections.

Let's now explore the bandwidth-flooding attack in more detail. Recalling our delay and loss analysis discussion in [Section 1.4.2](#), it's evident that if the server has an access rate of  $R$  bps, then the attacker will need to send traffic at a rate of approximately  $R$  bps to cause damage. If  $R$  is very large, a single attack source may not be able to generate enough traffic to harm the server. Furthermore, if all the

traffic emanates from a single source, an upstream router may be able to detect the attack and block all traffic from that source before the traffic gets near the server. In a **distributed DoS (DDoS)** attack, illustrated in **Figure 1.25**, the attacker controls multiple sources and has each source blast traffic at the target. With this approach, the aggregate traffic rate across all the controlled sources needs to be approximately  $R$  to cripple the service. DDoS attacks leveraging botnets with thousands of comprised hosts are a common occurrence today [DAM 2016]. DDoS attacks are much harder to detect and defend against than a DoS attack from a single host.

We encourage you to consider the following question as you work your way through this book: What can computer network designers do to defend against DoS attacks? We will see that different defenses are needed for the three types of DoS attacks.



**Figure 1.25 A distributed denial-of-service attack**

#### *The Bad Guys Can Sniff Packets*

Many users today access the Internet via wireless devices, such as WiFi-connected laptops or handheld devices with cellular Internet connections (covered in **Chapter 7**). While ubiquitous Internet access is extremely convenient and enables marvelous new applications for mobile users, it also creates a major security vulnerability—by placing a passive receiver in the vicinity of the wireless transmitter, that receiver can obtain a copy of every packet that is transmitted! These packets can contain all kinds of sensitive information, including passwords, social security numbers, trade secrets, and private personal messages. A passive receiver that records a copy of every packet that flies by is called a **packet sniffer**.

Sniffers can be deployed in wired environments as well. In wired broadcast environments, as in many Ethernet LANs, a packet sniffer can obtain copies of broadcast packets sent over the LAN. As described in [Section 1.2](#), cable access technologies also broadcast packets and are thus vulnerable to sniffing. Furthermore, a bad guy who gains access to an institution's access router or access link to the Internet may be able to plant a sniffer that makes a copy of every packet going to/from the organization. Sniffed packets can then be analyzed offline for sensitive information.

Packet-sniffing software is freely available at various Web sites and as commercial products. Professors teaching a networking course have been known to assign lab exercises that involve writing a packet-sniffing and application-layer data reconstruction program. Indeed, the Wireshark [\[Wireshark 2016\]](#) labs associated with this text (see the introductory [Wireshark lab](#) at the end of this chapter) use exactly such a packet sniffer!

Because packet sniffers are passive—that is, they do not inject packets into the channel—they are difficult to detect. So, when we send packets into a wireless channel, we must accept the possibility that some bad guy may be recording copies of our packets. As you may have guessed, some of the best defenses against packet sniffing involve cryptography. We will examine cryptography as it applies to network security in [Chapter 8](#).

### *The Bad Guys Can Masquerade as Someone You Trust*

It is surprisingly easy (*you* will have the knowledge to do so shortly as you proceed through this text!) to create a packet with an arbitrary source address, packet content, and destination address and then transmit this hand-crafted packet into the Internet, which will dutifully forward the packet to its destination. Imagine the unsuspecting receiver (say an Internet router) who receives such a packet, takes the (false) source address as being truthful, and then performs some command embedded in the packet's contents (say modifies its forwarding table). The ability to inject packets into the Internet with a false source address is known as [IP spoofing](#), and is but one of many ways in which one user can masquerade as another user.

To solve this problem, we will need *end-point authentication*, that is, a mechanism that will allow us to determine with certainty if a message originates from where we think it does. Once again, we encourage you to think about how this can be done for network applications and protocols as you progress through the chapters of this book. We will explore mechanisms for end-point authentication in [Chapter 8](#).

In closing this section, it's worth considering how the Internet got to be such an insecure place in the first place. The answer, in essence, is that the Internet was originally designed to be that way, based on the model of “a group of mutually trusting users attached to a transparent network” [\[Blumenthal 2001\]](#)—a model in which (by definition) there is no need for security. Many aspects of the original Internet architecture deeply reflect this notion of mutual trust. For example, the ability for one user to send a

packet to any other user is the default rather than a requested/granted capability, and user identity is taken at declared face value, rather than being authenticated by default.

But today's Internet certainly does not involve "mutually trusting users." Nonetheless, today's users still need to communicate when they don't necessarily trust each other, may wish to communicate anonymously, may communicate indirectly through third parties (e.g., Web caches, which we'll study in [Chapter 2](#), or mobility-assisting agents, which we'll study in [Chapter 7](#)), and may distrust the hardware, software, and even the air through which they communicate. We now have many security-related challenges before us as we progress through this book: We should seek defenses against sniffing, endpoint masquerading, man-in-the-middle attacks, DDoS attacks, malware, and more. We should keep in mind that communication among mutually trusted users is the exception rather than the rule. Welcome to the world of modern computer networking!

## 1.7 History of Computer Networking and the Internet

Sections 1.1 through 1.6 presented an overview of the technology of computer networking and the Internet. You should know enough now to impress your family and friends! However, if you really want to be a big hit at the next cocktail party, you should sprinkle your discourse with tidbits about the fascinating history of the Internet [Segaller 1998].

### 1.7.1 The Development of Packet Switching: 1961–1972

The field of computer networking and today’s Internet trace their beginnings back to the early 1960s, when the telephone network was the world’s dominant communication network. Recall from Section 1.3 that the telephone network uses circuit switching to transmit information from a sender to a receiver—an appropriate choice given that voice is transmitted at a constant rate between sender and receiver. Given the increasing importance of computers in the early 1960s and the advent of timeshared computers, it was perhaps natural to consider how to hook computers together so that they could be shared among geographically distributed users. The traffic generated by such users was likely to be *bursty*—intervals of activity, such as the sending of a command to a remote computer, followed by periods of inactivity while waiting for a reply or while contemplating the received response.

Three research groups around the world, each unaware of the others’ work [Leiner 1998], began inventing packet switching as an efficient and robust alternative to circuit switching. The first published work on packet-switching techniques was that of Leonard Kleinrock [Kleinrock 1961; Kleinrock 1964], then a graduate student at MIT. Using queuing theory, Kleinrock’s work elegantly demonstrated the effectiveness of the packet-switching approach for bursty traffic sources. In 1964, Paul Baran [Baran 1964] at the Rand Institute had begun investigating the use of packet switching for secure voice over military networks, and at the National Physical Laboratory in England, Donald Davies and Roger Scantlebury were also developing their ideas on packet switching.

The work at MIT, Rand, and the NPL laid the foundations for today’s Internet. But the Internet also has a long history of a let’s-build-it-and-demonstrate-it attitude that also dates back to the 1960s. J. C. R. Licklider [DEC 1990] and Lawrence Roberts, both colleagues of Kleinrock’s at MIT, went on to lead the computer science program at the Advanced Research Projects Agency (ARPA) in the United States. Roberts published an overall plan for the ARPAnet [Roberts 1967], the first packet-switched computer network and a direct ancestor of today’s public Internet. On Labor Day in 1969, the first packet switch was installed at UCLA under Kleinrock’s supervision, and three additional packet switches were installed

shortly thereafter at the Stanford Research Institute (SRI), UC Santa Barbara, and the University of Utah ([Figure 1.26](#)). The fledgling precursor to the Internet was four nodes large by the end of 1969. Kleinrock recalls the very first use of the network to perform a remote login from UCLA to SRI, crashing the system [[Kleinrock 2004](#)].

By 1972, ARPAnet had grown to approximately 15 nodes and was given its first public demonstration by Robert Kahn. The first host-to-host protocol between ARPAnet end systems, known as the network-control protocol (NCP), was completed [[RFC 001](#)]. With an end-to-end protocol available, applications could now be written. Ray Tomlinson wrote the first e-mail program in 1972.

## 1.7.2 Proprietary Networks and Internetworking: 1972–1980

The initial ARPAnet was a single, closed network. In order to communicate with an ARPAnet host, one had to be actually attached to another ARPAnet IMP. In the early to mid-1970s, additional stand-alone packet-switching networks besides ARPAnet came into being: ALOHANet, a microwave network linking universities on the Hawaiian islands [[Abramson 1970](#)], as well as DARPA's packet-satellite [[RFC 829](#)]

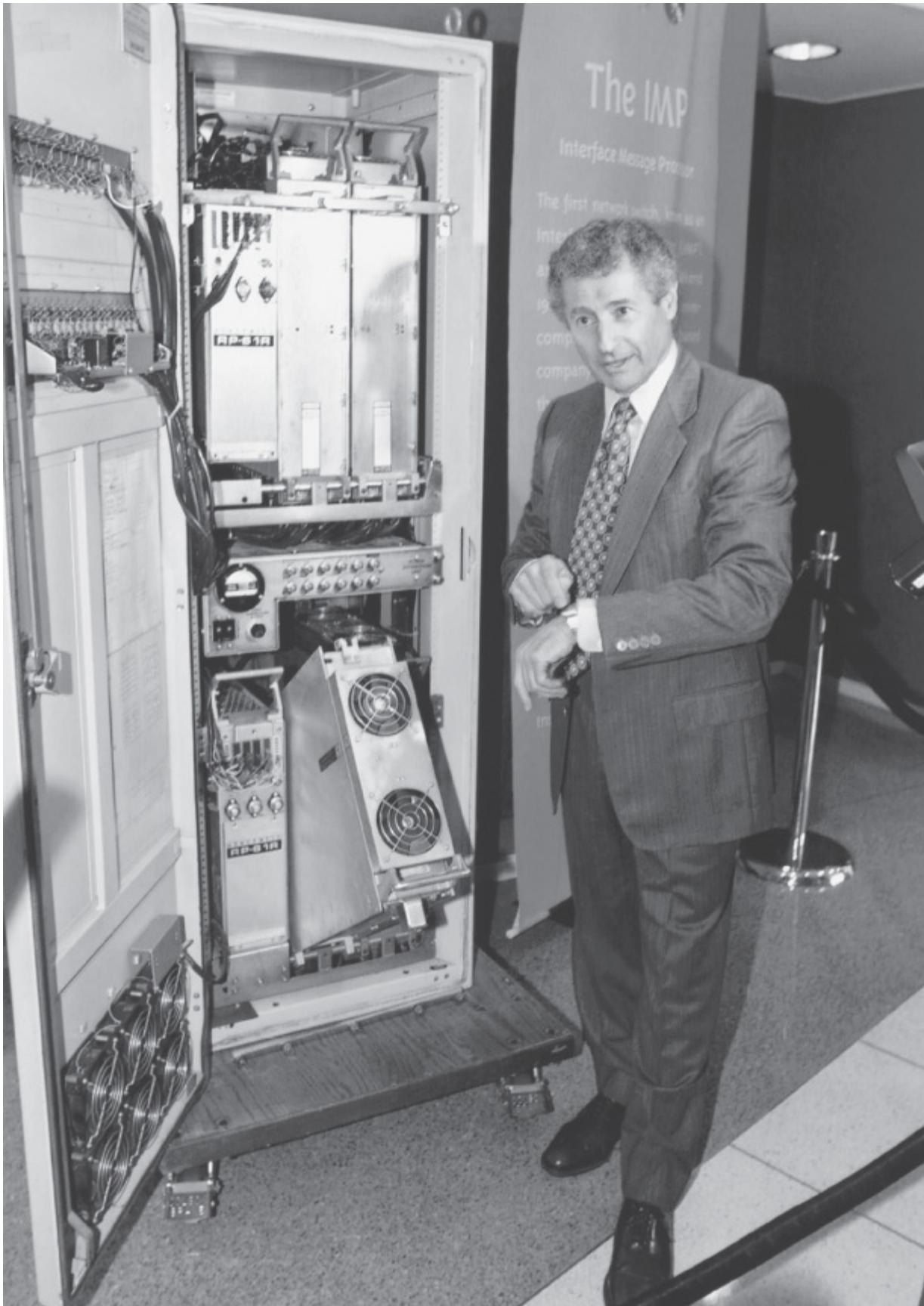


Figure 1.26 An early packet switch

and packet-radio networks [[Kahn 1978](#)]; Telenet, a BBN commercial packet-switching network based on ARPAnet technology; Cyclades, a French packet-switching network pioneered by Louis Pouzin [[Think 2012](#)]; Time-sharing networks such as Tymnet and the GE Information Services network, among others, in the late 1960s and early 1970s [[Schwartz 1977](#)]; IBM's SNA (1969–1974), which paralleled the ARPAnet work [[Schwartz 1977](#)].

The number of networks was growing. With perfect hindsight we can see that the time was ripe for developing an encompassing architecture for connecting networks together. Pioneering work on interconnecting networks (under the sponsorship of the Defense Advanced Research Projects Agency (DARPA)), in essence creating a *network of networks*, was done by Vinton Cerf and Robert Kahn [[Cerf 1974](#)]; the term *internetting* was coined to describe this work.

These architectural principles were embodied in TCP. The early versions of TCP, however, were quite different from today's TCP. The early versions of TCP combined a reliable in-sequence delivery of data via end-system retransmission (still part of today's TCP) with forwarding functions (which today are performed by IP). Early experimentation with TCP, combined with the recognition of the importance of an unreliable, non-flow-controlled, end-to-end transport service for applications such as packetized voice, led to the separation of IP out of TCP and the development of the UDP protocol. The three key Internet protocols that we see today—TCP, UDP, and IP—were conceptually in place by the end of the 1970s.

In addition to the DARPA Internet-related research, many other important networking activities were underway. In Hawaii, Norman Abramson was developing ALOHAnet, a packet-based radio network that allowed multiple remote sites on the Hawaiian Islands to communicate with each other. The ALOHA protocol [[Abramson 1970](#)] was the first multiple-access protocol, allowing geographically distributed users to share a single broadcast communication medium (a radio frequency). Metcalfe and Boggs built on Abramson's multiple-access protocol work when they developed the Ethernet protocol [[Metcalfe 1976](#)] for wire-based shared broadcast networks. Interestingly, Metcalfe and Boggs' Ethernet protocol was motivated by the need to connect multiple PCs, printers, and shared disks [[Perkins 1994](#)]. Twenty-five years ago, well before the PC revolution and the explosion of networks, Metcalfe and Boggs were laying the foundation for today's PC LANs.

### 1.7.3 A Proliferation of Networks: 1980–1990

By the end of the 1970s, approximately two hundred hosts were connected to the ARPAnet. By the end of the 1980s the number of hosts connected to the public Internet, a confederation of networks looking much like today's Internet, would reach a hundred thousand. The 1980s would be a time of tremendous growth.

Much of that growth resulted from several distinct efforts to create computer networks linking universities together. BITNET provided e-mail and file transfers among several universities in the Northeast. CSNET (computer science network) was formed to link university researchers who did not have access to ARPAnet. In 1986, NSFNET was created to provide access to NSF-sponsored supercomputing centers. Starting with an initial backbone speed of 56 kbps, NSFNET's backbone would be running at 1.5 Mbps by the end of the decade and would serve as a primary backbone linking regional networks.

In the ARPAnet community, many of the final pieces of today's Internet architecture were falling into place. January 1, 1983 saw the official deployment of TCP/IP as the new standard host protocol for ARPAnet (replacing the NCP protocol). The transition [\[RFC 801\]](#) from NCP to TCP/IP was a flag day event—all hosts were required to transfer over to TCP/IP as of that day. In the late 1980s, important extensions were made to TCP to implement host-based congestion control [\[Jacobson 1988\]](#). The DNS, used to map between a human-readable Internet name (for example, `gaia.cs.umass.edu`) and its 32-bit IP address, was also developed [\[RFC 1034\]](#).

Paralleling this development of the ARPAnet (which was for the most part a US effort), in the early 1980s the French launched the Minitel project, an ambitious plan to bring data networking into everyone's home. Sponsored by the French government, the Minitel system consisted of a public packet-switched network (based on the X.25 protocol suite), Minitel servers, and inexpensive terminals with built-in low-speed modems. The Minitel became a huge success in 1984 when the French government gave away a free Minitel terminal to each French household that wanted one. Minitel sites included free sites—such as a telephone directory site—as well as private sites, which collected a usage-based fee from each user. At its peak in the mid 1990s, it offered more than 20,000 services, ranging from home banking to specialized research databases. The Minitel was in a large proportion of French homes 10 years before most Americans had ever heard of the Internet.

#### 1.7.4 The Internet Explosion: The 1990s

The 1990s were ushered in with a number of events that symbolized the continued evolution and the soon-to-arrive commercialization of the Internet. ARPAnet, the progenitor of the Internet, ceased to exist. In 1991, NSFNET lifted its restrictions on the use of NSFNET for commercial purposes. NSFNET itself would be decommissioned in 1995, with Internet backbone traffic being carried by commercial Internet Service Providers.

The main event of the 1990s was to be the emergence of the World Wide Web application, which brought the Internet into the homes and businesses of millions of people worldwide. The Web served as a platform for enabling and deploying hundreds of new applications that we take for granted today, including search (e.g., Google and Bing) Internet commerce (e.g., Amazon and eBay) and social networks (e.g., Facebook).

The Web was invented at CERN by Tim Berners-Lee between 1989 and 1991 [\[Berners-Lee 1989\]](#), based on ideas originating in earlier work on hypertext from the 1940s by Vannevar Bush [\[Bush 1945\]](#) and since the 1960s by Ted Nelson [\[Xanadu 2012\]](#). Berners-Lee and his associates developed initial versions of HTML, HTTP, a Web server, and a browser—the four key components of the Web. Around the end of 1993 there were about two hundred Web servers in operation, this collection of servers being

just a harbinger of what was about to come. At about this time several researchers were developing Web browsers with GUI interfaces, including Marc Andreessen, who along with Jim Clark, formed Mosaic Communications, which later became Netscape Communications Corporation [Cusumano 1998; Quittner 1998]. By 1995, university students were using Netscape browsers to surf the Web on a daily basis. At about this time companies—big and small—began to operate Web servers and transact commerce over the Web. In 1996, Microsoft started to make browsers, which started the browser war between Netscape and Microsoft, which Microsoft won a few years later [Cusumano 1998].

The second half of the 1990s was a period of tremendous growth and innovation for the Internet, with major corporations and thousands of startups creating Internet products and services. By the end of the millennium the Internet was supporting hundreds of popular applications, including four killer applications:

- E-mail, including attachments and Web-accessible e-mail
- The Web, including Web browsing and Internet commerce
- Instant messaging, with contact lists
- Peer-to-peer file sharing of MP3s, pioneered by Napster

Interestingly, the first two killer applications came from the research community, whereas the last two were created by a few young entrepreneurs.

The period from 1995 to 2001 was a roller-coaster ride for the Internet in the financial markets. Before they were even profitable, hundreds of Internet startups made initial public offerings and started to be traded in a stock market. Many companies were valued in the billions of dollars without having any significant revenue streams. The Internet stocks collapsed in 2000–2001, and many startups shut down. Nevertheless, a number of companies emerged as big winners in the Internet space, including Microsoft, Cisco, Yahoo, e-Bay, Google, and Amazon.

### 1.7.5 The New Millennium

Innovation in computer networking continues at a rapid pace. Advances are being made on all fronts, including deployments of faster routers and higher transmission speeds in both access networks and in network backbones. But the following developments merit special attention:

- Since the beginning of the millennium, we have been seeing aggressive deployment of broadband Internet access to homes—not only cable modems and DSL but also fiber to the home, as discussed in **Section 1.2**. This high-speed Internet access has set the stage for a wealth of video applications, including the distribution of user-generated video (for example, YouTube), on-demand streaming of movies and television shows (e.g., Netflix), and multi-person video conference (e.g., Skype,

Facetime, and Google Hangouts).

- The increasing ubiquity of high-speed (54 Mbps and higher) public WiFi networks and medium-speed (tens of Mbps) Internet access via 4G cellular telephony networks is not only making it possible to remain constantly connected while on the move, but also enabling new location-specific applications such as Yelp, Tinder, Yik Yak, and Waz. The number of wireless devices connecting to the Internet surpassed the number of wired devices in 2011. This high-speed wireless access has set the stage for the rapid emergence of hand-held computers (iPhones, Androids, iPads, and so on), which enjoy constant and untethered access to the Internet.
- Online social networks—such as Facebook, Instagram, Twitter, and WeChat (hugely popular in China)—have created massive people networks on top of the Internet. Many of these social networks are extensively used for messaging as well as photo sharing. Many Internet users today “live” primarily within one or more social networks. Through their APIs, the online social networks create platforms for new networked applications and distributed games.
- As discussed in **Section 1.3.3**, online service providers, such as Google and Microsoft, have deployed their own extensive private networks, which not only connect together their globally distributed data centers, but are used to bypass the Internet as much as possible by peering directly with lower-tier ISPs. As a result, Google provides search results and e-mail access almost instantaneously, as if their data centers were running within one’s own computer.
- Many Internet commerce companies are now running their applications in the “cloud”—such as in Amazon’s EC2, in Google’s Application Engine, or in Microsoft’s Azure. Many companies and universities have also migrated their Internet applications (e.g., e-mail and Web hosting) to the cloud. Cloud companies not only provide applications scalable computing and storage environments, but also provide the applications implicit access to their high-performance private networks.

## 1.8 Summary

In this chapter we've covered a tremendous amount of material! We've looked at the various pieces of hardware and software that make up the Internet in particular and computer networks in general. We started at the edge of the network, looking at end systems and applications, and at the transport service provided to the applications running on the end systems. We also looked at the link-layer technologies and physical media typically found in the access network. We then dove deeper inside the network, into the network core, identifying packet switching and circuit switching as the two basic approaches for transporting data through a telecommunication network, and we examined the strengths and weaknesses of each approach. We also examined the structure of the global Internet, learning that the Internet is a network of networks. We saw that the Internet's hierarchical structure, consisting of higher- and lower-tier ISPs, has allowed it to scale to include thousands of networks.

In the second part of this introductory chapter, we examined several topics central to the field of computer networking. We first examined the causes of delay, throughput and packet loss in a packet-switched network. We developed simple quantitative models for transmission, propagation, and queuing delays as well as for throughput; we'll make extensive use of these delay models in the homework problems throughout this book. Next we examined protocol layering and service models, key architectural principles in networking that we will also refer back to throughout this book. We also surveyed some of the more prevalent security attacks in the Internet day. We finished our introduction to networking with a brief history of computer networking. The first chapter in itself constitutes a mini-course in computer networking.

So, we have indeed covered a tremendous amount of ground in this first chapter! If you're a bit overwhelmed, don't worry. In the following chapters we'll revisit all of these ideas, covering them in much more detail (that's a promise, not a threat!). At this point, we hope you leave this chapter with a still-developing intuition for the pieces that make up a network, a still-developing command of the vocabulary of networking (don't be shy about referring back to this chapter), and an ever-growing desire to learn more about networking. That's the task ahead of us for the rest of this book.

### Road-Mapping This Book

Before starting any trip, you should always glance at a road map in order to become familiar with the major roads and junctures that lie ahead. For the trip we are about to embark on, the ultimate destination is a deep understanding of the how, what, and why of computer networks. Our road map is

the sequence of chapters of this book:

1. Computer Networks and the Internet
2. Application Layer
3. Transport Layer
4. Network Layer: Data Plane
5. Network Layer: Control Plane
6. The Link Layer and LANs
7. Wireless and Mobile Networks
8. Security in Computer Networks
9. Multimedia Networking

**Chapters 2 through 6** are the five core chapters of this book. You should notice that these chapters are organized around the top four layers of the five-layer Internet protocol. Further note that our journey will begin at the top of the Internet protocol stack, namely, the application layer, and will work its way downward. The rationale behind this top-down journey is that once we understand the applications, we can understand the network services needed to support these applications. We can then, in turn, examine the various ways in which such services might be implemented by a network architecture. Covering applications early thus provides motivation for the remainder of the text.

The second half of the book—**Chapters 7 through 9**—zooms in on three enormously important (and somewhat independent) topics in modern computer networking. In **Chapter 7**, we examine wireless and mobile networks, including wireless LANs (including WiFi and Bluetooth), Cellular telephony networks (including GSM, 3G, and 4G), and mobility (in both IP and GSM networks). **Chapter 8**, which addresses security in computer networks, first looks at the underpinnings of encryption and network security, and then we examine how the basic theory is being applied in a broad range of Internet contexts. The last chapter, which addresses multimedia networking, examines audio and video applications such as Internet phone, video conferencing, and streaming of stored media. We also look at how a packet-switched network can be designed to provide consistent quality of service to audio and video applications.

# Homework Problems and Questions

## Chapter 1 Review Questions

### SECTION 1.1

- R1. What is the difference between a host and an end system? List several different types of end systems. Is a Web server an end system?
- R2. The word *protocol* is often used to describe diplomatic relations. How does Wikipedia describe diplomatic protocol?
- R3. Why are standards important for protocols?

### SECTION 1.2

- R4. List six access technologies. Classify each one as home access, enterprise access, or wide-area wireless access.
- R5. Is HFC transmission rate dedicated or shared among users? Are collisions possible in a downstream HFC channel? Why or why not?
- R6. List the available residential access technologies in your city. For each type of access, provide the advertised downstream rate, upstream rate, and monthly price.
- R7. What is the transmission rate of Ethernet LANs?
- R8. What are some of the physical media that Ethernet can run over?
- R9. Dial-up modems, HFC, DSL and FTTH are all used for residential access. For each of these access technologies, provide a range of transmission rates and comment on whether the transmission rate is shared or dedicated.
- R10. Describe the most popular wireless Internet access technologies today. Compare and contrast them.

### SECTION 1.3

- R11. Suppose there is exactly one packet switch between a sending host and a receiving host. The transmission rates between the sending host and the switch and between the switch and the receiving host are  $R_1$  and  $R_2$ , respectively. Assuming that the switch uses store-and-forward packet switching, what is the total end-to-end delay to send a packet of length  $L$ ? (Ignore queuing, propagation delay, and processing delay.)

R12. What advantage does a circuit-switched network have over a packet-switched network? What advantages does TDM have over FDM in a circuit-switched network?

R13. Suppose users share a 2 Mbps link. Also suppose each user transmits continuously at 1 Mbps when transmitting, but each user transmits only 20 percent of the time. (See the discussion of statistical multiplexing in [Section 1.3](#).)

- a. When circuit switching is used, how many users can be supported?
- b. For the remainder of this problem, suppose packet switching is used. Why will there be essentially no queuing delay before the link if two or fewer users transmit at the same time? Why will there be a queuing delay if three users transmit at the same time?
- c. Find the probability that a given user is transmitting.
- d. Suppose now there are three users. Find the probability that at any given time, all three users are transmitting simultaneously. Find the fraction of time during which the queue grows.

R14. Why will two ISPs at the same level of the hierarchy often peer with each other? How does an IXP earn money?

R15. Some content providers have created their own networks. Describe Google's network. What motivates content providers to create these networks?

#### SECTION 1.4

R16. Consider sending a packet from a source host to a destination host over a fixed route. List the delay components in the end-to-end delay. Which of these delays are constant and which are variable?

R17. Visit the Transmission Versus Propagation Delay applet at the companion Web site. Among the rates, propagation delay, and packet sizes available, find a combination for which the sender finishes transmitting before the first bit of the packet reaches the receiver. Find another combination for which the first bit of the packet reaches the receiver before the sender finishes transmitting.

R18. How long does it take a packet of length 1,000 bytes to propagate over a link of distance 2,500 km, propagation speed  $2.5 \cdot 10^8$  m/s, and transmission rate 2 Mbps? More generally, how long does it take a packet of length  $L$  to propagate over a link of distance  $d$ , propagation speed  $s$ , and transmission rate  $R$  bps? Does this delay depend on packet length? Does this delay depend on transmission rate?

R19. Suppose Host A wants to send a large file to Host B. The path from Host A to Host B has three links, of rates  $R_1=500$  kbps,  $R_2=2$  Mbps, and  $R_3=1$  Mbps.

- a. Assuming no other traffic in the network, what is the throughput for the file transfer?
- b. Suppose the file is 4 million bytes. Dividing the file size by the throughput, roughly how long will it take to transfer the file to Host B?
- c. Repeat (a) and (b), but now with  $R_2$  reduced to 100 kbps.

R20. Suppose end system A wants to send a large file to end system B. At a very high level, describe how end system A creates packets from the file. When one of these packets arrives to a router, what information in the packet does the router use to determine the link onto which the packet is forwarded? Why is packet switching in the Internet analogous to driving from one city to another and asking directions along the way?

R21. Visit the Queuing and Loss applet at the companion Web site. What is the maximum emission rate and the minimum transmission rate? With those rates, what is the traffic intensity? Run the applet with these rates and determine how long it takes for packet loss to occur. Then repeat the experiment a second time and determine again how long it takes for packet loss to occur. Are the values different? Why or why not?

## SECTION 1.5

R22. List five tasks that a layer can perform. Is it possible that one (or more) of these tasks could be performed by two (or more) layers?

R23. What are the five layers in the Internet protocol stack? What are the principal responsibilities of each of these layers?

R24. What is an application-layer message? A transport-layer segment? A network-layer datagram? A link-layer frame?

R25. Which layers in the Internet protocol stack does a router process? Which layers does a link-layer switch process? Which layers does a host process?

## SECTION 1.6

R26. What is the difference between a virus and a worm?

R27. Describe how a botnet can be created and how it can be used for a DDoS attack.

R28. Suppose Alice and Bob are sending packets to each other over a computer network. Suppose Trudy positions herself in the network so that she can capture all the packets sent by Alice and send whatever she wants to Bob; she can also capture all the packets sent by Bob and send whatever she wants to Alice. List some of the malicious things Trudy can do from this position.

## Problems

P1. Design and describe an application-level protocol to be used between an automatic teller machine and a bank's centralized computer. Your protocol should allow a user's card and password to be verified, the account balance (which is maintained at the centralized computer) to be queried, and an account withdrawal to be made (that is, money disbursed to the user).

Your protocol entities should be able to handle the all-too-common case in which there is not enough money in the account to cover the withdrawal. Specify your protocol by listing the messages exchanged and the action taken by the automatic teller machine or the bank's centralized computer on transmission and receipt of messages. Sketch the operation of your protocol for the case of a simple withdrawal with no errors, using a diagram similar to that in

[Figure 1.2](#). Explicitly state the assumptions made by your protocol about the underlying end-to-end transport service.

P2. [Equation 1.1](#) gives a formula for the end-to-end delay of sending one packet of length  $L$  over  $N$  links of transmission rate  $R$ . Generalize this formula for sending  $P$  such packets back-to-back over the  $N$  links.

P3. Consider an application that transmits data at a steady rate (for example, the sender generates an  $N$ -bit unit of data every  $k$  time units, where  $k$  is small and fixed). Also, when such an application starts, it will continue running for a relatively long period of time. Answer the following questions, briefly justifying your answer:

- a. Would a packet-switched network or a circuit-switched network be more appropriate for this application? Why?
- b. Suppose that a packet-switched network is used and the only traffic in this network comes from such applications as described above. Furthermore, assume that the sum of the application data rates is less than the capacities of each and every link. Is some form of congestion control needed? Why?

P4. Consider the circuit-switched network in [Figure 1.13](#). Recall that there are 4 circuits on each link. Label the four switches A, B, C, and D, going in the clockwise direction.

- a. What is the maximum number of simultaneous connections that can be in progress at any one time in this network?
- b. Suppose that all connections are between switches A and C. What is the maximum number of simultaneous connections that can be in progress?
- c. Suppose we want to make four connections between switches A and C, and another four connections between switches B and D. Can we route these calls through the four links to accommodate all eight connections?

P5. Review the car-caravan analogy in [Section 1.4](#). Assume a propagation speed of 100 km/hour.

- a. Suppose the caravan travels 150 km, beginning in front of one tollbooth, passing through a second tollbooth, and finishing just after a third tollbooth. What is the end-to-end delay?
- b. Repeat (a), now assuming that there are eight cars in the caravan instead of ten.

P6. This elementary problem begins to explore propagation delay and transmission delay, two central concepts in data networking. Consider two hosts, A and B, connected by a single link of rate  $R$  bps. Suppose that the two hosts are separated by  $m$  meters, and suppose the

propagation speed along the link is  $s$  meters/sec. Host A is to send a packet of size  $L$  bits to Host B.



VideoNote

Exploring propagation delay and transmission delay

- a. Express the propagation delay,  $d_{\text{prop}}$ , in terms of  $m$  and  $s$ .
- b. Determine the transmission time of the packet,  $d_{\text{trans}}$ , in terms of  $L$  and  $R$ .
- c. Ignoring processing and queuing delays, obtain an expression for the end-to-end delay.
- d. Suppose Host A begins to transmit the packet at time  $t=0$ . At time  $t=d_{\text{trans}}$ , where is the last bit of the packet?
- e. Suppose  $d_{\text{prop}}$  is greater than  $d_{\text{trans}}$ . At time  $t=d_{\text{trans}}$ , where is the first bit of the packet?
- f. Suppose  $d_{\text{prop}}$  is less than  $d_{\text{trans}}$ . At time  $t=d_{\text{trans}}$ , where is the first bit of the packet?
- g. Suppose  $s=2.5 \cdot 10^8$ ,  $L=120$  bits, and  $R=56$  kbps. Find the distance  $m$  so that  $d_{\text{prop}}$  equals  $d_{\text{trans}}$ .

P7. In this problem, we consider sending real-time voice from Host A to Host B over a packet-switched network (VoIP). Host A converts analog voice to a digital 64 kbps bit stream on the fly. Host A then groups the bits into 56-byte packets. There is one link between Hosts A and B; its transmission rate is 2 Mbps and its propagation delay is 10 msec. As soon as Host A gathers a packet, it sends it to Host B. As soon as Host B receives an entire packet, it converts the packet's bits to an analog signal. How much time elapses from the time a bit is created (from the original analog signal at Host A) until the bit is decoded (as part of the analog signal at Host B)?

P8. Suppose users share a 3 Mbps link. Also suppose each user requires 150 kbps when transmitting, but each user transmits only 10 percent of the time. (See the discussion of packet switching versus circuit switching in [Section 1.3](#).)

- a. When circuit switching is used, how many users can be supported?
- b. For the remainder of this problem, suppose packet switching is used. Find the probability that a given user is transmitting.
- c. Suppose there are 120 users. Find the probability that at any given time, exactly  $n$  users are transmitting simultaneously. (*Hint:* Use the binomial distribution.)
- d. Find the probability that there are 21 or more users transmitting simultaneously.

P9. Consider the discussion in [Section 1.3](#) of packet switching versus circuit switching in which an example is provided with a 1 Mbps link. Users are generating data at a rate of 100 kbps when busy, but are busy generating data only with probability  $p=0.1$ . Suppose that the 1 Mbps link is

replaced by a 1 Gbps link.

- a. What is  $N$ , the maximum number of users that can be supported simultaneously under circuit switching?
- b. Now consider packet switching and a user population of  $M$  users. Give a formula (in terms of  $p$ ,  $M$ ,  $N$ ) for the probability that more than  $N$  users are sending data.

P10. Consider a packet of length  $L$  that begins at end system A and travels over three links to a destination end system. These three links are connected by two packet switches. Let  $d_i$ ,  $s_i$ , and  $R_i$  denote the length, propagation speed, and the transmission rate of link  $i$ , for  $i=1,2,3$ . The packet switch delays each packet by  $d_{\text{proc}}$ . Assuming no queuing delays, in terms of  $d_i$ ,  $s_i$ ,  $R_i$ , ( $i=1,2,3$ ), and  $L$ , what is the total end-to-end delay for the packet? Suppose now the packet is 1,500 bytes, the propagation speed on all three links is 2.5·108m/s, the transmission rates of all three links are 2 Mbps, the packet switch processing delay is 3 msec, the length of the first link is 5,000 km, the length of the second link is 4,000 km, and the length of the last link is 1,000 km. For these values, what is the end-to-end delay?

P11. In the above problem, suppose  $R_1=R_2=R_3=R$  and  $d_{\text{proc}}=0$ . Further suppose the packet switch does not store-and-forward packets but instead immediately transmits each bit it receives before waiting for the entire packet to arrive. What is the end-to-end delay?

P12. A packet switch receives a packet and determines the outbound link to which the packet should be forwarded. When the packet arrives, one other packet is halfway done being transmitted on this outbound link and four other packets are waiting to be transmitted. Packets are transmitted in order of arrival. Suppose all packets are 1,500 bytes and the link rate is 2 Mbps. What is the queuing delay for the packet? More generally, what is the queuing delay when all packets have length  $L$ , the transmission rate is  $R$ ,  $x$  bits of the currently-being-transmitted packet have been transmitted, and  $n$  packets are already in the queue?

P13.

- a. Suppose  $N$  packets arrive simultaneously to a link at which no packets are currently being transmitted or queued. Each packet is of length  $L$  and the link has transmission rate  $R$ . What is the average queuing delay for the  $N$  packets?
- b. Now suppose that  $N$  such packets arrive to the link every  $LN/R$  seconds. What is the average queuing delay of a packet?

P14. Consider the queuing delay in a router buffer. Let  $\lambda$  denote traffic intensity; that is,  $\lambda=La/R$ . Suppose that the queuing delay takes the form  $\lambda L/R(1-\lambda)$  for  $\lambda < 1$ .

- a. Provide a formula for the total delay, that is, the queuing delay plus the transmission delay.
- b. Plot the total delay as a function of  $L/R$ .

P15. Let  $a$  denote the rate of packets arriving at a link in packets/sec, and let  $\mu$  denote the link's transmission rate in packets/sec. Based on the formula for the total delay (i.e., the queuing delay

plus the transmission delay) derived in the previous problem, derive a formula for the total delay in terms of  $a$  and  $\mu$ .

P16. Consider a router buffer preceding an outbound link. In this problem, you will use Little's formula, a famous formula from queuing theory. Let  $N$  denote the average number of packets in the buffer plus the packet being transmitted. Let  $a$  denote the rate of packets arriving at the link. Let  $d$  denote the average total delay (i.e., the queuing delay plus the transmission delay) experienced by a packet. Little's formula is  $N=a \cdot d$ . Suppose that on average, the buffer contains 10 packets, and the average packet queuing delay is 10 msec. The link's transmission rate is 100 packets/sec. Using Little's formula, what is the average packet arrival rate, assuming there is no packet loss?

P17.

- Generalize [Equation 1.2](#) in [Section 1.4.3](#) for heterogeneous processing rates, transmission rates, and propagation delays.
- Repeat (a), but now also suppose that there is an average queuing delay of  $d_{\text{queue}}$  at each node.

P18. Perform a Traceroute between source and destination on the same continent at three different hours of the day.



VideoNote

Using Traceroute to discover network paths and measure network delay

- Find the average and standard deviation of the round-trip delays at each of the three hours.
- Find the number of routers in the path at each of the three hours. Did the paths change during any of the hours?
- Try to identify the number of ISP networks that the Traceroute packets pass through from source to destination. Routers with similar names and/or similar IP addresses should be considered as part of the same ISP. In your experiments, do the largest delays occur at the peering interfaces between adjacent ISPs?
- Repeat the above for a source and destination on different continents. Compare the intra-continent and inter-continent results.

P19.

- Visit the site [www.traceroute.org](http://www.traceroute.org) and perform traceroutes from two different cities in France to the same destination host in the United States. How many links are the same

- in the two traceroutes? Is the transatlantic link the same?
- Repeat (a) but this time choose one city in France and another city in Germany.
  - Pick a city in the United States, and perform traceroutes to two hosts, each in a different city in China. How many links are common in the two traceroutes? Do the two traceroutes diverge before reaching China?

P20. Consider the throughput example corresponding to [Figure 1.20\(b\)](#). Now suppose that there are  $M$  client-server pairs rather than 10. Denote  $R_s$ ,  $R_c$ , and  $R$  for the rates of the server links, client links, and network link. Assume all other links have abundant capacity and that there is no other traffic in the network besides the traffic generated by the  $M$  client-server pairs. Derive a general expression for throughput in terms of  $R_s$ ,  $R_c$ ,  $R$ , and  $M$ .

P21. Consider [Figure 1.19\(b\)](#). Now suppose that there are  $M$  paths between the server and the client. No two paths share any link. Path  $k$  ( $k=1, \dots, M$ ) consists of  $N$  links with transmission rates  $R_{1k}, R_{2k}, \dots, R_{Nk}$ . If the server can only use one path to send data to the client, what is the maximum throughput that the server can achieve? If the server can use all  $M$  paths to send data, what is the maximum throughput that the server can achieve?

P22. Consider [Figure 1.19\(b\)](#). Suppose that each link between the server and the client has a packet loss probability  $p$ , and the packet loss probabilities for these links are independent. What is the probability that a packet (sent by the server) is successfully received by the receiver? If a packet is lost in the path from the server to the client, then the server will re-transmit the packet. On average, how many times will the server re-transmit the packet in order for the client to successfully receive the packet?

P23. Consider [Figure 1.19\(a\)](#). Assume that we know the bottleneck link along the path from the server to the client is the first link with rate  $R_s$  bits/sec. Suppose we send a pair of packets back to back from the server to the client, and there is no other traffic on this path. Assume each packet of size  $L$  bits, and both links have the same propagation delay  $d_{\text{prop}}$ .

- What is the packet inter-arrival time at the destination? That is, how much time elapses from when the last bit of the first packet arrives until the last bit of the second packet arrives?
- Now assume that the second link is the bottleneck link (i.e.,  $R_c < R_s$ ). Is it possible that the second packet queues at the input queue of the second link? Explain. Now suppose that the server sends the second packet  $T$  seconds after sending the first packet. How large must  $T$  be to ensure no queuing before the second link? Explain.

P24. Suppose you would like to urgently deliver 40 terabytes data from Boston to Los Angeles. You have available a 100 Mbps dedicated link for data transfer. Would you prefer to transmit the data via this link or instead use FedEx over-night delivery? Explain.

P25. Suppose two hosts, A and B, are separated by 20,000 kilometers and are connected by a direct link of  $R=2$  Mbps. Suppose the propagation speed over the link is  $2.5 \cdot 10^8$  meters/sec.

- Calculate the bandwidth-delay product,  $R \cdot d_{\text{prop}}$ .

- b. Consider sending a file of 800,000 bits from Host A to Host B. Suppose the file is sent continuously as one large message. What is the maximum number of bits that will be in the link at any given time?
- c. Provide an interpretation of the bandwidth-delay product.
- d. What is the width (in meters) of a bit in the link? Is it longer than a football field?
- e. Derive a general expression for the width of a bit in terms of the propagation speed  $s$ , the transmission rate  $R$ , and the length of the link  $m$ .

P26. Referring to problem P25, suppose we can modify  $R$ . For what value of  $R$  is the width of a bit as long as the length of the link?

P27. Consider problem P25 but now with a link of  $R=1$  Gbps.

- a. Calculate the bandwidth-delay product,  $R \cdot d_{\text{prop}}$ .
- b. Consider sending a file of 800,000 bits from Host A to Host B. Suppose the file is sent continuously as one big message. What is the maximum number of bits that will be in the link at any given time?
- c. What is the width (in meters) of a bit in the link?

P28. Refer again to problem P25.

- a. How long does it take to send the file, assuming it is sent continuously?
- b. Suppose now the file is broken up into 20 packets with each packet containing 40,000 bits. Suppose that each packet is acknowledged by the receiver and the transmission time of an acknowledgment packet is negligible. Finally, assume that the sender cannot send a packet until the preceding one is acknowledged. How long does it take to send the file?
- c. Compare the results from (a) and (b).

P29. Suppose there is a 10 Mbps microwave link between a geostationary satellite and its base station on Earth. Every minute the satellite takes a digital photo and sends it to the base station. Assume a propagation speed of  $2.4 \cdot 10^8$  meters/sec.

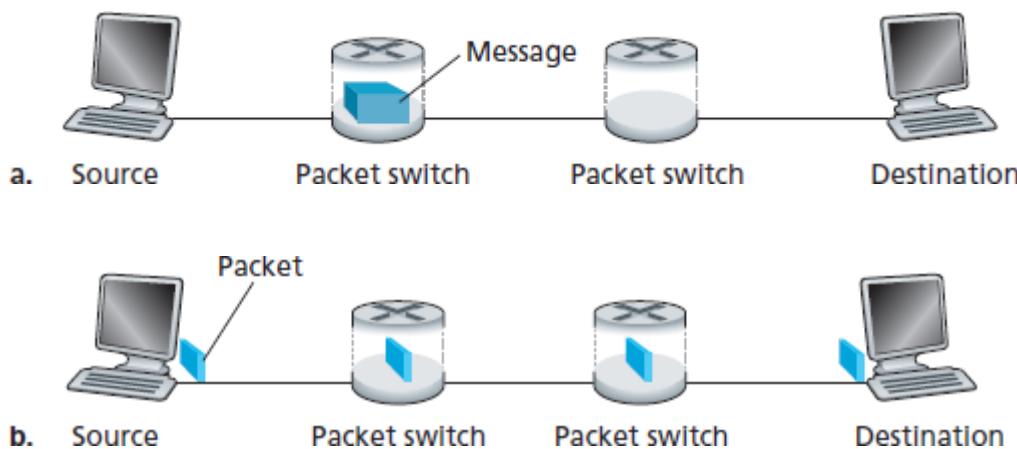
- a. What is the propagation delay of the link?
- b. What is the bandwidth-delay product,  $R \cdot d_{\text{prop}}$ ?
- c. Let  $x$  denote the size of the photo. What is the minimum value of  $x$  for the microwave link to be continuously transmitting?

P30. Consider the airline travel analogy in our discussion of layering in [Section 1.5](#), and the addition of headers to protocol data units as they flow down the protocol stack. Is there an equivalent notion of header information that is added to passengers and baggage as they move down the airline protocol stack?

P31. In modern packet-switched networks, including the Internet, the source host segments long, application-layer messages (for example, an image or a music file) into smaller packets

and sends the packets into the network. The receiver then reassembles the packets back into the original message. We refer to this process as *message segmentation*. [Figure 1.27](#) illustrates the end-to-end transport of a message with and without message segmentation. Consider a message that is 8·10<sup>6</sup> bits long that is to be sent from source to destination in [Figure 1.27](#). Suppose each link in the figure is 2 Mbps. Ignore propagation, queuing, and processing delays.

- a. Consider sending the message from source to destination *without* message segmentation. How long does it take to move the message from the source host to the first packet switch? Keeping in mind that each switch uses store-and-forward packet switching, what is the total time to move the message from source host to destination host?
- b. Now suppose that the message is segmented into 800 packets, with each packet being 10,000 bits long. How long does it take to move the first packet from source host to the first switch? When the first packet is being sent from the first switch to the second switch, the second packet is being sent from the source host to the first switch. At what time will the second packet be fully received at the first switch?
- c. How long does it take to move the file from source host to destination host when message segmentation is used? Compare this result with your answer in part (a) and comment.



**Figure 1.27 End-to-end message transport: (a) without message segmentation; (b) with message segmentation**

- d. In addition to reducing delay, what are reasons to use message segmentation?
- e. Discuss the drawbacks of message segmentation.

P32. Experiment with the Message Segmentation applet at the book's Web site. Do the delays in the applet correspond to the delays in the previous problem? How do link propagation delays affect the overall end-to-end delay for packet switching (with message segmentation) and for message switching?

P33. Consider sending a large file of  $F$  bits from Host A to Host B. There are three links (and two switches) between A and B, and the links are uncongested (that is, no queuing delays). Host A

segments the file into segments of  $S$  bits each and adds 80 bits of header to each segment, forming packets of  $L=80 + S$  bits. Each link has a transmission rate of  $R$  bps. Find the value of  $S$  that minimizes the delay of moving the file from Host A to Host B. Disregard propagation delay.

P34. Skype offers a service that allows you to make a phone call from a PC to an ordinary phone. This means that the voice call must pass through both the Internet and through a telephone network. Discuss how this might be done.

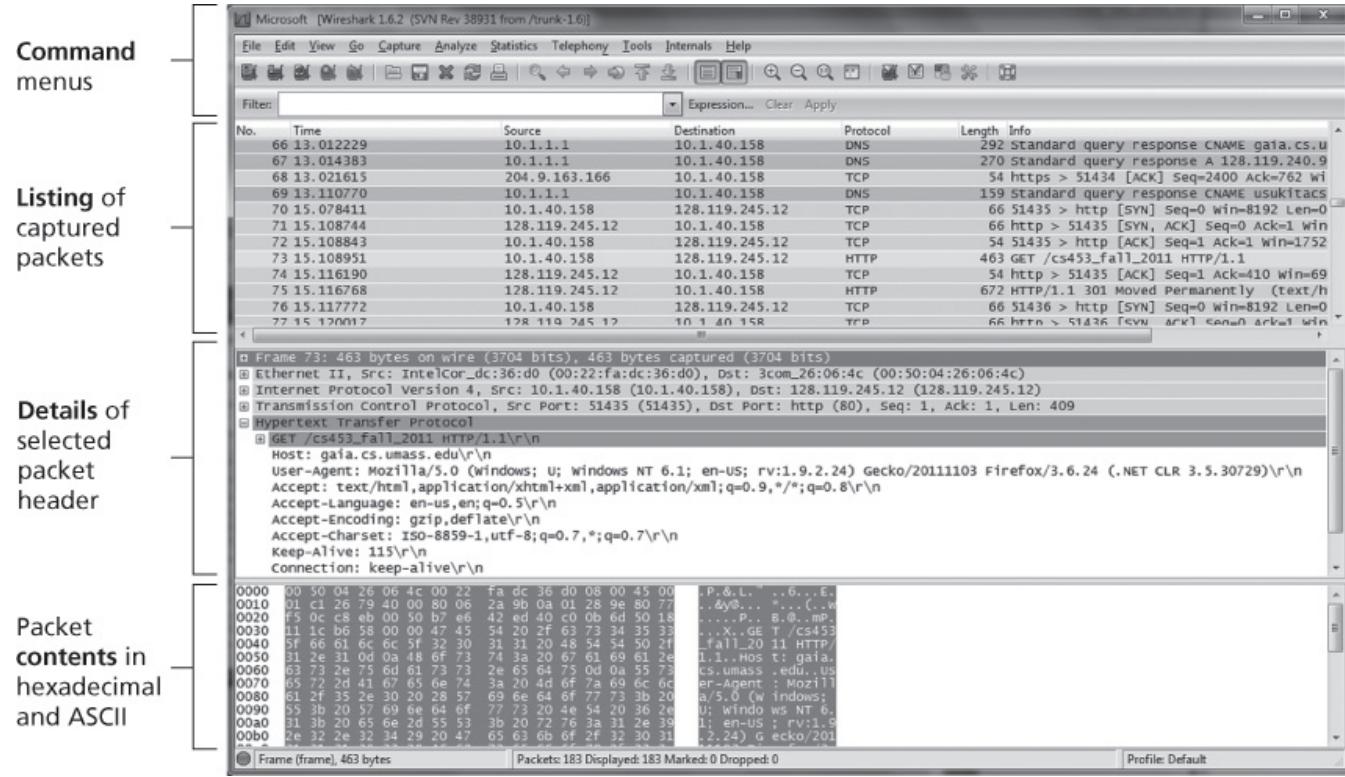
## Wireshark Lab

*“Tell me and I forget. Show me and I remember. Involve me and I understand.”*

*Chinese proverb*

One's understanding of network protocols can often be greatly deepened by seeing them in action and by playing around with them—observing the sequence of messages exchanged between two protocol entities, delving into the details of protocol operation, causing protocols to perform certain actions, and observing these actions and their consequences. This can be done in simulated scenarios or in a real network environment such as the Internet. The Java applets at the textbook Web site take the first approach. In the Wireshark labs, we'll take the latter approach. You'll run network applications in various scenarios using a computer on your desk, at home, or in a lab. You'll observe the network protocols in your computer, interacting and exchanging messages with protocol entities executing elsewhere in the Internet. Thus, you and your computer will be an integral part of these live labs. You'll observe—and you'll learn—by doing.

The basic tool for observing the messages exchanged between executing protocol entities is called a **packet sniffer**. As the name suggests, a packet sniffer passively copies (sniffs) messages being sent from and received by your computer; it also displays the contents of the various protocol fields of these captured messages. A screenshot of the Wireshark packet sniffer is shown in [Figure 1.28](#). Wireshark is a free packet sniffer that runs on Windows, Linux/Unix, and Mac computers.



**Figure 1.28 A Wireshark screenshot (Wireshark screenshot reprinted by permission of the Wireshark Foundation.)**

Throughout the textbook, you will find Wireshark labs that allow you to explore a number of the protocols studied in the chapter. In this first Wireshark lab, you'll obtain and install a copy of Wireshark, access a Web site, and capture and examine the protocol messages being exchanged between your Web browser and the Web server.

You can find full details about this first Wireshark lab (including instructions about how to obtain and install Wireshark) at the Web site <http://www.pearsonhighered.com/cs-resources/>.

#### AN INTERVIEW WITH...

##### **Leonard Kleinrock**

Leonard Kleinrock is a professor of computer science at the University of California, Los Angeles. In 1969, his computer at UCLA became the first node of the Internet. His creation of packet-switching principles in 1961 became the technology behind the Internet. He received his B.E.E. from the City College of New York (CCNY) and his masters and PhD in electrical engineering from MIT.



What made you decide to specialize in networking/Internet technology?

As a PhD student at MIT in 1959, I looked around and found that most of my classmates were doing research in the area of information theory and coding theory. At MIT, there was the great researcher, Claude Shannon, who had launched these fields and had solved most of the important problems already. The research problems that were left were hard and of lesser consequence. So I decided to launch out in a new area that no one else had yet conceived of. Remember that at MIT I was surrounded by lots of computers, and it was clear to me that soon these machines would need to communicate with each other. At the time, there was no effective way for them to do so, so I decided to develop the technology that would permit efficient and reliable data networks to be created.

What was your first job in the computer industry? What did it entail?

I went to the evening session at CCNY from 1951 to 1957 for my bachelor's degree in electrical engineering. During the day, I worked first as a technician and then as an engineer at a small, industrial electronics firm called Photobell. While there, I introduced digital technology to their product line. Essentially, we were using photoelectric devices to detect the presence of certain items (boxes, people, etc.) and the use of a circuit known then as a *bistable multivibrator* was just the kind of technology we needed to bring digital processing into this field of detection. These circuits happen to be the building blocks for computers, and have come to be known as *flip-flops* or *switches* in today's vernacular.

What was going through your mind when you sent the first host-to-host message (from UCLA to the Stanford Research Institute)?

Frankly, we had no idea of the importance of that event. We had not prepared a special message of historic significance, as did so many inventors of the past (Samuel Morse with "What hath God wrought." or Alexander Graham Bell with "Watson, come here! I want you." or Neal Armstrong with "That's one small step for a man, one giant leap for mankind.") Those guys were

*smart!* They understood media and public relations. All we wanted to do was to login to the SRI computer. So we typed the “L”, which was correctly received, we typed the “o” which was received, and then we typed the “g” which caused the SRI host computer to crash! So, it turned out that our message was the shortest and perhaps the most prophetic message ever, namely “Lo!” as in “Lo and behold!”

Earlier that year, I was quoted in a UCLA press release saying that once the network was up and running, it would be possible to gain access to computer utilities from our homes and offices as easily as we gain access to electricity and telephone connectivity. So my vision at that time was that the Internet would be ubiquitous, always on, always available, anyone with any device could connect from any location, and it would be invisible. However, I never anticipated that my 99-year-old mother would use the Internet—and indeed she did!

What is your vision for the future of networking?

The easy part of the vision is to predict the infrastructure itself. I anticipate that we see considerable deployment of nomadic computing, mobile devices, and smart spaces. Indeed, the availability of lightweight, inexpensive, high-performance, portable computing, and communication devices (plus the ubiquity of the Internet) has enabled us to become nomads. Nomadic computing refers to the technology that enables end users who travel from place to place to gain access to Internet services in a transparent fashion, no matter where they travel and no matter what device they carry or gain access to. The harder part of the vision is to predict the applications and services, which have consistently surprised us in dramatic ways (e-mail, search technologies, the World Wide Web, blogs, social networks, user generation, and sharing of music, photos, and videos, etc.). We are on the verge of a new class of surprising and innovative mobile applications delivered to our hand-held devices.

The next step will enable us to move out from the netherworld of cyberspace to the physical world of smart spaces. Our environments (desks, walls, vehicles, watches, belts, and so on) will come alive with technology, through actuators, sensors, logic, processing, storage, cameras, microphones, speakers, displays, and communication. This embedded technology will allow our environment to provide the IP services we want. When I walk into a room, the room will know I entered. I will be able to communicate with my environment naturally, as in spoken English; my requests will generate replies that present Web pages to me from wall displays, through my eyeglasses, as speech, holograms, and so forth.

Looking a bit further out, I see a networking future that includes the following additional key components. I see intelligent software agents deployed across the network whose function it is to mine data, act on that data, observe trends, and carry out tasks dynamically and adaptively. I see considerably more network traffic generated not so much by humans, but by these embedded devices and these intelligent software agents. I see large collections of self-organizing systems controlling this vast, fast network. I see huge amounts of information flashing

across this network instantaneously with this information undergoing enormous processing and filtering. The Internet will essentially be a pervasive global nervous system. I see all these things and more as we move headlong through the twenty-first century.

What people have inspired you professionally?

By far, it was Claude Shannon from MIT, a brilliant researcher who had the ability to relate his mathematical ideas to the physical world in highly intuitive ways. He was on my PhD thesis committee.

Do you have any advice for students entering the networking/Internet field?

The Internet and all that it enables is a vast new frontier, full of amazing challenges. There is room for great innovation. Don't be constrained by today's technology. Reach out and imagine what could be and then make it happen.

---

## Chapter 2 Application Layer

---

Network applications are the *raisons d'être* of a computer network—if we couldn't conceive of any useful applications, there wouldn't be any need for networking infrastructure and protocols to support them. Since the Internet's inception, numerous useful and entertaining applications have indeed been created. These applications have been the driving force behind the Internet's success, motivating people in homes, schools, governments, and businesses to make the Internet an integral part of their daily activities.

Internet applications include the classic text-based applications that became popular in the 1970s and 1980s: text e-mail, remote access to computers, file transfers, and newsgroups. They include the killer application of the mid-1990s, the World Wide Web, encompassing Web surfing, search, and electronic commerce. They include instant messaging and P2P file sharing, the two killer applications introduced at the end of the millennium. In the new millennium, new and highly compelling applications continue to emerge, including voice over IP and video conferencing such as Skype, Facetime, and Google Hangouts; user generated video such as YouTube and movies on demand such as Netflix; multiplayer online games such as Second Life and World of Warcraft. During this same period, we have seen the emergence of a new generation of social networking applications—such as Facebook, Instagram, Twitter, and WeChat—which have created engaging human networks on top of the Internet's network of routers and communication links. And most recently, along with the arrival of the smartphone, there has been a profusion of location based mobile apps, including popular check-in, dating, and road-traffic forecasting apps (such as Yelp, Tinder, Waz, and Yik Yak). Clearly, there has been no slowing down of new and exciting Internet applications. Perhaps some of the readers of this text will create the next generation of killer Internet applications!

In this chapter we study the conceptual and implementation aspects of network applications. We begin by defining key application-layer concepts, including network services required by applications, clients and servers, processes, and transport-layer interfaces. We examine several network applications in detail, including the Web, e-mail, DNS, peer-to-peer (P2P) file distribution, and video streaming.

([Chapter 9](#) will further examine multimedia applications, including streaming video and VoIP.) We then cover network application development, over both TCP and UDP. In particular, we study the socket interface and walk through some simple client-server applications in Python. We also provide several fun and interesting socket programming assignments at the end of the chapter.

The application layer is a particularly good place to start our study of protocols. It's familiar ground. We're acquainted with many of the applications that rely on the protocols we'll study. It will give us a good feel for what protocols are all about and will introduce us to many of the same issues that we'll see again when we study transport, network, and link layer protocols.

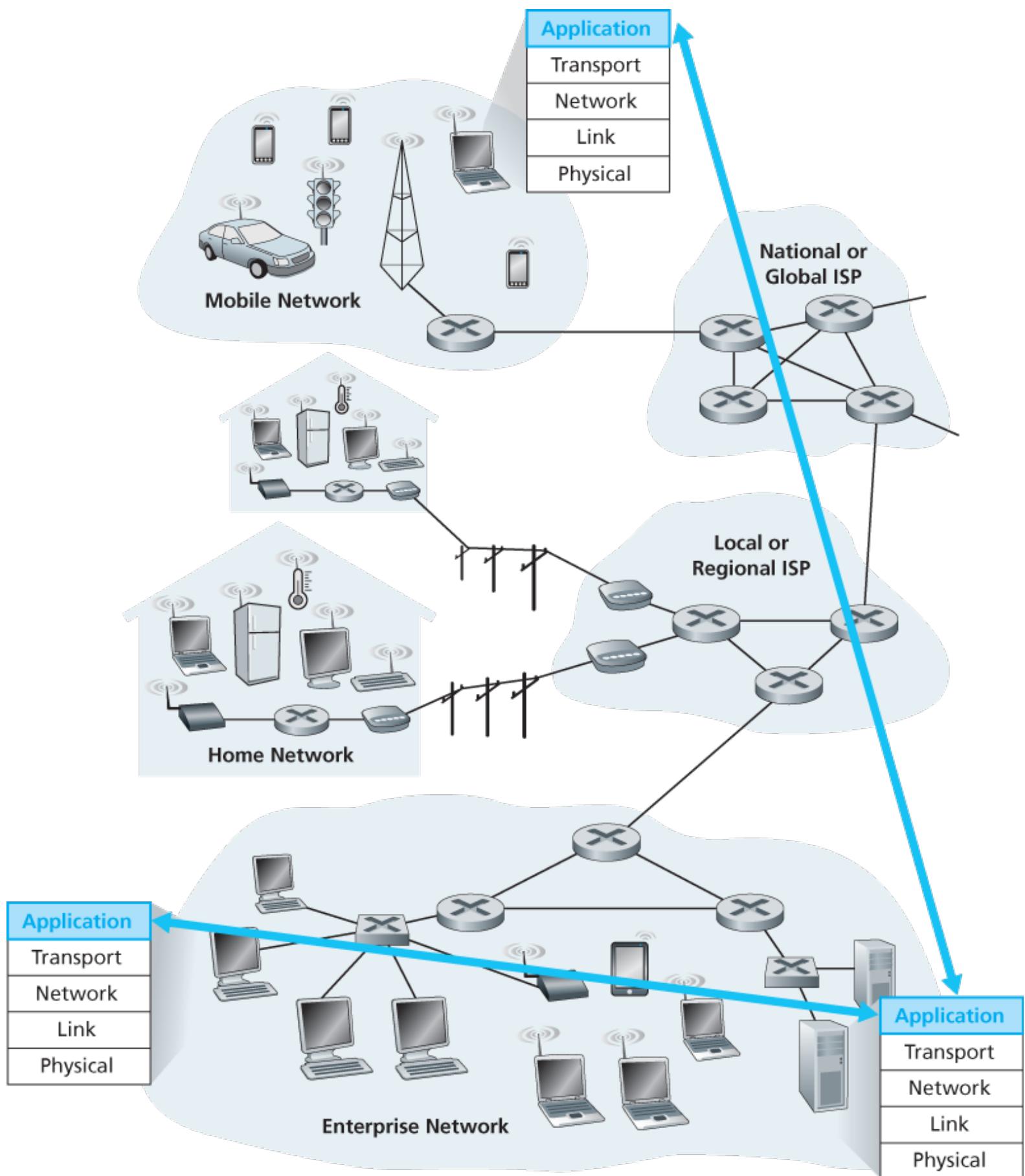
## 2.1 Principles of Network Applications

Suppose you have an idea for a new network application. Perhaps this application will be a great service to humanity, or will please your professor, or will bring you great wealth, or will simply be fun to develop. Whatever the motivation may be, let's now examine how you transform the idea into a real-world network application.

At the core of network application development is writing programs that run on different end systems and communicate with each other over the network. For example, in the Web application there are two distinct programs that communicate with each other: the browser program running in the user's host (desktop, laptop, tablet, smartphone, and so on); and the Web server program running in the Web server host. As another example, in a P2P file-sharing system there is a program in each host that participates in the file-sharing community. In this case, the programs in the various hosts may be similar or identical.

Thus, when developing your new application, you need to write software that will run on multiple end systems. This software could be written, for example, in C, Java, or Python. Importantly, you do not need to write software that runs on network-core devices, such as routers or link-layer switches. Even if you wanted to write application software for these network-core devices, you wouldn't be able to do so.

As we learned in [Chapter 1](#), and as shown earlier in [Figure 1.24](#), network-core devices do not function at the application layer but instead function at lower layers—specifically at the network layer and below. This basic design—namely, confining application software to the end systems—as shown in [Figure 2.1](#), has facilitated the rapid development and deployment of a vast array of network applications.



**Figure 2.1** Communication for a network application takes place between end systems at the application layer

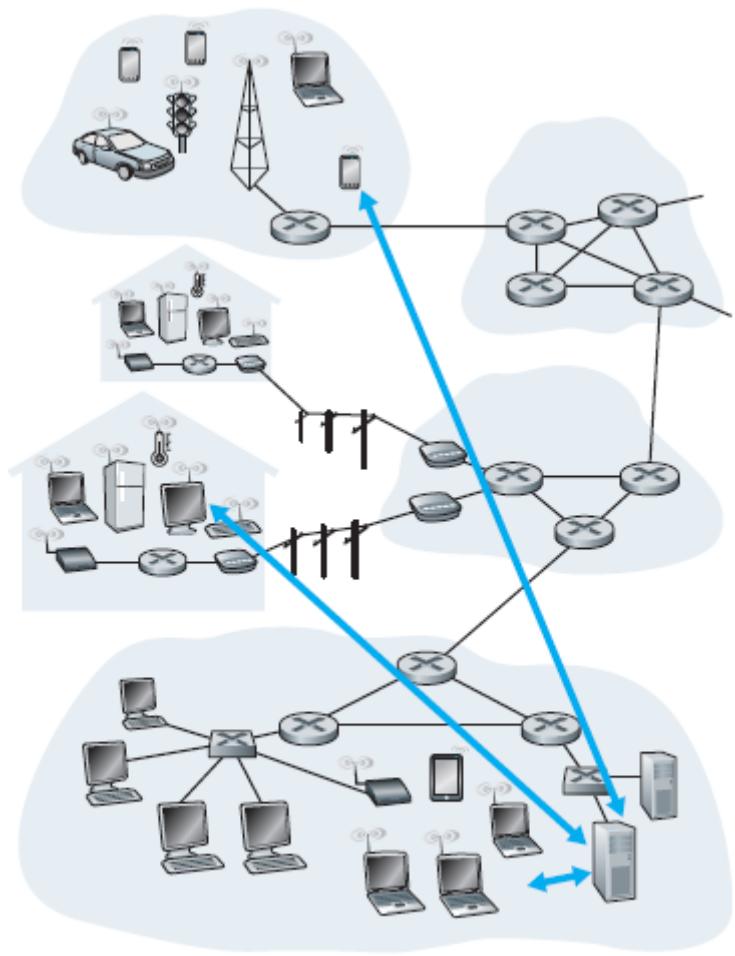
### 2.1.1 Network Application Architectures

Before diving into software coding, you should have a broad architectural plan for your application. Keep in mind that an application’s architecture is distinctly different from the network architecture (e.g., the five-layer Internet architecture discussed in [Chapter 1](#)). From the application developer’s perspective, the network architecture is fixed and provides a specific set of services to applications. The [application architecture](#), on the other hand, is designed by the application developer and dictates how the application is structured over the various end systems. In choosing the application architecture, an application developer will likely draw on one of the two predominant architectural paradigms used in modern network applications: the client-server architecture or the peer-to-peer (P2P) architecture.

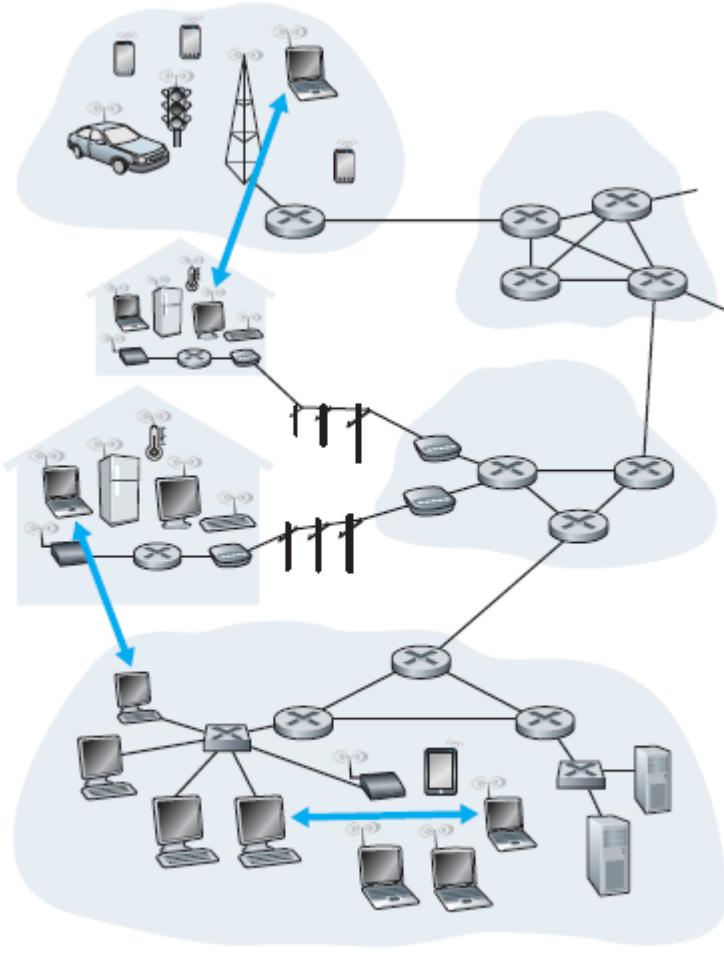
In a [client-server architecture](#), there is an always-on host, called the *server*, which services requests from many other hosts, called *clients*. A classic example is the Web application for which an always-on Web server services requests from browsers running on client hosts. When a Web server receives a request for an object from a client host, it responds by sending the requested object to the client host. Note that with the client-server architecture, clients do not directly communicate with each other; for example, in the Web application, two browsers do not directly communicate. Another characteristic of the client-server architecture is that the server has a fixed, well-known address, called an IP address (which we’ll discuss soon). Because the server has a fixed, well-known address, and because the server is always on, a client can always contact the server by sending a packet to the server’s IP address. Some of the better-known applications with a client-server architecture include the Web, FTP, Telnet, and e-mail. The client-server architecture is shown in [Figure 2.2\(a\)](#).

Often in a client-server application, a single-server host is incapable of keeping up with all the requests from clients. For example, a popular social-networking site can quickly become overwhelmed if it has only one server handling all of its requests. For this reason, a [data center](#), housing a large number of hosts, is often used to create a powerful virtual server. The most popular Internet services—such as search engines (e.g., Google, Bing, Baidu), Internet commerce (e.g., Amazon, eBay, Alibaba), Web-based e-mail (e.g., Gmail and Yahoo Mail), social networking (e.g., Facebook, Instagram, Twitter, and WeChat)—employ one or more data centers. As discussed in [Section 1.3.3](#), Google has 30 to 50 data centers distributed around the world, which collectively handle search, YouTube, Gmail, and other services. A data center can have hundreds of thousands of servers, which must be powered and maintained. Additionally, the service providers must pay recurring interconnection and bandwidth costs for sending data from their data centers.

In a [P2P architecture](#), there is minimal (or no) reliance on dedicated servers in data centers. Instead the application exploits direct communication between pairs of intermittently connected hosts, called *peers*. The peers are not owned by the service provider, but are instead desktops and laptops controlled by users, with most of the

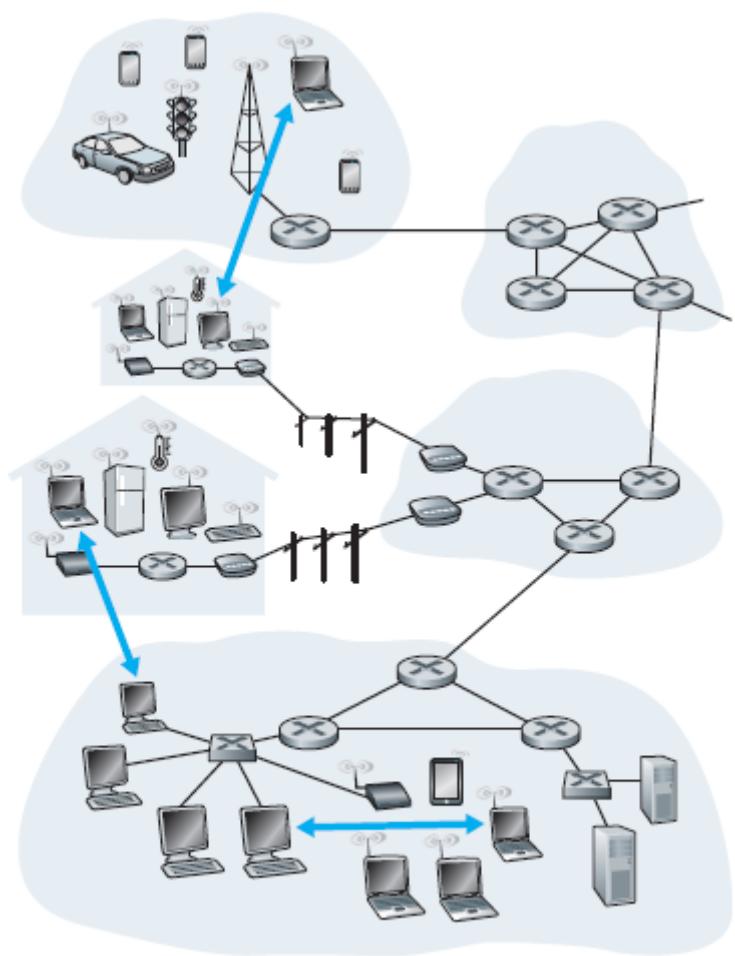


a. Client-server architecture



b. Peer-to-peer architecture

Figure 2.2 (a) Client-server architecture; (b) P2P architecture



b. Peer-to-peer architecture

peers residing in homes, universities, and offices. Because the peers communicate without passing through a dedicated server, the architecture is called peer-to-peer. Many of today's most popular and traffic-intensive applications are based on P2P architectures. These applications include file sharing (e.g., BitTorrent), peer-assisted download acceleration (e.g., Xunlei), and Internet telephony and video conference (e.g., Skype). The P2P architecture is illustrated in [Figure 2.2\(b\)](#). We mention that some applications have hybrid architectures, combining both client-server and P2P elements. For example, for many instant messaging applications, servers are used to track the IP addresses of users, but user-to-user messages are sent directly between user hosts (without passing through intermediate servers).

One of the most compelling features of P2P architectures is their **self-scalability**. For example, in a P2P file-sharing application, although each peer generates workload by requesting files, each peer also adds service capacity to the system by distributing files to other peers. P2P architectures are also cost effective, since they normally don't require significant server infrastructure and server bandwidth (in contrast with clients-server designs with datacenters). However, P2P applications face challenges of security, performance, and reliability due to their highly decentralized structure.

## 2.1.2 Processes Communicating

Before building your network application, you also need a basic understanding of how the programs, running in multiple end systems, communicate with each other. In the jargon of operating systems, it is not actually programs but **processes** that communicate. A process can be thought of as a program that is running within an end system. When processes are running on the same end system, they can communicate with each other with interprocess communication, using rules that are governed by the end system's operating system. But in this book we are not particularly interested in how processes in the same host communicate, but instead in how processes running on *different* hosts (with potentially different operating systems) communicate.

Processes on two different end systems communicate with each other by exchanging **messages** across the computer network. A sending process creates and sends messages into the network; a receiving process receives these messages and possibly responds by sending messages back. [Figure 2.1](#) illustrates that processes communicating with each other reside in the application layer of the five-layer protocol stack.

### *Client and Server Processes*

A network application consists of pairs of processes that send messages to each other over a network. For example, in the Web application a client browser process exchanges messages with a Web server

process. In a P2P file-sharing system, a file is transferred from a process in one peer to a process in another peer. For each pair of communicating processes, we typically label one of the two processes as the **client** and the other process as the **server**. With the Web, a browser is a client process and a Web server is a server process. With P2P file sharing, the peer that is downloading the file is labeled as the client, and the peer that is uploading the file is labeled as the server.

You may have observed that in some applications, such as in P2P file sharing, a process can be both a client and a server. Indeed, a process in a P2P file-sharing system can both upload and download files. Nevertheless, in the context of any given communication session between a pair of processes, we can still label one process as the client and the other process as the server. We define the client and server processes as follows:

*In the context of a communication session between a pair of processes, the process that initiates the communication (that is, initially contacts the other process at the beginning of the session) is labeled as the client. The process that waits to be contacted to begin the session is the server.*

In the Web, a browser process initializes contact with a Web server process; hence the browser process is the client and the Web server process is the server. In P2P file sharing, when Peer A asks Peer B to send a specific file, Peer A is the client and Peer B is the server in the context of this specific communication session. When there's no confusion, we'll sometimes also use the terminology "client side and server side of an application." At the end of this chapter, we'll step through simple code for both the client and server sides of network applications.

### *The Interface Between the Process and the Computer Network*

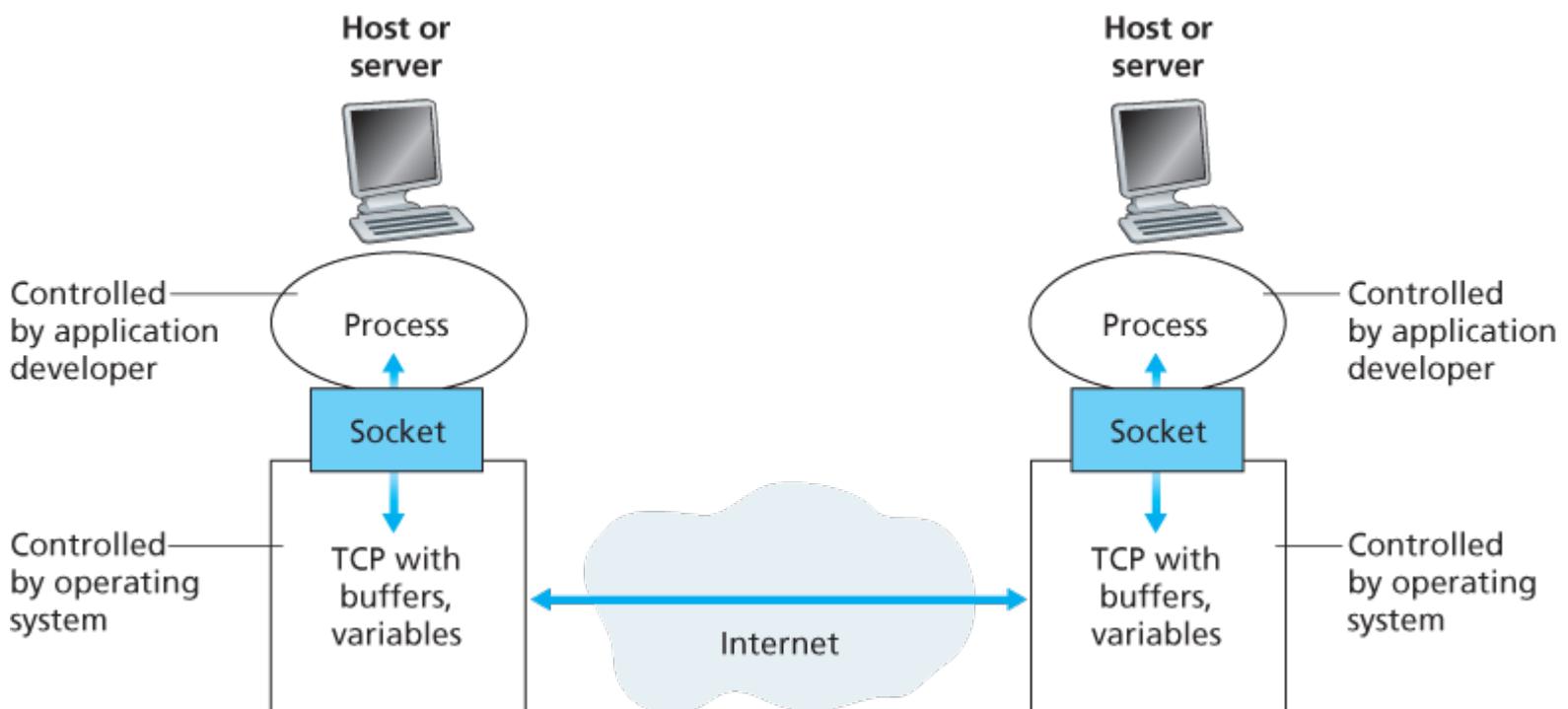
As noted above, most applications consist of pairs of communicating processes, with the two processes in each pair sending messages to each other. Any message sent from one process to another must go through the underlying network. A process sends messages into, and receives messages from, the network through a software interface called a **socket**. Let's consider an analogy to help us understand processes and sockets. A process is analogous to a house and its socket is analogous to its door. When a process wants to send a message to another process on another host, it shoves the message out its door (socket). This sending process assumes that there is a transportation infrastructure on the other side of its door that will transport the message to the door of the destination process. Once the message arrives at the destination host, the message passes through the receiving process's door (socket), and the receiving process then acts on the message.

**Figure 2.3** illustrates socket communication between two processes that communicate over the Internet. (**Figure 2.3** assumes that the underlying transport protocol used by the processes is the Internet's TCP protocol.) As shown in this figure, a socket is the interface between the application layer and the transport layer within a host. It is also referred to as the **Application Programming Interface (API)**

between the application and the network, since the socket is the programming interface with which network applications are built. The application developer has control of everything on the application-layer side of the socket but has little control of the transport-layer side of the socket. The only control that the application developer has on the transport-layer side is (1) the choice of transport protocol and (2) perhaps the ability to fix a few transport-layer parameters such as maximum buffer and maximum segment sizes (to be covered in [Chapter 3](#)). Once the application developer chooses a transport protocol (if a choice is available), the application is built using the transport-layer services provided by that protocol. We'll explore sockets in some detail in [Section 2.7](#).

### *Addressing Processes*

In order to send postal mail to a particular destination, the destination needs to have an address. Similarly, in order for a process running on one host to send packets to a process running on another host, the receiving process needs to have an address.



**Figure 2.3 Application processes, sockets, and underlying transport protocol**

To identify the receiving process, two pieces of information need to be specified: (1) the address of the host and (2) an identifier that specifies the receiving process in the destination host.

In the Internet, the host is identified by its **IP address**. We'll discuss IP addresses in great detail in [Chapter 4](#). For now, all we need to know is that an IP address is a 32-bit quantity that we can think of as uniquely identifying the host. In addition to knowing the address of the host to which a message is destined, the sending process must also identify the receiving process (more specifically, the receiving socket) running in the host. This information is needed because in general a host could be running many network applications. A destination **port number** serves this purpose. Popular applications have been

assigned specific port numbers. For example, a Web server is identified by port number 80. A mail server process (using the SMTP protocol) is identified by port number 25. A list of well-known port numbers for all Internet standard protocols can be found at [www.iana.org](http://www.iana.org). We'll examine port numbers in detail in [Chapter 3](#).

### 2.1.3 Transport Services Available to Applications

Recall that a socket is the interface between the application process and the transport-layer protocol. The application at the sending side pushes messages through the socket. At the other side of the socket, the transport-layer protocol has the responsibility of getting the messages to the socket of the receiving process.

Many networks, including the Internet, provide more than one transport-layer protocol. When you develop an application, you must choose one of the available transport-layer protocols. How do you make this choice? Most likely, you would study the services provided by the available transport-layer protocols, and then pick the protocol with the services that best match your application's needs. The situation is similar to choosing either train or airplane transport for travel between two cities. You have to choose one or the other, and each transportation mode offers different services. (For example, the train offers downtown pickup and drop-off, whereas the plane offers shorter travel time.)

What are the services that a transport-layer protocol can offer to applications invoking it? We can broadly classify the possible services along four dimensions: reliable data transfer, throughput, timing, and security.

#### *Reliable Data Transfer*

As discussed in [Chapter 1](#), packets can get lost within a computer network. For example, a packet can overflow a buffer in a router, or can be discarded by a host or router after having some of its bits corrupted. For many applications—such as electronic mail, file transfer, remote host access, Web document transfers, and financial applications—data loss can have devastating consequences (in the latter case, for either the bank or the customer!). Thus, to support these applications, something has to be done to guarantee that the data sent by one end of the application is delivered correctly and completely to the other end of the application. If a protocol provides such a guaranteed data delivery service, it is said to provide **reliable data transfer**. One important service that a transport-layer protocol can potentially provide to an application is process-to-process reliable data transfer. When a transport protocol provides this service, the sending process can just pass its data into the socket and know with complete confidence that the data will arrive without errors at the receiving process.

When a transport-layer protocol doesn't provide reliable data transfer, some of the data sent by the

sending process may never arrive at the receiving process. This may be acceptable for **loss-tolerant applications**, most notably multimedia applications such as conversational audio/video that can tolerate some amount of data loss. In these multimedia applications, lost data might result in a small glitch in the audio/video—not a crucial impairment.

### *Throughput*

In [Chapter 1](#) we introduced the concept of available throughput, which, in the context of a communication session between two processes along a network path, is the rate at which the sending process can deliver bits to the receiving process. Because other sessions will be sharing the bandwidth along the network path, and because these other sessions will be coming and going, the available throughput can fluctuate with time. These observations lead to another natural service that a transport-layer protocol could provide, namely, guaranteed available throughput at some specified rate. With such a service, the application could request a guaranteed throughput of  $r$  bits/sec, and the transport protocol would then ensure that the available throughput is always at least  $r$  bits/sec. Such a guaranteed throughput service would appeal to many applications. For example, if an Internet telephony application encodes voice at 32 kbps, it needs to send data into the network and have data delivered to the receiving application at this rate. If the transport protocol cannot provide this throughput, the application would need to encode at a lower rate (and receive enough throughput to sustain this lower coding rate) or may have to give up, since receiving, say, half of the needed throughput is of little or no use to this Internet telephony application. Applications that have throughput requirements are said to be **bandwidth-sensitive applications**. Many current multimedia applications are bandwidth sensitive, although some multimedia applications may use adaptive coding techniques to encode digitized voice or video at a rate that matches the currently available throughput.

While bandwidth-sensitive applications have specific throughput requirements, **elastic applications** can make use of as much, or as little, throughput as happens to be available. Electronic mail, file transfer, and Web transfers are all elastic applications. Of course, the more throughput, the better. There's an adage that says that one cannot be too rich, too thin, or have too much throughput!

### *Timing*

A transport-layer protocol can also provide timing guarantees. As with throughput guarantees, timing guarantees can come in many shapes and forms. An example guarantee might be that every bit that the sender pumps into the socket arrives at the receiver's socket no more than 100 msec later. Such a service would be appealing to interactive real-time applications, such as Internet telephony, virtual environments, teleconferencing, and multiplayer games, all of which require tight timing constraints on data delivery in order to be effective. (See [Chapter 9](#), [\[Gauthier 1999\]](#); [\[Ramjee 1994\]](#).) Long delays in Internet telephony, for example, tend to result in unnatural pauses in the conversation; in a multiplayer game or virtual interactive environment, a long delay between taking an action and seeing the response

from the environment (for example, from another player at the end of an end-to-end connection) makes the application feel less realistic. For non-real-time applications, lower delay is always preferable to higher delay, but no tight constraint is placed on the end-to-end delays.

### Security

Finally, a transport protocol can provide an application with one or more security services. For example, in the sending host, a transport protocol can encrypt all data transmitted by the sending process, and in the receiving host, the transport-layer protocol can decrypt the data before delivering the data to the receiving process. Such a service would provide confidentiality between the two processes, even if the data is somehow observed between sending and receiving processes. A transport protocol can also provide other security services in addition to confidentiality, including data integrity and end-point authentication, topics that we'll cover in detail in [Chapter 8](#).

#### 2.1.4 Transport Services Provided by the Internet

Up until this point, we have been considering transport services that a computer network *could* provide in general. Let's now get more specific and examine the type of transport services provided by the Internet. The Internet (and, more generally, TCP/IP networks) makes two transport protocols available to applications, UDP and TCP. When you (as an application developer) create a new network application for the Internet, one of the first decisions you have to make is whether to use UDP or TCP. Each of these protocols offers a different set of services to the invoking applications. [Figure 2.4](#) shows the service requirements for some selected applications.

### TCP Services

The TCP service model includes a connection-oriented service and a reliable data transfer service. When an application invokes TCP as its transport protocol, the application receives both of these services from TCP.

- **Connection-oriented service.** TCP has the client and server exchange transport-layer control information with each other *before* the application-level messages begin to flow. This so-called handshaking procedure alerts the client and server, allowing them to prepare for an onslaught of packets. After the handshaking phase, a [TCP connection](#) is said to exist between the sockets

Application	Data Loss	Throughput	Time-Sensitive
File transfer/download	No loss	Elastic	No
E-mail	No loss	Elastic	No
Web documents	No loss	Elastic (few kbps)	No
Internet telephony/ Video conferencing	Loss-tolerant	Audio: few kbps–1 Mbps Video: 10 kbps–5 Mbps	Yes: 100s of msec
Streaming stored audio/video	Loss-tolerant	Same as above	Yes: few seconds
Interactive games	Loss-tolerant	Few kbps–10 kbps	Yes: 100s of msec
Smartphone messaging	No loss	Elastic	Yes and no

**Figure 2.4 Requirements of selected network applications**

of the two processes. The connection is a full-duplex connection in that the two processes can send messages to each other over the connection at the same time. When the application finishes sending messages, it must tear down the connection. In [Chapter 3](#) we'll discuss connection-oriented service in detail and examine how it is implemented.

- **Reliable data transfer service.** The communicating processes can rely on TCP to deliver all data sent without error and in the proper order. When one side of the application passes a stream of bytes into a socket, it can count on TCP to deliver the same stream of bytes to the receiving socket, with no missing or duplicate bytes.

TCP also includes a congestion-control mechanism, a service for the general welfare of the Internet rather than for the direct benefit of the communicating processes. The TCP congestion-control mechanism throttles a sending process (client or server) when the network is congested between sender and receiver. As we will see

#### FOCUS ON SECURITY

##### SECURING TCP

Neither TCP nor UDP provides any encryption—the data that the sending process passes into its socket is the same data that travels over the network to the destination process. So, for example, if the sending process sends a password in cleartext (i.e., unencrypted) into its socket, the cleartext password will travel over all the links between sender and receiver, potentially getting sniffed and discovered at any of the intervening links. Because privacy and other security issues have become critical for many applications, the Internet community has developed an enhancement for TCP, called [Secure Sockets Layer \(SSL\)](#). TCP-enhanced-with-SSL not only

does everything that traditional TCP does but also provides critical process-to-process security services, including encryption, data integrity, and end-point authentication. We emphasize that SSL is not a third Internet transport protocol, on the same level as TCP and UDP, but instead is an enhancement of TCP, with the enhancements being implemented in the application layer. In particular, if an application wants to use the services of SSL, it needs to include SSL code (existing, highly optimized libraries and classes) in both the client and server sides of the application. SSL has its own socket API that is similar to the traditional TCP socket API. When an application uses SSL, the sending process passes cleartext data to the SSL socket; SSL in the sending host then encrypts the data and passes the encrypted data to the TCP socket. The encrypted data travels over the Internet to the TCP socket in the receiving process. The receiving socket passes the encrypted data to SSL, which decrypts the data. Finally, SSL passes the cleartext data through its SSL socket to the receiving process. We'll cover SSL in some detail in [Chapter 8](#).

in [Chapter 3](#), TCP congestion control also attempts to limit each TCP connection to its fair share of network bandwidth.

### UDP Services

UDP is a no-frills, lightweight transport protocol, providing minimal services. UDP is connectionless, so there is no handshaking before the two processes start to communicate. UDP provides an unreliable data transfer service—that is, when a process sends a message into a UDP socket, UDP provides *no* guarantee that the message will ever reach the receiving process. Furthermore, messages that do arrive at the receiving process may arrive out of order.

UDP does not include a congestion-control mechanism, so the sending side of UDP can pump data into the layer below (the network layer) at any rate it pleases. (Note, however, that the actual end-to-end throughput may be less than this rate due to the limited transmission capacity of intervening links or due to congestion).

### *Services Not Provided by Internet Transport Protocols*

We have organized transport protocol services along four dimensions: reliable data transfer, throughput, timing, and security. Which of these services are provided by TCP and UDP? We have already noted that TCP provides reliable end-to-end data transfer. And we also know that TCP can be easily enhanced at the application layer with SSL to provide security services. But in our brief description of TCP and UDP, conspicuously missing was any mention of throughput or timing guarantees—services *not* provided by today's Internet transport protocols. Does this mean that time-sensitive applications such as Internet telephony cannot run in today's Internet? The answer is clearly no—the Internet has been hosting time-sensitive applications for many years. These applications often work fairly well because

they have been designed to cope, to the greatest extent possible, with this lack of guarantee. We'll investigate several of these design tricks in [Chapter 9](#). Nevertheless, clever design has its limitations when delay is excessive, or the end-to-end throughput is limited. In summary, today's Internet can often provide satisfactory service to time-sensitive applications, but it cannot provide any timing or throughput guarantees.

**Figure 2.5** indicates the transport protocols used by some popular Internet applications. We see that e-mail, remote terminal access, the Web, and file transfer all use TCP. These applications have chosen TCP primarily because TCP provides reliable data transfer, guaranteeing that all data will eventually get to its destination. Because Internet telephony applications (such as Skype) can often tolerate some loss but require a minimal rate to be effective, developers of Internet telephony applications usually prefer to run their applications over UDP, thereby circumventing TCP's congestion control mechanism and packet overheads. But because many firewalls are configured to block (most types of) UDP traffic, Internet telephony applications often are designed to use TCP as a backup if UDP communication fails.

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP [RFC 5321]	TCP
Remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
File transfer	FTP [RFC 959]	TCP
Streaming multimedia	HTTP (e.g., YouTube)	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary (e.g., Skype)	UDP or TCP

**Figure 2.5 Popular Internet applications, their application-layer protocols, and their underlying transport protocols**

## 2.1.5 Application-Layer Protocols

We have just learned that network processes communicate with each other by sending messages into sockets. But how are these messages structured? What are the meanings of the various fields in the messages? When do the processes send the messages? These questions bring us into the realm of application-layer protocols. An **application-layer protocol** defines how an application's processes, running on different end systems, pass messages to each other. In particular, an application-layer protocol defines:

- The types of messages exchanged, for example, request messages and response messages
- The syntax of the various message types, such as the fields in the message and how the fields are delineated
- The semantics of the fields, that is, the meaning of the information in the fields
- Rules for determining when and how a process sends messages and responds to messages

Some application-layer protocols are specified in RFCs and are therefore in the public domain. For example, the Web's application-layer protocol, HTTP (the HyperText Transfer Protocol [\[RFC 2616\]](#)), is available as an RFC. If a browser developer follows the rules of the HTTP RFC, the browser will be able to retrieve Web pages from any Web server that has also followed the rules of the HTTP RFC. Many other application-layer protocols are proprietary and intentionally not available in the public domain. For example, Skype uses proprietary application-layer protocols.

It is important to distinguish between network applications and application-layer protocols. An application-layer protocol is only one piece of a network application (albeit, a very important piece of the application from our point of view!). Let's look at a couple of examples. The Web is a client-server application that allows users to obtain documents from Web servers on demand. The Web application consists of many components, including a standard for document formats (that is, HTML), Web browsers (for example, Firefox and Microsoft Internet Explorer), Web servers (for example, Apache and Microsoft servers), and an application-layer protocol. The Web's application-layer protocol, HTTP, defines the format and sequence of messages exchanged between browser and Web server. Thus, HTTP is only one piece (albeit, an important piece) of the Web application. As another example, an Internet e-mail application also has many components, including mail servers that house user mailboxes; mail clients (such as Microsoft Outlook) that allow users to read and create messages; a standard for defining the structure of an e-mail message; and application-layer protocols that define how messages are passed between servers, how messages are passed between servers and mail clients, and how the contents of message headers are to be interpreted. The principal application-layer protocol for electronic mail is SMTP (Simple Mail Transfer Protocol) [\[RFC 5321\]](#). Thus, e-mail's principal application-layer protocol, SMTP, is only one piece (albeit an important piece) of the e-mail application.

## 2.1.6 Network Applications Covered in This Book

New public domain and proprietary Internet applications are being developed every day. Rather than covering a large number of Internet applications in an encyclopedic manner, we have chosen to focus on a small number of applications that are both pervasive and important. In this chapter we discuss five important applications: the Web, electronic mail, directory service video streaming, and P2P applications. We first discuss the Web, not only because it is an enormously popular application, but also because its application-layer protocol, HTTP, is straightforward and easy to understand. We then discuss electronic mail, the Internet's first killer application. E-mail is more complex than the Web in the

sense that it makes use of not one but several application-layer protocols. After e-mail, we cover DNS, which provides a directory service for the Internet. Most users do not interact with DNS directly; instead, users invoke DNS indirectly through other applications (including the Web, file transfer, and electronic mail). DNS illustrates nicely how a piece of core network functionality (network-name to network-address translation) can be implemented at the application layer in the Internet. We then discuss P2P file sharing applications, and complete our application study by discussing video streaming on demand, including distributing stored video over content distribution networks. In **Chapter 9**, we'll cover multimedia applications in more depth, including voice over IP and video conferencing.

## 2.2 The Web and HTTP

Until the early 1990s the Internet was used primarily by researchers, academics, and university students to log in to remote hosts, to transfer files from local hosts to remote hosts and vice versa, to receive and send news, and to receive and send electronic mail. Although these applications were (and continue to be) extremely useful, the Internet was essentially unknown outside of the academic and research communities. Then, in the early 1990s, a major new application arrived on the scene—the World Wide Web [[Berners-Lee 1994](#)]. The Web was the first Internet application that caught the general public's eye. It dramatically changed, and continues to change, how people interact inside and outside their work environments. It elevated the Internet from just one of many data networks to essentially the one and only data network.

Perhaps what appeals the most to users is that the Web operates *on demand*. Users receive what they want, when they want it. This is unlike traditional broadcast radio and television, which force users to tune in when the content provider makes the content available. In addition to being available on demand, the Web has many other wonderful features that people love and cherish. It is enormously easy for any individual to make information available over the Web—everyone can become a publisher at extremely low cost. Hyperlinks and search engines help us navigate through an ocean of information. Photos and videos stimulate our senses. Forms, JavaScript, Java applets, and many other devices enable us to interact with pages and sites. And the Web and its protocols serve as a platform for YouTube, Web-based e-mail (such as Gmail), and most mobile Internet applications, including Instagram and Google Maps.

### 2.2.1 Overview of HTTP

The [HyperText Transfer Protocol \(HTTP\)](#), the Web's application-layer protocol, is at the heart of the Web. It is defined in [[RFC 1945](#)] and [[RFC 2616](#)]. HTTP is implemented in two programs: a client program and a server program. The client program and server program, executing on different end systems, talk to each other by exchanging HTTP messages. HTTP defines the structure of these messages and how the client and server exchange the messages. Before explaining HTTP in detail, we should review some Web terminology.

A [Web page](#) (also called a document) consists of objects. An [object](#) is simply a file—such as an HTML file, a JPEG image, a Java applet, or a video clip—that is addressable by a single URL. Most Web pages consist of a [base HTML file](#) and several referenced objects. For example, if a Web page

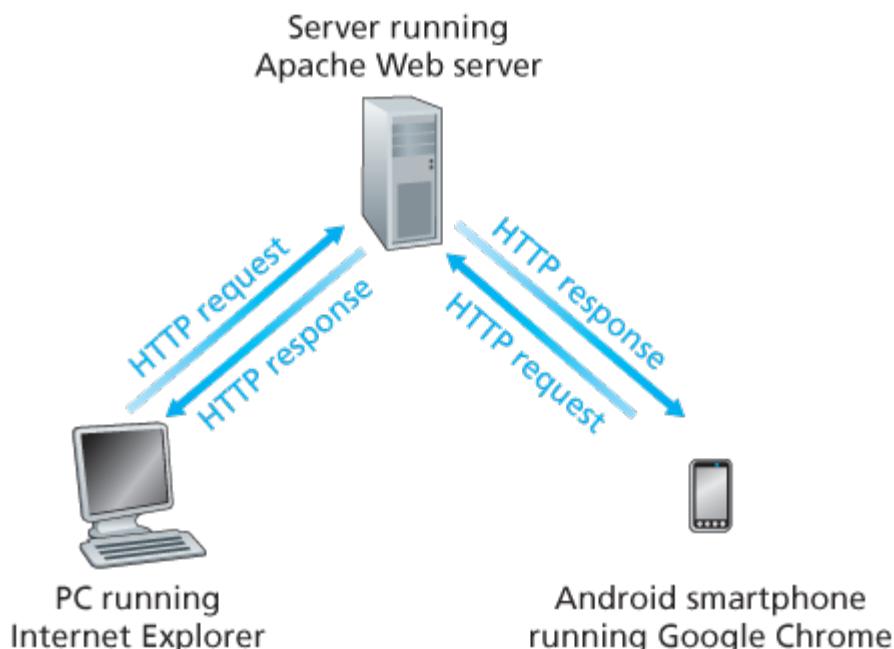
contains HTML text and five JPEG images, then the Web page has six objects: the base HTML file plus the five images. The base HTML file references the other objects in the page with the objects' URLs. Each URL has two components: the hostname of the server that houses the object and the object's path name. For example, the URL

`http://www.someSchool.edu/someDepartment/picture.gif`

has `www.someSchool.edu` for a hostname and `/someDepartment/picture.gif` for a path name. Because **Web browsers** (such as Internet Explorer and Firefox) implement the client side of HTTP, in the context of the Web, we will use the words *browser* and *client* interchangeably. **Web servers**, which implement the server side of HTTP, house Web objects, each addressable by a URL. Popular Web servers include Apache and Microsoft Internet Information Server.

HTTP defines how Web clients request Web pages from Web servers and how servers transfer Web pages to clients. We discuss the interaction between client and server in detail later, but the general idea is illustrated in **Figure 2.6**. When a user requests a Web page (for example, clicks on a hyperlink), the browser sends HTTP request messages for the objects in the page to the server. The server receives the requests and responds with HTTP response messages that contain the objects.

HTTP uses TCP as its underlying transport protocol (rather than running on top of UDP). The HTTP client first initiates a TCP connection with the server. Once the connection is established, the browser and the server processes access TCP through their socket interfaces. As described in **Section 2.1**, on the client side the socket interface is the door between the client process and the TCP connection; on the server side it is the door between the server process and the TCP connection. The client sends HTTP request messages into its socket interface and receives HTTP response messages from its socket interface. Similarly, the HTTP server receives request messages



## Figure 2.6 HTTP request-response behavior

from its socket interface and sends response messages into its socket interface. Once the client sends a message into its socket interface, the message is out of the client's hands and is "in the hands" of TCP.

Recall from [Section 2.1](#) that TCP provides a reliable data transfer service to HTTP. This implies that each HTTP request message sent by a client process eventually arrives intact at the server; similarly, each HTTP response message sent by the server process eventually arrives intact at the client. Here we see one of the great advantages of a layered architecture—HTTP need not worry about lost data or the details of how TCP recovers from loss or reordering of data within the network. That is the job of TCP and the protocols in the lower layers of the protocol stack.

It is important to note that the server sends requested files to clients without storing any state information about the client. If a particular client asks for the same object twice in a period of a few seconds, the server does not respond by saying that it just served the object to the client; instead, the server resends the object, as it has completely forgotten what it did earlier. Because an HTTP server maintains no information about the clients, HTTP is said to be a **stateless protocol**. We also remark that the Web uses the client-server application architecture, as described in [Section 2.1](#). A Web server is always on, with a fixed IP address, and it services requests from potentially millions of different browsers.

### 2.2.2 Non-Persistent and Persistent Connections

In many Internet applications, the client and server communicate for an extended period of time, with the client making a series of requests and the server responding to each of the requests. Depending on the application and on how the application is being used, the series of requests may be made back-to-back, periodically at regular intervals, or intermittently. When this client-server interaction is taking place over TCP, the application developer needs to make an important decision—should each request/response pair be sent over a *separate* TCP connection, or should all of the requests and their corresponding responses be sent over the *same* TCP connection? In the former approach, the application is said to use **non-persistent connections**; and in the latter approach, **persistent connections**. To gain a deep understanding of this design issue, let's examine the advantages and disadvantages of persistent connections in the context of a specific application, namely, HTTP, which can use both non-persistent connections and persistent connections. Although HTTP uses persistent connections in its default mode, HTTP clients and servers can be configured to use non-persistent connections instead.

#### *HTTP with Non-Persistent Connections*

Let's walk through the steps of transferring a Web page from server to client for the case of non-persistent connections. Let's suppose the page consists of a base HTML file and 10 JPEG images, and that all 11 of these objects reside on the same server. Further suppose the URL for the base HTML file is

*http://www.someSchool.edu/someDepartment/home.index*

Here is what happens:

1. The HTTP client process initiates a TCP connection to the server *www.someSchool.edu* on port number 80, which is the default port number for HTTP. Associated with the TCP connection, there will be a socket at the client and a socket at the server.
2. The HTTP client sends an HTTP request message to the server via its socket. The request message includes the path name */someDepartment/home .index*. (We will discuss HTTP messages in some detail below.)
3. The HTTP server process receives the request message via its socket, retrieves the object */someDepartment/home.index* from its storage (RAM or disk), encapsulates the object in an HTTP response message, and sends the response message to the client via its socket.
4. The HTTP server process tells TCP to close the TCP connection. (But TCP doesn't actually terminate the connection until it knows for sure that the client has received the response message intact.)
5. The HTTP client receives the response message. The TCP connection terminates. The message indicates that the encapsulated object is an HTML file. The client extracts the file from the response message, examines the HTML file, and finds references to the 10 JPEG objects.
6. The first four steps are then repeated for each of the referenced JPEG objects.

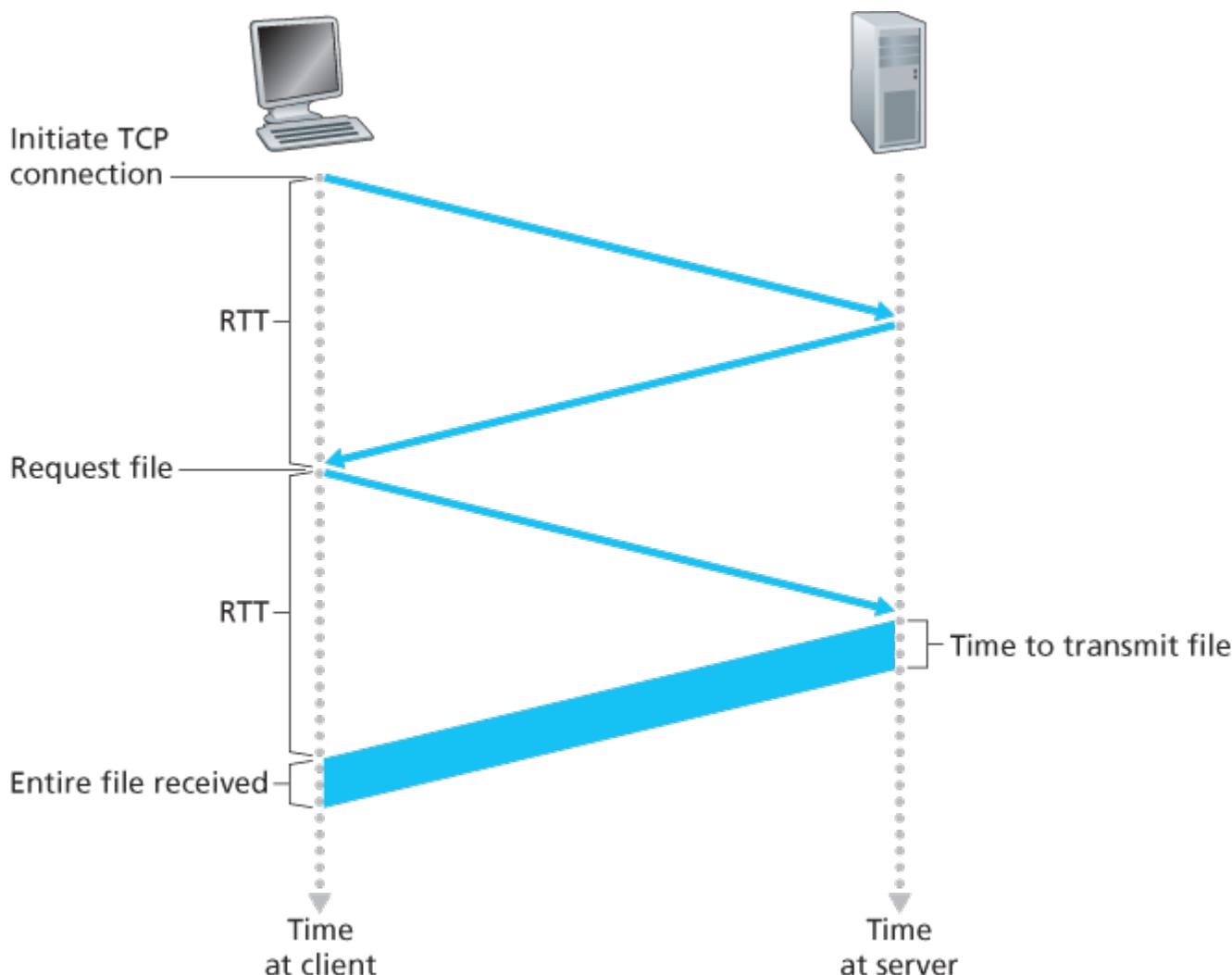
As the browser receives the Web page, it displays the page to the user. Two different browsers may interpret (that is, display to the user) a Web page in somewhat different ways. HTTP has nothing to do with how a Web page is interpreted by a client. The HTTP specifications ([\[RFC 1945\]](#) and [\[RFC 2616\]](#)) define only the communication protocol between the client HTTP program and the server HTTP program.

The steps above illustrate the use of non-persistent connections, where each TCP connection is closed after the server sends the object—the connection does not persist for other objects. Note that each TCP connection transports exactly one request message and one response message. Thus, in this example, when a user requests the Web page, 11 TCP connections are generated.

In the steps described above, we were intentionally vague about whether the client obtains the 10

JPEGs over 10 serial TCP connections, or whether some of the JPEGs are obtained over parallel TCP connections. Indeed, users can configure modern browsers to control the degree of parallelism. In their default modes, most browsers open 5 to 10 parallel TCP connections, and each of these connections handles one request-response transaction. If the user prefers, the maximum number of parallel connections can be set to one, in which case the 10 connections are established serially. As we'll see in the next chapter, the use of parallel connections shortens the response time.

Before continuing, let's do a back-of-the-envelope calculation to estimate the amount of time that elapses from when a client requests the base HTML file until the entire file is received by the client. To this end, we define the **round-trip time (RTT)**, which is the time it takes for a small packet to travel from client to server and then back to the client. The RTT includes packet-propagation delays, packet-queuing delays in intermediate routers and switches, and packet-processing delays. (These delays were discussed in [Section 1.4](#).) Now consider what happens when a user clicks on a hyperlink. As shown in [Figure 2.7](#), this causes the browser to initiate a TCP connection between the browser and the Web server; this involves a “three-way handshake”—the client sends a small TCP segment to the server, the server acknowledges and responds with a small TCP segment, and, finally, the client acknowledges back to the server. The first two parts of the three-way handshake take one RTT. After completing the first two parts of the handshake, the client sends the HTTP request message combined with the third part of the three-way handshake (the acknowledgment) into the TCP connection. Once the request message arrives at



**Figure 2.7 Back-of-the-envelope calculation for the time needed to request and receive an HTML file**

the server, the server sends the HTML file into the TCP connection. This HTTP request/response eats up another RTT. Thus, roughly, the total response time is two RTTs plus the transmission time at the server of the HTML file.

### *HTTP with Persistent Connections*

Non-persistent connections have some shortcomings. First, a brand-new connection must be established and maintained for *each requested object*. For each of these connections, TCP buffers must be allocated and TCP variables must be kept in both the client and server. This can place a significant burden on the Web server, which may be serving requests from hundreds of different clients simultaneously. Second, as we just described, each object suffers a delivery delay of two RTTs—one RTT to establish the TCP connection and one RTT to request and receive an object.

With HTTP 1.1 persistent connections, the server leaves the TCP connection open after sending a response. Subsequent requests and responses between the same client and server can be sent over the same connection. In particular, an entire Web page (in the example above, the base HTML file and the 10 images) can be sent over a single persistent TCP connection. Moreover, multiple Web pages residing on the same server can be sent from the server to the same client over a single persistent TCP connection. These requests for objects can be made back-to-back, without waiting for replies to pending requests (pipelining). Typically, the HTTP server closes a connection when it isn't used for a certain time (a configurable timeout interval). When the server receives the back-to-back requests, it sends the objects back-to-back. The default mode of HTTP uses persistent connections with pipelining. Most recently, HTTP/2 [[RFC 7540](#)] builds on HTTP 1.1 by allowing multiple requests and replies to be interleaved in the *same* connection, and a mechanism for prioritizing HTTP message requests and replies within this connection. We'll quantitatively compare the performance of non-persistent and persistent connections in the homework problems of [Chapters 2 and 3](#). You are also encouraged to see [[Heidemann 1997](#); [Nielsen 1997](#); [RFC 7540](#)].

### 2.2.3 HTTP Message Format

The HTTP specifications [[RFC 1945](#); [RFC 2616](#); [RFC 7540](#)] include the definitions of the HTTP message formats. There are two types of HTTP messages, request messages and response messages, both of which are discussed below.

#### *HTTP Request Message*

Below we provide a typical HTTP request message:

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
Connection: close
User-agent: Mozilla/5.0
Accept-language: fr
```

We can learn a lot by taking a close look at this simple request message. First of all, we see that the message is written in ordinary ASCII text, so that your ordinary computer-literate human being can read it. Second, we see that the message consists of five lines, each followed by a carriage return and a line feed. The last line is followed by an additional carriage return and line feed. Although this particular request message has five lines, a request message can have many more lines or as few as one line. The first line of an HTTP request message is called the **request line**; the subsequent lines are called the **header lines**. The request line has three fields: the method field, the URL field, and the HTTP version field. The method field can take on several different values, including `GET`, `POST`, `HEAD`, `PUT`, and `DELETE`. The great majority of HTTP request messages use the `GET` method. The `GET` method is used when the browser requests an object, with the requested object identified in the URL field. In this example, the browser is requesting the object `/somedir/page.html`. The version is self-explanatory; in this example, the browser implements version HTTP/1.1.

Now let's look at the header lines in the example. The header line `Host: www.someschool.edu` specifies the host on which the object resides. You might think that this header line is unnecessary, as there is already a TCP connection in place to the host. But, as we'll see in [Section 2.2.5](#), the information provided by the host header line is required by Web proxy caches. By including the `Connection: close` header line, the browser is telling the server that it doesn't want to bother with persistent connections; it wants the server to close the connection after sending the requested object. The `User-agent: Mozilla/5.0` header line specifies the user agent, that is, the browser type that is making the request to the server. Here the user agent is Mozilla/5.0, a Firefox browser. This header line is useful because the server can actually send different versions of the same object to different types of user agents. (Each of the versions is addressed by the same URL.) Finally, the `Accept-language: fr` header indicates that the user prefers to receive a French version of the object, if such an object exists on the server; otherwise, the server should send its default version. The `Accept-language: fr` header is just one of many content negotiation headers available in HTTP.

Having looked at an example, let's now look at the general format of a request message, as shown in [Figure 2.8](#). We see that the general format closely follows our earlier example. You may have noticed,

however, that after the header lines (and the additional carriage return and line feed) there is an “entity body.” The entity body is empty with the `GET` method, but is used with the `POST` method. An HTTP client often uses the `POST` method when the user fills out a form—for example, when a user provides search words to a search engine. With a `POST` message, the user is still requesting a Web page from the server, but the specific contents of the Web page

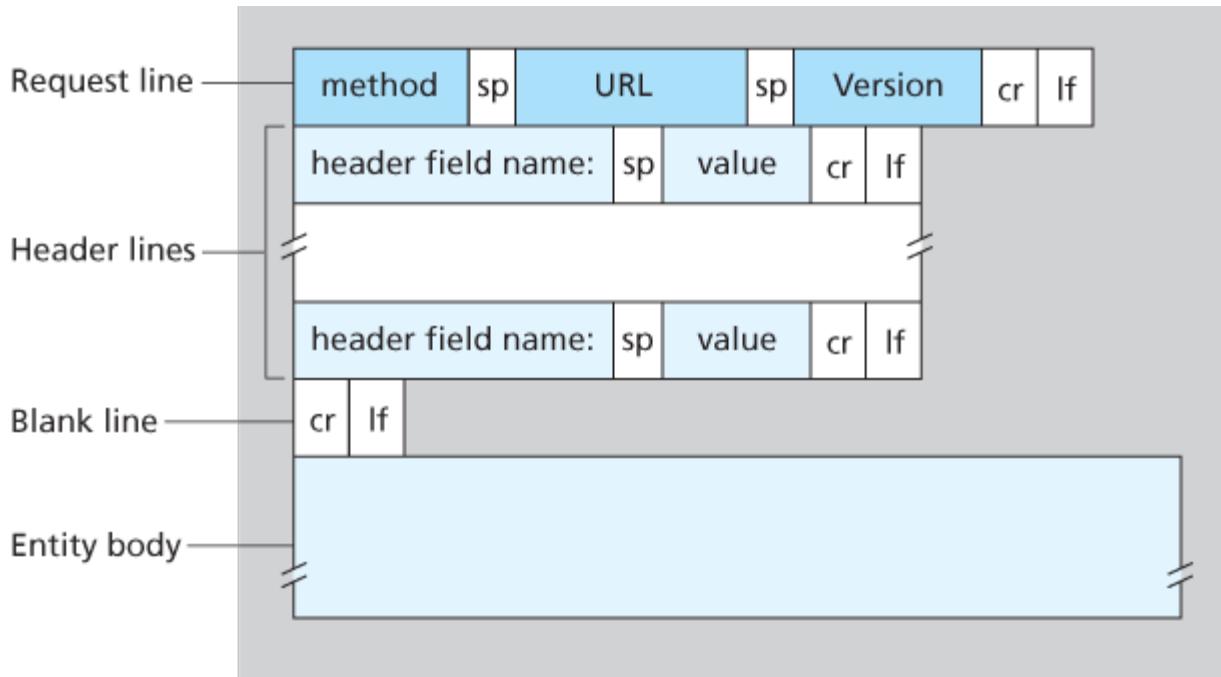


Figure 2.8 General format of an HTTP request message

depend on what the user entered into the form fields. If the value of the method field is `POST`, then the entity body contains what the user entered into the form fields.

We would be remiss if we didn't mention that a request generated with a form does not necessarily use the `POST` method. Instead, HTML forms often use the `GET` method and include the inputted data (in the form fields) in the requested URL. For example, if a form uses the `GET` method, has two fields, and the inputs to the two fields are `monkeys` and `bananas`, then the URL will have the structure

[www.somesite.com/animalsearch?monkeys&bananas](http://www.somesite.com/animalsearch?monkeys&bananas). In your day-to-day Web surfing, you have probably noticed extended URLs of this sort.

The `HEAD` method is similar to the `GET` method. When a server receives a request with the `HEAD` method, it responds with an HTTP message but it leaves out the requested object. Application developers often use the `HEAD` method for debugging. The `PUT` method is often used in conjunction with Web publishing tools. It allows a user to upload an object to a specific path (directory) on a specific Web server. The `PUT` method is also used by applications that need to upload objects to Web servers. The `DELETE` method allows a user, or an application, to delete an object on a Web server.

## HTTP Response Message

Below we provide a typical HTTP response message. This response message could be the response to the example request message just discussed.

```
HTTP/1.1 200 OK
Connection: close
Date: Tue, 18 Aug 2015 15:44:04 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 18 Aug 2015 15:11:03 GMT
Content-Length: 6821
Content-Type: text/html

(data data data data data ...)
```

Let's take a careful look at this response message. It has three sections: an initial **status line**, six **header lines**, and then the **entity body**. The entity body is the meat of the message—it contains the requested object itself (represented by `data data data data data ...`). The status line has three fields: the protocol version field, a status code, and a corresponding status message. In this example, the status line indicates that the server is using HTTP/1.1 and that everything is OK (that is, the server has found, and is sending, the requested object).

Now let's look at the header lines. The server uses the `Connection: close` header line to tell the client that it is going to close the TCP connection after sending the message. The `Date:` header line indicates the time and date when the HTTP response was created and sent by the server. Note that this is not the time when the object was created or last modified; it is the time when the server retrieves the object from its file system, inserts the object into the response message, and sends the response message. The `Server:` header line indicates that the message was generated by an Apache Web server; it is analogous to the `User-agent:` header line in the HTTP request message. The `Last-Modified:` header line indicates the time and date when the object was created or last modified. The `Last-Modified:` header, which we will soon cover in more detail, is critical for object caching, both in the local client and in network cache servers (also known as proxy servers). The `Content-Length:` header line indicates the number of bytes in the object being sent. The `Content-Type:` header line indicates that the object in the entity body is HTML text. (The object type is officially indicated by the `Content-Type:` header and not by the file extension.)

Having looked at an example, let's now examine the general format of a response message, which is shown in [Figure 2.9](#). This general format of the response message matches the previous example of a response message. Let's say a few additional words about status codes and their phrases. The status

code and associated phrase indicate the result of the request. Some common status codes and associated phrases include:

- *200 OK*: Request succeeded and the information is returned in the response.
- *301 Moved Permanently*: Requested object has been permanently moved; the new URL is specified in *Location*: header of the response message. The client software will automatically retrieve the new URL.
- *400 Bad Request*: This is a generic error code indicating that the request could not be understood by the server.

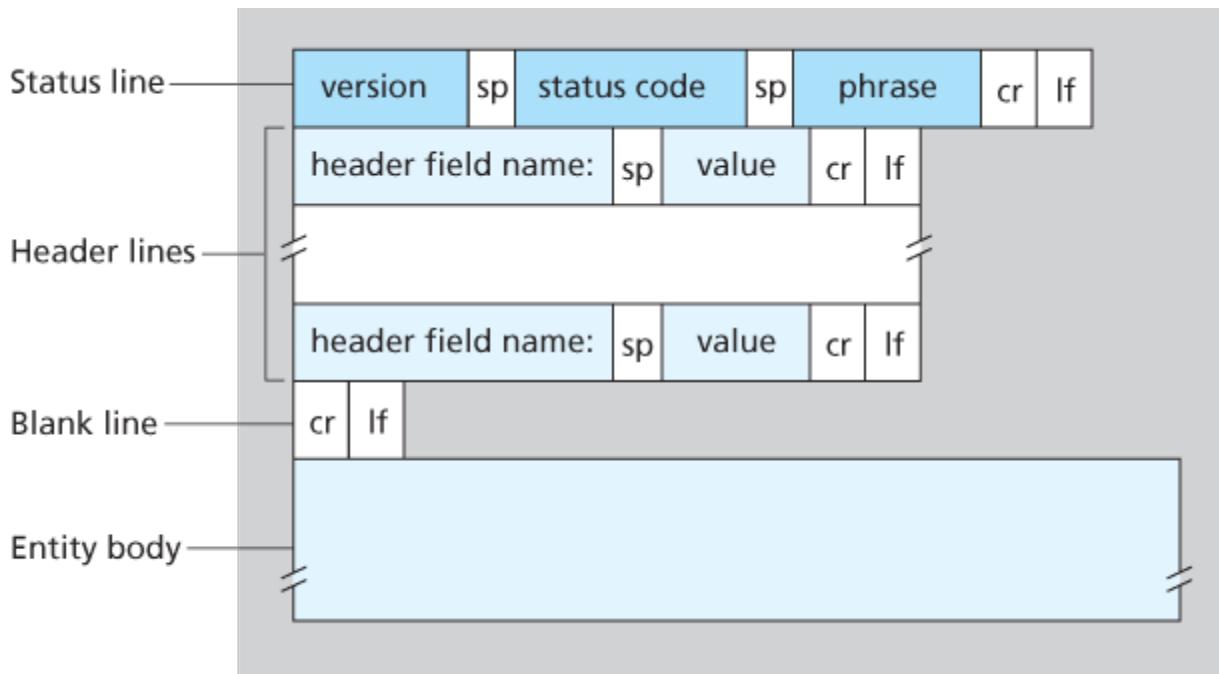


Figure 2.9 General format of an HTTP response message

- *404 Not Found*: The requested document does not exist on this server.
- *505 HTTP Version Not Supported*: The requested HTTP protocol version is not supported by the server.

How would you like to see a real HTTP response message? This is highly recommended and very easy to do! First Telnet into your favorite Web server. Then type in a one-line request message for some object that is housed on the server. For example, if you have access to a command prompt, type:



VideoNote

Using Wireshark to investigate the HTTP protocol

```
telnet gaia.cs.umass.edu 80  
  
GET /kurose_ross/interactive/index.php HTTP/1.1  
Host: gaia.cs.umass.edu
```

(Press the carriage return twice after typing the last line.) This opens a TCP connection to port 80 of the host [gaia.cs.umass.edu](http://gaia.cs.umass.edu) and then sends the HTTP request message. You should see a response message that includes the base HTML file for the interactive homework problems for this textbook. If you'd rather just see the HTTP message lines and not receive the object itself, replace `GET` with `HEAD`.

In this section we discussed a number of header lines that can be used within HTTP request and response messages. The HTTP specification defines many, many more header lines that can be inserted by browsers, Web servers, and network cache servers. We have covered only a small number of the totality of header lines. We'll cover a few more below and another small number when we discuss network Web caching in [Section 2.2.5](#). A highly readable and comprehensive discussion of the HTTP protocol, including its headers and status codes, is given in [\[Krishnamurthy 2001\]](#).

How does a browser decide which header lines to include in a request message? How does a Web server decide which header lines to include in a response message? A browser will generate header lines as a function of the browser type and version (for example, an HTTP/1.0 browser will not generate any 1.1 header lines), the user configuration of the browser (for example, preferred language), and whether the browser currently has a cached, but possibly out-of-date, version of the object. Web servers behave similarly: There are different products, versions, and configurations, all of which influence which header lines are included in response messages.

## 2.2.4 User-Server Interaction: Cookies

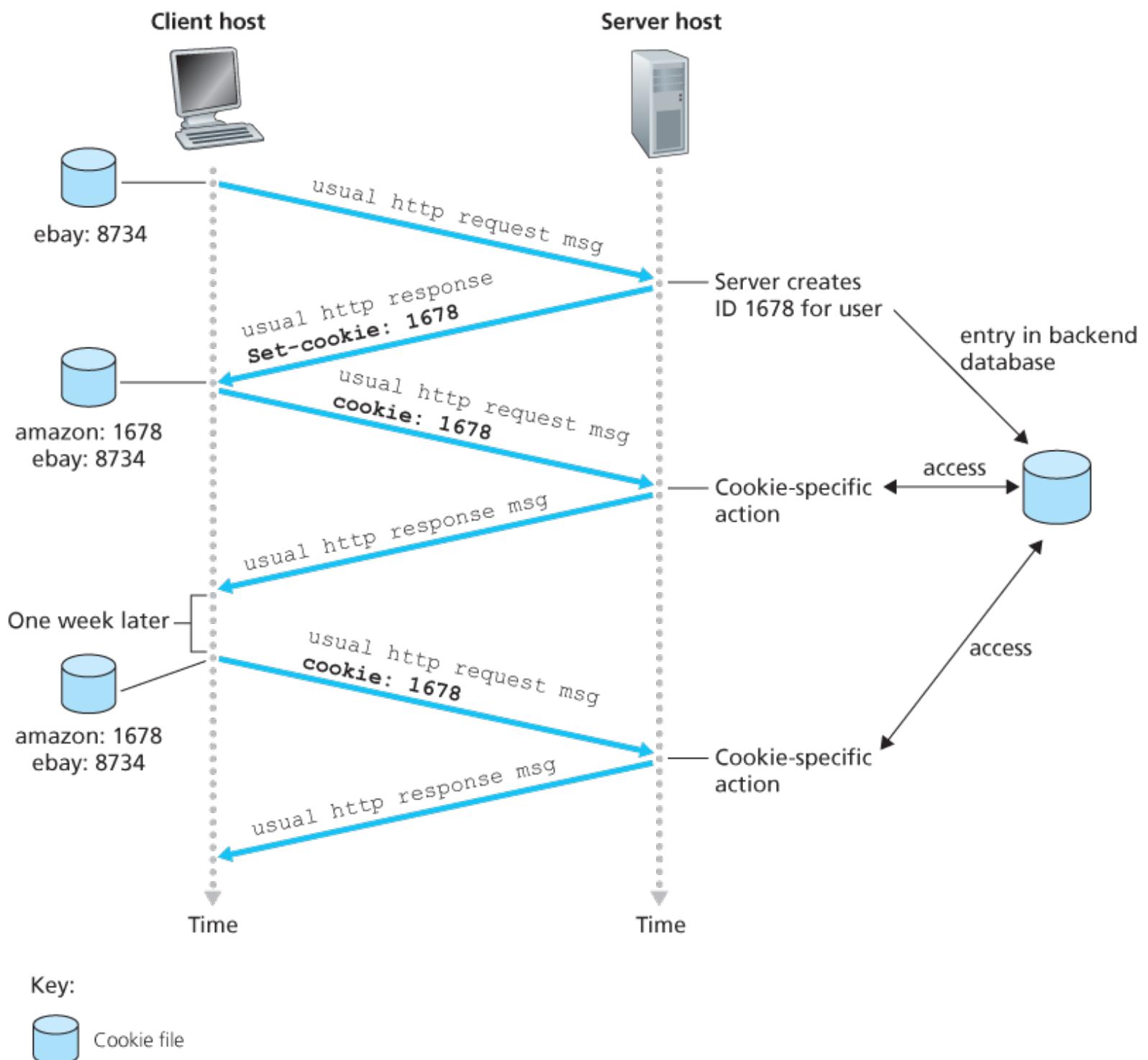
We mentioned above that an HTTP server is stateless. This simplifies server design and has permitted engineers to develop high-performance Web servers that can handle thousands of simultaneous TCP connections. However, it is often desirable for a Web site to identify users, either because the server wishes to restrict user access or because it wants to serve content as a function of the user identity. For these purposes, HTTP uses cookies. Cookies, defined in [\[RFC 6265\]](#), allow sites to keep track of users. Most major commercial Web sites use cookies today.

As shown in [Figure 2.10](#), cookie technology has four components: (1) a cookie header line in the HTTP response message; (2) a cookie header line in the HTTP request message; (3) a cookie file kept on the

user's end system and managed by the user's browser; and (4) a back-end database at the Web site. Using **Figure 2.10**, let's walk through an example of how cookies work. Suppose Susan, who always accesses the Web using Internet Explorer from her home PC, contacts [Amazon.com](#) for the first time. Let us suppose that in the past she has already visited the eBay site. When the request comes into the Amazon Web server, the server creates a unique identification number and creates an entry in its back-end database that is indexed by the identification number. The Amazon Web server then responds to Susan's browser, including in the HTTP response a *Set-cookie:* header, which contains the identification number. For example, the header line might be:

```
Set-cookie: 1678
```

When Susan's browser receives the HTTP response message, it sees the *Set-cookie:* header. The browser then appends a line to the special cookie file that it manages. This line includes the hostname of the server and the identification number in the *Set-cookie:* header. Note that the cookie file already has an entry for eBay, since Susan has visited that site in the past. As Susan continues to browse the Amazon site, each time she requests a Web page, her browser consults her cookie file, extracts her identification number for this site, and puts a cookie header line that



**Figure 2.10 Keeping user state with cookies**

includes the identification number in the HTTP request. Specifically, each of her HTTP requests to the Amazon server includes the header line:

*Cookie: 1678*

In this manner, the Amazon server is able to track Susan's activity at the Amazon site. Although the Amazon Web site does not necessarily know Susan's name, it knows exactly which pages user 1678 visited, in which order, and at what times! Amazon uses cookies to provide its shopping cart service—Amazon can maintain a list of all of Susan's intended purchases, so that she can pay for them.

collectively at the end of the session.

If Susan returns to Amazon's site, say, one week later, her browser will continue to put the header line *Cookie: 1678* in the request messages. Amazon also recommends products to Susan based on Web pages she has visited at Amazon in the past. If Susan also registers herself with Amazon—providing full name, e-mail address, postal address, and credit card information—Amazon can then include this information in its database, thereby associating Susan's name with her identification number (and all of the pages she has visited at the site in the past!). This is how Amazon and other e-commerce sites provide “one-click shopping”—when Susan chooses to purchase an item during a subsequent visit, she doesn't need to re-enter her name, credit card number, or address.

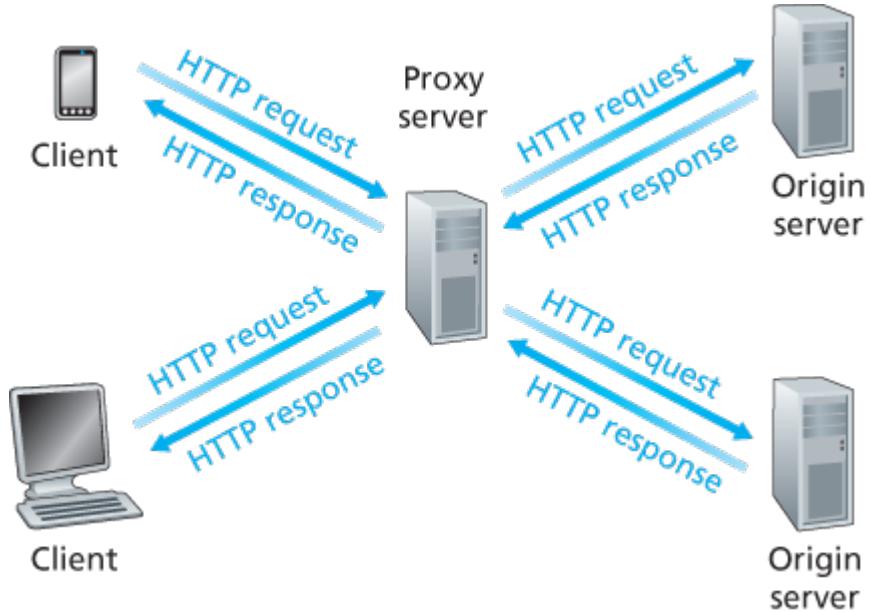
From this discussion we see that cookies can be used to identify a user. The first time a user visits a site, the user can provide a user identification (possibly his or her name). During the subsequent sessions, the browser passes a cookie header to the server, thereby identifying the user to the server. Cookies can thus be used to create a user session layer on top of stateless HTTP. For example, when a user logs in to a Web-based e-mail application (such as Hotmail), the browser sends cookie information to the server, permitting the server to identify the user throughout the user's session with the application.

Although cookies often simplify the Internet shopping experience for the user, they are controversial because they can also be considered as an invasion of privacy. As we just saw, using a combination of cookies and user-supplied account information, a Web site can learn a lot about a user and potentially sell this information to a third party. Cookie Central [[Cookie Central 2016](#)] includes extensive information on the cookie controversy.

## 2.2.5 Web Caching

A **Web cache**—also called a **proxy server**—is a network entity that satisfies HTTP requests on the behalf of an origin Web server. The Web cache has its own disk storage and keeps copies of recently requested objects in this storage. As shown in [Figure 2.11](#), a user's browser can be configured so that all of the user's HTTP requests are first directed to the Web cache. Once a browser is configured, each browser request for an object is first directed to the Web cache. As an example, suppose a browser is requesting the object *http://www.someschool.edu/campus.gif*. Here is what happens:

1. The browser establishes a TCP connection to the Web cache and sends an HTTP request for the object to the Web cache.
2. The Web cache checks to see if it has a copy of the object stored locally. If it does, the Web cache returns the object within an HTTP response message to the client browser.



**Figure 2.11 Clients requesting objects through a Web cache**

3. If the Web cache does not have the object, the Web cache opens a TCP connection to the origin server, that is, to `www.someschool.edu`. The Web cache then sends an HTTP request for the object into the cache-to-server TCP connection. After receiving this request, the origin server sends the object within an HTTP response to the Web cache.
4. When the Web cache receives the object, it stores a copy in its local storage and sends a copy, within an HTTP response message, to the client browser (over the existing TCP connection between the client browser and the Web cache).

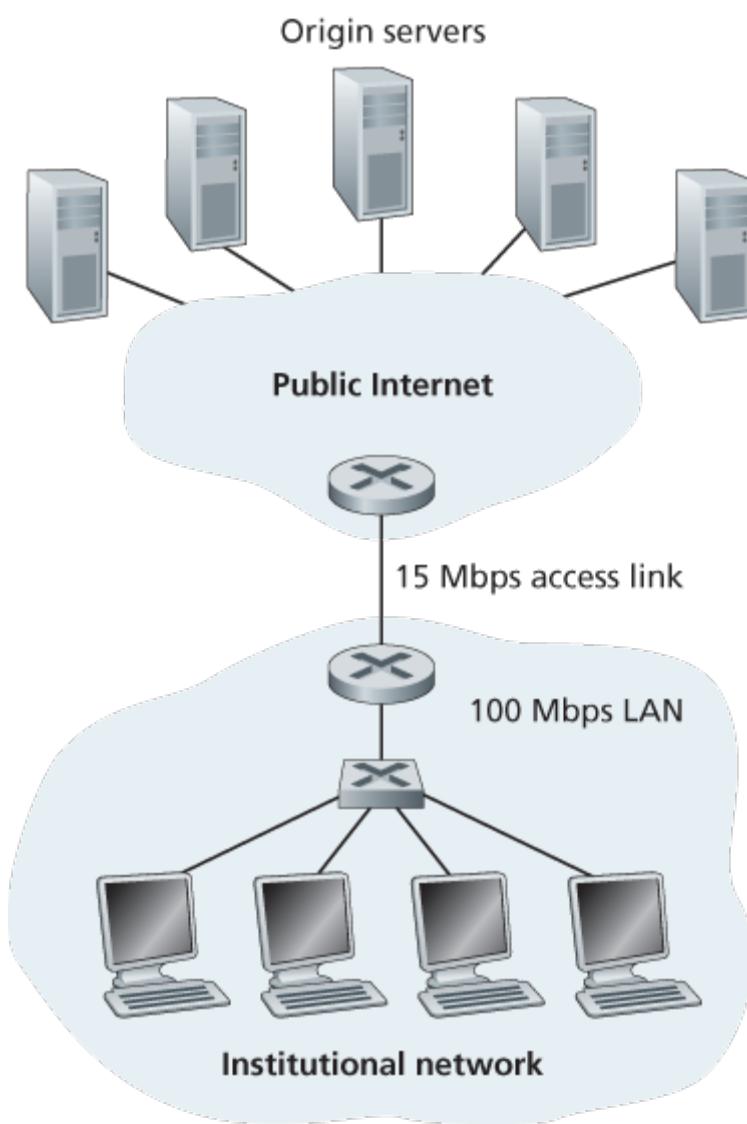
Note that a cache is both a server and a client at the same time. When it receives requests from and sends responses to a browser, it is a server. When it sends requests to and receives responses from an origin server, it is a client.

Typically a Web cache is purchased and installed by an ISP. For example, a university might install a cache on its campus network and configure all of the campus browsers to point to the cache. Or a major residential ISP (such as Comcast) might install one or more caches in its network and preconfigure its shipped browsers to point to the installed caches.

Web caching has seen deployment in the Internet for two reasons. First, a Web cache can substantially reduce the response time for a client request, particularly if the bottleneck bandwidth between the client and the origin server is much less than the bottleneck bandwidth between the client and the cache. If there is a high-speed connection between the client and the cache, as there often is, and if the cache has the requested object, then the cache will be able to deliver the object rapidly to the client. Second, as we will soon illustrate with an example, Web caches can substantially reduce traffic on an institution's access link to the Internet. By reducing traffic, the institution (for example, a company or a university) does not have to upgrade bandwidth as quickly, thereby reducing costs. Furthermore, Web caches can

substantially reduce Web traffic in the Internet as a whole, thereby improving performance for all applications.

To gain a deeper understanding of the benefits of caches, let's consider an example in the context of **Figure 2.12**. This figure shows two networks—the institutional network and the rest of the public Internet. The institutional network is a high-speed LAN. A router in the institutional network and a router in the Internet are connected by a 15 Mbps link. The origin servers are attached to the Internet but are located all over the globe. Suppose that the average object size is 1 Mbits and that the average request rate from the institution's browsers to the origin servers is 15 requests per second. Suppose that the HTTP request messages are negligibly small and thus create no traffic in the networks or in the access link (from institutional router to Internet router). Also suppose that the amount of time it takes from when the router on the Internet side of the access link in **Figure 2.12** forwards an HTTP request (within an IP datagram) until it receives the response (typically within many IP datagrams) is two seconds on average. Informally, we refer to this last delay as the “Internet delay.”



**Figure 2.12 Bottleneck between an institutional network and the Internet**

The total response time—that is, the time from the browser's request of an object until its receipt of the object—is the sum of the LAN delay, the access delay (that is, the delay between the two routers), and

the Internet delay. Let's now do a very crude calculation to estimate this delay. The traffic intensity on the LAN (see [Section 1.4.2](#)) is

$$(15 \text{ requests/sec}) \cdot (1 \text{ Mbits/request}) / (100 \text{ Mbps}) = 0.15$$

whereas the traffic intensity on the access link (from the Internet router to institution router) is

$$(15 \text{ requests/sec}) \cdot (1 \text{ Mbits/request}) / (15 \text{ Mbps}) = 1$$

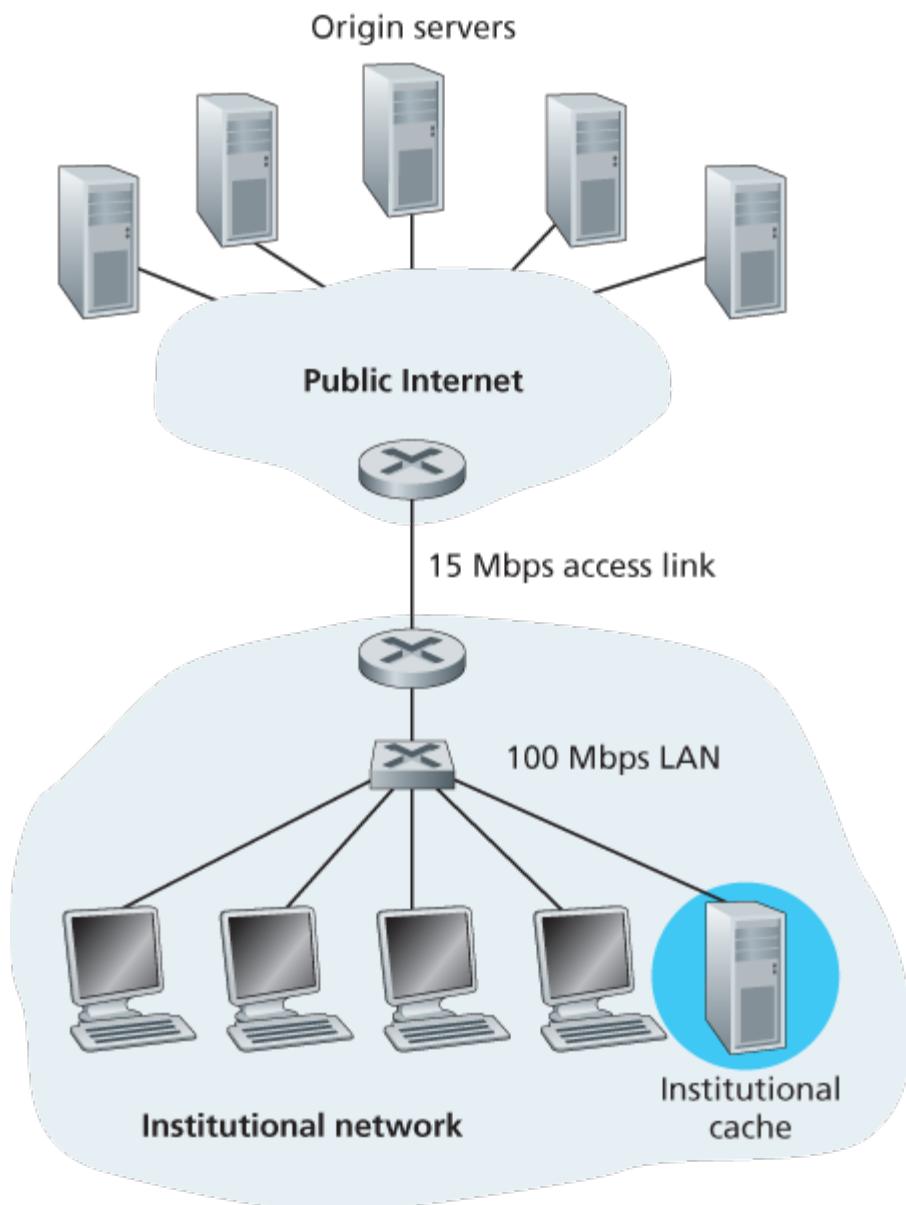
A traffic intensity of 0.15 on a LAN typically results in, at most, tens of milliseconds of delay; hence, we can neglect the LAN delay. However, as discussed in [Section 1.4.2](#), as the traffic intensity approaches 1 (as is the case of the access link in [Figure 2.12](#)), the delay on a link becomes very large and grows without bound. Thus, the average response time to satisfy requests is going to be on the order of minutes, if not more, which is unacceptable for the institution's users. Clearly something must be done.

One possible solution is to increase the access rate from 15 Mbps to, say, 100 Mbps. This will lower the traffic intensity on the access link to 0.15, which translates to negligible delays between the two routers. In this case, the total response time will roughly be two seconds, that is, the Internet delay. But this solution also means that the institution must upgrade its access link from 15 Mbps to 100 Mbps, a costly proposition.

Now consider the alternative solution of not upgrading the access link but instead installing a Web cache in the institutional network. This solution is illustrated in [Figure 2.13](#). Hit rates—the fraction of requests that are satisfied by a cache—typically range from 0.2 to 0.7 in practice. For illustrative purposes, let's suppose that the cache provides a hit rate of 0.4 for this institution. Because the clients and the cache are connected to the same high-speed LAN, 40 percent of the requests will be satisfied almost immediately, say, within 10 milliseconds, by the cache. Nevertheless, the remaining 60 percent of the requests still need to be satisfied by the origin servers. But with only 60 percent of the requested objects passing through the access link, the traffic intensity on the access link is reduced from 1.0 to 0.6. Typically, a traffic intensity less than 0.8 corresponds to a small delay, say, tens of milliseconds, on a 15 Mbps link. This delay is negligible compared with the two-second Internet delay. Given these considerations, average delay therefore is

$$0.4 \cdot (0.01 \text{ seconds}) + 0.6 \cdot (2.01 \text{ seconds})$$

which is just slightly greater than 1.2 seconds. Thus, this second solution provides an even lower response time than the first solution, and it doesn't require the institution



**Figure 2.13 Adding a cache to the institutional network**

to upgrade its link to the Internet. The institution does, of course, have to purchase and install a Web cache. But this cost is low—many caches use public-domain software that runs on inexpensive PCs.

Through the use of **Content Distribution Networks (CDNs)**, Web caches are increasingly playing an important role in the Internet. A CDN company installs many geographically distributed caches throughout the Internet, thereby localizing much of the traffic. There are shared CDNs (such as Akamai and Limelight) and dedicated CDNs (such as Google and Netflix). We will discuss CDNs in more detail in [Section 2.6](#).

#### *The Conditional GET*

Although caching can reduce user-perceived response times, it introduces a new problem—the copy of an object residing in the cache may be stale. In other words, the object housed in the Web server may have been modified since the copy was cached at the client. Fortunately, HTTP has a mechanism that allows a cache to verify that its objects are up to date. This mechanism is called the **conditional GET**.

An HTTP request message is a so-called conditional GET message if (1) the request message uses the `GET` method and (2) the request message includes an `If-Modified-Since:` header line.

To illustrate how the conditional GET operates, let's walk through an example. First, on the behalf of a requesting browser, a proxy cache sends a request message to a Web server:

```
GET /fruit/kiwi.gif HTTP/1.1  
Host: www.exotiquecuisine.com
```

Second, the Web server sends a response message with the requested object to the cache:

```
HTTP/1.1 200 OK  
Date: Sat, 3 Oct 2015 15:39:29  
Server: Apache/1.3.0 (Unix)  
Last-Modified: Wed, 9 Sep 2015 09:23:24  
Content-Type: image/gif  
  
(data data data data data ...)
```

The cache forwards the object to the requesting browser but also caches the object locally. Importantly, the cache also stores the last-modified date along with the object. Third, one week later, another browser requests the same object via the cache, and the object is still in the cache. Since this object may have been modified at the Web server in the past week, the cache performs an up-to-date check by issuing a conditional GET. Specifically, the cache sends:

```
GET /fruit/kiwi.gif HTTP/1.1  
Host: www.exotiquecuisine.com  
If-modified-since: Wed, 9 Sep 2015 09:23:24
```

Note that the value of the `If-modified-since:` header line is exactly equal to the value of the `Last-Modified:` header line that was sent by the server one week ago. This conditional GET is telling the server to send the object only if the object has been modified since the specified date. Suppose the object has not been modified since 9 Sep 2015 09:23:24. Then, fourth, the Web server sends a response message to the cache:

```
HTTP/1.1 304 Not Modified
Date: Sat, 10 Oct 2015 15:39:29
Server: Apache/1.3.0 (Unix)

(empty entity body)
```

We see that in response to the conditional GET, the Web server still sends a response message but does not include the requested object in the response message. Including the requested object would only waste bandwidth and increase user-perceived response time, particularly if the object is large. Note that this last response message has *304 Not Modified* in the status line, which tells the cache that it can go ahead and forward its (the proxy cache's) cached copy of the object to the requesting browser.

This ends our discussion of HTTP, the first Internet protocol (an application-layer protocol) that we've studied in detail. We've seen the format of HTTP messages and the actions taken by the Web client and server as these messages are sent and received. We've also studied a bit of the Web's application infrastructure, including caches, cookies, and back-end databases, all of which are tied in some way to the HTTP protocol.

## 2.3 Electronic Mail in the Internet

Electronic mail has been around since the beginning of the Internet. It was the most popular application when the Internet was in its infancy [Segaller 1998], and has become more elaborate and powerful over the years. It remains one of the Internet's most important and utilized applications.

As with ordinary postal mail, e-mail is an asynchronous communication medium—people send and read messages when it is convenient for them, without having to coordinate with other people's schedules. In contrast with postal mail, electronic mail is fast, easy to distribute, and inexpensive. Modern e-mail has many powerful features, including messages with attachments, hyperlinks, HTML-formatted text, and embedded photos.

In this section, we examine the application-layer protocols that are at the heart of Internet e-mail. But before we jump into an in-depth discussion of these protocols, let's take a high-level view of the Internet mail system and its key components.

**Figure 2.14** presents a high-level view of the Internet mail system. We see from this diagram that it has three major components: **user agents**, **mail servers**, and the **Simple Mail Transfer Protocol (SMTP)**. We now describe each of these components in the context of a sender, Alice, sending an e-mail message to a recipient, Bob. User agents allow users to read, reply to, forward, save, and compose messages. Microsoft Outlook and Apple Mail are examples of user agents for e-mail. When Alice is finished composing her message, her user agent sends the message to her mail server, where the message is placed in the mail server's outgoing message queue. When Bob wants to read a message, his user agent retrieves the message from his mailbox in his mail server.

Mail servers form the core of the e-mail infrastructure. Each recipient, such as Bob, has a **mailbox** located in one of the mail servers. Bob's mailbox manages and

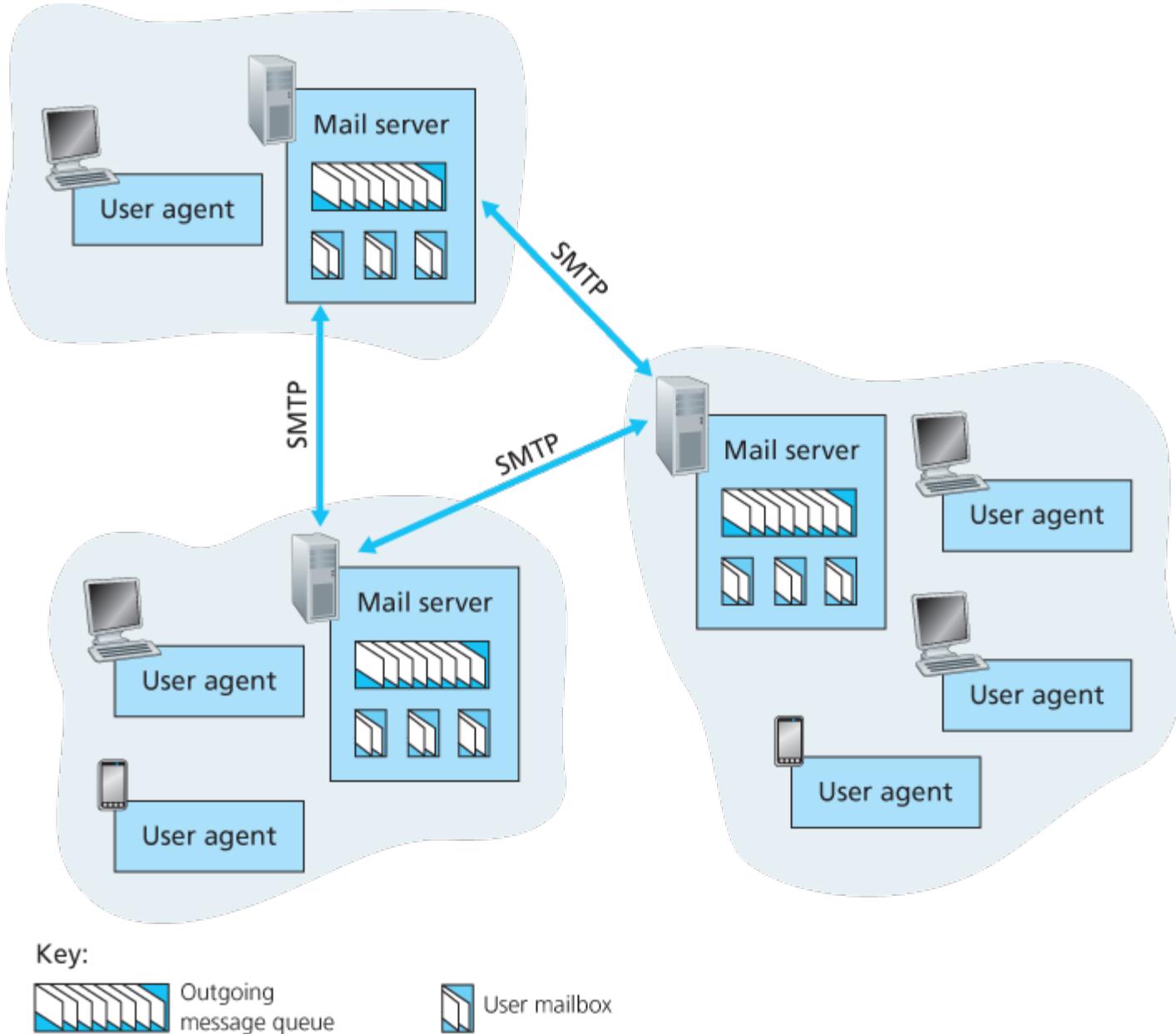


Figure 2.14 A high-level view of the Internet e-mail system

maintains the messages that have been sent to him. A typical message starts its journey in the sender's user agent, travels to the sender's mail server, and travels to the recipient's mail server, where it is deposited in the recipient's mailbox. When Bob wants to access the messages in his mailbox, the mail server containing his mailbox authenticates Bob (with usernames and passwords). Alice's mail server must also deal with failures in Bob's mail server. If Alice's server cannot deliver mail to Bob's server, Alice's server holds the message in a **message queue** and attempts to transfer the message later. Reattempts are often done every 30 minutes or so; if there is no success after several days, the server removes the message and notifies the sender (Alice) with an e-mail message.

SMTP is the principal application-layer protocol for Internet electronic mail. It uses the reliable data transfer service of TCP to transfer mail from the sender's mail server to the recipient's mail server. As with most application-layer protocols, SMTP has two sides: a client side, which executes on the sender's mail server, and a server side, which executes on the recipient's mail server. Both the client and server sides of SMTP run on every mail server. When a mail server sends mail to other mail servers, it acts as an SMTP client. When a mail server receives mail from other mail servers, it acts as an SMTP server.

### 2.3.1 SMTP

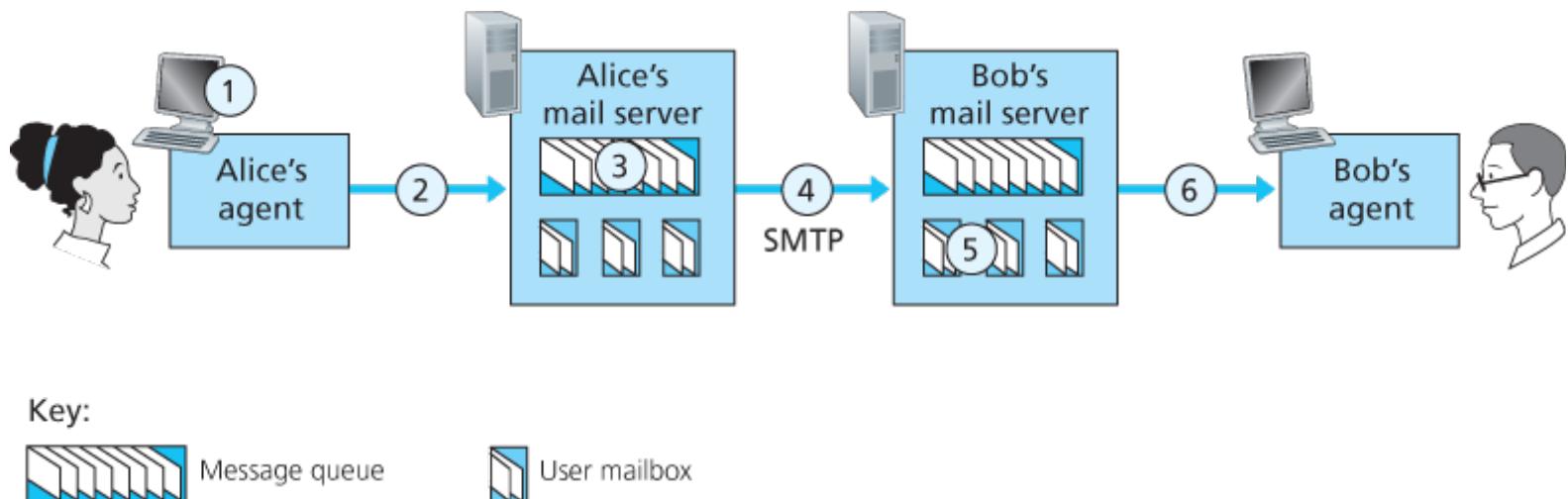
SMTP, defined in RFC 5321, is at the heart of Internet electronic mail. As mentioned above, SMTP transfers messages from senders' mail servers to the recipients' mail servers. SMTP is much older than HTTP. (The original SMTP RFC dates back to 1982, and SMTP was around long before that.) Although SMTP has numerous wonderful qualities, as evidenced by its ubiquity in the Internet, it is nevertheless a legacy technology that possesses certain archaic characteristics. For example, it restricts the body (not just the headers) of all mail messages to simple 7-bit ASCII. This restriction made sense in the early 1980s when transmission capacity was scarce and no one was e-mailing large attachments or large image, audio, or video files. But today, in the multimedia era, the 7-bit ASCII restriction is a bit of a pain—it requires binary multimedia data to be encoded to ASCII before being sent over SMTP; and it requires the corresponding ASCII message to be decoded back to binary after SMTP transport. Recall from [Section 2.2](#) that HTTP does not require multimedia data to be ASCII encoded before transfer.

To illustrate the basic operation of SMTP, let's walk through a common scenario. Suppose Alice wants to send Bob a simple ASCII message.

1. Alice invokes her user agent for e-mail, provides Bob's e-mail address (for example, [bob@someschool.edu](mailto:bob@someschool.edu)), composes a message, and instructs the user agent to send the message.
2. Alice's user agent sends the message to her mail server, where it is placed in a message queue.
3. The client side of SMTP, running on Alice's mail server, sees the message in the message queue. It opens a TCP connection to an SMTP server, running on Bob's mail server.
4. After some initial SMTP handshaking, the SMTP client sends Alice's message into the TCP connection.
5. At Bob's mail server, the server side of SMTP receives the message. Bob's mail server then places the message in Bob's mailbox.
6. Bob invokes his user agent to read the message at his convenience.

The scenario is summarized in [Figure 2.15](#).

It is important to observe that SMTP does not normally use intermediate mail servers for sending mail, even when the two mail servers are located at opposite ends of the world. If Alice's server is in Hong Kong and Bob's server is in St. Louis, the TCP



**Figure 2.15 Alice sends a message to Bob**

connection is a direct connection between the Hong Kong and St. Louis servers. In particular, if Bob’s mail server is down, the message remains in Alice’s mail server and waits for a new attempt—the message does not get placed in some intermediate mail server.

Let’s now take a closer look at how SMTP transfers a message from a sending mail server to a receiving mail server. We will see that the SMTP protocol has many similarities with protocols that are used for face-to-face human interaction. First, the client SMTP (running on the sending mail server host) has TCP establish a connection to port 25 at the server SMTP (running on the receiving mail server host). If the server is down, the client tries again later. Once this connection is established, the server and client perform some application-layer handshaking—just as humans often introduce themselves before transferring information from one to another, SMTP clients and servers introduce themselves before transferring information. During this SMTP handshaking phase, the SMTP client indicates the e-mail address of the sender (the person who generated the message) and the e-mail address of the recipient. Once the SMTP client and server have introduced themselves to each other, the client sends the message. SMTP can count on the reliable data transfer service of TCP to get the message to the server without errors. The client then repeats this process over the same TCP connection if it has other messages to send to the server; otherwise, it instructs TCP to close the connection.

Let’s next take a look at an example transcript of messages exchanged between an SMTP client (C) and an SMTP server (S). The hostname of the client is [crepes.fr](http://crepes.fr) and the hostname of the server is [hamburger.edu](http://hamburger.edu). The ASCII text lines prefaced with *C:* are exactly the lines the client sends into its TCP socket, and the ASCII text lines prefaced with *S:* are exactly the lines the server sends into its TCP socket. The following transcript begins as soon as the TCP connection is established.

```

S: 220 hamburger.edu
C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you

```

```
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr ... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```

In the example above, the client sends a message (“*Do you like ketchup? How about pickles?*”) from mail server **crepes.fr** to mail server *hamburger.edu*. As part of the dialogue, the client issued five commands: **HELO** (an abbreviation for HELLO), **MAIL FROM**, **RCPT TO**, **DATA**, and **QUIT**. These commands are self-explanatory. The client also sends a line consisting of a single period, which indicates the end of the message to the server. (In ASCII jargon, each message ends with **CRLF**, where **CR** and **LF** stand for carriage return and line feed, respectively.) The server issues replies to each command, with each reply having a reply code and some (optional) English-language explanation. We mention here that SMTP uses persistent connections: If the sending mail server has several messages to send to the same receiving mail server, it can send all of the messages over the same TCP connection. For each message, the client begins the process with a new **MAIL FROM: crepes.fr**, designates the end of message with an isolated period, and issues **QUIT** only after all messages have been sent.

It is highly recommended that you use Telnet to carry out a direct dialogue with an SMTP server. To do this, issue

```
telnet serverName 25
```

where **serverName** is the name of a local mail server. When you do this, you are simply establishing a TCP connection between your local host and the mail server. After typing this line, you should immediately receive the **220** reply from the server. Then issue the SMTP commands **HELO**, **MAIL FROM**, **RCPT TO**, **DATA**, **CRLF.CRLF**, and **QUIT** at the appropriate times. It is also highly recommended that you do Programming Assignment 3 at the end of this chapter. In that assignment, you’ll build a simple user agent that implements the client side of SMTP. It will allow you to send an e-

mail message to an arbitrary recipient via a local mail server.

### 2.3.2 Comparison with HTTP

Let's now briefly compare SMTP with HTTP. Both protocols are used to transfer files from one host to another: HTTP transfers files (also called objects) from a Web server to a Web client (typically a browser); SMTP transfers files (that is, e-mail messages) from one mail server to another mail server. When transferring the files, both persistent HTTP and SMTP use persistent connections. Thus, the two protocols have common characteristics. However, there are important differences. First, HTTP is mainly a **pull protocol**—someone loads information on a Web server and users use HTTP to pull the information from the server at their convenience. In particular, the TCP connection is initiated by the machine that wants to receive the file. On the other hand, SMTP is primarily a **push protocol**—the sending mail server pushes the file to the receiving mail server. In particular, the TCP connection is initiated by the machine that wants to send the file.

A second difference, which we alluded to earlier, is that SMTP requires each message, including the body of each message, to be in 7-bit ASCII format. If the message contains characters that are not 7-bit ASCII (for example, French characters with accents) or contains binary data (such as an image file), then the message has to be encoded into 7-bit ASCII. HTTP data does not impose this restriction.

A third important difference concerns how a document consisting of text and images (along with possibly other media types) is handled. As we learned in [Section 2.2](#), HTTP encapsulates each object in its own HTTP response message. SMTP places all of the message's objects into one message.

### 2.3.3 Mail Message Formats

When Alice writes an ordinary snail-mail letter to Bob, she may include all kinds of peripheral header information at the top of the letter, such as Bob's address, her own return address, and the date. Similarly, when an e-mail message is sent from one person to another, a header containing peripheral information precedes the body of the message itself. This peripheral information is contained in a series of header lines, which are defined in RFC 5322. The header lines and the body of the message are separated by a blank line (that is, by [\*CRLF\*](#)). RFC 5322 specifies the exact format for mail header lines as well as their semantic interpretations. As with HTTP, each header line contains readable text, consisting of a keyword followed by a colon followed by a value. Some of the keywords are required and others are optional. Every header must have a [\*From:\*](#) header line and a [\*To:\*](#) header line; a header may include a [\*Subject:\*](#) header line as well as other optional header lines. It is important to note that these header lines are *different* from the SMTP commands we studied in [Section 2.4.1](#) (even though

they contain some common words such as “*from*” and “*to*”). The commands in that section were part of the SMTP handshaking protocol; the header lines examined in this section are part of the mail message itself.

A typical message header looks like this:

```
From: alice@crepes.fr  
To: bob@hamburger.edu  
Subject: Searching for the meaning of life.
```

After the message header, a blank line follows; then the message body (in ASCII) follows. You should use Telnet to send a message to a mail server that contains some header lines, including the

*Subject*: header line. To do this, issue `telnet serverName 25`, as discussed in [Section 2.4.1](#).

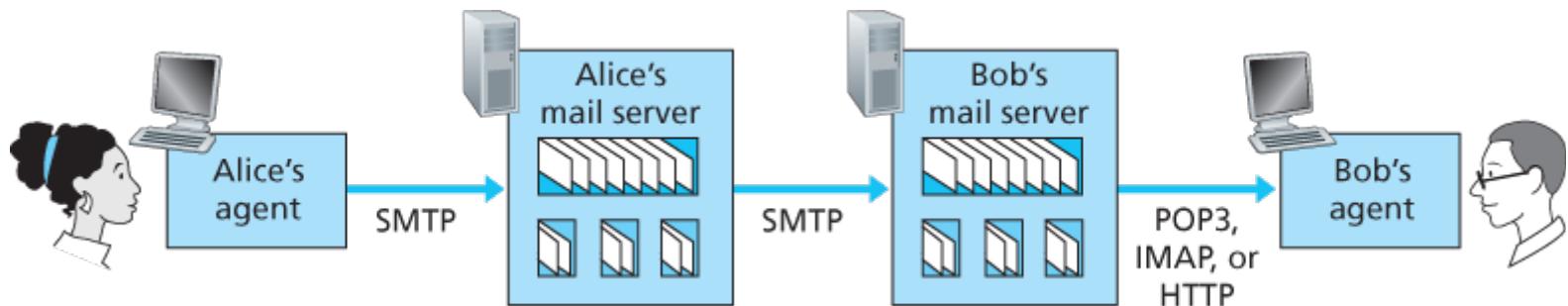
### 2.3.4 Mail Access Protocols

Once SMTP delivers the message from Alice’s mail server to Bob’s mail server, the message is placed in Bob’s mailbox. Throughout this discussion we have tacitly assumed that Bob reads his mail by logging onto the server host and then executing a mail reader that runs on that host. Up until the early 1990s this was the standard way of doing things. But today, mail access uses a client-server architecture—the typical user reads e-mail with a client that executes on the user’s end system, for example, on an office PC, a laptop, or a smartphone. By executing a mail client on a local PC, users enjoy a rich set of features, including the ability to view multimedia messages and attachments.

Given that Bob (the recipient) executes his user agent on his local PC, it is natural to consider placing a mail server on his local PC as well. With this approach, Alice’s mail server would dialogue directly with Bob’s PC. There is a problem with this approach, however. Recall that a mail server manages mailboxes and runs the client and server sides of SMTP. If Bob’s mail server were to reside on his local PC, then Bob’s PC would have to remain always on, and connected to the Internet, in order to receive new mail, which can arrive at any time. This is impractical for many Internet users. Instead, a typical user runs a user agent on the local PC but accesses its mailbox stored on an always-on shared mail server. This mail server is shared with other users and is typically maintained by the user’s ISP (for example, university or company).

Now let’s consider the path an e-mail message takes when it is sent from Alice to Bob. We just learned that at some point along the path the e-mail message needs to be deposited in Bob’s mail server. This could be done simply by having Alice’s user agent send the message directly to Bob’s mail server. And

this could be done with SMTP—indeed, SMTP has been designed for pushing e-mail from one host to another. However, typically the sender's user agent does not dialogue directly with the recipient's mail server. Instead, as shown in [Figure 2.16](#), Alice's user agent uses SMTP to push the e-mail message into her mail server, then Alice's mail server uses SMTP (as an SMTP client) to relay the e-mail message to Bob's mail server. Why the two-step procedure? Primarily because without relaying through Alice's mail server, Alice's user agent doesn't have any recourse to an unreachable destination.



**Figure 2.16 E-mail protocols and their communicating entities**

mail server. By having Alice first deposit the e-mail in her own mail server, Alice's mail server can repeatedly try to send the message to Bob's mail server, say every 30 minutes, until Bob's mail server becomes operational. (And if Alice's mail server is down, then she has the recourse of complaining to her system administrator!) The SMTP RFC defines how the SMTP commands can be used to relay a message across multiple SMTP servers.

But there is still one missing piece to the puzzle! How does a recipient like Bob, running a user agent on his local PC, obtain his messages, which are sitting in a mail server within Bob's ISP? Note that Bob's user agent can't use SMTP to obtain the messages because obtaining the messages is a pull operation, whereas SMTP is a push protocol. The puzzle is completed by introducing a special mail access protocol that transfers messages from Bob's mail server to his local PC. There are currently a number of popular mail access protocols, including [Post Office Protocol—Version 3 \(POP3\)](#), [Internet Mail Access Protocol \(IMAP\)](#), and [HTTP](#).

[Figure 2.16](#) provides a summary of the protocols that are used for Internet mail: SMTP is used to transfer mail from the sender's mail server to the recipient's mail server; SMTP is also used to transfer mail from the sender's user agent to the sender's mail server. A mail access protocol, such as POP3, is used to transfer mail from the recipient's mail server to the recipient's user agent.

### POP3

POP3 is an extremely simple mail access protocol. It is defined in [\[RFC 1939\]](#), which is short and quite readable. Because the protocol is so simple, its functionality is rather limited. POP3 begins when the user agent (the client) opens a TCP connection to the mail server (the server) on port 110. With the TCP

connection established, POP3 progresses through three phases: authorization, transaction, and update. During the first phase, authorization, the user agent sends a username and a password (in the clear) to authenticate the user. During the second phase, transaction, the user agent retrieves messages; also during this phase, the user agent can mark messages for deletion, remove deletion marks, and obtain mail statistics. The third phase, update, occurs after the client has issued the `quit` command, ending the POP3 session; at this time, the mail server deletes the messages that were marked for deletion.

In a POP3 transaction, the user agent issues commands, and the server responds to each command with a reply. There are two possible responses: `+OK` (sometimes followed by server-to-client data), used by the server to indicate that the previous command was fine; and `-ERR`, used by the server to indicate that something was wrong with the previous command.

The authorization phase has two principal commands: `user <username>` and `pass <password>`. To illustrate these two commands, we suggest that you Telnet directly into a POP3 server, using port 110, and issue these commands. Suppose that `mailServer` is the name of your mail server. You will see something like:

```
telnet mailServer 110
+OK POP3 server ready
user bob
+OK
pass hungry
+OK user successfully logged on
```

If you misspell a command, the POP3 server will reply with an `-ERR` message.

Now let's take a look at the transaction phase. A user agent using POP3 can often be configured (by the user) to “download and delete” or to “download and keep.” The sequence of commands issued by a POP3 user agent depends on which of these two modes the user agent is operating in. In the download-and-delete mode, the user agent will issue the `list`, `retr`, and `dele` commands. As an example, suppose the user has two messages in his or her mailbox. In the dialogue below, `C:` (standing for client) is the user agent and `S:` (standing for server) is the mail server. The transaction will look something like:

```
C: list
S: 1 498
S: 2 912
```

```
S: .
C: retr 1
S: (blah blah ...
S: .....
S: .....blah)
S: .
C: dele 1
C: retr 2
S: (blah blah ...
S: .....
S: .....blah)
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```

The user agent first asks the mail server to list the size of each of the stored messages. The user agent then retrieves and deletes each message from the server. Note that after the authorization phase, the user agent employed only four commands: *list*, *retr*, *dele*, and *quit*. The syntax for these commands is defined in RFC 1939. After processing the *quit* command, the POP3 server enters the update phase and removes messages 1 and 2 from the mailbox.

A problem with this download-and-delete mode is that the recipient, Bob, may be nomadic and may want to access his mail messages from multiple machines, for example, his office PC, his home PC, and his portable computer. The download-and-delete mode partitions Bob's mail messages over these three machines; in particular, if Bob first reads a message on his office PC, he will not be able to reread the message from his portable at home later in the evening. In the download-and-keep mode, the user agent leaves the messages on the mail server after downloading them. In this case, Bob can reread messages from different machines; he can access a message from work and access it again later in the week from home.

During a POP3 session between a user agent and the mail server, the POP3 server maintains some state information; in particular, it keeps track of which user messages have been marked deleted. However, the POP3 server does not carry state information across POP3 sessions. This lack of state information across sessions greatly simplifies the implementation of a POP3 server.

## IMAP

With POP3 access, once Bob has downloaded his messages to the local machine, he can create mail

folders and move the downloaded messages into the folders. Bob can then delete messages, move messages across folders, and search for messages (by sender name or subject). But this paradigm—namely, folders and messages in the local machine—poses a problem for the nomadic user, who would prefer to maintain a folder hierarchy on a remote server that can be accessed from any computer. This is not possible with POP3—the POP3 protocol does not provide any means for a user to create remote folders and assign messages to folders.

To solve this and other problems, the IMAP protocol, defined in [\[RFC 3501\]](#), was invented. Like POP3, IMAP is a mail access protocol. It has many more features than POP3, but it is also significantly more complex. (And thus the client and server side implementations are significantly more complex.)

An IMAP server will associate each message with a folder; when a message first arrives at the server, it is associated with the recipient's INBOX folder. The recipient can then move the message into a new, user-created folder, read the message, delete the message, and so on. The IMAP protocol provides commands to allow users to create folders and move messages from one folder to another. IMAP also provides commands that allow users to search remote folders for messages matching specific criteria. Note that, unlike POP3, an IMAP server maintains user state information across IMAP sessions—for example, the names of the folders and which messages are associated with which folders.

Another important feature of IMAP is that it has commands that permit a user agent to obtain components of messages. For example, a user agent can obtain just the message header of a message or just one part of a multipart MIME message. This feature is useful when there is a low-bandwidth connection (for example, a slow-speed modem link) between the user agent and its mail server. With a low-bandwidth connection, the user may not want to download all of the messages in its mailbox, particularly avoiding long messages that might contain, for example, an audio or video clip.

### *Web-Based E-Mail*

More and more users today are sending and accessing their e-mail through their Web browsers. Hotmail introduced Web-based access in the mid 1990s. Now Web-based e-mail is also provided by Google, Yahoo!, as well as just about every major university and corporation. With this service, the user agent is an ordinary Web browser, and the user communicates with its remote mailbox via HTTP. When a recipient, such as Bob, wants to access a message in his mailbox, the e-mail message is sent from Bob's mail server to Bob's browser using the HTTP protocol rather than the POP3 or IMAP protocol. When a sender, such as Alice, wants to send an e-mail message, the e-mail message is sent from her browser to her mail server over HTTP rather than over SMTP. Alice's mail server, however, still sends messages to, and receives messages from, other mail servers using SMTP.

## 2.4 DNS—The Internet’s Directory Service

We human beings can be identified in many ways. For example, we can be identified by the names that appear on our birth certificates. We can be identified by our social security numbers. We can be identified by our driver’s license numbers. Although each of these identifiers can be used to identify people, within a given context one identifier may be more appropriate than another. For example, the computers at the IRS (the infamous tax-collecting agency in the United States) prefer to use fixed-length social security numbers rather than birth certificate names. On the other hand, ordinary people prefer the more mnemonic birth certificate names rather than social security numbers. (Indeed, can you imagine saying, “Hi. My name is 132-67-9875. Please meet my husband, 178-87-1146.”)

Just as humans can be identified in many ways, so too can Internet hosts. One identifier for a host is its **hostname**. Hostnames—such as `www.facebook.com`, `www.google.com`, `gaias.cs.umass.edu`—are mnemonic and are therefore appreciated by humans. However, hostnames provide little, if any, information about the location within the Internet of the host. (A hostname such as `www.eurecom.fr`, which ends with the country code `.fr`, tells us that the host is probably in France, but doesn’t say much more.) Furthermore, because hostnames can consist of variable-length alphanumeric characters, they would be difficult to process by routers. For these reasons, hosts are also identified by so-called **IP addresses**.

We discuss IP addresses in some detail in [Chapter 4](#), but it is useful to say a few brief words about them now. An IP address consists of four bytes and has a rigid hierarchical structure. An IP address looks like `121.7.106.83`, where each period separates one of the bytes expressed in decimal notation from 0 to 255. An IP address is hierarchical because as we scan the address from left to right, we obtain more and more specific information about where the host is located in the Internet (that is, within which network, in the network of networks). Similarly, when we scan a postal address from bottom to top, we obtain more and more specific information about where the addressee is located.

### 2.4.1 Services Provided by DNS

We have just seen that there are two ways to identify a host—by a hostname and by an IP address. People prefer the more mnemonic hostname identifier, while routers prefer fixed-length, hierarchically structured IP addresses. In order to reconcile these preferences, we need a directory service that translates hostnames to IP addresses. This is the main task of the Internet’s **domain name system (DNS)**. The DNS is (1) a distributed database implemented in a hierarchy of **DNS servers**, and (2) an

application-layer protocol that allows hosts to query the distributed database. The DNS servers are often UNIX machines running the Berkeley Internet Name Domain (BIND) software [BIND 2016]. The DNS protocol runs over UDP and uses port 53.

DNS is commonly employed by other application-layer protocols—including HTTP and SMTP to translate user-supplied hostnames to IP addresses. As an example, consider what happens when a browser (that is, an HTTP client), running on some user’s host, requests the URL

`www.someschool.edu/index.html`. In order for the user’s host to be able to send an HTTP request message to the Web server `www.someschool.edu`, the user’s host must first obtain the IP address of `www.someschool.edu`. This is done as follows.

1. The same user machine runs the client side of the DNS application.
2. The browser extracts the hostname, `www.someschool.edu`, from the URL and passes the hostname to the client side of the DNS application.
3. The DNS client sends a query containing the hostname to a DNS server.
4. The DNS client eventually receives a reply, which includes the IP address for the hostname.
5. Once the browser receives the IP address from DNS, it can initiate a TCP connection to the HTTP server process located at port 80 at that IP address.

We see from this example that DNS adds an additional delay—sometimes substantial—to the Internet applications that use it. Fortunately, as we discuss below, the desired IP address is often cached in a “nearby” DNS server, which helps to reduce DNS network traffic as well as the average DNS delay.

DNS provides a few other important services in addition to translating hostnames to IP addresses:

- **Host aliasing.** A host with a complicated hostname can have one or more alias names. For example, a hostname such as `relay1.west-coast.enterprise.com` could have, say, two aliases such as `enterprise.com` and `www.enterprise.com`. In this case, the hostname `relay1.west-coast.enterprise.com` is said to be a **canonical hostname**. Alias hostnames, when present, are typically more mnemonic than canonical hostnames. DNS can be invoked by an application to obtain the canonical hostname for a supplied alias hostname as well as the IP address of the host.
- **Mail server aliasing.** For obvious reasons, it is highly desirable that e-mail addresses be mnemonic. For example, if Bob has an account with Yahoo Mail, Bob’s e-mail address might be as simple as `bob@yahoo.mail`. However, the hostname of the Yahoo mail server is more complicated and much less mnemonic than simply `yahoo.com` (for example, the canonical hostname might be something like `relay1.west-coast.yahoo.com`). DNS can be invoked by a mail application to obtain the canonical hostname for a supplied alias hostname as well as the IP address of the host. In fact, the MX record (see below) permits a company’s mail server and Web server to have identical (aliased) hostnames; for example, a company’s Web server and mail server can both be called

[enterprise.com](#).

- **Load distribution.** DNS is also used to perform load distribution among replicated servers, such as replicated Web servers. Busy sites, such as [cnn.com](#), are replicated over multiple servers, with each server running on a different end system and each having a different IP address. For replicated Web servers, a set of IP addresses is thus associated with one canonical hostname. The DNS database contains this set of IP addresses. When clients make a DNS query for a name mapped to a set of addresses, the server responds with the entire set of IP addresses, but rotates the ordering of the addresses within each reply. Because a client typically sends its HTTP request message to the IP address that is listed first in the set, DNS rotation distributes the traffic among the replicated servers. DNS rotation is also used for e-mail so that multiple mail servers can have the same alias name. Also, content distribution companies such as Akamai have used DNS in more sophisticated ways [\[Dilley 2002\]](#) to provide Web content distribution (see [Section 2.6.3](#)).

The DNS is specified in RFC 1034 and RFC 1035, and updated in several additional RFCs. It is a complex system, and we only touch upon key aspects of its

#### PRINCIPLES IN PRACTICE

##### *DNS: CRITICAL NETWORK FUNCTIONS VIA THE CLIENT-SERVER PARADIGM*

Like HTTP, FTP, and SMTP, the DNS protocol is an application-layer protocol since it (1) runs between communicating end systems using the client-server paradigm and (2) relies on an underlying end-to-end transport protocol to transfer DNS messages between communicating end systems. In another sense, however, the role of the DNS is quite different from Web, file transfer, and e-mail applications. Unlike these applications, the DNS is not an application with which a user directly interacts. Instead, the DNS provides a core Internet function—namely, translating hostnames to their underlying IP addresses, for user applications and other software in the Internet. We noted in [Section 1.2](#) that much of the complexity in the Internet architecture is located at the “edges” of the network. The DNS, which implements the critical name-to-address translation process using clients and servers located at the edge of the network, is yet another example of that design philosophy.

operation here. The interested reader is referred to these RFCs and the book by Albitz and Liu [\[Albitz 1993\]](#); see also the retrospective paper [\[Mockapetris 1988\]](#), which provides a nice description of the what and why of DNS, and [\[Mockapetris 2005\]](#).

#### 2.4.2 Overview of How DNS Works

We now present a high-level overview of how DNS works. Our discussion will focus on the hostname-to-

IP-address translation service.

Suppose that some application (such as a Web browser or a mail reader) running in a user's host needs to translate a hostname to an IP address. The application will invoke the client side of DNS, specifying the hostname that needs to be translated. (On many UNIX-based machines, `gethostbyname()` is the function call that an application calls in order to perform the translation.) DNS in the user's host then takes over, sending a query message into the network. All DNS query and reply messages are sent within UDP datagrams to port 53. After a delay, ranging from milliseconds to seconds, DNS in the user's host receives a DNS reply message that provides the desired mapping. This mapping is then passed to the invoking application. Thus, from the perspective of the invoking application in the user's host, DNS is a black box providing a simple, straightforward translation service. But in fact, the black box that implements the service is complex, consisting of a large number of DNS servers distributed around the globe, as well as an application-layer protocol that specifies how the DNS servers and querying hosts communicate.

A simple design for DNS would have one DNS server that contains all the mappings. In this centralized design, clients simply direct all queries to the single DNS server, and the DNS server responds directly to the querying clients. Although the simplicity of this design is attractive, it is inappropriate for today's Internet, with its vast (and growing) number of hosts. The problems with a centralized design include:

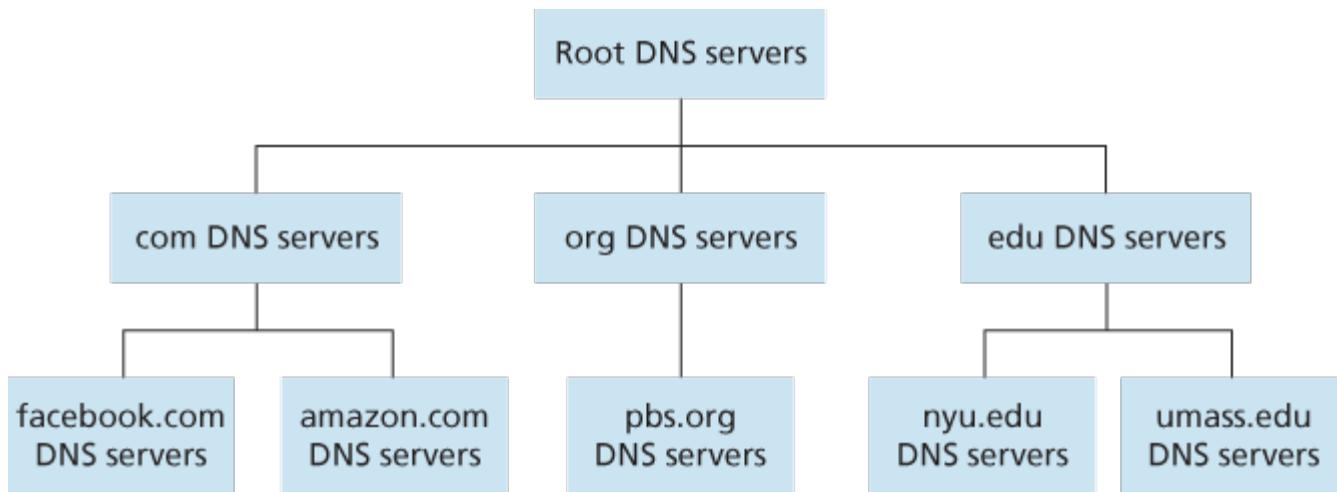
- **A single point of failure.** If the DNS server crashes, so does the entire Internet!
- **Traffic volume.** A single DNS server would have to handle all DNS queries (for all the HTTP requests and e-mail messages generated from hundreds of millions of hosts).
- **Distant centralized database.** A single DNS server cannot be "close to" all the querying clients. If we put the single DNS server in New York City, then all queries from Australia must travel to the other side of the globe, perhaps over slow and congested links. This can lead to significant delays.
- **Maintenance.** The single DNS server would have to keep records for all Internet hosts. Not only would this centralized database be huge, but it would have to be updated frequently to account for every new host.

In summary, a centralized database in a single DNS server simply *doesn't scale*. Consequently, the DNS is distributed by design. In fact, the DNS is a wonderful example of how a distributed database can be implemented in the Internet.

### *A Distributed, Hierarchical Database*

In order to deal with the issue of scale, the DNS uses a large number of servers, organized in a hierarchical fashion and distributed around the world. No single DNS server has all of the mappings for all of the hosts in the Internet. Instead, the mappings are distributed across the DNS servers. To a first approximation, there are three classes of DNS servers—root DNS servers, top-level domain (TLD) DNS

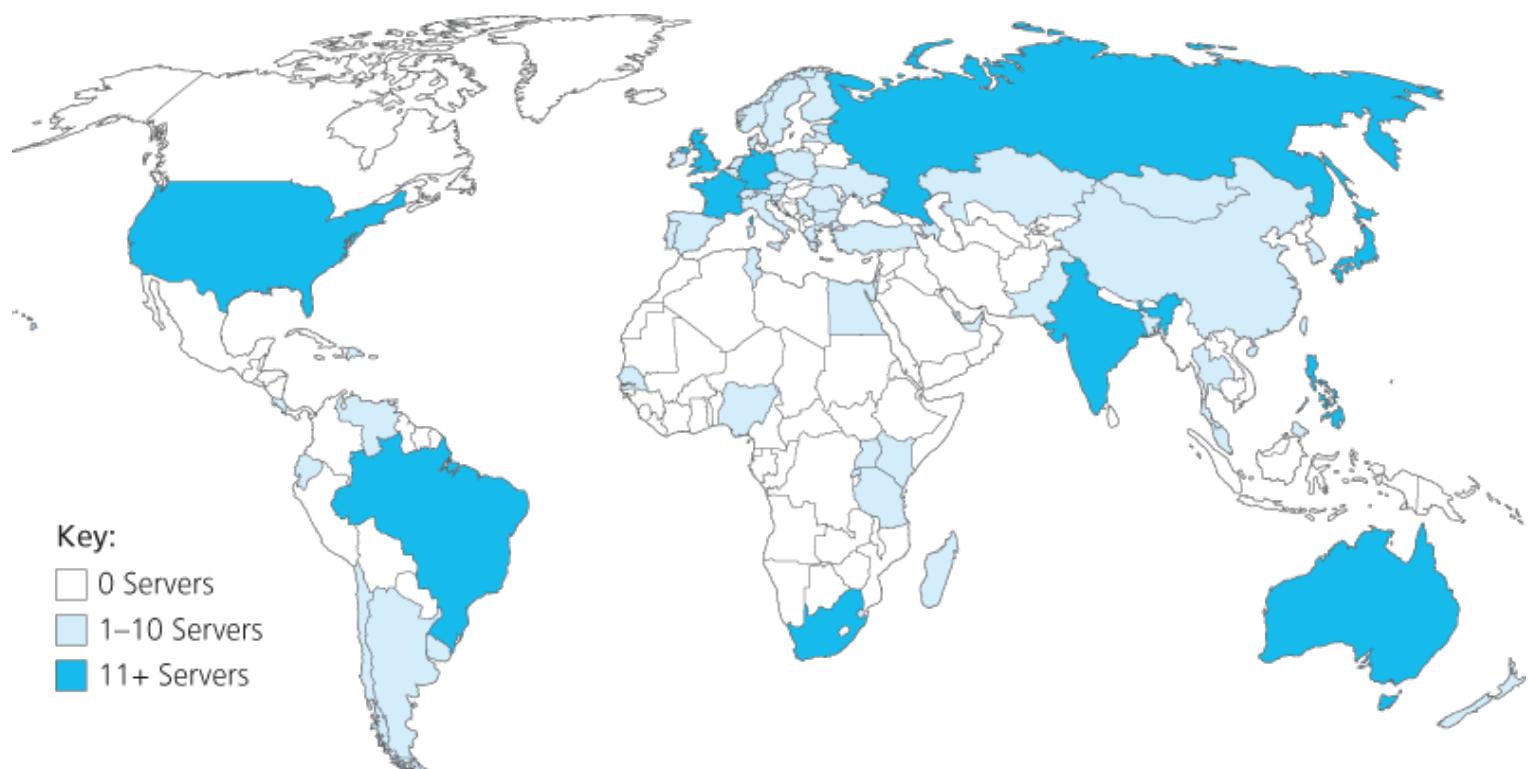
servers, and authoritative DNS servers—organized in a hierarchy as shown in [Figure 2.17](#). To understand how these three classes of servers interact, suppose a DNS client wants to determine the IP address for the hostname [www.amazon.com](#). To a first



**Figure 2.17 Portion of the hierarchy of DNS servers**

approximation, the following events will take place. The client first contacts one of the root servers, which returns IP addresses for TLD servers for the top-level domain [com](#). The client then contacts one of these TLD servers, which returns the IP address of an authoritative server for [amazon.com](#). Finally, the client contacts one of the authoritative servers for [amazon.com](#), which returns the IP address for the hostname [www.amazon.com](#). We'll soon examine this DNS lookup process in more detail. But let's first take a closer look at these three classes of DNS servers:

- **Root DNS servers.** There are over 400 root name servers scattered all over the world. [Figure 2.18](#) shows the countries that have root names servers, with countries having more than ten darkly shaded. These root name servers are managed by 13 different organizations. The full list of root name servers, along with the organizations that manage them and their IP addresses can be found at [\[Root Servers 2016\]](#). Root name servers provide the IP addresses of the TLD servers.
- **Top-level domain (TLD) servers.** For each of the top-level domains — top-level domains such as com, org, net, edu, and gov, and all of the country top-level domains such as uk, fr, ca, and jp — there is TLD server (or server cluster). The company Verisign Global Registry Services maintains the TLD servers for the [com](#) top-level domain, and the company Educause maintains the TLD servers for the [edu](#) top-level domain. The network infrastructure supporting a TLD can be large and complex; see [\[Osterweil 2012\]](#) for a nice overview of the Verisign network. See [\[TLD list 2016\]](#) for a list of all top-level domains. TLD servers provide the IP addresses for authoritative DNS servers.



**Figure 2.18 DNS root servers in 2016**

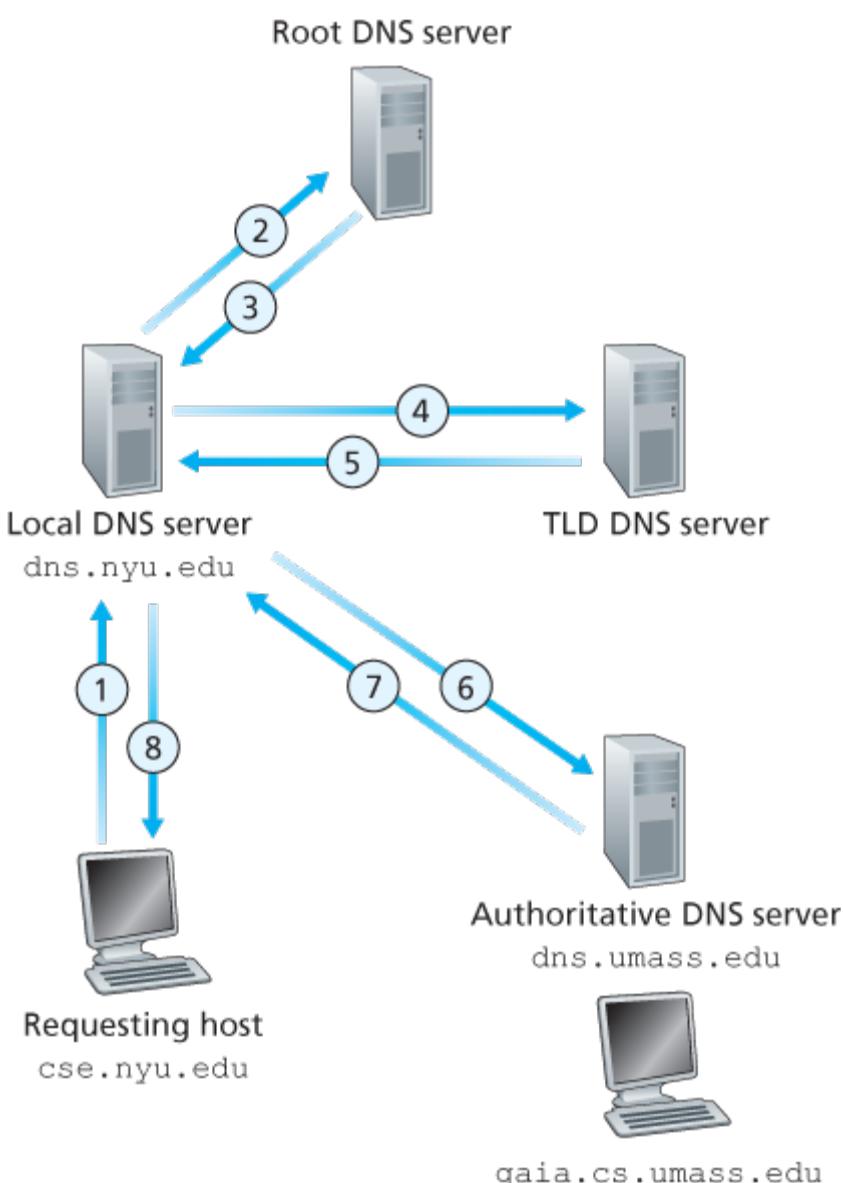
- **Authoritative DNS servers.** Every organization with publicly accessible hosts (such as Web servers and mail servers) on the Internet must provide publicly accessible DNS records that map the names of those hosts to IP addresses. An organization's authoritative DNS server houses these DNS records. An organization can choose to implement its own authoritative DNS server to hold these records; alternatively, the organization can pay to have these records stored in an authoritative DNS server of some service provider. Most universities and large companies implement and maintain their own primary and secondary (backup) authoritative DNS server.

The root, TLD, and authoritative DNS servers all belong to the hierarchy of DNS servers, as shown in [Figure 2.17](#). There is another important type of DNS server called the [local DNS server](#). A local DNS server does not strictly belong to the hierarchy of servers but is nevertheless central to the DNS architecture. Each ISP—such as a residential ISP or an institutional ISP—has a local DNS server (also called a default name server). When a host connects to an ISP, the ISP provides the host with the IP addresses of one or more of its local DNS servers (typically through DHCP, which is discussed in [Chapter 4](#)). You can easily determine the IP address of your local DNS server by accessing network status windows in Windows or UNIX. A host's local DNS server is typically “close to” the host. For an institutional ISP, the local DNS server may be on the same LAN as the host; for a residential ISP, it is typically separated from the host by no more than a few routers. When a host makes a DNS query, the query is sent to the local DNS server, which acts a proxy, forwarding the query into the DNS server hierarchy, as we'll discuss in more detail below.

Let's take a look at a simple example. Suppose the host `cse.nyu.edu` desires the IP address of `gaias.cs.umass.edu`. Also suppose that NYU's local DNS server for `cse.nyu.edu` is called

`dns.nyu.edu` and that an authoritative DNS server for `gaia.cs.umass.edu` is called `dns.umass.edu`. As shown in **Figure 2.19**, the host `cse.nyu.edu` first sends a DNS query message to its local DNS server, `dns.nyu.edu`. The query message contains the hostname to be translated, namely, `gaia.cs.umass.edu`. The local DNS server forwards the query message to a root DNS server. The root DNS server takes note of the `edu` suffix and returns to the local DNS server a list of IP addresses for TLD servers responsible for `edu`. The local DNS server then resends the query message to one of these TLD servers. The TLD server takes note of the `umass.edu` suffix and responds with the IP address of the authoritative DNS server for the University of Massachusetts, namely, `dns.umass.edu`. Finally, the local DNS server resends the query message directly to `dns.umass.edu`, which responds with the IP address of `gaia.cs.umass.edu`. Note that in this example, in order to obtain the mapping for one hostname, eight DNS messages were sent: four query messages and four reply messages! We'll soon see how DNS caching reduces this query traffic.

Our previous example assumed that the TLD server knows the authoritative DNS server for the hostname. In general this not always true. Instead, the TLD server



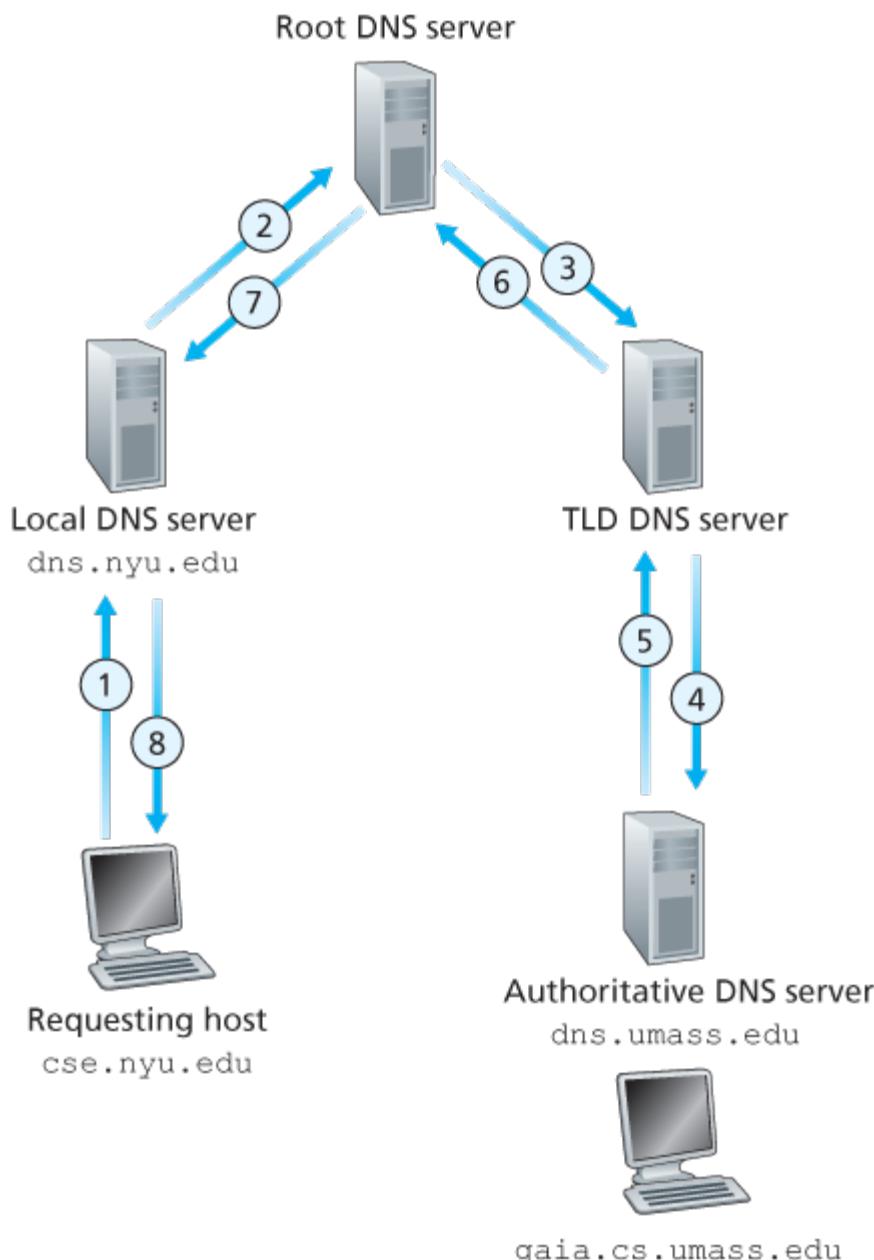
**Figure 2.19 Interaction of the various DNS servers**

may know only of an intermediate DNS server, which in turn knows the authoritative DNS server for the hostname. For example, suppose again that the University of Massachusetts has a DNS server for the university, called `dns.umass.edu`. Also suppose that each of the departments at the University of Massachusetts has its own DNS server, and that each departmental DNS server is authoritative for all hosts in the department. In this case, when the intermediate DNS server, `dns.umass.edu`, receives a query for a host with a hostname ending with `cs.umass.edu`, it returns to `dns.nyu.edu` the IP address of `dns.cs.umass.edu`, which is authoritative for all hostnames ending with `cs.umass.edu`. The local DNS server `dns.nyu.edu` then sends the query to the authoritative DNS server, which returns the desired mapping to the local DNS server, which in turn returns the mapping to the requesting host. In this case, a total of 10 DNS messages are sent!

The example shown in [Figure 2.19](#) makes use of both **recursive queries** and **iterative queries**. The query sent from `cse.nyu.edu` to `dns.nyu.edu` is a recursive query, since the query asks `dns.nyu.edu` to obtain the mapping on its behalf. But the subsequent three queries are iterative since all of the replies are directly returned to `dns.nyu.edu`. In theory, any DNS query can be iterative or recursive. For example, [Figure 2.20](#) shows a DNS query chain for which all of the queries are recursive. In practice, the queries typically follow the pattern in [Figure 2.19](#): The query from the requesting host to the local DNS server is recursive, and the remaining queries are iterative.

### *DNS Caching*

Our discussion thus far has ignored **DNS caching**, a critically important feature of the DNS system. In truth, DNS extensively exploits DNS caching in order to improve the delay performance and to reduce the number of DNS messages



**Figure 2.20 Recursive queries in DNS**

ricocheting around the Internet. The idea behind DNS caching is very simple. In a query chain, when a DNS server receives a DNS reply (containing, for example, a mapping from a hostname to an IP address), it can cache the mapping in its local memory. For example, in [Figure 2.19](#), each time the local DNS server `dns.nyu.edu` receives a reply from some DNS server, it can cache any of the information contained in the reply. If a hostname/IP address pair is cached in a DNS server and another query arrives to the DNS server for the same hostname, the DNS server can provide the desired IP address, even if it is not authoritative for the hostname. Because hosts and mappings between hostnames and IP addresses are by no means permanent, DNS servers discard cached information after a period of time (often set to two days).

As an example, suppose that a host `apricot.nyu.edu` queries `dns.nyu.edu` for the IP address for the hostname `cnn.com`. Furthermore, suppose that a few hours later, another NYU host, say, `kiwi.nyu.edu`, also queries `dns.nyu.edu` with the same hostname. Because of caching, the local DNS server will be able to immediately return the IP address of `cnn.com` to this second requesting

host without having to query any other DNS servers. A local DNS server can also cache the IP addresses of TLD servers, thereby allowing the local DNS server to bypass the root DNS servers in a query chain. In fact, because of caching, root servers are bypassed for all but a very small fraction of DNS queries.

### 2.4.3 DNS Records and Messages

The DNS servers that together implement the DNS distributed database store **resource records (RRs)**, including RRs that provide hostname-to-IP address mappings. Each DNS reply message carries one or more resource records. In this and the following subsection, we provide a brief overview of DNS resource records and messages; more details can be found in [Albitz 1993] or in the DNS RFCs [RFC 1034; RFC 1035].

A resource record is a four-tuple that contains the following fields:

*(Name, Value, Type, TTL)*

*TTL* is the time to live of the resource record; it determines when a resource should be removed from a cache. In the example records given below, we ignore the *TTL* field. The meaning of *Name* and *Value* depend on *Type*:

- If *Type=A*, then *Name* is a hostname and *Value* is the IP address for the hostname. Thus, a Type A record provides the standard hostname-to-IP address mapping. As an example, *(relay1.bar.foo.com, 145.37.93.126, A)* is a Type A record.
- If *Type=NS*, then *Name* is a domain (such as *foo.com*) and *Value* is the hostname of an authoritative DNS server that knows how to obtain the IP addresses for hosts in the domain. This record is used to route DNS queries further along in the query chain. As an example, *(foo.com, dns.foo.com, NS)* is a Type NS record.
- If *Type=CNAME*, then *Value* is a canonical hostname for the alias hostname *Name*. This record can provide querying hosts the canonical name for a hostname. As an example, *(foo.com, relay1.bar.foo.com, CNAME)* is a CNAME record.
- If *Type=MX*, then *Value* is the canonical name of a mail server that has an alias hostname *Name*. As an example, *(foo.com, mail.bar.foo.com, MX)* is an MX record. MX records allow the hostnames of mail servers to have simple aliases. Note that by using the MX record, a company can have the same aliased name for its mail server and for one of its other servers (such as its Web server). To obtain the canonical name for the mail server, a DNS client would query for an MX

record; to obtain the canonical name for the other server, the DNS client would query for the CNAME record.

If a DNS server is authoritative for a particular hostname, then the DNS server will contain a Type A record for the hostname. (Even if the DNS server is not authoritative, it may contain a Type A record in its cache.) If a server is not authoritative for a hostname, then the server will contain a Type NS record for the domain that includes the hostname; it will also contain a Type A record that provides the IP address of the DNS server in the *Value* field of the NS record. As an example, suppose an edu TLD server is not authoritative for the host `gaia.cs.umass.edu`. Then this server will contain a record for a domain that includes the host `gaia.cs.umass.edu`, for example, (`umass.edu`, `dns.umass.edu`, `NS`). The edu TLD server would also contain a Type A record, which maps the DNS server `dns.umass.edu` to an IP address, for example, (`dns.umass.edu`, `128.119.40.111`, `A`).

### DNS Messages

Earlier in this section, we referred to DNS query and reply messages. These are the only two kinds of DNS messages. Furthermore, both query and reply messages have the same format, as shown in

**Figure 2.21.**The semantics of the various fields in a DNS message are as follows:

- The first 12 bytes is the *header section*, which has a number of fields. The first field is a 16-bit number that identifies the query. This identifier is copied into the reply message to a query, allowing the client to match received replies with sent queries. There are a number of flags in the flag field. A 1-bit query/reply flag indicates whether the message is a query (0) or a reply (1). A 1-bit authoritative flag is

Identification	Flags	
Number of questions	Number of answer RRs	12 bytes
Number of authority RRs	Number of additional RRs	
Questions (variable number of questions)		Name, type fields for a query
Answers (variable number of resource records)		RRs in response to query
Authority (variable number of resource records)		Records for authoritative servers
Additional information (variable number of resource records)		Additional “helpful” info that may be used

## Figure 2.21 DNS message format

set in a reply message when a DNS server is an authoritative server for a queried name. A 1-bit recursion-desired flag is set when a client (host or DNS server) desires that the DNS server perform recursion when it doesn't have the record. A 1-bit recursion-available field is set in a reply if the DNS server supports recursion. In the header, there are also four number-of fields. These fields indicate the number of occurrences of the four types of data sections that follow the header.

- The *question section* contains information about the query that is being made. This section includes (1) a name field that contains the name that is being queried, and (2) a type field that indicates the type of question being asked about the name—for example, a host address associated with a name (Type A) or the mail server for a name (Type MX).
- In a reply from a DNS server, the *answer section* contains the resource records for the name that was originally queried. Recall that in each resource record there is the *Type* (for example, A, NS, CNAME, and MX), the *Value*, and the *TTL*. A reply can return multiple RRs in the answer, since a hostname can have multiple IP addresses (for example, for replicated Web servers, as discussed earlier in this section).
- The *authority section* contains records of other authoritative servers.
- The *additional section* contains other helpful records. For example, the answer field in a reply to an MX query contains a resource record providing the canonical hostname of a mail server. The additional section contains a Type A record providing the IP address for the canonical hostname of the mail server.

How would you like to send a DNS query message directly from the host you're working on to some DNS server? This can easily be done with the [nslookup program](#), which is available from most Windows and UNIX platforms. For example, from a Windows host, open the Command Prompt and invoke the nslookup program by simply typing “nslookup.” After invoking nslookup, you can send a DNS query to any DNS server (root, TLD, or authoritative). After receiving the reply message from the DNS server, nslookup will display the records included in the reply (in a human-readable format). As an alternative to running nslookup from your own host, you can visit one of many Web sites that allow you to remotely employ nslookup. (Just type “nslookup” into a search engine and you'll be brought to one of these sites.) The DNS Wireshark lab at the end of this chapter will allow you to explore the DNS in much more detail.

### *Inserting Records into the DNS Database*

The discussion above focused on how records are retrieved from the DNS database. You might be wondering how records get into the database in the first place. Let's look at how this is done in the context of a specific example. Suppose you have just created an exciting new startup company called Network Utopia. The first thing you'll surely want to do is register the domain name

[networkutopia.com](http://networkutopia.com) at a registrar. A **registrar** is a commercial entity that verifies the uniqueness of the domain name, enters the domain name into the DNS database (as discussed below), and collects a small fee from you for its services. Prior to 1999, a single registrar, Network Solutions, had a monopoly on domain name registration for [com](http://com), [net](http://net), and [org](http://org) domains. But now there are many registrars competing for customers, and the Internet Corporation for Assigned Names and Numbers (ICANN) accredits the various registrars. A complete list of accredited registrars is available at <http://www.internic.net>.

When you register the domain name [networkutopia.com](http://networkutopia.com) with some registrar, you also need to provide the registrar with the names and IP addresses of your primary and secondary authoritative DNS servers. Suppose the names and IP addresses are [dns1.networkutopia.com](http://dns1.networkutopia.com), [dns2.networkutopia.com](http://dns2.networkutopia.com), 212.2.212.1, and 212.212.212.2. For each of these two authoritative DNS servers, the registrar would then make sure that a Type NS and a Type A record are entered into the TLD com servers. Specifically, for the primary authoritative server for [networkutopia.com](http://networkutopia.com), the registrar would insert the following two resource records into the DNS system:

([networkutopia.com](http://networkutopia.com), [dns1.networkutopia.com](http://dns1.networkutopia.com), NS)

([dns1.networkutopia.com](http://dns1.networkutopia.com), 212.212.212.1, A)

You'll also have to make sure that the Type A resource record for your Web server [www.networkutopia.com](http://www.networkutopia.com) and the Type MX resource record for your mail server [mail.networkutopia.com](http://mail.networkutopia.com) are entered into your authoritative DNS

## FOCUS ON SECURITY

### DNS VULNERABILITIES

We have seen that DNS is a critical component of the Internet infrastructure, with many important services—including the Web and e-mail—simply incapable of functioning without it. We therefore naturally ask, how can DNS be attacked? Is DNS a sitting duck, waiting to be knocked out of service, while taking most Internet applications down with it?

The first type of attack that comes to mind is a DDoS bandwidth-flooding attack (see [Section 1.6](#)) against DNS servers. For example, an attacker could attempt to send to each DNS root server a deluge of packets, so many that the majority of legitimate DNS queries never get answered. Such a large-scale DDoS attack against DNS root servers actually took place on October 21, 2002. In this attack, the attackers leveraged a botnet to send truck loads of ICMP ping messages to each of the 13 DNS root IP addresses. (ICMP messages are discussed in

**Section 5.6.** For now, it suffices to know that ICMP packets are special types of IP datagrams.) Fortunately, this large-scale attack caused minimal damage, having little or no impact on users' Internet experience. The attackers did succeed at directing a deluge of packets at the root servers. But many of the DNS root servers were protected by packet filters, configured to always block all ICMP ping messages directed at the root servers. These protected servers were thus spared and functioned as normal. Furthermore, most local DNS servers cache the IP addresses of top-level-domain servers, allowing the query process to often bypass the DNS root servers.

A potentially more effective DDoS attack against DNS would be send a deluge of DNS queries to top-level-domain servers, for example, to all the top-level-domain servers that handle the .com domain. It would be harder to filter DNS queries directed to DNS servers; and top-level-domain servers are not as easily bypassed as are root servers. But the severity of such an attack would be partially mitigated by caching in local DNS servers.

DNS could potentially be attacked in other ways. In a man-in-the-middle attack, the attacker intercepts queries from hosts and returns bogus replies. In the DNS poisoning attack, the attacker sends bogus replies to a DNS server, tricking the server into accepting bogus records into its cache. Either of these attacks could be used, for example, to redirect an unsuspecting Web user to the attacker's Web site. These attacks, however, are difficult to implement, as they require intercepting packets or throttling servers [\[Skoudis 2006\]](#).

In summary, DNS has demonstrated itself to be surprisingly robust against attacks. To date, there hasn't been an attack that has successfully impeded the DNS service.

servers. (Until recently, the contents of each DNS server were configured statically, for example, from a configuration file created by a system manager. More recently, an UPDATE option has been added to the DNS protocol to allow data to be dynamically added or deleted from the database via DNS messages. [\[RFC 2136\]](#) and [\[RFC 3007\]](#) specify DNS dynamic updates.)

Once all of these steps are completed, people will be able to visit your Web site and send e-mail to the employees at your company. Let's conclude our discussion of DNS by verifying that this statement is true. This verification also helps to solidify what we have learned about DNS. Suppose Alice in Australia wants to view the Web page [www.networkutopia.com](http://www.networkutopia.com). As discussed earlier, her host will first send a DNS query to her local DNS server. The local DNS server will then contact a TLD `.com` server. (The local DNS server will also have to contact a root DNS server if the address of a TLD `.com` server is not cached.) This TLD server contains the Type NS and Type A resource records listed above, because the registrar had these resource records inserted into all of the TLD com servers. The TLD com server sends a reply to Alice's local DNS server, with the reply containing the two resource records. The local DNS server then sends a DNS query to `212.212.212.1`, asking for the Type A record corresponding to [www.networkutopia.com](http://www.networkutopia.com). This record provides the IP address of the desired Web server, say, `212.212.71.4`, which the local DNS server passes back to Alice's host. Alice's browser can now

initiate a TCP connection to the host `212.212.71.4` and send an HTTP request over the connection.

Whew! There's a lot more going on than what meets the eye when one surfs the Web!

## 2.5 Peer-to-Peer File Distribution

The applications described in this chapter thus far—including the Web, e-mail, and DNS—all employ client-server architectures with significant reliance on always-on infrastructure servers. Recall from [Section 2.1.1](#) that with a P2P architecture, there is minimal (or no) reliance on always-on infrastructure servers. Instead, pairs of intermittently connected hosts, called peers, communicate directly with each other. The peers are not owned by a service provider, but are instead desktops and laptops controlled by users.

In this section we consider a very natural P2P application, namely, distributing a large file from a single server to a large number of hosts (called peers). The file might be a new version of the Linux operating system, a software patch for an existing operating system or application, an MP3 music file, or an MPEG video file. In client-server file distribution, the server must send a copy of the file to each of the peers—placing an enormous burden on the server and consuming a large amount of server bandwidth. In P2P file distribution, each peer can redistribute any portion of the file it has received to any other peers, thereby assisting the server in the distribution process. As of 2016, the most popular P2P file distribution protocol is BitTorrent. Originally developed by Bram Cohen, there are now many different independent BitTorrent clients conforming to the BitTorrent protocol, just as there are a number of Web browser clients that conform to the HTTP protocol. In this subsection, we first examine the self-scalability of P2P architectures in the context of file distribution. We then describe BitTorrent in some detail, highlighting its most important characteristics and features.

### *Scalability of P2P Architectures*

To compare client-server architectures with peer-to-peer architectures, and illustrate the inherent self-scalability of P2P, we now consider a simple quantitative model for distributing a file to a fixed set of peers for both architecture types. As shown in [Figure 2.22](#), the server and the peers are connected to the Internet with access links. Denote the upload rate of the server’s access link by  $u_s$ , the upload rate of the  $i$ th peer’s access link by  $u_i$ , and the download rate of the  $i$ th peer’s access link by  $d_i$ . Also denote the size of the file to be distributed (in bits) by  $F$  and the number of peers that want to obtain a copy of the file by  $N$ . The **distribution time** is the time it takes to get

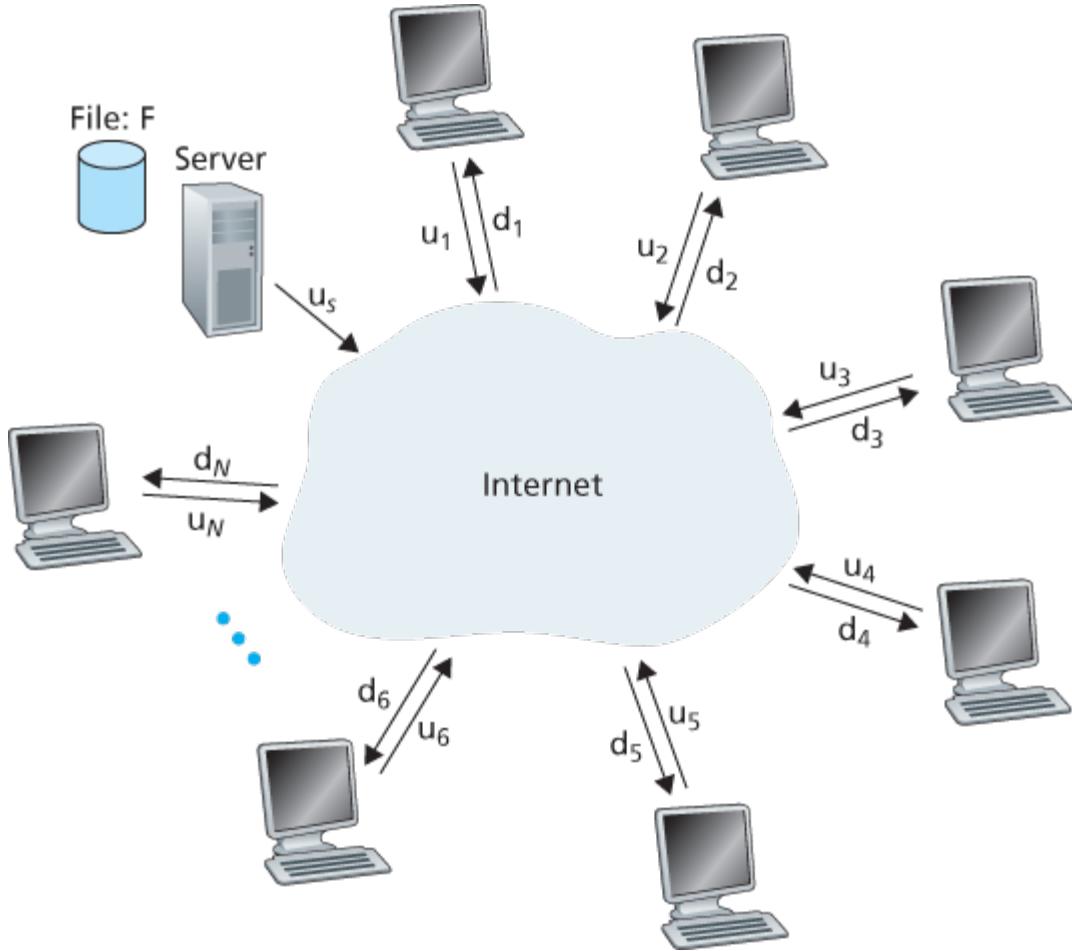


Figure 2.22 An illustrative file distribution problem

a copy of the file to all  $N$  peers. In our analysis of the distribution time below, for both client-server and P2P architectures, we make the simplifying (and generally accurate [Akella 2003]) assumption that the Internet core has abundant bandwidth, implying that all of the bottlenecks are in access networks. We also suppose that the server and clients are not participating in any other network applications, so that all of their upload and download access bandwidth can be fully devoted to distributing this file.

Let's first determine the distribution time for the client-server architecture, which we denote by  $D_{cs}$ . In the client-server architecture, none of the peers aids in distributing the file. We make the following observations:

- The server must transmit one copy of the file to each of the  $N$  peers. Thus the server must transmit  $NF$  bits. Since the server's upload rate is  $u_s$ , the time to distribute the file must be at least  $NF/u_s$ .
- Let  $d_{\min}$  denote the download rate of the peer with the lowest download rate, that is,  $d_{\min} = \min\{d_1, d_2, \dots, d_N\}$ . The peer with the lowest download rate cannot obtain all  $F$  bits of the file in less than  $F/d_{\min}$  seconds. Thus the minimum distribution time is at least  $F/d_{\min}$ .

Putting these two observations together, we obtain

$$D_{cs} \geq \max\{NF/u_s, F/d_{\min}\}.$$

This provides a lower bound on the minimum distribution time for the client-server architecture. In the homework problems you will be asked to show that the server can schedule its transmissions so that the lower bound is actually achieved. So let's take this lower bound provided above as the actual distribution time, that is,

$$D_{\text{CS}} = \max\{NFu_s, Fd_{\min}\} \quad (2.1)$$

We see from [Equation 2.1](#) that for  $N$  large enough, the client-server distribution time is given by  $NF/u_s$ . Thus, the distribution time increases linearly with the number of peers  $N$ . So, for example, if the number of peers from one week to the next increases a thousand-fold from a thousand to a million, the time required to distribute the file to all peers increases by 1,000.

Let's now go through a similar analysis for the P2P architecture, where each peer can assist the server in distributing the file. In particular, when a peer receives some file data, it can use its own upload capacity to redistribute the data to other peers. Calculating the distribution time for the P2P architecture is somewhat more complicated than for the client-server architecture, since the distribution time depends on how each peer distributes portions of the file to the other peers. Nevertheless, a simple expression for the minimal distribution time can be obtained [\[Kumar 2006\]](#). To this end, we first make the following observations:

- At the beginning of the distribution, only the server has the file. To get this file into the community of peers, the server must send each bit of the file at least once into its access link. Thus, the minimum distribution time is at least  $F/u_s$ . (Unlike the client-server scheme, a bit sent once by the server may not have to be sent by the server again, as the peers may redistribute the bit among themselves.)
- As with the client-server architecture, the peer with the lowest download rate cannot obtain all  $F$  bits of the file in less than  $F/d_{\min}$  seconds. Thus the minimum distribution time is at least  $F/d_{\min}$ .
- Finally, observe that the total upload capacity of the system as a whole is equal to the upload rate of the server plus the upload rates of each of the individual peers, that is,  $u_{\text{total}} = u_s + u_1 + \dots + u_N$ . The system must deliver (upload)  $F$  bits to each of the  $N$  peers, thus delivering a total of  $NF$  bits. This cannot be done at a rate faster than  $u_{\text{total}}$ . Thus, the minimum distribution time is also at least  $NF/(u_s + u_1 + \dots + u_N)$ .

Putting these three observations together, we obtain the minimum distribution time for P2P, denoted by  $D_{\text{P2P}}$ .

$$D_{\text{P2P}} \geq \max\{Fu_s, Fd_{\min}, NFu_s + \sum_{i=1}^N u_i\} \quad (2.2)$$

[Equation 2.2](#) provides a lower bound for the minimum distribution time for the P2P architecture. It turns out that if we imagine that each peer can redistribute a bit as soon as it receives the bit, then there is a

redistribution scheme that actually achieves this lower bound [Kumar 2006]. (We will prove a special case of this result in the homework.) In reality, where chunks of the file are redistributed rather than individual bits, [Equation 2.2](#) serves as a good approximation of the actual minimum distribution time. Thus, let's take the lower bound provided by [Equation 2.2](#) as the actual minimum distribution time, that is,

$$DP2P = \max\{Fus, F_{\min}, NFus + \sum_{i=1}^N u_i\} \quad (2.3)$$

[Figure 2.23](#) compares the minimum distribution time for the client-server and P2P architectures assuming that all peers have the same upload rate  $u$ . In [Figure 2.23](#), we have set  $F/u = 1$  hour,  $us = 10u$ , and  $d_{\min} \geq us$ . Thus, a peer can transmit the entire file in one hour, the server transmission rate is 10 times the peer upload rate,



**Figure 2.23 Distribution time for P2P and client-server architectures**

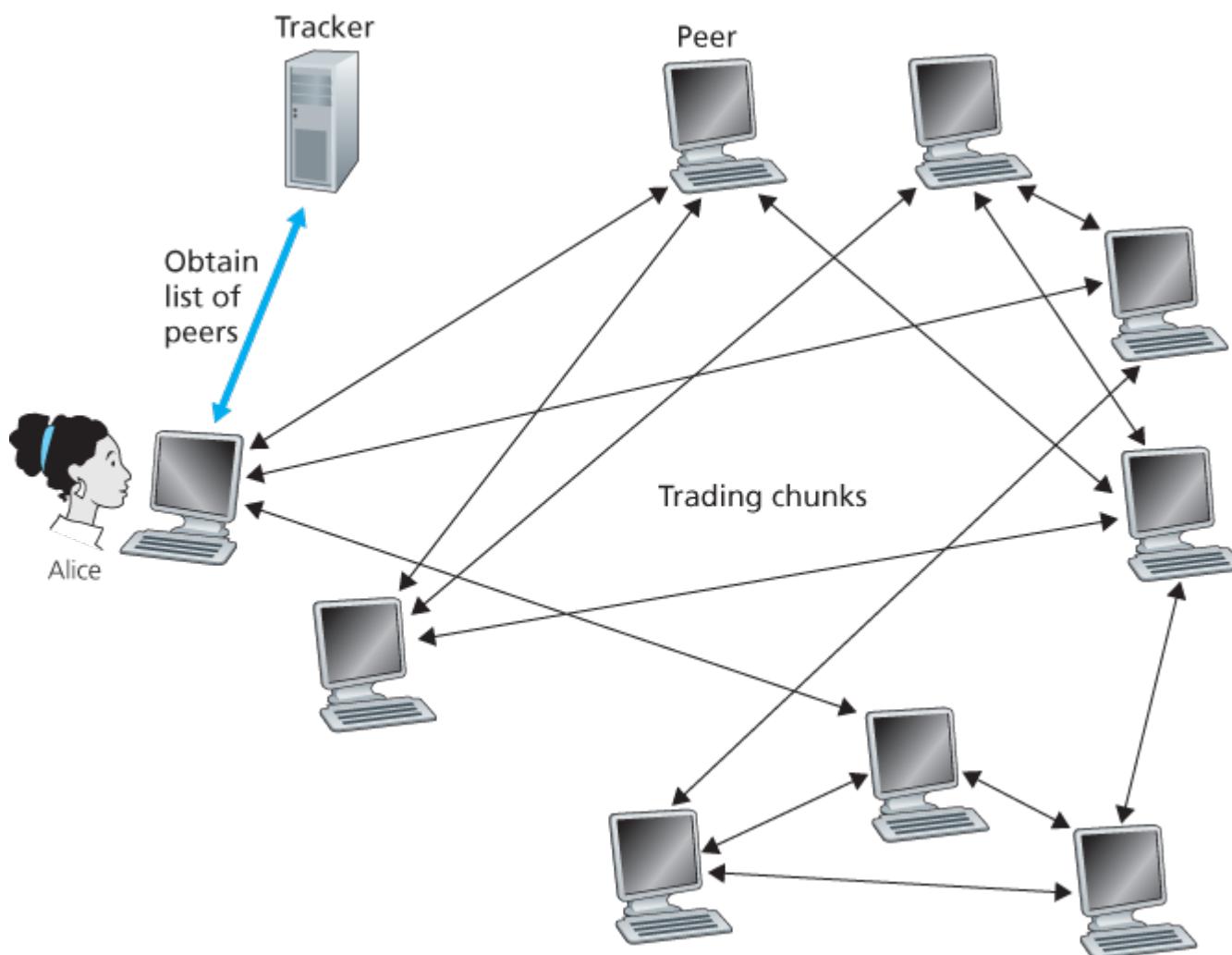
and (for simplicity) the peer download rates are set large enough so as not to have an effect. We see from [Figure 2.23](#) that for the client-server architecture, the distribution time increases linearly and without bound as the number of peers increases. However, for the P2P architecture, the minimal distribution time is not only always less than the distribution time of the client-server architecture; it is also less than one hour for *any* number of peers  $N$ . Thus, applications with the P2P architecture can be self-scaling. This scalability is a direct consequence of peers being redistributors as well as consumers of bits.

### BitTorrent

BitTorrent is a popular P2P protocol for file distribution [[Chao 2011](#)]. In BitTorrent lingo, the collection of

all peers participating in the distribution of a particular file is called a *torrent*. Peers in a torrent download equal-size *chunks* of the file from one another, with a typical chunk size of 256 KBytes. When a peer first joins a torrent, it has no chunks. Over time it accumulates more and more chunks. While it downloads chunks it also uploads chunks to other peers. Once a peer has acquired the entire file, it may (selfishly) leave the torrent, or (altruistically) remain in the torrent and continue to upload chunks to other peers. Also, any peer may leave the torrent at any time with only a subset of chunks, and later rejoin the torrent.

Let's now take a closer look at how BitTorrent operates. Since BitTorrent is a rather complicated protocol and system, we'll only describe its most important mechanisms, sweeping some of the details under the rug; this will allow us to see the forest through the trees. Each torrent has an infrastructure node called a *tracker*.



**Figure 2.24 File distribution with BitTorrent**

When a peer joins a torrent, it registers itself with the tracker and periodically informs the tracker that it is still in the torrent. In this manner, the tracker keeps track of the peers that are participating in the torrent. A given torrent may have fewer than ten or more than a thousand peers participating at any instant of time.

As shown in [Figure 2.24](#), when a new peer, Alice, joins the torrent, the tracker randomly selects a subset of peers (for concreteness, say 50) from the set of participating peers, and sends the IP addresses of these 50 peers to Alice. Possessing this list of peers, Alice attempts to establish concurrent TCP connections with all the peers on this list. Let's call all the peers with which Alice succeeds in establishing a TCP connection "neighboring peers." (In [Figure 2.24](#), Alice is shown to have only three neighboring peers. Normally, she would have many more.) As time evolves, some of these peers may leave and other peers (outside the initial 50) may attempt to establish TCP connections with Alice. So a peer's neighboring peers will fluctuate over time.

At any given time, each peer will have a subset of chunks from the file, with different peers having different subsets. Periodically, Alice will ask each of her neighboring peers (over the TCP connections) for the list of the chunks they have. If Alice has  $L$  different neighbors, she will obtain  $L$  lists of chunks. With this knowledge, Alice will issue requests (again over the TCP connections) for chunks she currently does not have.

So at any given instant of time, Alice will have a subset of chunks and will know which chunks her neighbors have. With this information, Alice will have two important decisions to make. First, which chunks should she request first from her neighbors? And second, to which of her neighbors should she send requested chunks? In deciding which chunks to request, Alice uses a technique called [rarest first](#). The idea is to determine, from among the chunks she does not have, the chunks that are the rarest among her neighbors (that is, the chunks that have the fewest repeated copies among her neighbors) and then request those rarest chunks first. In this manner, the rarest chunks get more quickly redistributed, aiming to (roughly) equalize the numbers of copies of each chunk in the torrent.

To determine which requests she responds to, BitTorrent uses a clever trading algorithm. The basic idea is that Alice gives priority to the neighbors that are currently supplying her data *at the highest rate*. Specifically, for each of her neighbors, Alice continually measures the rate at which she receives bits and determines the four peers that are feeding her bits at the highest rate. She then reciprocates by sending chunks to these same four peers. Every 10 seconds, she recalculates the rates and possibly modifies the set of four peers. In BitTorrent lingo, these four peers are said to be [unchoked](#). Importantly, every 30 seconds, she also picks one additional neighbor at random and sends it chunks. Let's call the randomly chosen peer Bob. In BitTorrent lingo, Bob is said to be [optimistically unchoked](#). Because Alice is sending data to Bob, she may become one of Bob's top four uploaders, in which case Bob would start to send data to Alice. If the rate at which Bob sends data to Alice is high enough, Bob could then, in turn, become one of Alice's top four uploaders. In other words, every 30 seconds, Alice will randomly choose a new trading partner and initiate trading with that partner. If the two peers are satisfied with the trading, they will put each other in their top four lists and continue trading with each other until one of the peers finds a better partner. The effect is that peers capable of uploading at compatible rates tend to find each other. The random neighbor selection also allows new peers to get chunks, so that they can have something to trade. All other neighboring peers besides these five peers

(four “top” peers and one probing peer) are “choked,” that is, they do not receive any chunks from Alice. BitTorrent has a number of interesting mechanisms that are not discussed here, including pieces (mini-chunks), pipelining, random first selection, endgame mode, and anti-snubbing [Cohen 2003].

The incentive mechanism for trading just described is often referred to as tit-for-tat [Cohen 2003]. It has been shown that this incentive scheme can be circumvented [[Liogkas 2006](#); [Locher 2006](#); [Piatek 2007](#)]. Nevertheless, the BitTorrent ecosystem is wildly successful, with millions of simultaneous peers actively sharing files in hundreds of thousands of torrents. If BitTorrent had been designed without tit-for-tat (or a variant), but otherwise exactly the same, BitTorrent would likely not even exist now, as the majority of the users would have been freeriders [[Saroiu 2002](#)].

We close our discussion on P2P by briefly mentioning another application of P2P, namely, Distributed Hash Table (DHT). A distributed hash table is a simple database, with the database records being distributed over the peers in a P2P system. DHTs have been widely implemented (e.g., in BitTorrent) and have been the subject of extensive research. An overview is provided in a Video Note in the companion website.



VideoNote

Walking though distributed hash tables

## 2.6 Video Streaming and Content Distribution Networks

Streaming prerecorded video now accounts for the majority of the traffic in residential ISPs in North America. In particular, the Netflix and YouTube services alone consumed a whopping 37% and 16%, respectively, of residential ISP traffic in 2015 [[Sandvine 2015](#)]. In this section we will provide an overview of how popular video streaming services are implemented in today's Internet. We will see they are implemented using application-level protocols and servers that function in some ways like a cache. In [Chapter 9](#), devoted to multimedia networking, we will further examine Internet video as well as other Internet multimedia services.

### 2.6.1 Internet Video

In streaming stored video applications, the underlying medium is prerecorded video, such as a movie, a television show, a prerecorded sporting event, or a prerecorded user-generated video (such as those commonly seen on YouTube). These prerecorded videos are placed on servers, and users send requests to the servers to view the videos *on demand*. Many Internet companies today provide streaming video, including, Netflix, YouTube (Google), Amazon, and Youku.

But before launching into a discussion of video streaming, we should first get a quick feel for the video medium itself. A video is a sequence of images, typically being displayed at a constant rate, for example, at 24 or 30 images per second. An uncompressed, digitally encoded image consists of an array of pixels, with each pixel encoded into a number of bits to represent luminance and color. An important characteristic of video is that it can be compressed, thereby trading off video quality with bit rate. Today's off-the-shelf compression algorithms can compress a video to essentially any bit rate desired. Of course, the higher the bit rate, the better the image quality and the better the overall user viewing experience.

From a networking perspective, perhaps the most salient characteristic of video is its high bit rate. Compressed Internet video typically ranges from 100 kbps for low-quality video to over 3 Mbps for streaming high-definition movies; 4K streaming envisions a bitrate of more than 10 Mbps. This can translate to huge amount of traffic and storage, particularly for high-end video. For example, a single 2 Mbps video with a duration of 67 minutes will consume 1 gigabyte of storage and traffic. By far, the most important performance measure for streaming video is average end-to-end throughput. In order to provide continuous playout, the network must provide an average throughput to the streaming application that is at least as large as the bit rate of the compressed video.

We can also use compression to create multiple versions of the same video, each at a different quality level. For example, we can use compression to create, say, three versions of the same video, at rates of 300 kbps, 1 Mbps, and 3 Mbps. Users can then decide which version they want to watch as a function of their current available bandwidth. Users with high-speed Internet connections might choose the 3 Mbps version; users watching the video over 3G with a smartphone might choose the 300 kbps version.

## 2.6.2 HTTP Streaming and DASH

In HTTP streaming, the video is simply stored at an HTTP server as an ordinary file with a specific URL. When a user wants to see the video, the client establishes a TCP connection with the server and issues an HTTP `GET` request for that URL. The server then sends the video file, within an HTTP response message, as quickly as the underlying network protocols and traffic conditions will allow. On the client side, the bytes are collected in a client application buffer. Once the number of bytes in this buffer exceeds a predetermined threshold, the client application begins playback—specifically, the streaming video application periodically grabs video frames from the client application buffer, decompresses the frames, and displays them on the user's screen. Thus, the video streaming application is displaying video as it is receiving and buffering frames corresponding to latter parts of the video.

Although HTTP streaming, as described in the previous paragraph, has been extensively deployed in practice (for example, by YouTube since its inception), it has a major shortcoming: All clients receive the same encoding of the video, despite the large variations in the amount of bandwidth available to a client, both across different clients and also over time for the same client. This has led to the development of a new type of HTTP-based streaming, often referred to as **Dynamic Adaptive Streaming over HTTP (DASH)**. In DASH, the video is encoded into several different versions, with each version having a different bit rate and, correspondingly, a different quality level. The client dynamically requests chunks of video segments of a few seconds in length. When the amount of available bandwidth is high, the client naturally selects chunks from a high-rate version; and when the available bandwidth is low, it naturally selects from a low-rate version. The client selects different chunks one at a time with HTTP GET request messages [\[Akhshabi 2011\]](#).

DASH allows clients with different Internet access rates to stream in video at different encoding rates. Clients with low-speed 3G connections can receive a low bit-rate (and low-quality) version, and clients with fiber connections can receive a high-quality version. DASH also allows a client to adapt to the available bandwidth if the available end-to-end bandwidth changes during the session. This feature is particularly important for mobile users, who typically see their bandwidth availability fluctuate as they move with respect to the base stations.

With DASH, each video version is stored in the HTTP server, each with a different URL. The HTTP

server also has a **manifest file**, which provides a URL for each version along with its bit rate. The client first requests the manifest file and learns about the various versions. The client then selects one chunk at a time by specifying a URL and a byte range in an HTTP GET request message for each chunk. While downloading chunks, the client also measures the received bandwidth and runs a rate determination algorithm to select the chunk to request next. Naturally, if the client has a lot of video buffered and if the measured receive bandwidth is high, it will choose a chunk from a high-bitrate version. And naturally if the client has little video buffered and the measured received bandwidth is low, it will choose a chunk from a low-bitrate version. DASH therefore allows the client to freely switch among different quality levels.

### 2.6.3 Content Distribution Networks

Today, many Internet video companies are distributing on-demand multi-Mbps streams to millions of users on a daily basis. YouTube, for example, with a library of hundreds of millions of videos, distributes hundreds of millions of video streams to users around the world every day. Streaming all this traffic to locations all over the world while providing continuous playout and high interactivity is clearly a challenging task.

For an Internet video company, perhaps the most straightforward approach to providing streaming video service is to build a single massive data center, store all of its videos in the data center, and stream the videos directly from the data center to clients worldwide. But there are three major problems with this approach. First, if the client is far from the data center, server-to-client packets will cross many communication links and likely pass through many ISPs, with some of the ISPs possibly located on different continents. If one of these links provides a throughput that is less than the video consumption rate, the end-to-end throughput will also be below the consumption rate, resulting in annoying freezing delays for the user. (Recall from [Chapter 1](#) that the end-to-end throughput of a stream is governed by the throughput at the bottleneck link.) The likelihood of this happening increases as the number of links in the end-to-end path increases. A second drawback is that a popular video will likely be sent many times over the same communication links. Not only does this waste network bandwidth, but the Internet video company itself will be paying its provider ISP (connected to the data center) for sending the *same* bytes into the Internet over and over again. A third problem with this solution is that a single data center represents a single point of failure—if the data center or its links to the Internet goes down, it would not be able to distribute *any* video streams.

In order to meet the challenge of distributing massive amounts of video data to users distributed around the world, almost all major video-streaming companies make use of **Content Distribution Networks (CDNs)**. A CDN manages servers in multiple geographically distributed locations, stores copies of the videos (and other types of Web content, including documents, images, and audio) in its servers, and attempts to direct each user request to a CDN location that will provide the best user experience. The

CDN may be a **private CDN**, that is, owned by the content provider itself; for example, Google's CDN distributes YouTube videos and other types of content. The CDN may alternatively be a **third-party CDN** that distributes content on behalf of multiple content providers; Akamai, Limelight and Level-3 all operate third-party CDNs. A very readable overview of modern CDNs is [[Leighton 2009; Nygren 2010](#)].

CDNs typically adopt one of two different server placement philosophies [[Huang 2008](#)]:

- **Enter Deep.** One philosophy, pioneered by Akamai, is to *enter deep* into the access networks of Internet Service Providers, by deploying server clusters in access ISPs all over the world. (Access networks are described in [Section 1.3](#).) Akamai takes this approach with clusters in approximately 1,700 locations. The goal is to get close to end users, thereby improving user-perceived delay and throughput by decreasing the number of links and routers between the end user and the CDN server from which it receives content. Because of this highly distributed design, the task of maintaining and managing the clusters becomes challenging.
- **Bring Home.** A second design philosophy, taken by Limelight and many other CDN companies, is to *bring the ISPs home* by building large clusters at a smaller number (for example, tens) of sites. Instead of getting inside the access ISPs, these CDNs typically place their clusters in Internet Exchange Points (IXPs) (see [Section 1.3](#)). Compared with the enter-deep design philosophy, the bring-home design typically results in lower maintenance and management overhead, possibly at the expense of higher delay and lower throughput to end users.

Once its clusters are in place, the CDN replicates content across its clusters. The CDN may not want to place a copy of every video in each cluster, since some videos are rarely viewed or are only popular in some countries. In fact, many CDNs do not push videos to their clusters but instead use a simple pull strategy: If a client requests a video from a cluster that is not storing the video, then the cluster retrieves the video (from a central repository or from another cluster) and stores a copy locally while streaming the video to the client at the same time. Similar Web caching (see [Section 2.2.5](#)), when a cluster's storage becomes full, it removes videos that are not frequently requested.

### *CDN Operation*

Having identified the two major approaches toward deploying a CDN, let's now dive down into the nuts and bolts of how a CDN operates. When a browser in a user's

#### CASE STUDY

#### GOOGLE'S NETWORK INFRASTRUCTURE

To support its vast array of cloud services—including search, Gmail, calendar, YouTube video, maps, documents, and social networks—Google has deployed an extensive private network and CDN infrastructure. Google's CDN infrastructure has three tiers of server clusters:

- Fourteen “mega data centers,” with eight in North America, four in Europe, and two in Asia [[Google Locations 2016](#)], with each data center having on the order of 100,000 servers. These mega data centers are responsible for serving dynamic (and often personalized) content, including search results and Gmail messages.
- An estimated 50 clusters in IXPs scattered throughout the world, with each cluster consisting on the order of 100–500 servers [[Adhikari 2011a](#)]. These clusters are responsible for serving static content, including YouTube videos [[Adhikari 2011a](#)].
- Many hundreds of “enter-deep” clusters located within an access ISP. Here a cluster typically consists of tens of servers within a single rack. These enter-deep servers perform TCP splitting (see [Section 3.7](#)) and serve static content [[Chen 2011](#)], including the static portions of Web pages that embody search results.

All of these data centers and cluster locations are networked together with Google’s own private network. When a user makes a search query, often the query is first sent over the local ISP to a nearby enter-deep cache, from where the static content is retrieved; while providing the static content to the client, the nearby cache also forwards the query over Google’s private network to one of the mega data centers, from where the personalized search results are retrieved. For a YouTube video, the video itself may come from one of the bring-home caches, whereas portions of the Web page surrounding the video may come from the nearby enter-deep cache, and the advertisements surrounding the video come from the data centers. In summary, except for the local ISPs, the Google cloud services are largely provided by a network infrastructure that is independent of the public Internet.

host is instructed to retrieve a specific video (identified by a URL), the CDN must intercept the request so that it can (1) determine a suitable CDN server cluster for that client at that time, and (2) redirect the client’s request to a server in that cluster. We’ll shortly discuss how a CDN can determine a suitable cluster. But first let’s examine the mechanics behind intercepting and redirecting a request.

Most CDNs take advantage of DNS to intercept and redirect requests; an interesting discussion of such a use of the DNS is [[Vixie 2009](#)]. Let’s consider a simple example to illustrate how the DNS is typically involved. Suppose a content provider, NetCinema, employs the third-party CDN company, KingCDN, to distribute its videos to its customers. On the NetCinema Web pages, each of its videos is assigned a URL that includes the string “video” and a unique identifier for the video itself; for example, Transformers 7 might be assigned <http://video.netcinema.com/6Y7B23V>. Six steps then occur, as shown in [Figure 2.25](#):

1. The user visits the Web page at NetCinema.
2. When the user clicks on the link <http://video.netcinema.com/6Y7B23V>, the user’s host sends a DNS query for [video.netcinema.com](http://video.netcinema.com).

- The user's Local DNS Server (LDNS) relays the DNS query to an authoritative DNS server for NetCinema, which observes the string "video" in the hostname **video.netcinema.com**. To "hand over" the DNS query to KingCDN, instead of returning an IP address, the NetCinema authoritative DNS server returns to the LDNS a hostname in the KingCDN's domain, for example, **a1105.kingcdn.com**.
- From this point on, the DNS query enters into KingCDN's private DNS infrastructure. The user's LDNS then sends a second query, now for **a1105.kingcdn.com**, and KingCDN's DNS system eventually returns the IP addresses of a KingCDN content server to the LDNS. It is thus here, within the KingCDN's DNS system, that the CDN server from which the client will receive its content is specified.

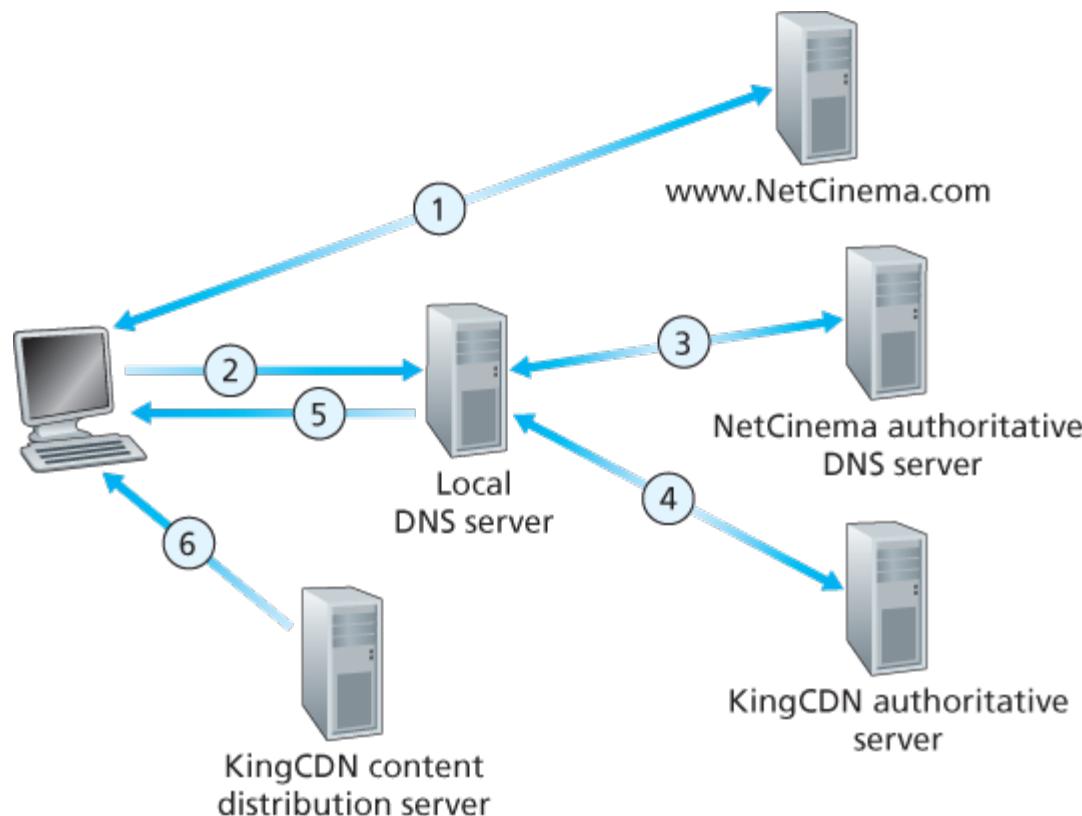


Figure 2.25 DNS redirects a user's request to a CDN server

- The LDNS forwards the IP address of the content-serving CDN node to the user's host.
- Once the client receives the IP address for a KingCDN content server, it establishes a direct TCP connection with the server at that IP address and issues an HTTP GET request for the video. If DASH is used, the server will first send to the client a manifest file with a list of URLs, one for each version of the video, and the client will dynamically select chunks from the different versions.

#### *Cluster Selection Strategies*

At the core of any CDN deployment is a **cluster selection strategy**, that is, a mechanism for dynamically directing clients to a server cluster or a data center within the CDN. As we just saw, the

CDN learns the IP address of the client's LDNS server via the client's DNS lookup. After learning this IP address, the CDN needs to select an appropriate cluster based on this IP address. CDNs generally employ proprietary cluster selection strategies. We now briefly survey a few approaches, each of which has its own advantages and disadvantages.

One simple strategy is to assign the client to the cluster that is **geographically closest**. Using commercial geo-location databases (such as Quova [[Quova 2016](#)] and Max-Mind [[MaxMind 2016](#)]), each LDNS IP address is mapped to a geographic location. When a DNS request is received from a particular LDNS, the CDN chooses the geographically closest cluster, that is, the cluster that is the fewest kilometers from the LDNS "as the bird flies." Such a solution can work reasonably well for a large fraction of the clients [[Agarwal 2009](#)]. However, for some clients, the solution may perform poorly, since the geographically closest cluster may not be the closest cluster in terms of the length or number of hops of the network path. Furthermore, a problem inherent with all DNS-based approaches is that some end-users are configured to use remotely located LDNSs [[Shaikh 2001](#); Mao 2002], in which case the LDNS location may be far from the client's location. Moreover, this simple strategy ignores the variation in delay and available bandwidth over time of Internet paths, always assigning the same cluster to a particular client.

In order to determine the best cluster for a client based on the *current* traffic conditions, CDNs can instead perform periodic **real-time measurements** of delay and loss performance between their clusters and clients. For instance, a CDN can have each of its clusters periodically send probes (for example, ping messages or DNS queries) to all of the LDNSs around the world. One drawback of this approach is that many LDNSs are configured to not respond to such probes.

#### 2.6.4 Case Studies: Netflix, YouTube, and Kankan

We conclude our discussion of streaming stored video by taking a look at three highly successful large-scale deployments: Netflix, YouTube, and Kankan. We'll see that each of these systems take a very different approach, yet employ many of the underlying principles discussed in this section.

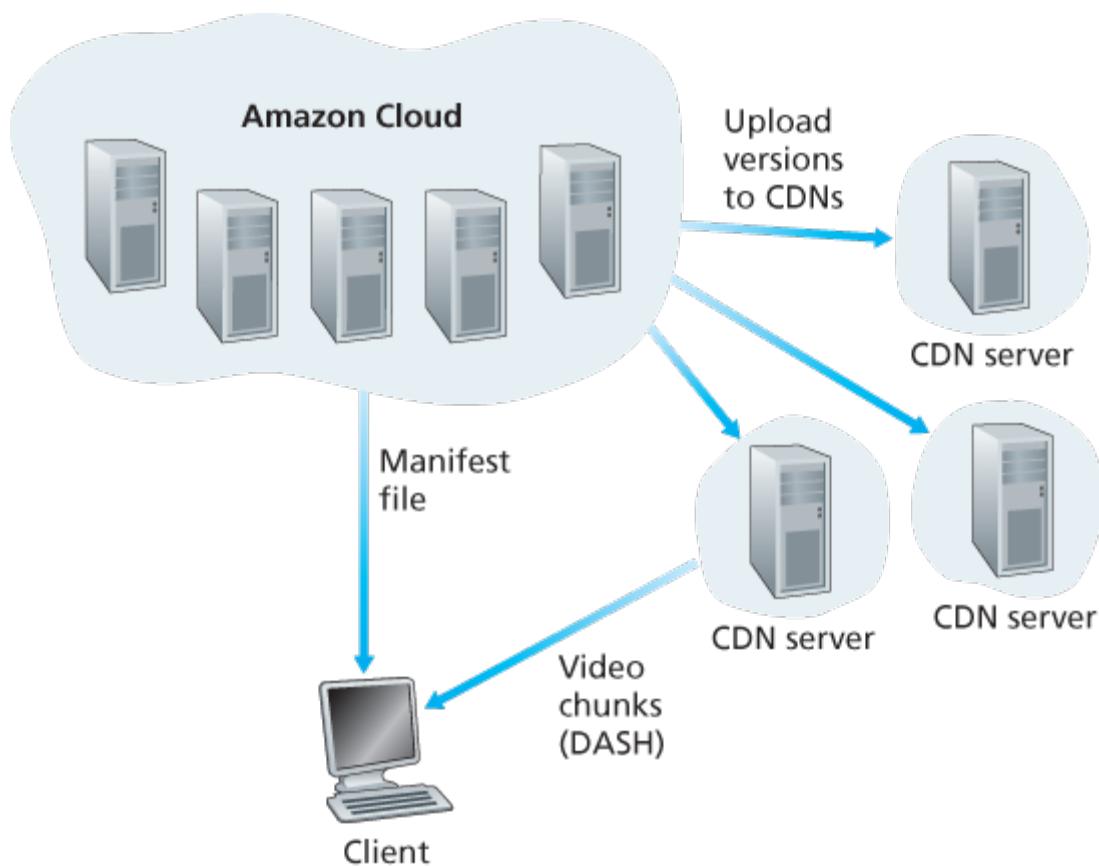
##### *Netflix*

Generating 37% of the downstream traffic in residential ISPs in North America in 2015, Netflix has become the leading service provider for online movies and TV series in the United States [[Sandvine 2015](#)]. As we discuss below, Netflix video distribution has two major components: the Amazon cloud and its own private CDN infrastructure.

Netflix has a Web site that handles numerous functions, including user registration and login, billing, movie catalogue for browsing and searching, and a movie recommendation system. As shown in [Figure](#)

**2.26**, this Web site (and its associated backend databases) run entirely on Amazon servers in the Amazon cloud. Additionally, the Amazon cloud handles the following critical functions:

- **Content ingestion.** Before Netflix can distribute a movie to its customers, it must first ingest and process the movie. Netflix receives studio master versions of movies and uploads them to hosts in the Amazon cloud.
- **Content processing.** The machines in the Amazon cloud create many different formats for each movie, suitable for a diverse array of client video players running on desktop computers, smartphones, and game consoles connected to televisions. A different version is created for each of these formats and at multiple bit rates, allowing for adaptive streaming over HTTP using DASH.
- **Uploading versions to its CDN.** Once all of the versions of a movie have been created, the hosts in the Amazon cloud upload the versions to its CDN.



**Figure 2.26** Netflix video streaming platform

When Netflix first rolled out its video streaming service in 2007, it employed three third-party CDN companies to distribute its video content. Netflix has since created its own private CDN, from which it now streams all of its videos. (Netflix still uses Akamai to distribute its Web pages, however.) To create its own CDN, Netflix has installed server racks both in IXPs and within residential ISPs themselves.

Netflix currently has server racks in over 50 IXP locations; see [\[Netflix Open Connect 2016\]](#) for a current list of IXPs housing Netflix racks. There are also hundreds of ISP locations housing Netflix racks; also see [\[Netflix Open Connect 2016\]](#), where Netflix provides to potential ISP partners instructions about installing a (free) Netflix rack for their networks. Each server in the rack has several 10 Gbps

Ethernet ports and over 100 terabytes of storage. The number of servers in a rack varies: IXPs installations often have tens of servers and contain the entire Netflix streaming video library, including multiple versions of the videos to support DASH; local IXPs may only have one server and contain only the most popular videos. Netflix does not use pull-caching ([Section 2.2.5](#)) to populate its CDN servers in the IXPs and ISPs. Instead, Netflix distributes by pushing the videos to its CDN servers during off-peak hours. For those locations that cannot hold the entire library, Netflix pushes only the most popular videos, which are determined on a day-to-day basis. The Netflix CDN design is described in some detail in the YouTube videos [[Netflix Video 1](#)] and [[Netflix Video 2](#)].

Having described the components of the Netflix architecture, let's take a closer look at the interaction between the client and the various servers that are involved in movie delivery. As indicated earlier, the Web pages for browsing the Netflix video library are served from servers in the Amazon cloud. When a user selects a movie to play, the Netflix software, running in the Amazon cloud, first determines which of its CDN servers have copies of the movie. Among the servers that have the movie, the software then determines the “best” server for that client request. If the client is using a residential ISP that has a Netflix CDN server rack installed in that ISP, and this rack has a copy of the requested movie, then a server in this rack is typically selected. If not, a server at a nearby IXP is typically selected.

Once Netflix determines the CDN server that is to deliver the content, it sends the client the IP address of the specific server as well as a manifest file, which has the URLs for the different versions of the requested movie. The client and that CDN server then directly interact using a proprietary version of DASH. Specifically, as described in [Section 2.6.2](#), the client uses the byte-range header in HTTP GET request messages, to request chunks from the different versions of the movie. Netflix uses chunks that are approximately four-seconds long [[Adhikari 2012](#)]. While the chunks are being downloaded, the client measures the received throughput and runs a rate-determination algorithm to determine the quality of the next chunk to request.

Netflix embodies many of the key principles discussed earlier in this section, including adaptive streaming and CDN distribution. However, because Netflix uses its own private CDN, which distributes only video (and not Web pages), Netflix has been able to simplify and tailor its CDN design. In particular, Netflix does not need to employ DNS redirect, as discussed in [Section 2.6.3](#), to connect a particular client to a CDN server; instead, the Netflix software (running in the Amazon cloud) directly tells the client to use a particular CDN server. Furthermore, the Netflix CDN uses push caching rather than pull caching ([Section 2.2.5](#)): content is pushed into the servers at scheduled times at off-peak hours, rather than dynamically during cache misses.

## YouTube

With 300 hours of video uploaded to YouTube every minute and several billion video views per day [[YouTube 2016](#)], YouTube is indisputably the world's largest video-sharing site. YouTube began its

service in April 2005 and was acquired by Google in November 2006. Although the Google/YouTube design and protocols are proprietary, through several independent measurement efforts we can gain a basic understanding about how YouTube operates [[Zink 2009](#); [Torres 2011](#); [Adhikari 2011a](#)]. As with Netflix, YouTube makes extensive use of CDN technology to distribute its videos [[Torres 2011](#)]. Similar to Netflix, Google uses its own private CDN to distribute YouTube videos, and has installed server clusters in many hundreds of different IXP and ISP locations. From these locations and directly from its huge data centers, Google distributes YouTube videos [[Adhikari 2011a](#)]. Unlike Netflix, however, Google uses pull caching, as described in [Section 2.2.5](#), and DNS redirect, as described in [Section 2.6.3](#). Most of the time, Google's cluster-selection strategy directs the client to the cluster for which the RTT between client and cluster is the lowest; however, in order to balance the load across clusters, sometimes the client is directed (via DNS) to a more distant cluster [[Torres 2011](#)].

YouTube employs HTTP streaming, often making a small number of different versions available for a video, each with a different bit rate and corresponding quality level. YouTube does not employ adaptive streaming (such as DASH), but instead requires the user to manually select a version. In order to save bandwidth and server resources that would be wasted by repositioning or early termination, YouTube uses the HTTP byte range request to limit the flow of transmitted data after a target amount of video is prefetched.

Several million videos are uploaded to YouTube every day. Not only are YouTube videos streamed from server to client over HTTP, but YouTube uploaders also upload their videos from client to server over HTTP. YouTube processes each video it receives, converting it to a YouTube video format and creating multiple versions at different bit rates. This processing takes place entirely within Google data centers. (See the case study on Google's network infrastructure in [Section 2.6.3](#).)

### Kankan

We just saw that dedicated servers, operated by private CDNs, stream Netflix and YouTube videos to clients. Netflix and YouTube have to pay not only for the server hardware but also for the bandwidth the servers use to distribute the videos. Given the scale of these services and the amount of bandwidth they are consuming, such a CDN deployment can be costly.

We conclude this section by describing an entirely different approach for providing video on demand over the Internet at a large scale—one that allows the service provider to significantly reduce its infrastructure and bandwidth costs. As you might suspect, this approach uses P2P delivery instead of (or along with) client-server delivery. Since 2011, Kankan (owned and operated by Xunlei) has been deploying P2P video delivery with great success, with tens of millions of users every month [[Zhang 2015](#)].

At a high level, P2P video streaming is very similar to BitTorrent file downloading. When a peer wants to

see a video, it contacts a tracker to discover other peers in the system that have a copy of that video. This requesting peer then requests chunks of the video in parallel from the other peers that have the video. Different from downloading with BitTorrent, however, requests are preferentially made for chunks that are to be played back in the near future in order to ensure continuous playback [\[Dhungel 2012\]](#).

Recently, Kankan has migrated to a hybrid CDN-P2P streaming system [\[Zhang 2015\]](#). Specifically, Kankan now deploys a few hundred servers within China and pushes video content to these servers. This Kankan CDN plays a major role in the start-up stage of video streaming. In most cases, the client requests the beginning of the content from CDN servers, and in parallel requests content from peers. When the total P2P traffic is sufficient for video playback, the client will cease streaming from the CDN and only stream from peers. But if the P2P streaming traffic becomes insufficient, the client will restart CDN connections and return to the mode of hybrid CDN-P2P streaming. In this manner, Kankan can ensure short initial start-up delays while minimally relying on costly infrastructure servers and bandwidth.

## 2.7 Socket Programming: Creating Network Applications

Now that we've looked at a number of important network applications, let's explore how network application programs are actually created. Recall from [Section 2.1](#) that a typical network application consists of a pair of programs—a client program and a server program—residing in two different end systems. When these two programs are executed, a client process and a server process are created, and these processes communicate with each other by reading from, and writing to, sockets. When creating a network application, the developer's main task is therefore to write the code for both the client and server programs.

There are two types of network applications. One type is an implementation whose operation is specified in a protocol standard, such as an RFC or some other standards document; such an application is sometimes referred to as “open,” since the rules specifying its operation are known to all. For such an implementation, the client and server programs must conform to the rules dictated by the RFC. For example, the client program could be an implementation of the client side of the HTTP protocol, described in [Section 2.2](#) and precisely defined in RFC 2616; similarly, the server program could be an implementation of the HTTP server protocol, also precisely defined in RFC 2616. If one developer writes code for the client program and another developer writes code for the server program, and both developers carefully follow the rules of the RFC, then the two programs will be able to interoperate. Indeed, many of today's network applications involve communication between client and server programs that have been created by independent developers—for example, a Google Chrome browser communicating with an Apache Web server, or a BitTorrent client communicating with BitTorrent tracker.

The other type of network application is a proprietary network application. In this case the client and server programs employ an application-layer protocol that has *not* been openly published in an RFC or elsewhere. A single developer (or development team) creates both the client and server programs, and the developer has complete control over what goes in the code. But because the code does not implement an open protocol, other independent developers will not be able to develop code that interoperates with the application.

In this section, we'll examine the key issues in developing a client-server application, and we'll “get our hands dirty” by looking at code that implements a very simple client-server application. During the development phase, one of the first decisions the developer must make is whether the application is to run over TCP or over UDP. Recall that TCP is connection oriented and provides a reliable byte-stream channel through which data flows between two end systems. UDP is connectionless and sends independent packets of data from one end system to the other, without any guarantees about delivery.

Recall also that when a client or server program implements a protocol defined by an RFC, it should use the well-known port number associated with the protocol; conversely, when developing a proprietary application, the developer must be careful to avoid using such well-known port numbers. (Port numbers were briefly discussed in [Section 2.1](#). They are covered in more detail in [Chapter 3](#).)

We introduce UDP and TCP socket programming by way of a simple UDP application and a simple TCP application. We present the simple UDP and TCP applications in Python 3. We could have written the code in Java, C, or C++, but we chose Python mostly because Python clearly exposes the key socket concepts. With Python there are fewer lines of code, and each line can be explained to the novice programmer without difficulty. But there's no need to be frightened if you are not familiar with Python. You should be able to easily follow the code if you have experience programming in Java, C, or C++.

If you are interested in client-server programming with Java, you are encouraged to see the Companion Website for this textbook; in fact, you can find there all the examples in this section (and associated labs) in Java. For readers who are interested in client-server programming in C, there are several good references available [[Donahoo 2001](#); [Stevens 1997](#); [Frost 1994](#); Kurose 1996]; our Python examples below have a similar look and feel to C.

### 2.7.1 Socket Programming with UDP

In this subsection, we'll write simple client-server programs that use UDP; in the following section, we'll write similar programs that use TCP.

Recall from [Section 2.1](#) that processes running on different machines communicate with each other by sending messages into sockets. We said that each process is analogous to a house and the process's socket is analogous to a door. The application resides on one side of the door in the house; the transport-layer protocol resides on the other side of the door in the outside world. The application developer has control of everything on the application-layer side of the socket; however, it has little control of the transport-layer side.

Now let's take a closer look at the interaction between two communicating processes that use UDP sockets. Before the sending process can push a packet of data out the socket door, when using UDP, it must first attach a destination address to the packet. After the packet passes through the sender's socket, the Internet will use this destination address to route the packet through the Internet to the socket in the receiving process. When the packet arrives at the receiving socket, the receiving process will retrieve the packet through the socket, and then inspect the packet's contents and take appropriate action.

So you may be now wondering, what goes into the destination address that is attached to the packet?

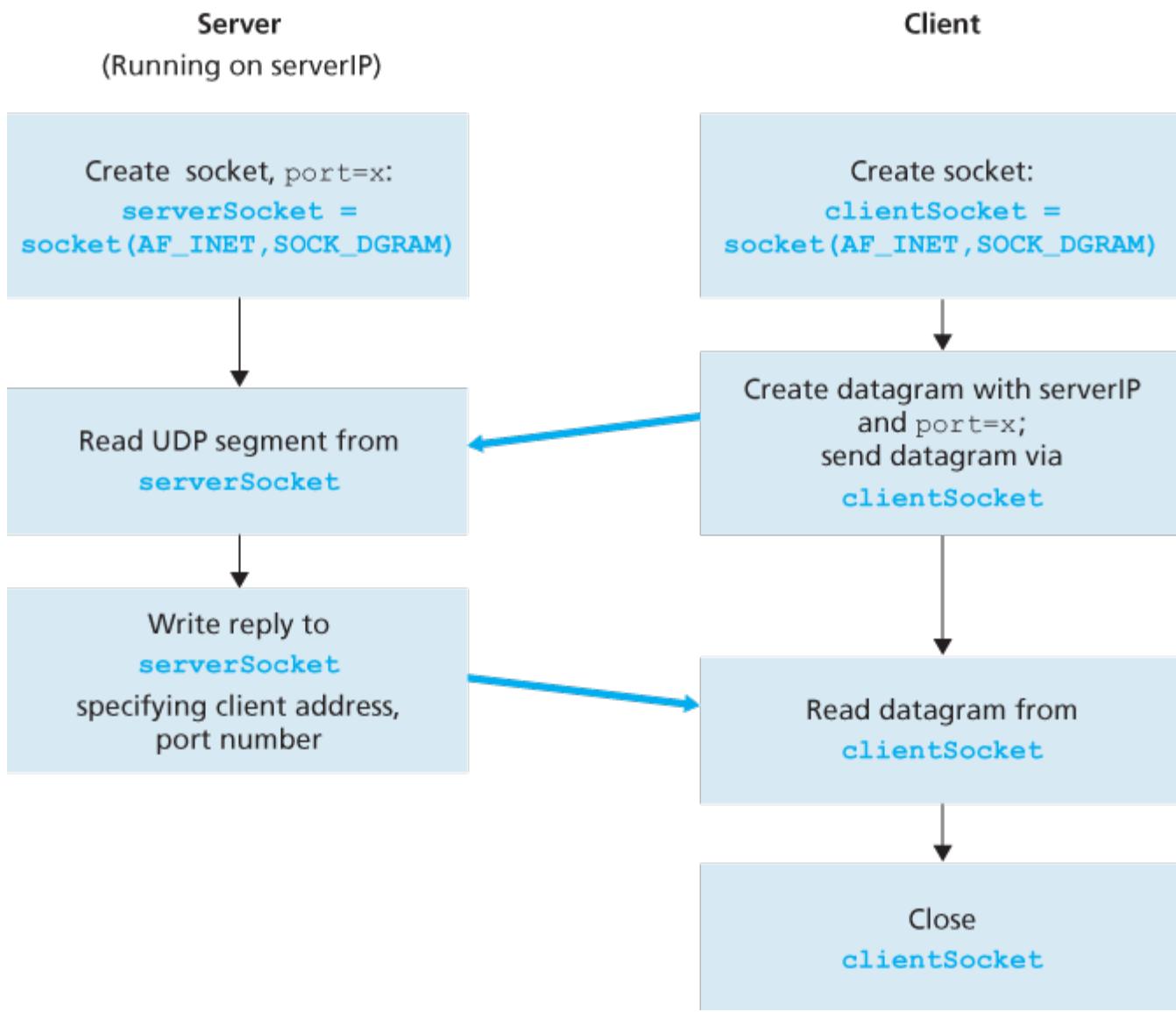
As you might expect, the destination host's IP address is part of the destination address. By including the destination IP address in the packet, the routers in the Internet will be able to route the packet through the Internet to the destination host. But because a host may be running many network application processes, each with one or more sockets, it is also necessary to identify the particular socket in the destination host. When a socket is created, an identifier, called a **port number**, is assigned to it. So, as you might expect, the packet's destination address also includes the socket's port number. In summary, the sending process attaches to the packet a destination address, which consists of the destination host's IP address and the destination socket's port number. Moreover, as we shall soon see, the sender's source address—consisting of the IP address of the source host and the port number of the source socket—are also attached to the packet. However, attaching the source address to the packet is typically *not* done by the UDP application code; instead it is automatically done by the underlying operating system.

We'll use the following simple client-server application to demonstrate socket programming for both UDP and TCP:

1. The client reads a line of characters (data) from its keyboard and sends the data to the server.
2. The server receives the data and converts the characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the line on its screen.

**Figure 2.27** highlights the main socket-related activity of the client and server that communicate over the UDP transport service.

Now let's get our hands dirty and take a look at the client-server program pair for a UDP implementation of this simple application. We also provide a detailed, line-by-line analysis after each program. We'll begin with the UDP client, which will send a simple application-level message to the server. In order for



**Figure 2.27 The client-server application using UDP**

the server to be able to receive and reply to the client’s message, it must be ready and running—that is, it must be running as a process before the client sends its message.

The client program is called `UDPClient.py`, and the server program is called `UDPServer.py`. In order to emphasize the key issues, we intentionally provide code that is minimal. “Good code” would certainly have a few more auxiliary lines, in particular for handling error cases. For this application, we have arbitrarily chosen 12000 for the server port number.

### *UDPClient.py*

Here is the code for the client side of the application:

```

from socket import *
serverName = 'hostname'
serverPort = 12000

```

```
clientSocket = socket(AF_INET, SOCK_DGRAM)
message = raw_input('Input lowercase sentence:')
clientSocket.sendto(message.encode(), (serverName, serverPort))
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print(modifiedMessage.decode())
clientSocket.close()
```

Now let's take a look at the various lines of code in `UDPClient.py`.

```
from socket import *
```

The `socket` module forms the basis of all network communications in Python. By including this line, we will be able to create sockets within our program.

```
serverName = 'hostname'
serverPort = 12000
```

The first line sets the variable `serverName` to the string ‘hostname’. Here, we provide a string containing either the IP address of the server (e.g., “128.138.32.126”) or the hostname of the server (e.g., “`cis.poly.edu`”). If we use the hostname, then a DNS lookup will automatically be performed to get the IP address.) The second line sets the integer variable `serverPort` to 12000.

```
clientSocket = socket(AF_INET, SOCK_DGRAM)
```

This line creates the client's socket, called `clientSocket`. The first parameter indicates the address family; in particular, `AF_INET` indicates that the underlying network is using IPv4. (Do not worry about this now—we will discuss IPv4 in [Chapter 4](#).) The second parameter indicates that the socket is of type `SOCK_DGRAM`, which means it is a UDP socket (rather than a TCP socket). Note that we are not specifying the port number of the client socket when we create it; we are instead letting the operating system do this for us. Now that the client process's door has been created, we will want to create a message to send through the door.

```
message = raw_input('Input lowercase sentence:')
```

`raw_input()` is a built-in function in Python. When this command is executed, the user at the client is prompted with the words “Input lowercase sentence:” The user then uses her keyboard to input a line, which is put into the variable `message`. Now that we have a socket and a message, we will want to send the message through the socket to the destination host.

```
clientSocket.sendto(message.encode(), (serverName, serverPort))
```

In the above line, we first convert the message from string type to byte type, as we need to send bytes into a socket; this is done with the `encode()` method. The method `sendto()` attaches the destination address (`serverName, serverPort`) to the message and sends the resulting packet into the process’s socket, `clientSocket`. (As mentioned earlier, the source address is also attached to the packet, although this is done automatically rather than explicitly by the code.) Sending a client-to-server message via a UDP socket is that simple! After sending the packet, the client waits to receive data from the server.

```
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
```

With the above line, when a packet arrives from the Internet at the client’s socket, the packet’s data is put into the variable `modifiedMessage` and the packet’s source address is put into the variable `serverAddress`. The variable `serverAddress` contains both the server’s IP address and the server’s port number. The program UDPClient doesn’t actually need this server address information, since it already knows the server address from the outset; but this line of Python provides the server address nevertheless. The method `recvfrom` also takes the buffer size 2048 as input. (This buffer size works for most purposes.)

```
print(modifiedMessage.decode())
```

This line prints out `modifiedMessage` on the user’s display, after converting the message from bytes to string. It should be the original line that the user typed, but now capitalized.

```
clientSocket.close()
```

This line closes the socket. The process then terminates.

## UDPServer.py

Let's now take a look at the server side of the application:

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(), clientAddress)
```

Note that the beginning of UDPServer is similar to UDPClient. It also imports the socket module, also sets the integer variable `serverPort` to 12000, and also creates a socket of type `SOCK_DGRAM` (a UDP socket). The first line of code that is significantly different from UDPClient is:

```
serverSocket.bind(('', serverPort))
```

The above line binds (that is, assigns) the port number 12000 to the server's socket. Thus in UDPServer, the code (written by the application developer) is explicitly assigning a port number to the socket. In this manner, when anyone sends a packet to port 12000 at the IP address of the server, that packet will be directed to this socket. UDPServer then enters a while loop; the while loop will allow UDPServer to receive and process packets from clients indefinitely. In the while loop, UDPServer waits for a packet to arrive.

```
message, clientAddress = serverSocket.recvfrom(2048)
```

This line of code is similar to what we saw in UDPClient. When a packet arrives at the server's socket, the packet's data is put into the variable `message` and the packet's source address is put into the variable `clientAddress`. The variable `clientAddress` contains both the client's IP address and the client's port number. Here, UDPServer *will* make use of this address information, as it provides a return

address, similar to the return address with ordinary postal mail. With this source address information, the server now knows to where it should direct its reply.

```
modifiedMessage = message.decode().upper()
```

This line is the heart of our simple application. It takes the line sent by the client and, after converting the message to a string, uses the method `upper()` to capitalize it.

```
serverSocket.sendto(modifiedMessage.encode(), clientAddress)
```

This last line attaches the client's address (IP address and port number) to the capitalized message (after converting the string to bytes), and sends the resulting packet into the server's socket. (As mentioned earlier, the server address is also attached to the packet, although this is done automatically rather than explicitly by the code.) The Internet will then deliver the packet to this client address. After the server sends the packet, it remains in the while loop, waiting for another UDP packet to arrive (from any client running on any host).

To test the pair of programs, you run `UDPClient.py` on one host and `UDPServer.py` on another host. Be sure to include the proper hostname or IP address of the server in `UDPClient.py`. Next, you execute `UDPServer.py`, the compiled server program, in the server host. This creates a process in the server that idles until it is contacted by some client. Then you execute `UDPClient.py`, the compiled client program, in the client. This creates a process in the client. Finally, to use the application at the client, you type a sentence followed by a carriage return.

To develop your own UDP client-server application, you can begin by slightly modifying the client or server programs. For example, instead of converting all the letters to uppercase, the server could count the number of times the letter `s` appears and return this number. Or you can modify the client so that after receiving a capitalized sentence, the user can continue to send more sentences to the server.

## 2.7.2 Socket Programming with TCP

Unlike UDP, TCP is a connection-oriented protocol. This means that before the client and server can start to send data to each other, they first need to handshake and establish a TCP connection. One end of the TCP connection is attached to the client socket and the other end is attached to a server socket. When creating the TCP connection, we associate with it the client socket address (IP address and port

number) and the server socket address (IP address and port number). With the TCP connection established, when one side wants to send data to the other side, it just drops the data into the TCP connection via its socket. This is different from UDP, for which the server must attach a destination address to the packet before dropping it into the socket.

Now let's take a closer look at the interaction of client and server programs in TCP. The client has the job of initiating contact with the server. In order for the server to be able to react to the client's initial contact, the server has to be ready. This implies two things. First, as in the case of UDP, the TCP server must be running as a process before the client attempts to initiate contact. Second, the server program must have a special door—more precisely, a special socket—that welcomes some initial contact from a client process running on an arbitrary host. Using our house/door analogy for a process/socket, we will sometimes refer to the client's initial contact as "knocking on the welcoming door."

With the server process running, the client process can initiate a TCP connection to the server. This is done in the client program by creating a TCP socket. When the client creates its TCP socket, it specifies the address of the welcoming socket in the server, namely, the IP address of the server host and the port number of the socket. After creating its socket, the client initiates a three-way handshake and establishes a TCP connection with the server. The three-way handshake, which takes place within the transport layer, is completely invisible to the client and server programs.

During the three-way handshake, the client process knocks on the welcoming door of the server process. When the server "hears" the knocking, it creates a new door—more precisely, a *new* socket that is dedicated to that particular client. In our example below, the welcoming door is a TCP socket object that we call `ServerSocket`; the newly created socket dedicated to the client making the connection is called `connectionSocket`. Students who are encountering TCP sockets for the first time sometimes confuse the welcoming socket (which is the initial point of contact for all clients wanting to communicate with the server), and each newly created server-side connection socket that is subsequently created for communicating with each client.

From the application's perspective, the client's socket and the server's connection socket are directly connected by a pipe. As shown in [Figure 2.28](#), the client process can send arbitrary bytes into its socket, and TCP guarantees that the server process will receive (through the connection socket) each byte in the order sent. TCP thus provides a reliable service between the client and server processes. Furthermore, just as people can go in and out the same door, the client process not only sends bytes into but also receives bytes from its socket; similarly, the server process not only receives bytes from but also sends bytes into its connection socket.

We use the same simple client-server application to demonstrate socket programming with TCP: The client sends one line of data to the server, the server capitalizes the line and sends it back to the client.

[Figure 2.29](#) highlights the main socket-related activity of the client and server that communicate over

the TCP transport service.

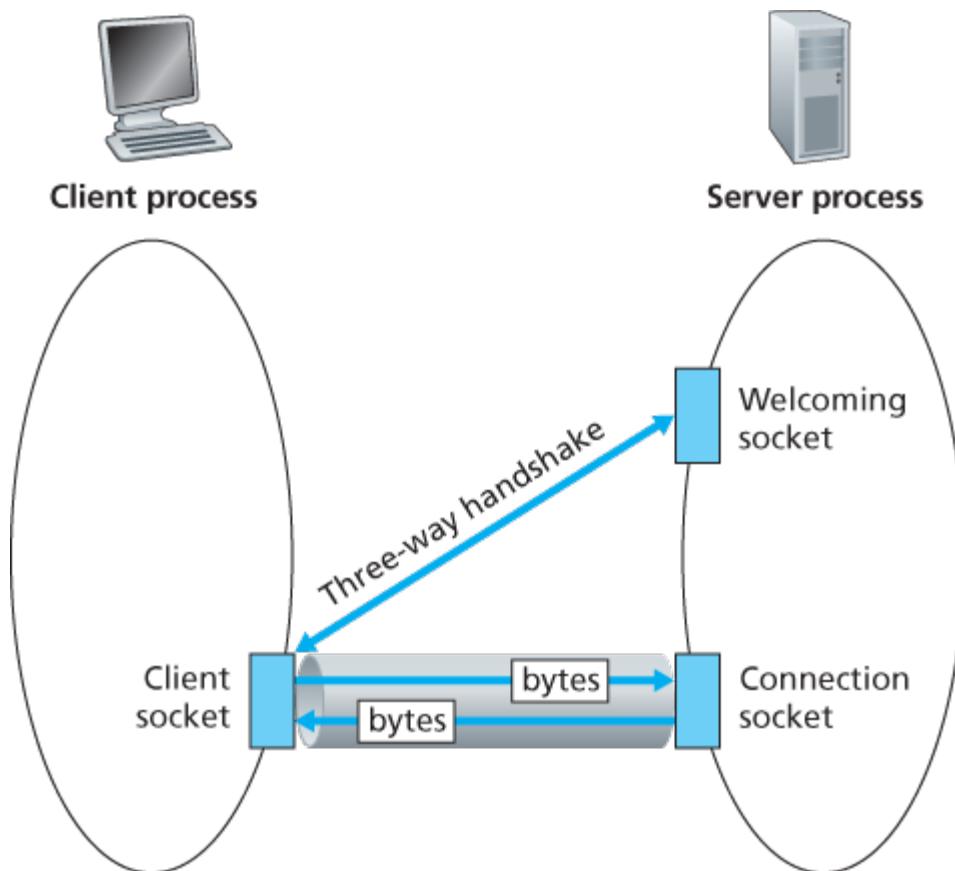


Figure 2.28 *The TCP Server* process has two sockets

### TCPClient.py

Here is the code for the client side of the application:

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print('From Server: ', modifiedSentence.decode())
clientSocket.close()
```

Let's now take a look at the various lines in the code that differ significantly from the UDP implementation. The first such line is the creation of the client socket.

```
clientSocket = socket (AF_INET, SOCK_STREAM)
```

This line creates the client's socket, called `clientSocket`. The first parameter again indicates that the underlying network is using IPv4. The second parameter

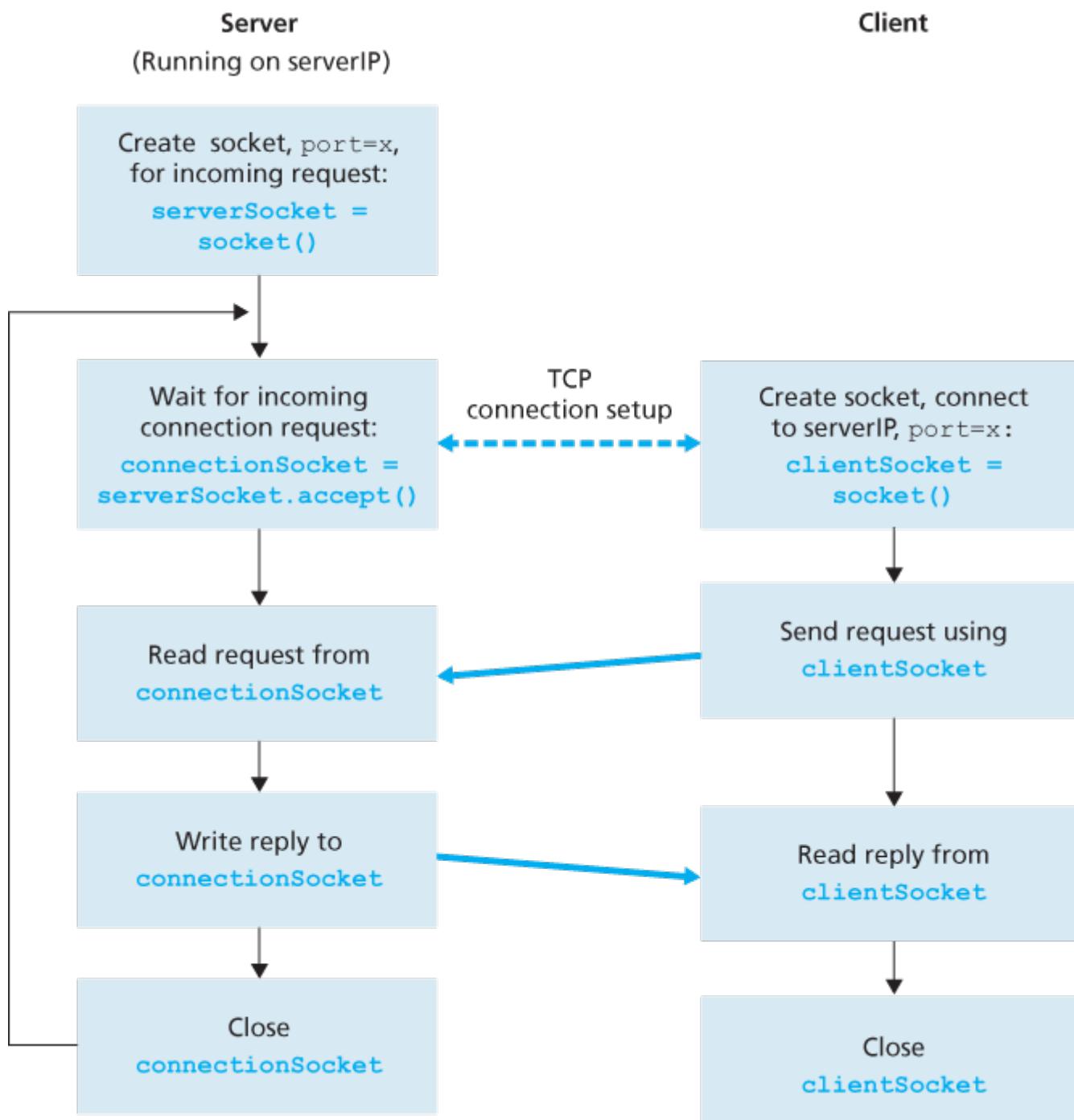


Figure 2.29 The client-server application using TCP

indicates that the socket is of type `SOCK_STREAM`, which means it is a TCP socket (rather than a UDP socket). Note that we are again not specifying the port number of the client socket when we create it; we are instead letting the operating system do this for us. Now the next line of code is very different from what we saw in `UDPClient`:

```
clientSocket.connect((serverName, serverPort))
```

Recall that before the client can send data to the server (or vice versa) using a TCP socket, a TCP connection must first be established between the client and server. The above line initiates the TCP connection between the client and server. The parameter of the `connect()` method is the address of the server side of the connection. After this line of code is executed, the three-way handshake is performed and a TCP connection is established between the client and server.

```
sentence = raw_input('Input lowercase sentence:')
```

As with UDPClient, the above obtains a sentence from the user. The string `sentence` continues to gather characters until the user ends the line by typing a carriage return. The next line of code is also very different from UDPClient:

```
clientSocket.send(sentence.encode())
```

The above line sends the `sentence` through the client's socket and into the TCP connection. Note that the program does *not* explicitly create a packet and attach the destination address to the packet, as was the case with UDP sockets. Instead the client program simply drops the bytes in the string `sentence` into the TCP connection. The client then waits to receive bytes from the server.

```
modifiedSentence = clientSocket.recv(2048)
```

When characters arrive from the server, they get placed into the string `modifiedSentence`. Characters continue to accumulate in `modifiedSentence` until the line ends with a carriage return character. After printing the capitalized sentence, we close the client's socket:

```
clientSocket.close()
```

This last line closes the socket and, hence, closes the TCP connection between the client and the server. It causes TCP in the client to send a TCP message to TCP in the server (see [Section 3.5](#)).

## *TCP*Server.py

Now let's take a look at the server program.

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('', serverPort))
serverSocket.listen(1)
print('The server is ready to receive')
while True:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.encode())
    connectionSocket.close()
```

Let's now take a look at the lines that differ significantly from UDPServer and TCPClient. As with TCPClient, the server creates a TCP socket with:

```
serverSocket=socket(AF_INET, SOCK_STREAM)
```

Similar to UDPServer, we associate the server port number, *serverPort*, with this socket:

```
serverSocket.bind(('', serverPort))
```

But with TCP, *serverSocket* will be our welcoming socket. After establishing this welcoming door, we will wait and listen for some client to knock on the door:

```
serverSocket.listen(1)
```

This line has the server listen for TCP connection requests from the client. The parameter specifies the maximum number of queued connections (at least 1).

```
connectionSocket, addr = serverSocket.accept()
```

When a client knocks on this door, the program invokes the `accept()` method for `serverSocket`, which creates a new socket in the server, called `connectionSocket`, dedicated to this particular client. The client and server then complete the handshaking, creating a TCP connection between the client's `clientSocket` and the server's `connectionSocket`. With the TCP connection established, the client and server can now send bytes to each other over the connection. With TCP, all bytes sent from one side are not only guaranteed to arrive at the other side but also guaranteed to arrive in order.

```
connectionSocket.close()
```

In this program, after sending the modified sentence to the client, we close the connection socket. But since `serverSocket` remains open, another client can now knock on the door and send the server a sentence to modify.

This completes our discussion of socket programming in TCP. You are encouraged to run the two programs in two separate hosts, and also to modify them to achieve slightly different goals. You should compare the UDP program pair with the TCP program pair and see how they differ. You should also do many of the socket programming assignments described at the ends of [Chapter 2](#), [4](#), and [9](#). Finally, we hope someday, after mastering these and more advanced socket programs, you will write your own popular network application, become very rich and famous, and remember the authors of this textbook!

## 2.8 Summary

In this chapter, we've studied the conceptual and the implementation aspects of network applications. We've learned about the ubiquitous client-server architecture adopted by many Internet applications and seen its use in the HTTP, SMTP, POP3, and DNS protocols. We've studied these important application-level protocols, and their corresponding associated applications (the Web, file transfer, e-mail, and DNS) in some detail. We've learned about the P2P architecture and how it is used in many applications. We've also learned about streaming video, and how modern video distribution systems leverage CDNs. We've examined how the socket API can be used to build network applications. We've walked through the use of sockets for connection-oriented (TCP) and connectionless (UDP) end-to-end transport services. The first step in our journey down the layered network architecture is now complete!

At the very beginning of this book, in [Section 1.1](#), we gave a rather vague, bare-bones definition of a protocol: “the format and the order of messages exchanged between two or more communicating entities, as well as the actions taken on the transmission and/or receipt of a message or other event.” The material in this chapter, and in particular our detailed study of the HTTP, SMTP, POP3, and DNS protocols, has now added considerable substance to this definition. Protocols are a key concept in networking; our study of application protocols has now given us the opportunity to develop a more intuitive feel for what protocols are all about.

In [Section 2.1](#), we described the service models that TCP and UDP offer to applications that invoke them. We took an even closer look at these service models when we developed simple applications that run over TCP and UDP in [Section 2.7](#). However, we have said little about how TCP and UDP provide these service models. For example, we know that TCP provides a reliable data service, but we haven't said yet how it does so. In the next chapter we'll take a careful look at not only the *what*, but also the *how* and *why* of transport protocols.

Equipped with knowledge about Internet application structure and application-level protocols, we're now ready to head further down the protocol stack and examine the transport layer in [Chapter 3](#).

# Homework Problems and Questions

## Chapter 2 Review Questions

### SECTION 2.1

- R1. List five nonproprietary Internet applications and the application-layer protocols that they use.
- R2. What is the difference between network architecture and application architecture?
- R3. For a communication session between a pair of processes, which process is the client and which is the server?
- R4. For a P2P file-sharing application, do you agree with the statement, “There is no notion of client and server sides of a communication session”? Why or why not?
- R5. What information is used by a process running on one host to identify a process running on another host?
- R6. Suppose you wanted to do a transaction from a remote client to a server as fast as possible. Would you use UDP or TCP? Why?
- R7. Referring to [Figure 2.4](#), we see that none of the applications listed in [Figure 2.4](#) requires both no data loss and timing. Can you conceive of an application that requires no data loss and that is also highly time-sensitive?
- R8. List the four broad classes of services that a transport protocol can provide. For each of the service classes, indicate if either UDP or TCP (or both) provides such a service.
- R9. Recall that TCP can be enhanced with SSL to provide process-to-process security services, including encryption. Does SSL operate at the transport layer or the application layer? If the application developer wants TCP to be enhanced with SSL, what does the developer have to do?

### SECTION 2.2–2.5

- R10. What is meant by a handshaking protocol?
- R11. Why do HTTP, SMTP, and POP3 run on top of TCP rather than on UDP?
- R12. Consider an e-commerce site that wants to keep a purchase record for each of its customers. Describe how this can be done with cookies.
- R13. Describe how Web caching can reduce the delay in receiving a requested object. Will Web caching reduce the delay for all objects requested by a user or for only some of the objects?

Why?

R14. Telnet into a Web server and send a multiline request message. Include in the request message the *If-modified-since:* header line to force a response message with the [304 Not Modified](#) status code.

R15. List several popular messaging apps. Do they use the same protocols as SMS?

R16. Suppose Alice, with a Web-based e-mail account (such as Hotmail or Gmail), sends a message to Bob, who accesses his mail from his mail server using POP3. Discuss how the message gets from Alice's host to Bob's host. Be sure to list the series of application-layer protocols that are used to move the message between the two hosts.

R17. Print out the header of an e-mail message you have recently received. How many *Received:* header lines are there? Analyze each of the header lines in the message.

R18. From a user's perspective, what is the difference between the download-and-delete mode and the download-and-keep mode in POP3?

R19. Is it possible for an organization's Web server and mail server to have exactly the same alias for a hostname (for example, [foo.com](#))? What would be the type for the RR that contains the hostname of the mail server?

R20. Look over your received e-mails, and examine the header of a message sent from a user with a .edu e-mail address. Is it possible to determine from the header the IP address of the host from which the message was sent? Do the same for a message sent from a Gmail account.

## SECTION 2.5

R21. In BitTorrent, suppose Alice provides chunks to Bob throughout a 30-second interval. Will Bob necessarily return the favor and provide chunks to Alice in this same interval? Why or why not?

R22. Consider a new peer Alice that joins BitTorrent without possessing any chunks. Without any chunks, she cannot become a top-four uploader for any of the other peers, since she has nothing to upload. How then will Alice get her first chunk?

R23. What is an overlay network? Does it include routers? What are the edges in the overlay network?

## SECTION 2.6

R24. CDNs typically adopt one of two different server placement philosophies. Name and briefly describe them.

R25. Besides network-related considerations such as delay, loss, and bandwidth performance, there are other important factors that go into designing a CDN server selection strategy. What are they?

## SECTION 2.7

R26. In Section 2.7, the UDP server described needed only one socket, whereas the TCP server needed two sockets. Why? If the TCP server were to support  $n$  simultaneous connections, each from a different client host, how many sockets would the TCP server need?

R27. For the client-server application over TCP described in [Section 2.7](#), why must the server program be executed before the client program? For the client-server application over UDP, why may the client program be executed before the server program?

## Problems

P1. True or false?

- a. A user requests a Web page that consists of some text and three images. For this page, the client will send one request message and receive four response messages.
- b. Two distinct Web pages (for example, [www.mit.edu/research.html](http://www.mit.edu/research.html) and [www.mit.edu/students.html](http://www.mit.edu/students.html)) can be sent over the same persistent connection.
- c. With nonpersistent connections between browser and origin server, it is possible for a single TCP segment to carry two distinct HTTP request messages.
- d. The `Date:` header in the HTTP response message indicates when the object in the response was last modified.
- e. HTTP response messages never have an empty message body.

P2. SMS, iMessage, and WhatsApp are all smartphone real-time messaging systems. After doing some research on the Internet, for each of these systems write one paragraph about the protocols they use. Then write a paragraph explaining how they differ.

P3. Consider an HTTP client that wants to retrieve a Web document at a given URL. The IP address of the HTTP server is initially unknown. What transport and application-layer protocols besides HTTP are needed in this scenario?

P4. Consider the following string of ASCII characters that were captured by Wireshark when the browser sent an HTTP GET message (i.e., this is the actual content of an HTTP GET message).

The characters `<cr><lf>` are carriage return and line-feed characters (that is, the italicized character string `<cr>` in the text below represents the single carriage-return character that was contained at that point in the HTTP header). Answer the following questions, indicating where in the HTTP GET message below you find the answer.

```
GET /cs453/index.html HTTP/1.1<cr><lf>Host: gai  
a.cs.umass.edu<cr><lf>User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.2) Gec  
ko/20040804 Netscape/7.2 (ax) <cr><lf>Accept:ex
```

```
t/xml, application/xml, application/xhtml+xml, text  
/html; q=0.9, text/plain; q=0.8, image/png, */*; q=0.5  
<cr><lf>Accept-Language: en-us, en; q=0.5<cr><lf>Accept-  
Encoding: zip, deflate<cr><lf>Accept-Charset: ISO  
-8859-1, utf-8; q=0.7, *; q=0.7<cr><lf>Keep-Alive: 300<cr>  
<lf>Connection: keep-alive<cr><lf><cr><lf>
```

- a. What is the URL of the document requested by the browser?
- b. What version of HTTP is the browser running?
- c. Does the browser request a non-persistent or a persistent connection?
- d. What is the IP address of the host on which the browser is running?
- e. What type of browser initiates this message? Why is the browser type needed in an HTTP request message?

P5. The text below shows the reply sent from the server in response to the HTTP GET message in the question above. Answer the following questions, indicating where in the message below you find the answer.

```
HTTP/1.1 200 OK<cr><lf>Date: Tue, 07 Mar 2008  
12:39:45GMT<cr><lf>Server: Apache/2.0.52 (Fedora)  
<cr><lf>Last-Modified: Sat, 10 Dec 2005 18:27:46  
GMT<cr><lf>ETag: "526c3-f22-a88a4c80"<cr><lf>Accept-  
Ranges: bytes<cr><lf>Content-Length: 3874<cr><lf>  
Keep-Alive: timeout=max=100<cr><lf>Connection:  
Keep-Alive<cr><lf>Content-Type: text/html; charset=  
ISO-8859-1<cr><lf><cr><lf><!doctype html public "-//  
w3c//dtd html 4.0 transitional//en"><lf><html><lf>  
<head><lf> <meta http-equiv="Content-Type"  
content="text/html; charset=iso-8859-1"><lf> <meta  
name="GENERATOR" content="Mozilla/4.79 [en] (Windows NT  
5.0; U) Netscape]"><lf> <title>CMPSCI 453 / 591 /  
NTU-ST550ASpring 2005 homepage</title><lf></head><lf>  
<much more document text following here (not shown)>
```

- a. Was the server able to successfully find the document or not? What time was the document reply provided?
- b. When was the document last modified?
- c. How many bytes are there in the document being returned?
- d. What are the first 5 bytes of the document being returned? Did the server agree to a

persistent connection?

P6. Obtain the HTTP/1.1 specification (RFC 2616). Answer the following questions:

- a. Explain the mechanism used for signaling between the client and server to indicate that a persistent connection is being closed. Can the client, the server, or both signal the close of a connection?
- b. What encryption services are provided by HTTP?
- c. Can a client open three or more simultaneous connections with a given server?
- d. Either a server or a client may close a transport connection between them if either one detects the connection has been idle for some time. Is it possible that one side starts closing a connection while the other side is transmitting data via this connection?

Explain.

P7. Suppose within your Web browser you click on a link to obtain a Web page. The IP address for the associated URL is not cached in your local host, so a DNS lookup is necessary to obtain the IP address. Suppose that  $n$  DNS servers are visited before your host receives the IP address from DNS; the successive visits incur an RTT of  $RTT_1, \dots, RTT_n$ . Further suppose that the Web page associated with the link contains exactly one object, consisting of a small amount of HTML text. Let  $RTT_0$  denote the RTT between the local host and the server containing the object. Assuming zero transmission time of the object, how much time elapses from when the client clicks on the link until the client receives the object?

P8. Referring to Problem P7, suppose the HTML file references eight very small objects on the same server. Neglecting transmission times, how much time elapses with

- a. Non-persistent HTTP with no parallel TCP connections?
- b. Non-persistent HTTP with the browser configured for 5 parallel connections?
- c. Persistent HTTP?

P9. Consider [Figure 2.12](#), for which there is an institutional network connected to the Internet. Suppose that the average object size is 850,000 bits and that the average request rate from the institution's browsers to the origin servers is 16 requests per second. Also suppose that the amount of time it takes from when the router on the Internet side of the access link forwards an HTTP request until it receives the response is three seconds on average (see Section 2.2.5). Model the total average response time as the sum of the average access delay (that is, the delay from Internet router to institution router) and the average Internet delay. For the average access delay, use  $\Delta/(1-\Delta\beta)$ , where  $\Delta$  is the average time required to send an object over the access link and  $\beta$  is the arrival rate of objects to the access link.

- a. Find the total average response time.
- b. Now suppose a cache is installed in the institutional LAN. Suppose the miss rate is 0.4. Find the total response time.

P10. Consider a short, 10-meter link, over which a sender can transmit at a rate of 150 bits/sec in both directions. Suppose that packets containing data are 100,000 bits long, and packets containing only control (e.g., ACK or handshaking) are 200 bits long. Assume that  $N$  parallel connections each get  $1/N$  of the link bandwidth. Now consider the HTTP protocol, and suppose that each downloaded object is 100 Kbits long, and that the initial downloaded object contains 10 referenced objects from the same sender. Would parallel downloads via parallel instances of non-persistent HTTP make sense in this case? Now consider persistent HTTP. Do you expect significant gains over the non-persistent case? Justify and explain your answer.

P11. Consider the scenario introduced in the previous problem. Now suppose that the link is shared by Bob with four other users. Bob uses parallel instances of non-persistent HTTP, and the other four users use non-persistent HTTP without parallel downloads.

- a. Do Bob's parallel connections help him get Web pages more quickly? Why or why not?
- b. If all five users open five parallel instances of non-persistent HTTP, then would Bob's parallel connections still be beneficial? Why or why not?

P12. Write a simple TCP program for a server that accepts lines of input from a client and prints the lines onto the server's standard output. (You can do this by modifying the TCPServer.py program in the text.) Compile and execute your program. On any other machine that contains a Web browser, set the proxy server in the browser to the host that is running your server program; also configure the port number appropriately. Your browser should now send its GET request messages to your server, and your server should display the messages on its standard output. Use this platform to determine whether your browser generates conditional GET messages for objects that are locally cached.

P13. What is the difference between *MAIL FROM*: in SMTP and *From*: in the mail message itself?

P14. How does SMTP mark the end of a message body? How about HTTP? Can HTTP use the same method as SMTP to mark the end of a message body? Explain.

P15. Read RFC 5321 for SMTP. What does MTA stand for? Consider the following received spam e-mail (modified from a real spam e-mail). Assuming only the originator of this spam e-mail is malicious and all other hosts are honest, identify the malicious host that has generated this spam e-mail.

```
From - Fri Nov 07 13:41:30 2008
Return-Path: <tennis5@pp33head.com>
Received: from barmail.cs.umass.edu (barmail.cs.umass.edu
[128.119.240.3]) by cs.umass.edu (8.13.1/8.12.6) for
<hg@cs.umass.edu>; Fri, 7 Nov 2008 13:27:10 -0500
Received: from asusus-4b96 (localhost [127.0.0.1]) by
barmail.cs.umass.edu (Spam Firewall) for <hg@cs.umass.edu>; Fri, 7
```

*Nov 2008 13:27:07 -0500 (EST)*  
*Received: from asusus-4b96 ([58.88.21.177]) by barmail.cs.umass.edu*  
*for <hg@cs.umass.edu>; Fri, 07 Nov 2008 13:27:07 -0500 (EST)*  
*Received: from [58.88.21.177] by inbnd55.exchangeddd.com; Sat, 8*  
*Nov 2008 01:27:07 +0700*  
*From: "Jonny" <tennis5@pp33head.com>*  
*To: <hg@cs.umass.edu>*

*Subject: How to secure your savings*

P16. Read the POP3 RFC, RFC 1939. What is the purpose of the UIDL POP3 command?

P17. Consider accessing your e-mail with POP3.

- a. Suppose you have configured your POP mail client to operate in the download-and-delete mode. Complete the following transaction:

```
C: list  
S: 1 498  
S: 2 912  
S: .  
C: retr 1  
S: blah blah ...  
S: .....blah  
S: .  
?  
?
```

- b. Suppose you have configured your POP mail client to operate in the download-and-keep mode. Complete the following transaction:

```
C: list  
S: 1 498  
S: 2 912  
S: .  
C: retr 1  
S: blah blah ...  
S: .....blah  
S: .  
?
```

- c. Suppose you have configured your POP mail client to operate in the download-and-keep mode. Using your transcript in part (b), suppose you retrieve messages 1 and 2, exit POP, and then five minutes later you again access POP to retrieve new e-mail. Suppose that in the five-minute interval no new messages have been sent to you. Provide a transcript of this second POP session.

P18.

- a. What is a *whois* database?
- b. Use various whois databases on the Internet to obtain the names of two DNS servers. Indicate which whois databases you used.
- c. Use nslookup on your local host to send DNS queries to three DNS servers: your local DNS server and the two DNS servers you found in part (b). Try querying for Type A, NS, and MX reports. Summarize your findings.
- d. Use nslookup to find a Web server that has multiple IP addresses. Does the Web server of your institution (school or company) have multiple IP addresses?
- e. Use the ARIN whois database to determine the IP address range used by your university.
- f. Describe how an attacker can use whois databases and the nslookup tool to perform reconnaissance on an institution before launching an attack.
- g. Discuss why whois databases should be publicly available.

P19. In this problem, we use the useful *dig* tool available on Unix and Linux hosts to explore the hierarchy of DNS servers. Recall that in [Figure 2.19](#), a DNS server in the DNS hierarchy delegates a DNS query to a DNS server lower in the hierarchy, by sending back to the DNS client the name of that lower-level DNS server. First read the man page for *dig*, and then answer the following questions.

- a. Starting with a root DNS server (from one of the root servers [a-m].root-servers.net), initiate a sequence of queries for the IP address for your department's Web server by using *dig*. Show the list of the names of DNS servers in the delegation chain in answering your query.
- b. Repeat part (a) for several popular Web sites, such as [google.com](http://google.com), [yahoo.com](http://yahoo.com), or [amazon.com](http://amazon.com).

P20. Suppose you can access the caches in the local DNS servers of your department. Can you propose a way to roughly determine the Web servers (outside your department) that are most popular among the users in your department? Explain.

P21. Suppose that your department has a local DNS server for all computers in the department.

You are an ordinary user (i.e., not a network/system administrator). Can you determine if an external Web site was likely accessed from a computer in your department a couple of seconds ago? Explain.

P22. Consider distributing a file of  $F=15$  Gbits to  $N$  peers. The server has an upload rate of  $u_s=30$  Mbps, and each peer has a download rate of  $d_i=2$  Mbps and an upload rate of  $u$ . For  $N=10, 100,$  and  $1,000$  and  $u=300$  Kbps,  $700$  Kbps, and  $2$  Mbps, prepare a chart giving the minimum distribution time for each of the combinations of  $N$  and  $u$  for both client-server distribution and P2P distribution.

P23. Consider distributing a file of  $F$  bits to  $N$  peers using a client-server architecture. Assume a fluid model where the server can simultaneously transmit to multiple peers, transmitting to each peer at different rates, as long as the combined rate does not exceed  $u_s$ .

- a. Suppose that  $u_s/N \leq d_{\min}$ . Specify a distribution scheme that has a distribution time of  $NF/u_s$ .
- b. Suppose that  $u_s/N \geq d_{\min}$ . Specify a distribution scheme that has a distribution time of  $F/d_{\min}$ .
- c. Conclude that the minimum distribution time is in general given by  $\max\{NF/u_s, F/d_{\min}\}$ .

P24. Consider distributing a file of  $F$  bits to  $N$  peers using a P2P architecture. Assume a fluid model. For simplicity assume that  $d_{\min}$  is very large, so that peer download bandwidth is never a bottleneck.

- a. Suppose that  $u_s \leq (u_s + u_1 + \dots + u_N)/N$ . Specify a distribution scheme that has a distribution time of  $F/u_s$ .
- b. Suppose that  $u_s \geq (u_s + u_1 + \dots + u_N)/N$ . Specify a distribution scheme that has a distribution time of  $NF/(u_s + u_1 + \dots + u_N)$ .
- c. Conclude that the minimum distribution time is in general given by  $\max\{F/u_s, NF/(u_s + u_1 + \dots + u_N)\}$ .

P25. Consider an overlay network with  $N$  active peers, with each pair of peers having an active TCP connection. Additionally, suppose that the TCP connections pass through a total of  $M$  routers. How many nodes and edges are there in the corresponding overlay network?

P26. Suppose Bob joins a BitTorrent torrent, but he does not want to upload any data to any other peers (so called free-riding).

- a. Bob claims that he can receive a complete copy of the file that is shared by the swarm. Is Bob's claim possible? Why or why not?
- b. Bob further claims that he can further make his "free-riding" more efficient by using a collection of multiple computers (with distinct IP addresses) in the computer lab in his department. How can he do that?

P27. Consider a DASH system for which there are  $N$  video versions (at  $N$  different rates and qualities) and  $N$  audio versions (at  $N$  different rates and qualities). Suppose we want to allow the

player to choose at any time any of the  $N$  video versions and any of the  $N$  audio versions.

- a. If we create files so that the audio is mixed in with the video, so server sends only one media stream at given time, how many files will the server need to store (each a different URL)?
- b. If the server instead sends the audio and video streams separately and has the client synchronize the streams, how many files will the server need to store?

P28. Install and compile the Python programs TCPClient and UDPClient on one host and TCPServer and UDPServer on another host.

- a. Suppose you run TCPClient before you run TCPServer. What happens? Why?
- b. Suppose you run UDPClient before you run UDPServer. What happens? Why?
- c. What happens if you use different port numbers for the client and server sides?

P29. Suppose that in UDPClient.py, after we create the socket, we add the line:

```
clientSocket.bind(('', 5432))
```

Will it become necessary to change UDPServer.py? What are the port numbers for the sockets in UDPClient and UDPServer? What were they before making this change?

P30. Can you configure your browser to open multiple simultaneous connections to a Web site? What are the advantages and disadvantages of having a large number of simultaneous TCP connections?

P31. We have seen that Internet TCP sockets treat the data being sent as a byte stream but UDP sockets recognize message boundaries. What are one advantage and one disadvantage of byte-oriented API versus having the API explicitly recognize and preserve application-defined message boundaries?

P32. What is the Apache Web server? How much does it cost? What functionality does it currently have? You may want to look at Wikipedia to answer this question.

### *Socket Programming Assignments*

The Companion Website includes six socket programming assignments. The first four assignments are summarized below. The fifth assignment makes use of the ICMP protocol and is summarized at the end of [Chapter 5](#). The sixth assignment employs multimedia protocols and is summarized at the end of [Chapter 9](#). It is highly recommended that students complete several, if not all, of these assignments. Students can find full details of these assignments, as well as important snippets of the Python code, at the Web site [www.pearsonhighered.com/cs-resources](http://www.pearsonhighered.com/cs-resources).

### *Assignment 1: Web Server*

In this assignment, you will develop a simple Web server in Python that is capable of processing only one request. Specifically, your Web server will (i) create a connection socket when contacted by a client (browser); (ii) receive the HTTP request from this connection; (iii) parse the request to determine the specific file being requested; (iv) get the requested file from the server's file system; (v) create an HTTP response message consisting of the requested file preceded by header lines; and (vi) send the response over the TCP connection to the requesting browser. If a browser requests a file that is not present in your server, your server should return a "404 Not Found" error message.

In the Companion Website, we provide the skeleton code for your server. Your job is to complete the code, run your server, and then test your server by sending requests from browsers running on different hosts. If you run your server on a host that already has a Web server running on it, then you should use a different port than port 80 for your Web server.

#### *Assignment 2: UDP Pinger*

In this programming assignment, you will write a client ping program in Python. Your client will send a simple ping message to a server, receive a corresponding pong message back from the server, and determine the delay between when the client sent the ping message and received the pong message. This delay is called the Round Trip Time (RTT). The functionality provided by the client and server is similar to the functionality provided by standard ping program available in modern operating systems. However, standard ping programs use the Internet Control Message Protocol (ICMP) (which we will study in [Chapter 5](#)). Here we will create a nonstandard (but simple!) UDP-based ping program.

Your ping program is to send 10 ping messages to the target server over UDP. For each message, your client is to determine and print the RTT when the corresponding pong message is returned. Because UDP is an unreliable protocol, a packet sent by the client or server may be lost. For this reason, the client cannot wait indefinitely for a reply to a ping message. You should have the client wait up to one second for a reply from the server; if no reply is received, the client should assume that the packet was lost and print a message accordingly.

In this assignment, you will be given the complete code for the server (available in the Companion Website). Your job is to write the client code, which will be very similar to the server code. It is recommended that you first study carefully the server code. You can then write your client code, liberally cutting and pasting lines from the server code.

#### *Assignment 3: Mail Client*

The goal of this programming assignment is to create a simple mail client that sends e-mail to any recipient. Your client will need to establish a TCP connection with a mail server (e.g., a Google mail server), dialogue with the mail server using the SMTP protocol, send an e-mail message to a recipient

(e.g., your friend) via the mail server, and finally close the TCP connection with the mail server.

For this assignment, the Companion Website provides the skeleton code for your client. Your job is to complete the code and test your client by sending e-mail to different user accounts. You may also try sending through different servers (for example, through a Google mail server and through your university mail server).

#### *Assignment 4: Multi-Threaded Web Proxy*

In this assignment, you will develop a Web proxy. When your proxy receives an HTTP request for an object from a browser, it generates a new HTTP request for the same object and sends it to the origin server. When the proxy receives the corresponding HTTP response with the object from the origin server, it creates a new HTTP response, including the object, and sends it to the client. This proxy will be multi-threaded, so that it will be able to handle multiple requests at the same time.

For this assignment, the Companion Website provides the skeleton code for the proxy server. Your job is to complete the code, and then test it by having different browsers request Web objects via your proxy.

#### *Wireshark Lab: HTTP*

Having gotten our feet wet with the Wireshark packet sniffer in Lab 1, we're now ready to use Wireshark to investigate protocols in operation. In this lab, we'll explore several aspects of the HTTP protocol: the basic GET/reply interaction, HTTP message formats, retrieving large HTML files, retrieving HTML files with embedded URLs, persistent and non-persistent connections, and HTTP authentication and security.

As is the case with all Wireshark labs, the full description of this lab is available at this book's Web site, [www.pearsonhighered.com/cs-resources](http://www.pearsonhighered.com/cs-resources).

#### *Wireshark Lab: DNS*

In this lab, we take a closer look at the client side of the DNS, the protocol that translates Internet hostnames to IP addresses. Recall from [Section 2.5](#) that the client's role in the DNS is relatively simple—a client sends a query to its local DNS server and receives a response back. Much can go on under the covers, invisible to the DNS clients, as the hierarchical DNS servers communicate with each other to either recursively or iteratively resolve the client's DNS query. From the DNS client's standpoint, however, the protocol is quite simple—a query is formulated to the local DNS server and a response is received from that server. We observe DNS in action in this lab.

As is the case with all Wireshark labs, the full description of this lab is available at this book's Web site, [www.pearsonhighered.com/cs-resources](http://www.pearsonhighered.com/cs-resources).

### An Interview With...

#### **Marc Andreessen**

Marc Andreessen is the co-creator of Mosaic, the Web browser that popularized the World Wide Web in 1993. Mosaic had a clean, easily understood interface and was the first browser to display images in-line with text. In 1994, Marc Andreessen and Jim Clark founded Netscape, whose browser was by far the most popular browser through the mid-1990s. Netscape also developed the Secure Sockets Layer (SSL) protocol and many Internet server products, including mail servers and SSL-based Web servers. He is now a co-founder and general partner of venture capital firm Andreessen Horowitz, overseeing portfolio development with holdings that include Facebook, Foursquare, Groupon, Jawbone, Twitter, and Zynga. He serves on numerous boards, including Bump, eBay, Glam Media, Facebook, and Hewlett-Packard. He holds a BS in Computer Science from the University of Illinois at Urbana-Champaign.



How did you become interested in computing? Did you always know that you wanted to work in information technology?

The video game and personal computing revolutions hit right when I was growing up—personal computing was the new technology frontier in the late 70's and early 80's. And it wasn't just Apple and the IBM PC, but hundreds of new companies like Commodore and Atari as well. I taught myself to program out of a book called "Instant Freeze-Dried BASIC" at age 10, and got my first computer (a TRS-80 Color Computer—look it up!) at age 12.

Please describe one or two of the most exciting projects you have worked on during your career.

What were the biggest challenges?

Undoubtedly the most exciting project was the original Mosaic web browser in '92-'93—and the biggest challenge was getting anyone to take it seriously back then. At the time, everyone thought the interactive future would be delivered as “interactive television” by huge companies, not as the Internet by startups.

What excites you about the future of networking and the Internet? What are your biggest concerns?

The most exciting thing is the huge unexplored frontier of applications and services that programmers and entrepreneurs are able to explore—the Internet has unleashed creativity at a level that I don’t think we’ve ever seen before. My biggest concern is the principle of unintended consequences—we don’t always know the implications of what we do, such as the Internet being used by governments to run a new level of surveillance on citizens.

Is there anything in particular students should be aware of as Web technology advances?

The rate of change—the most important thing to learn is how to learn—how to flexibly adapt to changes in the specific technologies, and how to keep an open mind on the new opportunities and possibilities as you move through your career.

What people inspired you professionally?

Vannevar Bush, Ted Nelson, Doug Engelbart, Nolan Bushnell, Bill Hewlett and Dave Packard, Ken Olsen, Steve Jobs, Steve Wozniak, Andy Grove, Grace Hopper, Hedy Lamarr, Alan Turing, Richard Stallman.

What are your recommendations for students who want to pursue careers in computing and information technology?

Go as deep as you possibly can on understanding how technology is created, and then complement with learning how business works.

Can technology solve the world’s problems?

No, but we advance the standard of living of people through economic growth, and most economic growth throughout history has come from technology—so that’s as good as it gets.



---

## Chapter 3 Transport Layer

---

Residing between the application and network layers, the transport layer is a central piece of the layered network architecture. It has the critical role of providing communication services directly to the application processes running on different hosts. The pedagogic approach we take in this chapter is to alternate between discussions of transport-layer principles and discussions of how these principles are implemented in existing protocols; as usual, particular emphasis will be given to Internet protocols, in particular the TCP and UDP transport-layer protocols.

We'll begin by discussing the relationship between the transport and network layers. This sets the stage for examining the first critical function of the transport layer—extending the network layer's delivery service between two end systems to a delivery service between two application-layer processes running on the end systems. We'll illustrate this function in our coverage of the Internet's connectionless transport protocol, UDP.

We'll then return to principles and confront one of the most fundamental problems in computer networking—how two entities can communicate reliably over a medium that may lose and corrupt data. Through a series of increasingly complicated (and realistic!) scenarios, we'll build up an array of techniques that transport protocols use to solve this problem. We'll then show how these principles are embodied in TCP, the Internet's connection-oriented transport protocol.

We'll next move on to a second fundamentally important problem in networking—controlling the transmission rate of transport-layer entities in order to avoid, or recover from, congestion within the network. We'll consider the causes and consequences of congestion, as well as commonly used congestion-control techniques. After obtaining a solid understanding of the issues behind congestion control, we'll study TCP's approach to congestion control.

## 3.1 Introduction and Transport-Layer Services

In the previous two chapters we touched on the role of the transport layer and the services that it provides. Let's quickly review what we have already learned about the transport layer.

A transport-layer protocol provides for **logical communication** between application processes running on different hosts. By *logical communication*, we mean that from an application's perspective, it is as if the hosts running the processes were directly connected; in reality, the hosts may be on opposite sides of the planet, connected via numerous routers and a wide range of link types. Application processes use the logical communication provided by the transport layer to send messages to each other, free from the worry of the details of the physical infrastructure used to carry these messages. **Figure 3.1** illustrates the notion of logical communication.

As shown in **Figure 3.1**, transport-layer protocols are implemented in the end systems but not in network routers. On the sending side, the transport layer converts the application-layer messages it receives from a sending application process into transport-layer packets, known as transport-layer **segments** in Internet terminology. This is done by (possibly) breaking the application messages into smaller chunks and adding a transport-layer header to each chunk to create the transport-layer segment. The transport layer then passes the segment to the network layer at the sending end system, where the segment is encapsulated within a network-layer packet (a datagram) and sent to the destination. It's important to note that network routers act only on the network-layer fields of the datagram; that is, they do not examine the fields of the transport-layer segment encapsulated with the datagram. On the receiving side, the network layer extracts the transport-layer segment from the datagram and passes the segment up to the transport layer. The transport layer then processes the received segment, making the data in the segment available to the receiving application.

More than one transport-layer protocol may be available to network applications. For example, the Internet has two protocols—TCP and UDP. Each of these protocols provides a different set of transport-layer services to the invoking application.

### 3.1.1 Relationship Between Transport and Network Layers

Recall that the transport layer lies just above the network layer in the protocol stack. Whereas a transport-layer protocol provides logical communication between

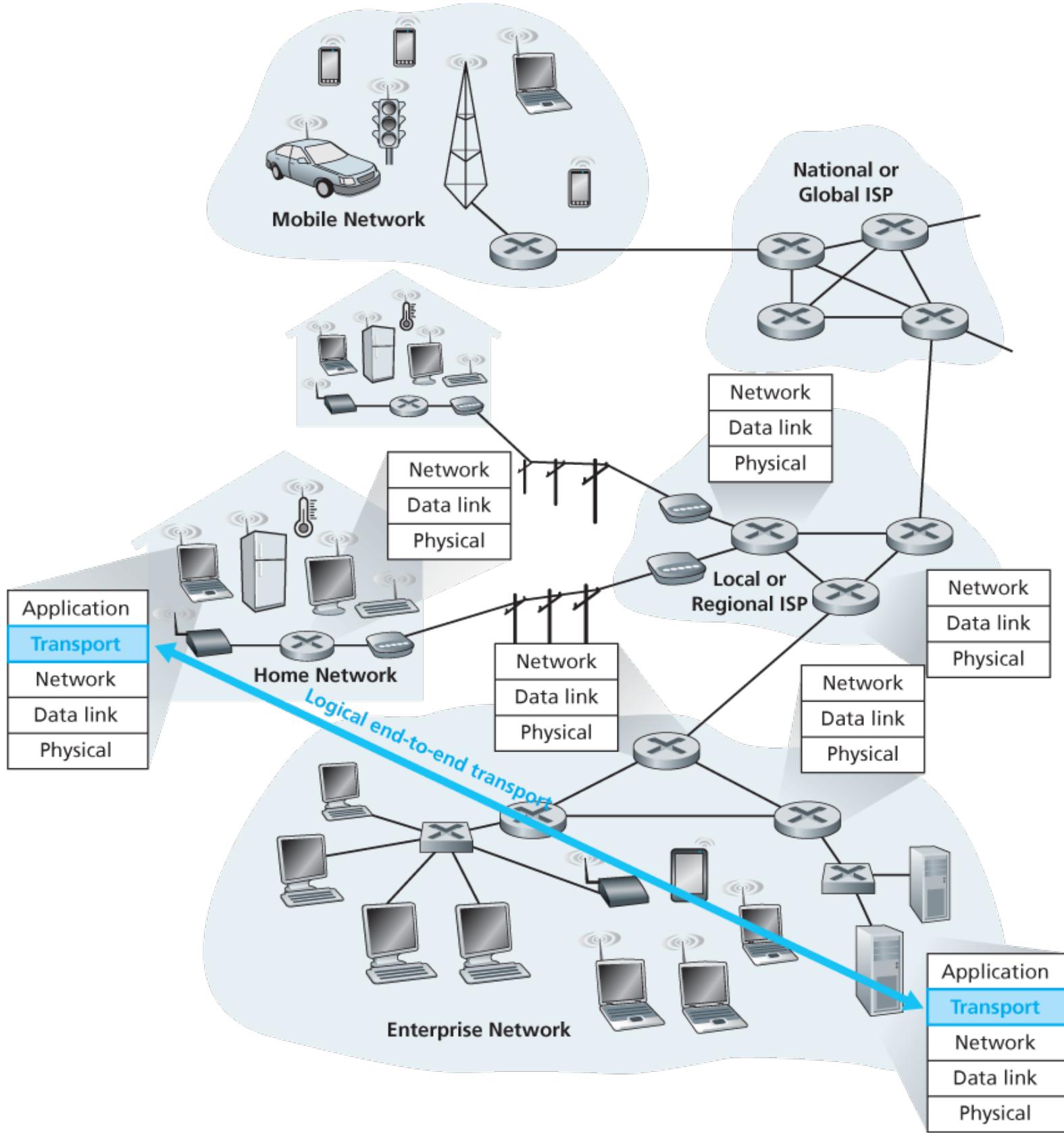


Figure 3.1 The transport layer provides logical rather than physical communication between application processes

processes running on different hosts, a network-layer protocol provides logical-communication between hosts. This distinction is subtle but important. Let's examine this distinction with the aid of a household analogy.

Consider two houses, one on the East Coast and the other on the West Coast, with each house being home to a dozen kids. The kids in the East Coast household are cousins of the kids in the West Coast

household. The kids in the two households love to write to each other—each kid writes each cousin every week, with each letter delivered by the traditional postal service in a separate envelope. Thus, each household sends 144 letters to the other household every week. (These kids would save a lot of money if they had e-mail!) In each of the households there is one kid—Ann in the West Coast house and Bill in the East Coast house—responsible for mail collection and mail distribution. Each week Ann visits all her brothers and sisters, collects the mail, and gives the mail to a postal-service mail carrier, who makes daily visits to the house. When letters arrive at the West Coast house, Ann also has the job of distributing the mail to her brothers and sisters. Bill has a similar job on the East Coast.

In this example, the postal service provides logical communication between the two houses—the postal service moves mail from house to house, not from person to person. On the other hand, Ann and Bill provide logical communication among the cousins—Ann and Bill pick up mail from, and deliver mail to, their brothers and sisters. Note that from the cousins' perspective, Ann and Bill are the mail service, even though Ann and Bill are only a part (the end-system part) of the end-to-end delivery process. This household example serves as a nice analogy for explaining how the transport layer relates to the network layer:

application messages = letters in envelopes

processes = cousins

hosts (also called end systems) = houses

transport-layer protocol = Ann and Bill

network-layer protocol = postal service (including mail carriers)

Continuing with this analogy, note that Ann and Bill do all their work within their respective homes; they are not involved, for example, in sorting mail in any intermediate mail center or in moving mail from one mail center to another. Similarly, transport-layer protocols live in the end systems. Within an end system, a transport protocol moves messages from application processes to the network edge (that is, the network layer) and vice versa, but it doesn't have any say about how the messages are moved within the network core. In fact, as illustrated in [Figure 3.1](#), intermediate routers neither act on, nor recognize, any information that the transport layer may have added to the application messages.

Continuing with our family saga, suppose now that when Ann and Bill go on vacation, another cousin pair—say, Susan and Harvey—substitute for them and provide the household-internal collection and delivery of mail. Unfortunately for the two families, Susan and Harvey do not do the collection and delivery in exactly the same way as Ann and Bill. Being younger kids, Susan and Harvey pick up and drop off the mail less frequently and occasionally lose letters (which are sometimes chewed up by the family dog). Thus, the cousin-pair Susan and Harvey do not provide the same set of services (that is, the same service model) as Ann and Bill. In an analogous manner, a computer network may make

available multiple transport protocols, with each protocol offering a different service model to applications.

The possible services that Ann and Bill can provide are clearly constrained by the possible services that the postal service provides. For example, if the postal service doesn't provide a maximum bound on how long it can take to deliver mail between the two houses (for example, three days), then there is no way that Ann and Bill can guarantee a maximum delay for mail delivery between any of the cousin pairs. In a similar manner, the services that a transport protocol can provide are often constrained by the service model of the underlying network-layer protocol. If the network-layer protocol cannot provide delay or bandwidth guarantees for transport-layer segments sent between hosts, then the transport-layer protocol cannot provide delay or bandwidth guarantees for application messages sent between processes.

Nevertheless, certain services *can* be offered by a transport protocol even when the underlying network protocol doesn't offer the corresponding service at the network layer. For example, as we'll see in this chapter, a transport protocol can offer reliable data transfer service to an application even when the underlying network protocol is unreliable, that is, even when the network protocol loses, garbles, or duplicates packets. As another example (which we'll explore in [Chapter 8](#) when we discuss network security), a transport protocol can use encryption to guarantee that application messages are not read by intruders, even when the network layer cannot guarantee the confidentiality of transport-layer segments.

### 3.1.2 Overview of the Transport Layer in the Internet

Recall that the Internet makes two distinct transport-layer protocols available to the application layer. One of these protocols is **UDP** (User Datagram Protocol), which provides an unreliable, connectionless service to the invoking application. The second of these protocols is **TCP** (Transmission Control Protocol), which provides a reliable, connection-oriented service to the invoking application. When designing a network application, the application developer must specify one of these two transport protocols. As we saw in [Section 2.7](#), the application developer selects between UDP and TCP when creating sockets.

To simplify terminology, we refer to the transport-layer packet as a *segment*. We mention, however, that the Internet literature (for example, the RFCs) also refers to the transport-layer packet for TCP as a segment but often refers to the packet for UDP as a datagram. But this same Internet literature also uses the term *datagram* for the network-layer packet! For an introductory book on computer networking such as this, we believe that it is less confusing to refer to both TCP and UDP packets as segments, and reserve the term *datagram* for the network-layer packet.

Before proceeding with our brief introduction of UDP and TCP, it will be useful to say a few words about the Internet's network layer. (We'll learn about the network layer in detail in [Chapters 4 and 5](#).) The Internet's network-layer protocol has a name—IP, for Internet Protocol. IP provides logical communication between hosts. The IP service model is a **best-effort delivery service**. This means that IP makes its “best effort” to deliver segments between communicating hosts, *but it makes no guarantees*. In particular, it does not guarantee segment delivery, it does not guarantee orderly delivery of segments, and it does not guarantee the integrity of the data in the segments. For these reasons, IP is said to be an **unreliable service**. We also mention here that every host has at least one network-layer address, a so-called IP address. We'll examine IP addressing in detail in [Chapter 4](#); for this chapter we need only keep in mind that *each host has an IP address*.

Having taken a glimpse at the IP service model, let's now summarize the service models provided by UDP and TCP. The most fundamental responsibility of UDP and TCP is to extend IP's delivery service between two end systems to a delivery service between two processes running on the end systems. Extending host-to-host delivery to process-to-process delivery is called **transport-layer multiplexing** and **demultiplexing**. We'll discuss transport-layer multiplexing and demultiplexing in the next section. UDP and TCP also provide integrity checking by including error-detection fields in their segments' headers. These two minimal transport-layer services—process-to-process data delivery and error checking—are the only two services that UDP provides! In particular, like IP, UDP is an unreliable service—it does not guarantee that data sent by one process will arrive intact (or at all!) to the destination process. UDP is discussed in detail in [Section 3.3](#).

TCP, on the other hand, offers several additional services to applications. First and foremost, it provides **reliable data transfer**. Using flow control, sequence numbers, acknowledgments, and timers (techniques we'll explore in detail in this chapter), TCP ensures that data is delivered from sending process to receiving process, correctly and in order. TCP thus converts IP's unreliable service between end systems into a reliable data transport service between processes. TCP also provides **congestion control**. Congestion control is not so much a service provided to the invoking application as it is a service for the Internet as a whole, a service for the general good. Loosely speaking, TCP congestion control prevents any one TCP connection from swamping the links and routers between communicating hosts with an excessive amount of traffic. TCP strives to give each connection traversing a congested link an equal share of the link bandwidth. This is done by regulating the rate at which the sending sides of TCP connections can send traffic into the network. UDP traffic, on the other hand, is unregulated. An application using UDP transport can send at any rate it pleases, for as long as it pleases.

A protocol that provides reliable data transfer and congestion control is necessarily complex. We'll need several sections to cover the principles of reliable data transfer and congestion control, and additional sections to cover the TCP protocol itself. These topics are investigated in [Sections 3.4 through 3.8](#). The approach taken in this chapter is to alternate between basic principles and the TCP protocol. For example, we'll first discuss reliable data transfer in a general setting and then discuss how TCP

specifically provides reliable data transfer. Similarly, we'll first discuss congestion control in a general setting and then discuss how TCP performs congestion control. But before getting into all this good stuff, let's first look at transport-layer multiplexing and demultiplexing.

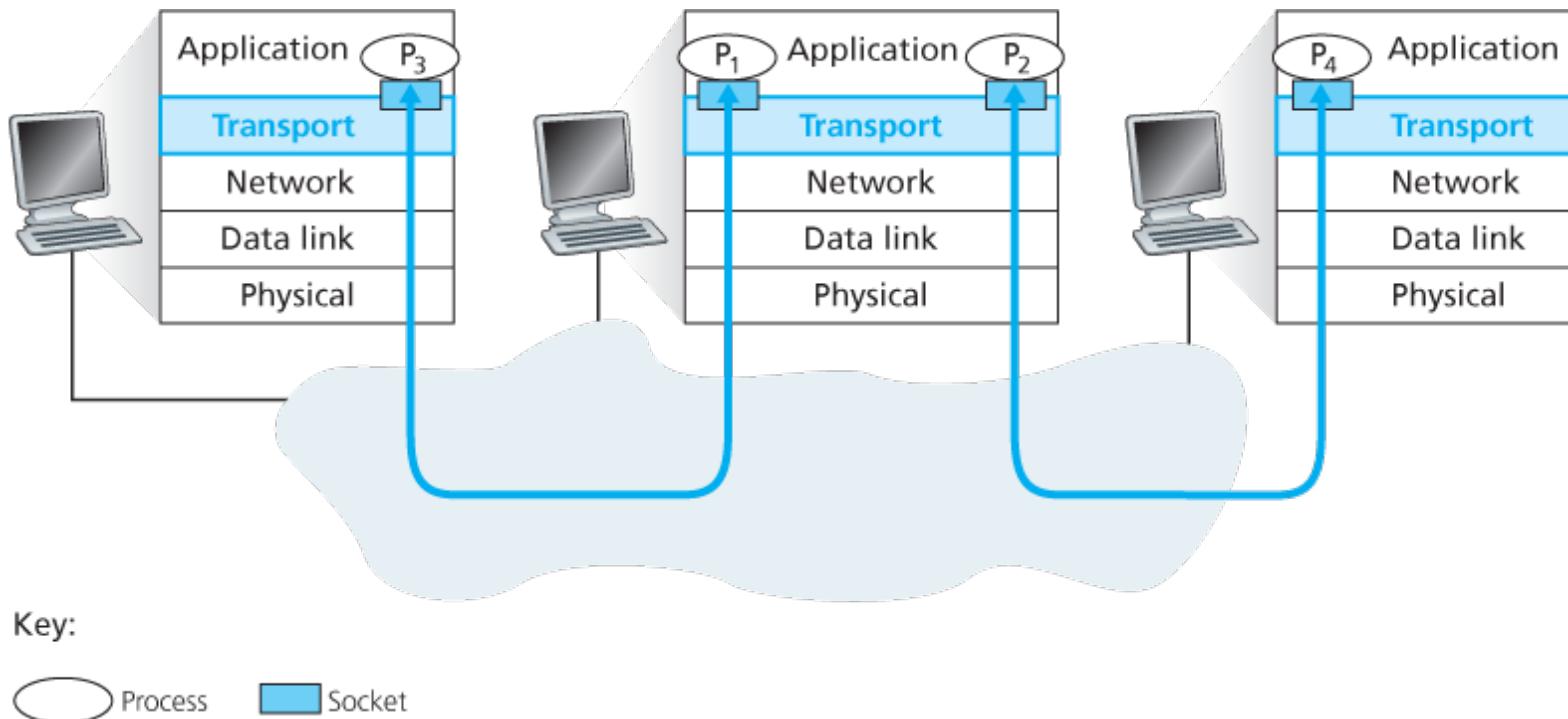
## 3.2 Multiplexing and Demultiplexing

In this section, we discuss transport-layer multiplexing and demultiplexing, that is, extending the host-to-host delivery service provided by the network layer to a process-to-process delivery service for applications running on the hosts. In order to keep the discussion concrete, we'll discuss this basic transport-layer service in the context of the Internet. We emphasize, however, that a multiplexing/demultiplexing service is needed for all computer networks.

At the destination host, the transport layer receives segments from the network layer just below. The transport layer has the responsibility of delivering the data in these segments to the appropriate application process running in the host. Let's take a look at an example. Suppose you are sitting in front of your computer, and you are downloading Web pages while running one FTP session and two Telnet sessions. You therefore have four network application processes running—two Telnet processes, one FTP process, and one HTTP process. When the transport layer in your computer receives data from the network layer below, it needs to direct the received data to one of these four processes. Let's now examine how this is done.

First recall from [Section 2.7](#) that a process (as part of a network application) can have one or more **sockets**, doors through which data passes from the network to the process and through which data passes from the process to the network. Thus, as shown in [Figure 3.2](#), the transport layer in the receiving host does not actually deliver data directly to a process, but instead to an intermediary socket. Because at any given time there can be more than one socket in the receiving host, each socket has a unique identifier. The format of the identifier depends on whether the socket is a UDP or a TCP socket, as we'll discuss shortly.

Now let's consider how a receiving host directs an incoming transport-layer segment to the appropriate socket. Each transport-layer segment has a set of fields in the segment for this purpose. At the receiving end, the transport layer examines these fields to identify the receiving socket and then directs the segment to that socket. This job of delivering the data in a transport-layer segment to the correct socket is called **demultiplexing**. The job of gathering data chunks at the source host from different sockets, encapsulating each data chunk with header information (that will later be used in demultiplexing) to create segments, and passing the segments to the network layer is called **multiplexing**. Note that the transport layer in the middle host

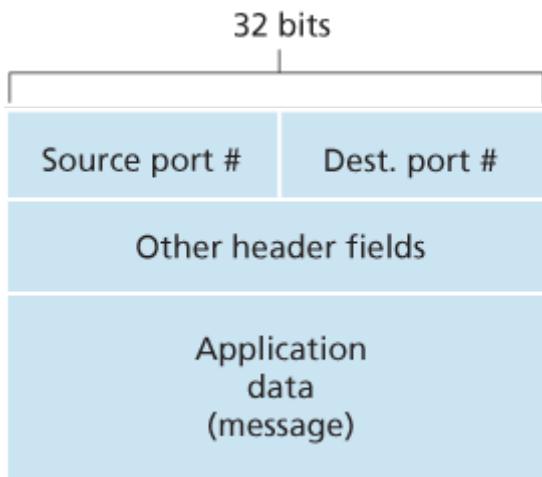


**Figure 3.2 Transport-layer multiplexing and demultiplexing**

in [Figure 3.2](#) must demultiplex segments arriving from the network layer below to either process P<sub>1</sub> or P<sub>2</sub> above; this is done by directing the arriving segment's data to the corresponding process's socket. The transport layer in the middle host must also gather outgoing data from these sockets, form transport-layer segments, and pass these segments down to the network layer. Although we have introduced multiplexing and demultiplexing in the context of the Internet transport protocols, it's important to realize that they are concerns whenever a single protocol at one layer (at the transport layer or elsewhere) is used by multiple protocols at the next higher layer.

To illustrate the demultiplexing job, recall the household analogy in the previous section. Each of the kids is identified by his or her name. When Bill receives a batch of mail from the mail carrier, he performs a demultiplexing operation by observing to whom the letters are addressed and then hand delivering the mail to his brothers and sisters. Ann performs a multiplexing operation when she collects letters from her brothers and sisters and gives the collected mail to the mail person.

Now that we understand the roles of transport-layer multiplexing and demultiplexing, let us examine how it is actually done in a host. From the discussion above, we know that transport-layer multiplexing requires (1) that sockets have unique identifiers, and (2) that each segment have special fields that indicate the socket to which the segment is to be delivered. These special fields, illustrated in [Figure 3.3](#), are the **source port number field** and the **destination port number field**. (The UDP and TCP segments have other fields as well, as discussed in the subsequent sections of this chapter.) Each port number is a 16-bit number, ranging from 0 to 65535. The port numbers ranging from 0 to 1023 are called **well-known port numbers** and are restricted, which means that they are reserved for use by well-known



**Figure 3.3 Source and destination port-number fields in a transport-layer segment**

application protocols such as HTTP (which uses port number 80) and FTP (which uses port number 21). The list of well-known port numbers is given in RFC 1700 and is updated at <http://www.iana.org> [RFC 3232]. When we develop a new application (such as the simple application developed in [Section 2.7](#)), we must assign the application a port number.

It should now be clear how the transport layer *could* implement the demultiplexing service: Each socket in the host could be assigned a port number, and when a segment arrives at the host, the transport layer examines the destination port number in the segment and directs the segment to the corresponding socket. The segment's data then passes through the socket into the attached process. As we'll see, this is basically how UDP does it. However, we'll also see that multiplexing/demultiplexing in TCP is yet more subtle.

#### *Connectionless Multiplexing and Demultiplexing*

Recall from [Section 2.7.1](#) that the Python program running in a host can create a UDP socket with the line

```
clientSocket = socket(AF_INET, SOCK_DGRAM)
```

When a UDP socket is created in this manner, the transport layer automatically assigns a port number to the socket. In particular, the transport layer assigns a port number in the range 1024 to 65535 that is currently not being used by any other UDP port in the host. Alternatively, we can add a line into our Python program after we create the socket to associate a specific port number (say, 19157) to this UDP socket via the socket **bind()** method:

```
clientSocket.bind(('', 19157))
```

If the application developer writing the code were implementing the server side of a “well-known protocol,” then the developer would have to assign the corresponding well-known port number. Typically, the client side of the application lets the transport layer automatically (and transparently) assign the port number, whereas the server side of the application assigns a specific port number.

With port numbers assigned to UDP sockets, we can now precisely describe UDP multiplexing/demultiplexing. Suppose a process in Host A, with UDP port 19157, wants to send a chunk of application data to a process with UDP port 46428 in Host B. The transport layer in Host A creates a transport-layer segment that includes the application data, the source port number (19157), the destination port number (46428), and two other values (which will be discussed later, but are unimportant for the current discussion). The transport layer then passes the resulting segment to the network layer. The network layer encapsulates the segment in an IP datagram and makes a best-effort attempt to deliver the segment to the receiving host. If the segment arrives at the receiving Host B, the transport layer at the receiving host examines the destination port number in the segment (46428) and delivers the segment to its socket identified by port 46428. Note that Host B could be running multiple processes, each with its own UDP socket and associated port number. As UDP segments arrive from the network, Host B directs (demultiplexes) each segment to the appropriate socket by examining the segment’s destination port number.

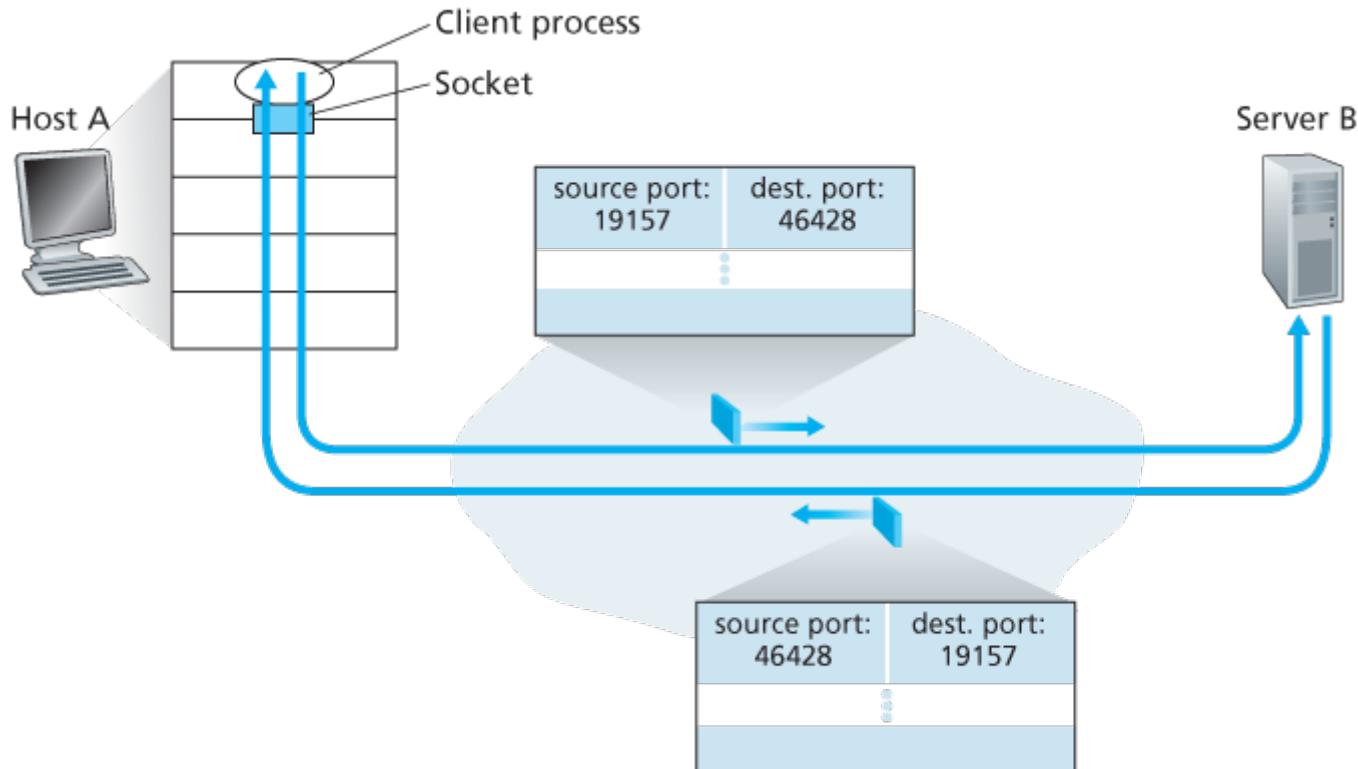
It is important to note that a UDP socket is fully identified by a two-tuple consisting of a destination IP address and a destination port number. As a consequence, if two UDP segments have different source IP addresses and/or source port numbers, but have the same *destination* IP address and *destination* port number, then the two segments will be directed to the same destination process via the same destination socket.

You may be wondering now, what is the purpose of the source port number? As shown in [Figure 3.4](#), in the A-to-B segment the source port number serves as part of a “return address”—when B wants to send a segment back to A, the destination port in the B-to-A segment will take its value from the source port value of the A-to-B segment. (The complete return address is A’s IP address and the source port number.) As an example, recall the UDP server program studied in [Section 2.7](#). In [`UDPServer.py`](#), the server uses the `recvfrom()` method to extract the client-side (source) port number from the segment it receives from the client; it then sends a new segment to the client, with the extracted source port number serving as the destination port number in this new segment.

### *Connection-Oriented Multiplexing and Demultiplexing*

In order to understand TCP demultiplexing, we have to take a close look at TCP sockets and TCP connection establishment. One subtle difference between a TCP socket and a UDP socket is that a TCP

socket is identified by a four-tuple: (source IP address, source port number, destination IP address, destination port number). Thus, when a TCP segment arrives from the network to a host, the host uses all four values to direct (demultiplex) the segment to the appropriate socket.



**Figure 3.4 The inversion of source and destination port numbers**

In particular, and in contrast with UDP, two arriving TCP segments with different source IP addresses or source port numbers will (with the exception of a TCP segment carrying the original connection-establishment request) be directed to two different sockets. To gain further insight, let's reconsider the TCP client-server programming example in [Section 2.7.2](#):

- The TCP server application has a “welcoming socket,” that waits for connection-establishment requests from TCP clients (see [Figure 2.29](#)) on port number 12000.
- The TCP client creates a socket and sends a connection establishment request segment with the lines:

```
clientSocket = socket (AF_INET, SOCK_STREAM)
clientSocket.connect ((serverName, 12000))
```

- A connection-establishment request is nothing more than a TCP segment with destination port number 12000 and a special connection-establishment bit set in the TCP header (discussed in [Section 3.5](#)). The segment also includes a source port number that was chosen by the client.
- When the host operating system of the computer running the server process receives the incoming

connection-request segment with destination port 12000, it locates the server process that is waiting to accept a connection on port number 12000. The server process then creates a new socket:

```
connectionSocket, addr = serverSocket.accept()
```

- Also, the transport layer at the server notes the following four values in the connection-request segment: (1) the source port number in the segment, (2) the IP address of the source host, (3) the destination port number in the segment, and (4) its own IP address. The newly created connection socket is identified by these four values; all subsequently arriving segments whose source port, source IP address, destination port, and destination IP address match these four values will be demultiplexed to this socket. With the TCP connection now in place, the client and server can now send data to each other.

The server host may support many simultaneous TCP connection sockets, with each socket attached to a process, and with each socket identified by its own four-tuple. When a TCP segment arrives at the host, all four fields (source IP address, source port, destination IP address, destination port) are used to direct (demultiplex) the segment to the appropriate socket.

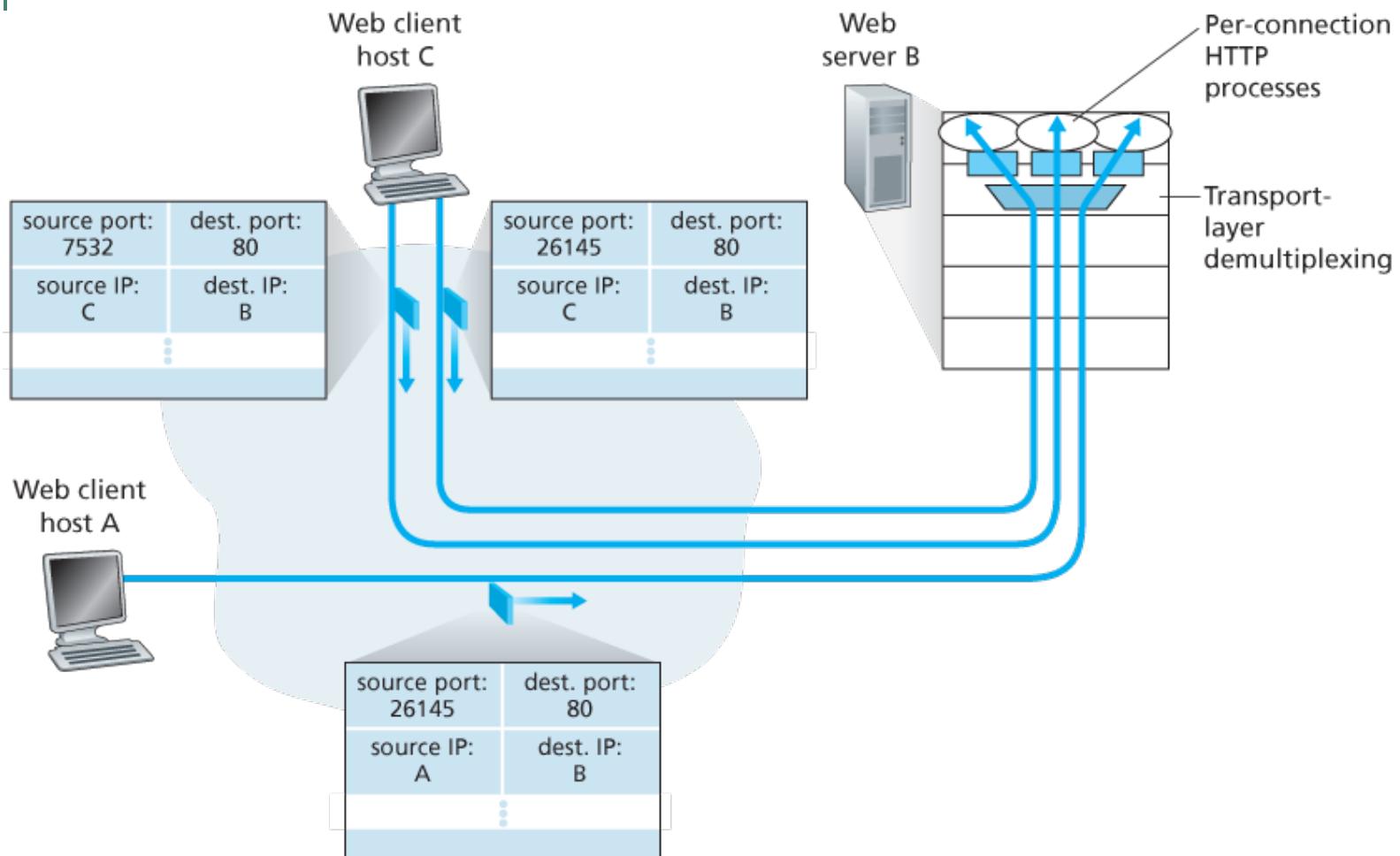
## FOCUS ON SECURITY

### Port Scanning

We've seen that a server process waits patiently on an open port for contact by a remote client. Some ports are reserved for well-known applications (e.g., Web, FTP, DNS, and SMTP servers); other ports are used by convention by popular applications (e.g., the Microsoft 2000 SQL server listens for requests on UDP port 1434). Thus, if we determine that a port is open on a host, we may be able to map that port to a specific application running on the host. This is very useful for system administrators, who are often interested in knowing which network applications are running on the hosts in their networks. But attackers, in order to "case the joint," also want to know which ports are open on target hosts. If a host is found to be running an application with a known security flaw (e.g., a SQL server listening on port 1434 was subject to a buffer overflow, allowing a remote user to execute arbitrary code on the vulnerable host, a flaw exploited by the Slammer worm [\[CERT 2003–04\]](#)), then that host is ripe for attack.

Determining which applications are listening on which ports is a relatively easy task. Indeed there are a number of public domain programs, called port scanners, that do just that. Perhaps the most widely used of these is nmap, freely available at <http://nmap.org> and included in most Linux distributions. For TCP, nmap sequentially scans ports, looking for ports that are accepting TCP connections. For UDP, nmap again sequentially scans ports, looking for UDP ports that respond to transmitted UDP segments. In both cases, nmap returns a list of open, closed, or unreachable ports. A host running nmap can attempt to scan any target host *anywhere* in the

Internet. We'll revisit nmap in [Section 3.5.6](#), when we discuss TCP connection management.



**Figure 3.5 Two clients, using the same destination port number (80) to communicate with the same Web server application**

The situation is illustrated in [Figure 3.5](#), in which Host C initiates two HTTP sessions to server B, and Host A initiates one HTTP session to B. Hosts A and C and server B each have their own unique IP address—A, C, and B, respectively. Host C assigns two different source port numbers (26145 and 7532) to its two HTTP connections. Because Host A is choosing source port numbers independently of C, it might also assign a source port of 26145 to its HTTP connection. But this is not a problem—server B will still be able to correctly demultiplex the two connections having the same source port number, since the two connections have different source IP addresses.

### Web Servers and TCP

Before closing this discussion, it's instructive to say a few additional words about Web servers and how they use port numbers. Consider a host running a Web server, such as an Apache Web server, on port 80. When clients (for example, browsers) send segments to the server, *all* segments will have destination port 80. In particular, both the initial connection-establishment segments and the segments carrying HTTP request messages will have destination port 80. As we have just described, the server distinguishes the segments from the different clients using source IP addresses and source port

numbers.

**Figure 3.5** shows a Web server that spawns a new process for each connection. As shown in **Figure 3.5**, each of these processes has its own connection socket through which HTTP requests arrive and HTTP responses are sent. We mention, however, that there is not always a one-to-one correspondence between connection sockets and processes. In fact, today's high-performing Web servers often use only one process, and create a new thread with a new connection socket for each new client connection. (A thread can be viewed as a lightweight subprocess.) If you did the first programming assignment in **Chapter 2**, you built a Web server that does just this. For such a server, at any given time there may be many connection sockets (with different identifiers) attached to the same process.

If the client and server are using persistent HTTP, then throughout the duration of the persistent connection the client and server exchange HTTP messages via the same server socket. However, if the client and server use non-persistent HTTP, then a new TCP connection is created and closed for every request/response, and hence a new socket is created and later closed for every request/response. This frequent creating and closing of sockets can severely impact the performance of a busy Web server (although a number of operating system tricks can be used to mitigate the problem). Readers interested in the operating system issues surrounding persistent and non-persistent HTTP are encouraged to see [\[Nielsen 1997; Nahum 2002\]](#).

Now that we've discussed transport-layer multiplexing and demultiplexing, let's move on and discuss one of the Internet's transport protocols, UDP. In the next section we'll see that UDP adds little more to the network-layer protocol than a multiplexing/demultiplexing service.

### 3.3 Connectionless Transport: UDP

In this section, we'll take a close look at UDP, how it works, and what it does. We encourage you to refer back to [Section 2.1](#), which includes an overview of the UDP service model, and to [Section 2.7.1](#), which discusses socket programming using UDP.

To motivate our discussion about UDP, suppose you were interested in designing a no-frills, bare-bones transport protocol. How might you go about doing this? You might first consider using a vacuous transport protocol. In particular, on the sending side, you might consider taking the messages from the application process and passing them directly to the network layer; and on the receiving side, you might consider taking the messages arriving from the network layer and passing them directly to the application process. But as we learned in the previous section, we have to do a little more than nothing! At the very least, the transport layer has to provide a multiplexing/demultiplexing service in order to pass data between the network layer and the correct application-level process.

UDP, defined in [\[RFC 768\]](#), does just about as little as a transport protocol can do. Aside from the multiplexing/demultiplexing function and some light error checking, it adds nothing to IP. In fact, if the application developer chooses UDP instead of TCP, then the application is almost directly talking with IP. UDP takes messages from the application process, attaches source and destination port number fields for the multiplexing/demultiplexing service, adds two other small fields, and passes the resulting segment to the network layer. The network layer encapsulates the transport-layer segment into an IP datagram and then makes a best-effort attempt to deliver the segment to the receiving host. If the segment arrives at the receiving host, UDP uses the destination port number to deliver the segment's data to the correct application process. Note that with UDP there is no handshaking between sending and receiving transport-layer entities before sending a segment. For this reason, UDP is said to be *connectionless*.

DNS is an example of an application-layer protocol that typically uses UDP. When the DNS application in a host wants to make a query, it constructs a DNS query message and passes the message to UDP. Without performing any handshaking with the UDP entity running on the destination end system, the host-side UDP adds header fields to the message and passes the resulting segment to the network layer. The network layer encapsulates the UDP segment into a datagram and sends the datagram to a name server. The DNS application at the querying host then waits for a reply to its query. If it doesn't receive a reply (possibly because the underlying network lost the query or the reply), it might try resending the query, try sending the query to another name server, or inform the invoking application that it can't get a reply.

Now you might be wondering why an application developer would ever choose to build an application over UDP rather than over TCP. Isn't TCP always preferable, since TCP provides a reliable data transfer service, while UDP does not? The answer is no, as some applications are better suited for UDP for the following reasons:

- **Finer application-level control over what data is sent, and when.** Under UDP, as soon as an application process passes data to UDP, UDP will package the data inside a UDP segment and immediately pass the segment to the network layer. TCP, on the other hand, has a congestion-control mechanism that throttles the transport-layer TCP sender when one or more links between the source and destination hosts become excessively congested. TCP will also continue to resend a segment until the receipt of the segment has been acknowledged by the destination, regardless of how long reliable delivery takes. Since real-time applications often require a minimum sending rate, do not want to overly delay segment transmission, and can tolerate some data loss, TCP's service model is not particularly well matched to these applications' needs. As discussed below, these applications can use UDP and implement, as part of the application, any additional functionality that is needed beyond UDP's no-frills segment-delivery service.
- **No connection establishment.** As we'll discuss later, TCP uses a three-way handshake before it starts to transfer data. UDP just blasts away without any formal preliminaries. Thus UDP does not introduce any delay to establish a connection. This is probably the principal reason why DNS runs over UDP rather than TCP—DNS would be much slower if it ran over TCP. HTTP uses TCP rather than UDP, since reliability is critical for Web pages with text. But, as we briefly discussed in [Section 2.2](#), the TCP connection-establishment delay in HTTP is an important contributor to the delays associated with downloading Web documents. Indeed, the QUIC protocol (Quick UDP Internet Connection, [\[Iyengar 2015\]](#)), used in Google's Chrome browser, uses UDP as its underlying transport protocol and implements reliability in an application-layer protocol on top of UDP.
- **No connection state.** TCP maintains connection state in the end systems. This connection state includes receive and send buffers, congestion-control parameters, and sequence and acknowledgment number parameters. We will see in [Section 3.5](#) that this state information is needed to implement TCP's reliable data transfer service and to provide congestion control. UDP, on the other hand, does not maintain connection state and does not track any of these parameters. For this reason, a server devoted to a particular application can typically support many more active clients when the application runs over UDP rather than TCP.
- **Small packet header overhead.** The TCP segment has 20 bytes of header overhead in every segment, whereas UDP has only 8 bytes of overhead.

[Figure 3.6](#) lists popular Internet applications and the transport protocols that they use. As we expect, e-mail, remote terminal access, the Web, and file transfer run over TCP—all these applications need the reliable data transfer service of TCP. Nevertheless, many important applications run over UDP rather than TCP. For example, UDP is used to carry network management (SNMP; see [Section 5.7](#)) data. UDP is preferred to TCP in this case, since network management applications must often run when the

network is in a stressed state—precisely when reliable, congestion-controlled data transfer is difficult to achieve. Also, as we mentioned earlier, DNS runs over UDP, thereby avoiding TCP’s connection-establishment delays.

As shown in [Figure 3.6](#), both UDP and TCP are sometimes used today with multimedia applications, such as Internet phone, real-time video conferencing, and streaming of stored audio and video. We’ll take a close look at these applications in [Chapter 9](#). We just mention now that all of these applications can tolerate a small amount of packet loss, so that reliable data transfer is not absolutely critical for the application’s success. Furthermore, real-time applications, like Internet phone and video conferencing, react very poorly to TCP’s congestion control. For these reasons, developers of multimedia applications may choose to run their applications over UDP instead of TCP. When packet loss rates are low, and with some organizations blocking UDP traffic for security reasons (see [Chapter 8](#)), TCP becomes an increasingly attractive protocol for streaming media transport.

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	typically proprietary	UDP or TCP
Internet telephony	typically proprietary	UDP or TCP
Network management	SNMP	Typically UDP
Name translation	DNS	Typically UDP

**Figure 3.6 Popular Internet applications and their underlying transport protocols**

Although commonly done today, running multimedia applications over UDP is controversial. As we mentioned above, UDP has no congestion control. But congestion control is needed to prevent the network from entering a congested state in which very little useful work is done. If everyone were to start streaming high-bit-rate video without using any congestion control, there would be so much packet overflow at routers that very few UDP packets would successfully traverse the source-to-destination path. Moreover, the high loss rates induced by the uncontrolled UDP senders would cause the TCP senders (which, as we’ll see, *do* decrease their sending rates in the face of congestion) to dramatically decrease their rates. Thus, the lack of congestion control in UDP can result in high loss rates between a UDP sender and receiver, and the crowding out of TCP sessions—a potentially serious problem [[Floyd](#)

[1999](#). Many researchers have proposed new mechanisms to force all sources, including UDP sources, to perform adaptive congestion control [[Mahdavi 1997](#); [Floyd 2000](#); [Kohler 2006](#): [RFC 4340](#)].

Before discussing the UDP segment structure, we mention that it *is* possible for an application to have reliable data transfer when using UDP. This can be done if reliability is built into the application itself (for example, by adding acknowledgment and retransmission mechanisms, such as those we'll study in the next section). We mentioned earlier that the QUIC protocol [[Iyengar 2015](#)] used in Google's Chrome browser implements reliability in an application-layer protocol on top of UDP. But this is a nontrivial task that would keep an application developer busy debugging for a long time. Nevertheless, building reliability directly into the application allows the application to "have its cake and eat it too. That is, application processes can communicate reliably without being subjected to the transmission-rate constraints imposed by TCP's congestion-control mechanism.

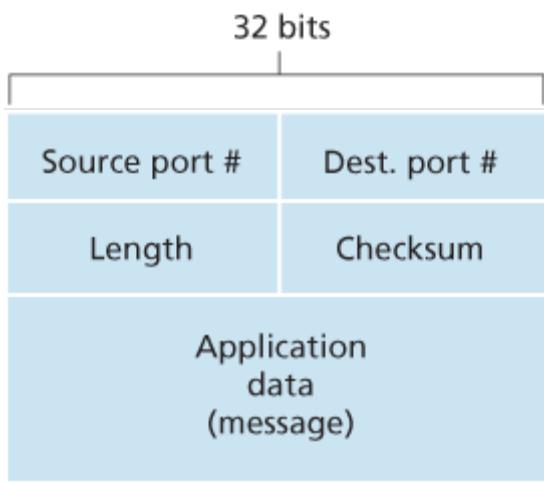
### 3.3.1 UDP Segment Structure

The UDP segment structure, shown in [Figure 3.7](#), is defined in RFC 768. The application data occupies the data field of the UDP segment. For example, for DNS, the data field contains either a query message or a response message. For a streaming audio application, audio samples fill the data field. The UDP header has only four fields, each consisting of two bytes. As discussed in the previous section, the port numbers allow the destination host to pass the application data to the correct process running on the destination end system (that is, to perform the demultiplexing function). The length field specifies the number of bytes in the UDP segment (header plus data). An explicit length value is needed since the size of the data field may differ from one UDP segment to the next. The checksum is used by the receiving host to check whether errors have been introduced into the segment. In truth, the checksum is also calculated over a few of the fields in the IP header in addition to the UDP segment. But we ignore this detail in order to see the forest through the trees. We'll discuss the checksum calculation below.

Basic principles of error detection are described in [Section 6.2](#). The length field specifies the length of the UDP segment, including the header, in bytes.

### 3.3.2 UDP Checksum

The UDP checksum provides for error detection. That is, the checksum is used to determine whether bits within the UDP segment have been altered (for example, by noise in the links or while stored in a router) as it moved from source to destination.



**Figure 3.7 UDP segment structure**

UDP at the sender side performs the 1s complement of the sum of all the 16-bit words in the segment, with any overflow encountered during the sum being wrapped around. This result is put in the checksum field of the UDP segment. Here we give a simple example of the checksum calculation. You can find details about efficient implementation of the calculation in RFC 1071 and performance over real data in [\[Stone 1998; Stone 2000\]](#). As an example, suppose that we have the following three 16-bit words:

0110011001100000

0101010101010101

100011100001100

The sum of first two of these 16-bit words is

0110011001100000

0101010101010101

1011101110110101

Adding the third word to the above sum gives

1011101110110101

100011100001100

0100101011000010

Note that this last addition had overflow, which was wrapped around. The 1s complement is obtained by converting all the 0s to 1s and converting all the 1s to 0s. Thus the 1s complement of the sum 0100101011000010 is 101101010011101, which becomes the checksum. At the receiver, all four 16-

bit words are added, including the checksum. If no errors are introduced into the packet, then clearly the sum at the receiver will be 1111111111111111. If one of the bits is a 0, then we know that errors have been introduced into the packet.

You may wonder why UDP provides a checksum in the first place, as many link-layer protocols (including the popular Ethernet protocol) also provide error checking. The reason is that there is no guarantee that all the links between source and destination provide error checking; that is, one of the links may use a link-layer protocol that does not provide error checking. Furthermore, even if segments are correctly transferred across a link, it's possible that bit errors could be introduced when a segment is stored in a router's memory. Given that neither link-by-link reliability nor in-memory error detection is guaranteed, UDP must provide error detection at the transport layer, *on an end-end basis*, if the end-end data transfer service is to provide error detection. This is an example of the celebrated **end-end principle** in system design [\[Saltzer 1984\]](#), which states that since certain functionality (error detection, in this case) must be implemented on an end-end basis: "functions placed at the lower levels may be redundant or of little value when compared to the cost of providing them at the higher level."

Because IP is supposed to run over just about any layer-2 protocol, it is useful for the transport layer to provide error checking as a safety measure. Although UDP provides error checking, it does not do anything to recover from an error. Some implementations of UDP simply discard the damaged segment; others pass the damaged segment to the application with a warning.

That wraps up our discussion of UDP. We will soon see that TCP offers reliable data transfer to its applications as well as other services that UDP doesn't offer. Naturally, TCP is also more complex than UDP. Before discussing TCP, however, it will be useful to step back and first discuss the underlying principles of reliable data transfer.

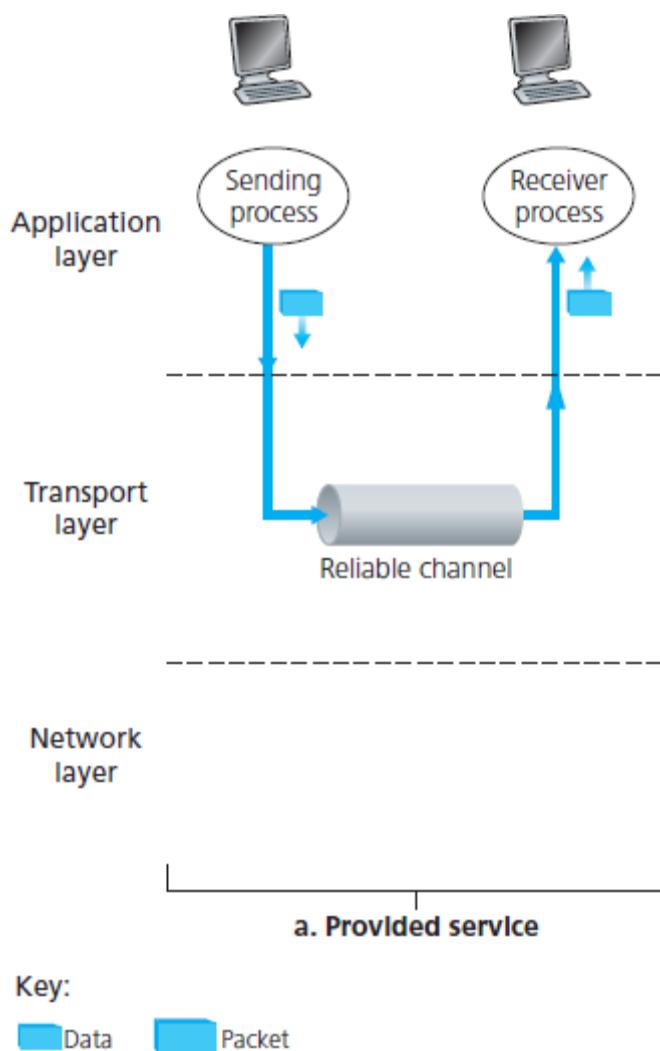
## 3.4 Principles of Reliable Data Transfer

In this section, we consider the problem of reliable data transfer in a general context. This is appropriate since the problem of implementing reliable data transfer occurs not only at the transport layer, but also at the link layer and the application layer as well. The general problem is thus of central importance to networking. Indeed, if one had to identify a “top-ten” list of fundamentally important problems in all of networking, this would be a candidate to lead the list. In the next section we’ll examine TCP and show, in particular, that TCP exploits many of the principles that we are about to describe.

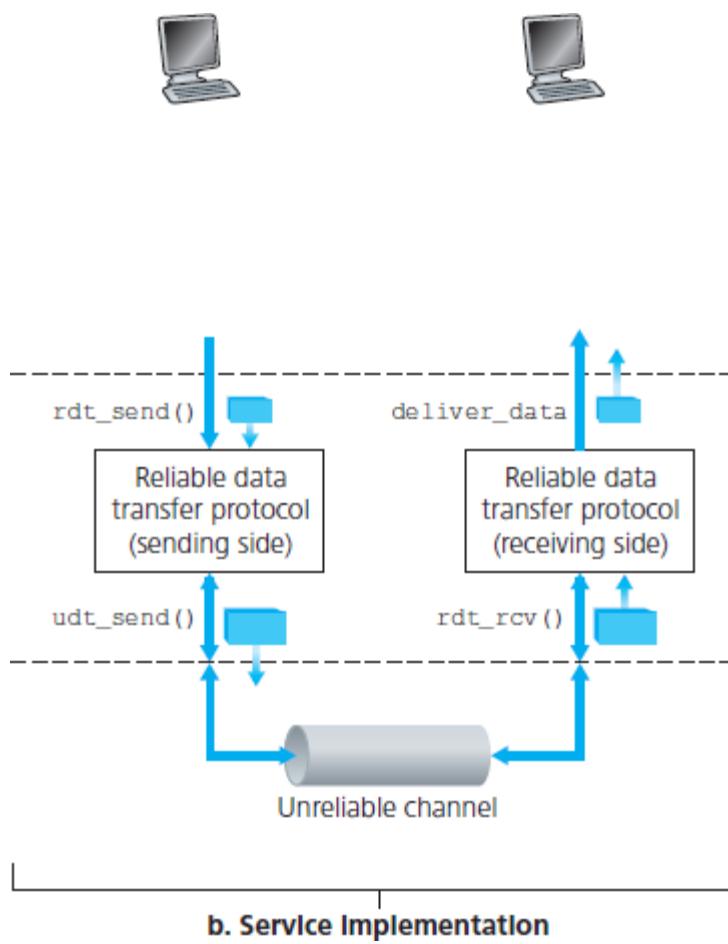
**Figure 3.8** illustrates the framework for our study of reliable data transfer. The service abstraction provided to the upper-layer entities is that of a reliable channel through which data can be transferred. With a reliable channel, no transferred data bits are corrupted (flipped from 0 to 1, or vice versa) or lost, and all are delivered in the order in which they were sent. This is precisely the service model offered by TCP to the Internet applications that invoke it.

It is the responsibility of a **reliable data transfer protocol** to implement this service abstraction. This task is made difficult by the fact that the layer *below* the reliable data transfer protocol may be unreliable. For example, TCP is a reliable data transfer protocol that is implemented on top of an unreliable (IP) end-to-end network layer. More generally, the layer beneath the two reliably communicating end points might consist of a single physical link (as in the case of a link-level data transfer protocol) or a global internetwork (as in the case of a transport-level protocol). For our purposes, however, we can view this lower layer simply as an unreliable point-to-point channel.

In this section, we will incrementally develop the sender and receiver sides of a reliable data transfer protocol, considering increasingly complex models of the underlying channel. For example, we’ll consider what protocol mechanisms are



**Figure 3.8 Reliable data transfer: Service model and service implementation**



needed when the underlying channel can corrupt bits or lose entire packets. One assumption we'll adopt throughout our discussion here is that packets will be delivered in the order in which they were sent, with some packets possibly being lost; that is, the underlying channel will not reorder packets. **Figure 3.8(b)** illustrates the interfaces for our data transfer protocol. The sending side of the data transfer protocol will be invoked from above by a call to `rdt_send()`. It will pass the data to be delivered to the upper layer at the receiving side. (Here `rdt` stands for *reliable data transfer* protocol and `_send` indicates that the sending side of `rdt` is being called. The first step in developing any protocol is to choose a good name!) On the receiving side, `rdt_rcv()` will be called when a packet arrives from the receiving side of the channel. When the `rdt` protocol wants to deliver data to the upper layer, it will do so by calling `deliver_data()`. In the following we use the terminology “packet” rather than transport-layer “segment.” Because the theory developed in this section applies to computer networks in general and not just to the Internet transport layer, the generic term “packet” is perhaps more appropriate here.

In this section we consider only the case of **unidirectional data transfer**, that is, data transfer from the sending to the receiving side. The case of reliable **bidirectional** (that is, full-duplex) **data transfer** is conceptually no more difficult but considerably more tedious to explain. Although we consider only unidirectional data transfer, it is important to note that the sending and receiving sides of our protocol will nonetheless need to transmit packets in *both* directions, as indicated in **Figure 3.8**. We will see shortly that, in addition to exchanging packets containing the data to be transferred, the sending and receiving sides of `rdt` will also need to exchange control packets back and forth. Both the send and receive sides of `rdt` send packets to the other side by a call to `udt_send()` (where `udt` stands for *unreliable data transfer*).

### 3.4.1 Building a Reliable Data Transfer Protocol

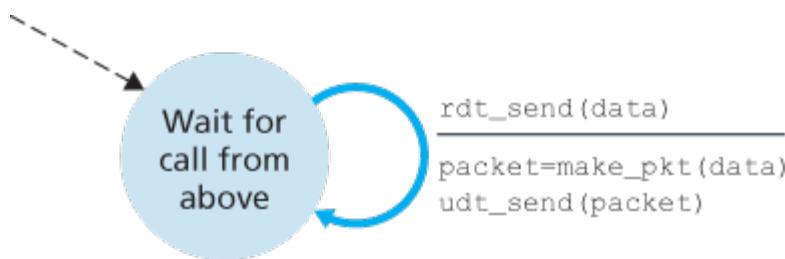
We now step through a series of protocols, each one becoming more complex, arriving at a flawless, reliable data transfer protocol.

*Reliable Data Transfer over a Perfectly Reliable Channel:* `rdt1.0`

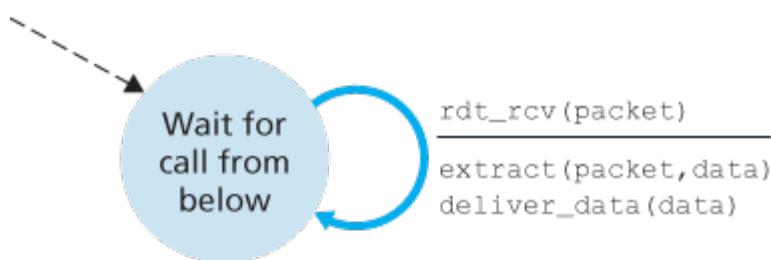
We first consider the simplest case, in which the underlying channel is completely reliable. The protocol itself, which we'll call `rdt1.0`, is trivial. The **finite-state machine (FSM)** definitions for the `rdt1.0` sender and receiver are shown in **Figure 3.9**. The FSM in **Figure 3.9(a)** defines the operation of the sender, while the FSM in **Figure 3.9(b)** defines the operation of the receiver. It is important to note that there are *separate* FSMs for the sender and for the receiver. The sender and receiver FSMs in **Figure 3.9** each have just one state. The arrows in the FSM description indicate the transition of the protocol from one state to another. (Since each FSM in **Figure 3.9** has just one state, a transition is necessarily from the one state back to itself; we'll see more complicated state diagrams shortly.) The event causing

the transition is shown above the horizontal line labeling the transition, and the actions taken when the event occurs are shown below the horizontal line. When no action is taken on an event, or no event occurs and an action is taken, we'll use the symbol  $\Lambda$  below or above the horizontal, respectively, to explicitly denote the lack of an action or event. The initial state of the FSM is indicated by the dashed arrow. Although the FSMs in **Figure 3.9** have but one state, the FSMs we will see shortly have multiple states, so it will be important to identify the initial state of each FSM.

The sending side of `rdt` simply accepts data from the upper layer via the `rdt_send(data)` event, creates a packet containing the data (via the action `make_pkt(data)`) and sends the packet into the channel. In practice, the `rdt_send(data)` event would result from a procedure call (for example, to `rdt_send()`) by the upper-layer application.



a. **rdt1.0: sending side**



b. **rdt1.0: receiving side**

**Figure 3.9 `rdt1.0` – A protocol for a completely reliable channel**

On the receiving side, `rdt` receives a packet from the underlying channel via the `rdt_rcv(packet)` event, removes the data from the packet (via the action `extract(packet, data)`) and passes the data up to the upper layer (via the action `deliver_data(data)`). In practice, the `rdt_rcv(packet)` event would result from a procedure call (for example, to `rdt_rcv()`) from the lower-layer protocol.

In this simple protocol, there is no difference between a unit of data and a packet. Also, all packet flow is from the sender to receiver; with a perfectly reliable channel there is no need for the receiver side to provide any feedback to the sender since nothing can go wrong! Note that we have also assumed that

the receiver is able to receive data as fast as the sender happens to send data. Thus, there is no need for the receiver to ask the sender to slow down!

### *Reliable Data Transfer over a Channel with Bit Errors: `rdt2.0`*

A more realistic model of the underlying channel is one in which bits in a packet may be corrupted. Such bit errors typically occur in the physical components of a network as a packet is transmitted, propagates, or is buffered. We'll continue to assume for the moment that all transmitted packets are received (although their bits may be corrupted) in the order in which they were sent.

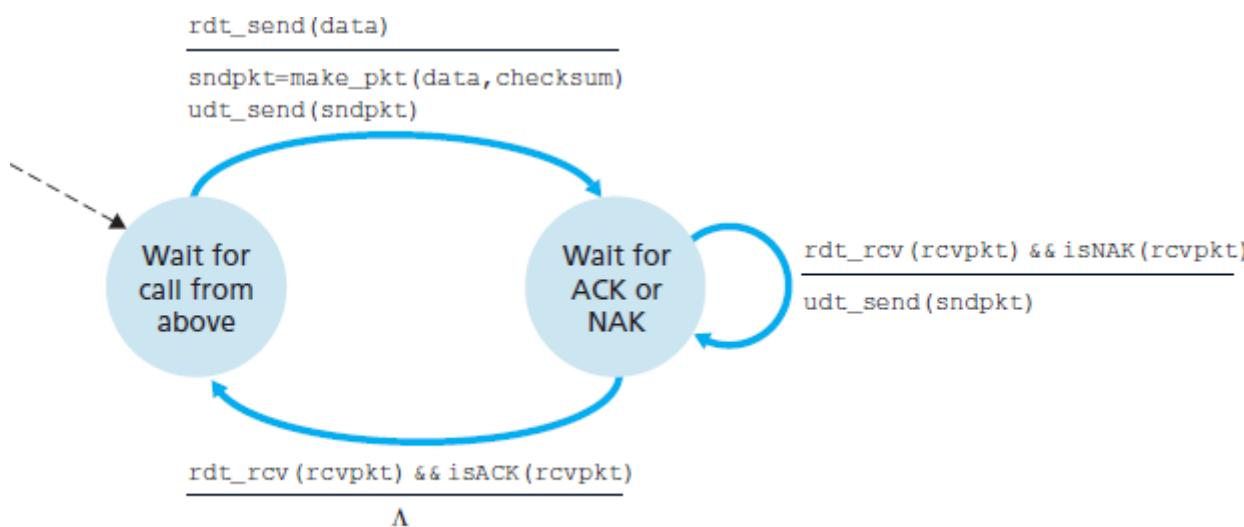
Before developing a protocol for reliably communicating over such a channel, first consider how people might deal with such a situation. Consider how you yourself might dictate a long message over the phone. In a typical scenario, the message taker might say "OK" after each sentence has been heard, understood, and recorded. If the message taker hears a garbled sentence, you're asked to repeat the garbled sentence. This message-dictation protocol uses both **positive acknowledgments** ("OK") and **negative acknowledgments** ("Please repeat that."). These control messages allow the receiver to let the sender know what has been received correctly, and what has been received in error and thus requires repeating. In a computer network setting, reliable data transfer protocols based on such retransmission are known as **ARQ (Automatic Repeat reQuest) protocols**.

Fundamentally, three additional protocol capabilities are required in ARQ protocols to handle the presence of bit errors:

- **Error detection.** First, a mechanism is needed to allow the receiver to detect when bit errors have occurred. Recall from the previous section that UDP uses the Internet checksum field for exactly this purpose. In [Chapter 6](#) we'll examine error-detection and -correction techniques in greater detail; these techniques allow the receiver to detect and possibly correct packet bit errors. For now, we need only know that these techniques require that extra bits (beyond the bits of original data to be transferred) be sent from the sender to the receiver; these bits will be gathered into the packet checksum field of the `rdt2.0` data packet.
- **Receiver feedback.** Since the sender and receiver are typically executing on different end systems, possibly separated by thousands of miles, the only way for the sender to learn of the receiver's view of the world (in this case, whether or not a packet was received correctly) is for the receiver to provide explicit feedback to the sender. The positive (ACK) and negative (NAK) acknowledgment replies in the message-dictation scenario are examples of such feedback. Our `rdt2.0` protocol will similarly send ACK and NAK packets back from the receiver to the sender. In principle, these packets need only be one bit long; for example, a 0 value could indicate a NAK and a value of 1 could indicate an ACK.
- **Retransmission.** A packet that is received in error at the receiver will be retransmitted by the sender.

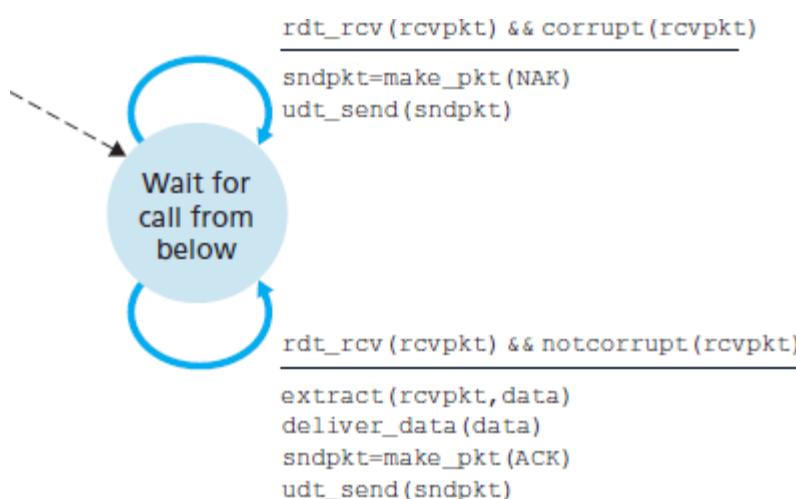
**Figure 3.10** shows the FSM representation of *rdt2.0*, a data transfer protocol employing error detection, positive acknowledgments, and negative acknowledgments.

The send side of *rdt2.0* has two states. In the leftmost state, the send-side protocol is waiting for data to be passed down from the upper layer. When the *rdt\_send(data)* event occurs, the sender will create a packet (*sndpkt*) containing the data to be sent, along with a packet checksum (for example, as discussed in **Section 3.3.2** for the case of a UDP segment), and then send the packet via the *udt\_send(sndpkt)* operation. In the rightmost state, the sender protocol is waiting for an ACK or a NAK packet from the receiver. If an ACK packet is received



a. **rdt2.0: sending side**

**Figure 3.10 *rdt2.0* – A protocol for a channel with bit errors**



b. **rdt2.0: receiving side**

(the notation *rdt\_recv(rcvpkt) && isACK(rcvpkt)* in **Figure 3.10** corresponds to this event), the sender knows that the most recently transmitted packet has been received correctly and thus the protocol returns to the state of waiting for data from the upper layer. If a NAK is received, the protocol retransmits the last packet and waits for an ACK or NAK to be returned by the receiver in response to

the retransmitted data packet. It is important to note that when the sender is in the wait-for-ACK-or-NAK state, it *cannot* get more data from the upper layer; that is, the `rdt_send()` event can not occur; that will happen only after the sender receives an ACK and leaves this state. Thus, the sender will not send a new piece of data until it is sure that the receiver has correctly received the current packet. Because of this behavior, protocols such as `rdt2.0` are known as **stop-and-wait** protocols.

The receiver-side FSM for `rdt2.0` still has a single state. On packet arrival, the receiver replies with either an ACK or a NAK, depending on whether or not the received packet is corrupted. In **Figure 3.10**, the notation `rdt_rcv(rcvpkt) && corrupt(rcvpkt)` corresponds to the event in which a packet is received and is found to be in error.

Protocol `rdt2.0` may look as if it works but, unfortunately, it has a fatal flaw. In particular, we haven't accounted for the possibility that the ACK or NAK packet could be corrupted! (Before proceeding on, you should think about how this problem may be fixed.) Unfortunately, our slight oversight is not as innocuous as it may seem. Minimally, we will need to add checksum bits to ACK/NAK packets in order to detect such errors. The more difficult question is how the protocol should recover from errors in ACK or NAK packets. The difficulty here is that if an ACK or NAK is corrupted, the sender has no way of knowing whether or not the receiver has correctly received the last piece of transmitted data.

Consider three possibilities for handling corrupted ACKs or NAKs:

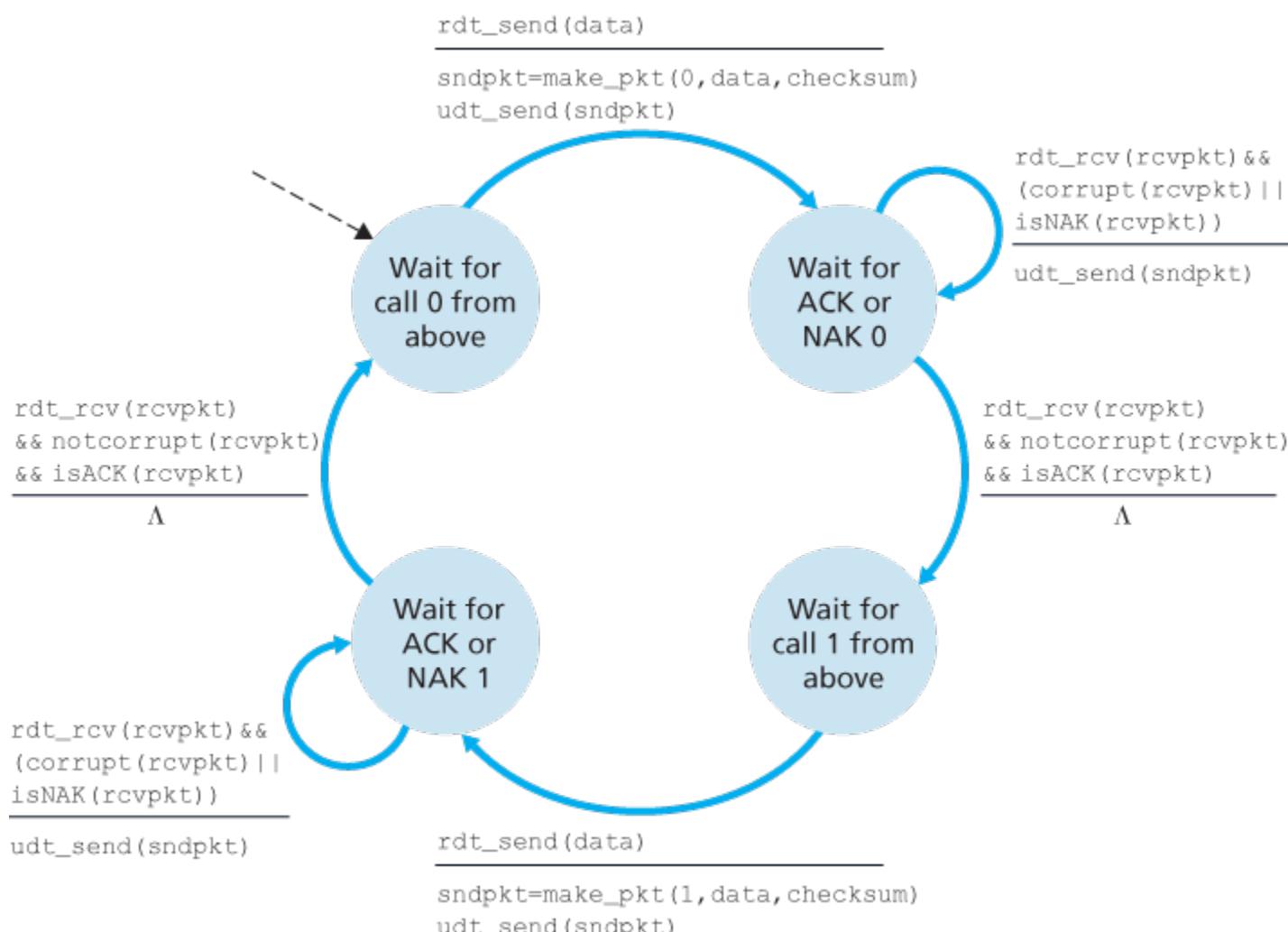
- For the first possibility, consider what a human might do in the message-dictation scenario. If the speaker didn't understand the "OK" or "Please repeat that" reply from the receiver, the speaker would probably ask, "What did you say?" (thus introducing a new type of sender-to-receiver packet to our protocol). The receiver would then repeat the reply. But what if the speaker's "What did you say?" is corrupted? The receiver, having no idea whether the garbled sentence was part of the dictation or a request to repeat the last reply, would probably then respond with "What did you say?" And then, of course, that response might be garbled. Clearly, we're heading down a difficult path.
- A second alternative is to add enough checksum bits to allow the sender not only to detect, but also to recover from, bit errors. This solves the immediate problem for a channel that can corrupt packets but not lose them.
- A third approach is for the sender simply to resend the current data packet when it receives a garbled ACK or NAK packet. This approach, however, introduces **duplicate packets** into the sender-to-receiver channel. The fundamental difficulty with duplicate packets is that the receiver doesn't know whether the ACK or NAK it last sent was received correctly at the sender. Thus, it cannot know *a priori* whether an arriving packet contains new data or is a retransmission!

A simple solution to this new problem (and one adopted in almost all existing data transfer protocols, including TCP) is to add a new field to the data packet and have the sender number its data packets by putting a **sequence number** into this field. The receiver then need only check this sequence number to

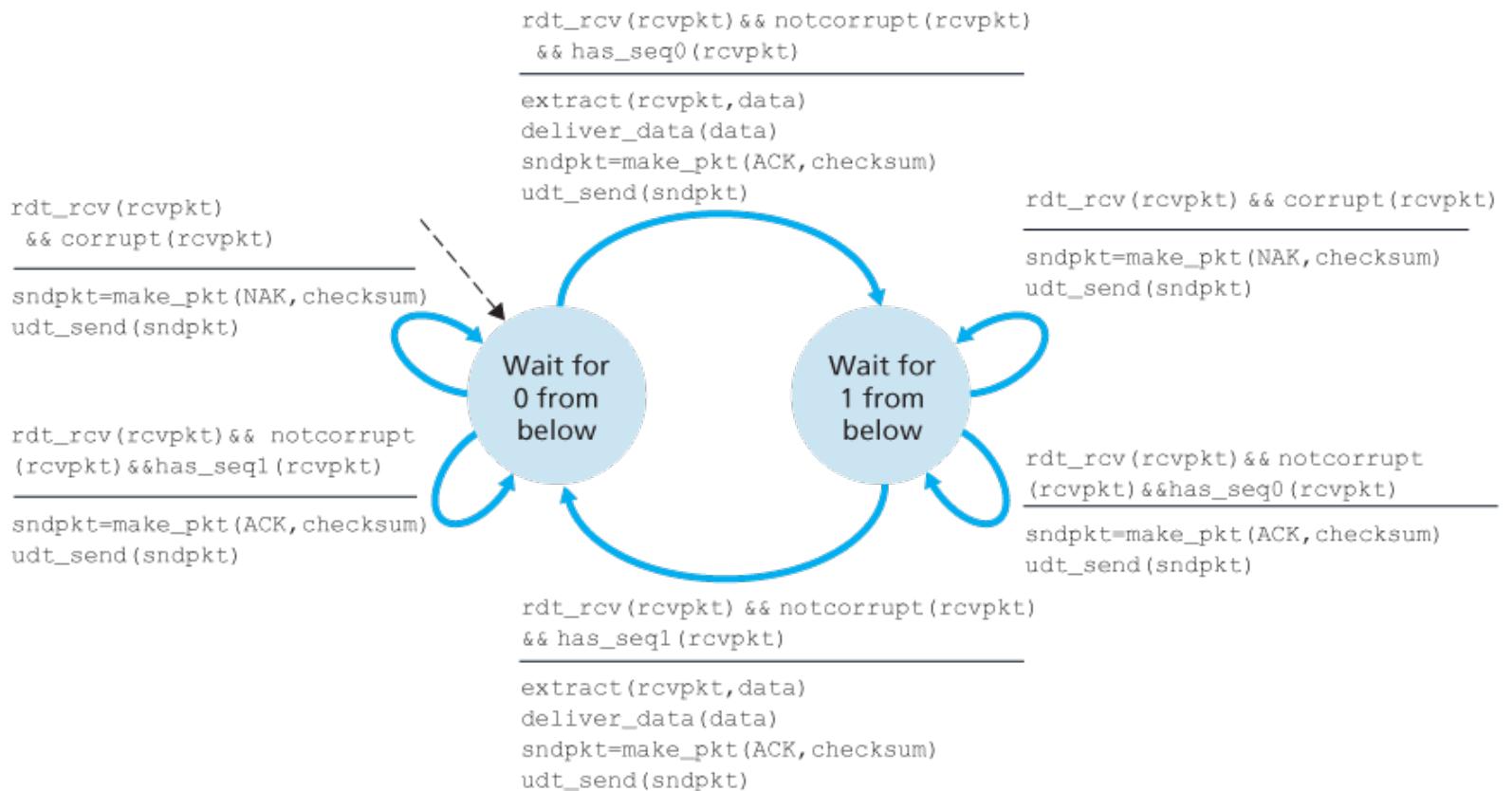
determine whether or not the received packet is a retransmission. For this simple case of a stop-and-wait protocol, a 1-bit sequence number will suffice, since it will allow the receiver to know whether the sender is resending the previously transmitted packet (the sequence number of the received packet has the same sequence number as the most recently received packet) or a new packet (the sequence number changes, moving “forward” in modulo-2 arithmetic). Since we are currently assuming a channel that does not lose packets, ACK and NAK packets do not themselves need to indicate the sequence number of the packet they are acknowledging. The sender knows that a received ACK or NAK packet (whether garbled or not) was generated in response to its most recently transmitted data packet.

**Figures 3.11 and 3.12** show the FSM description for `rdt2.1`, our fixed version of `rdt2.0`. The `rdt2.1` sender and receiver FSMs each now have twice as many states as before. This is because the protocol state must now reflect whether the packet currently being sent (by the sender) or expected (at the receiver) should have a sequence number of 0 or 1. Note that the actions in those states where a 0-numbered packet is being sent or expected are mirror images of those where a 1-numbered packet is being sent or expected; the only differences have to do with the handling of the sequence number.

Protocol `rdt2.1` uses both positive and negative acknowledgments from the receiver to the sender. When an out-of-order packet is received, the receiver sends a positive acknowledgment for the packet it has received. When a corrupted packet



**Figure 3.11** `rdt2.1` sender

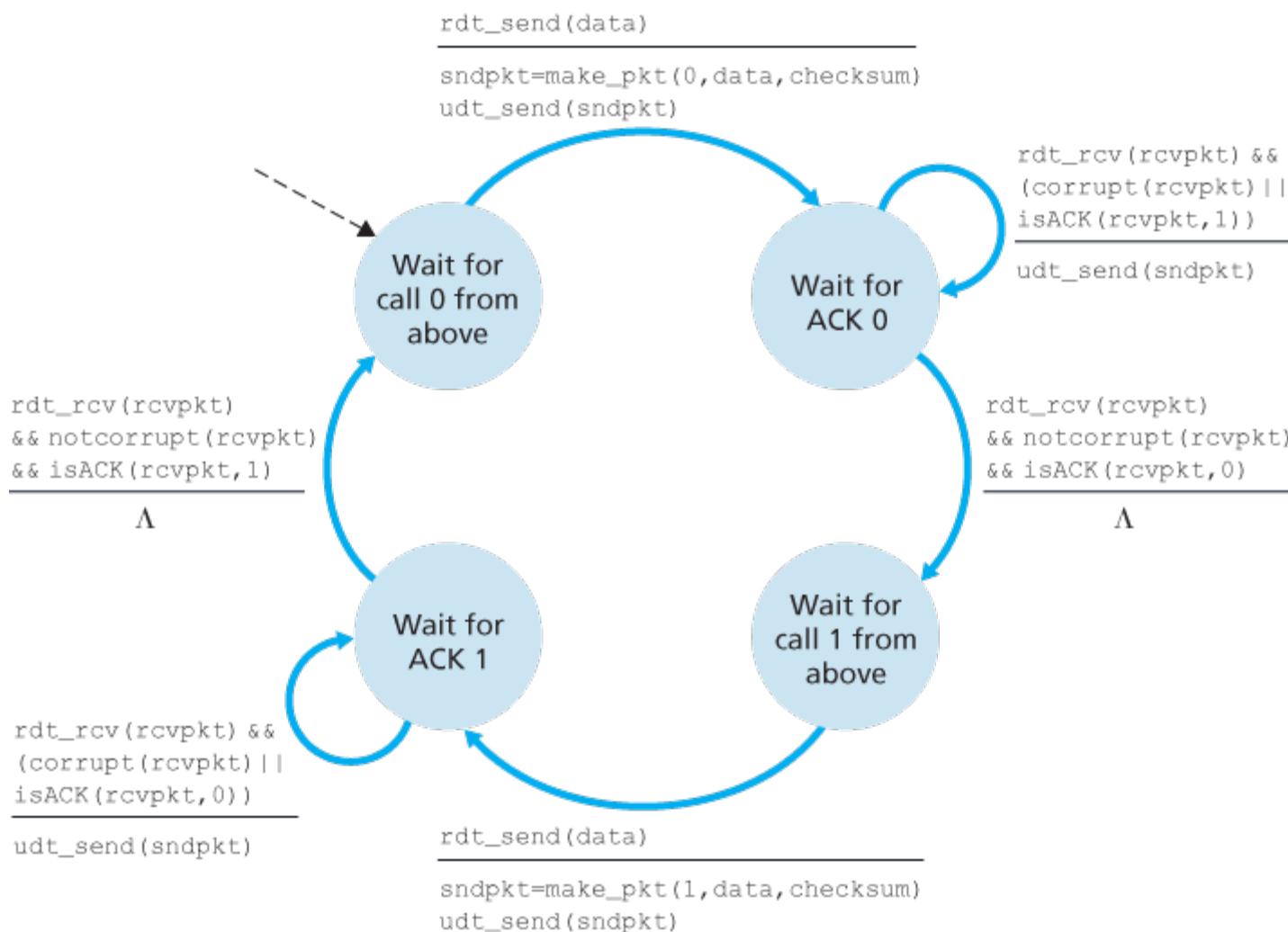


**Figure 3.12 *rdt2.1* receiver**

is received, the receiver sends a negative acknowledgment. We can accomplish the same effect as a NAK if, instead of sending a NAK, we send an ACK for the last correctly received packet. A sender that receives two ACKs for the same packet (that is, receives **duplicate ACKs**) knows that the receiver did not correctly receive the packet following the packet that is being ACKed twice. Our NAK-free reliable data transfer protocol for a channel with bit errors is *rdt2.2*, shown in **Figures 3.13** and **3.14**. One subtle change between *rdt2.1* and *rdt2.2* is that the receiver must now include the sequence number of the packet being acknowledged by an ACK message (this is done by including the *ACK, 0* or *ACK, 1* argument in *make\_pkt()* in the receiver FSM), and the sender must now check the sequence number of the packet being acknowledged by a received ACK message (this is done by including the *0* or *1* argument in *isACK()* in the sender FSM).

#### *Reliable Data Transfer over a Lossy Channel with Bit Errors: rdt3.0*

Suppose now that in addition to corrupting bits, the underlying channel can *lose* packets as well, a not-uncommon event in today's computer networks (including the Internet). Two additional concerns must now be addressed by the protocol: how to detect packet loss and what to do when packet loss occurs. The use of checksumming, sequence numbers, ACK packets, and retransmissions—the techniques

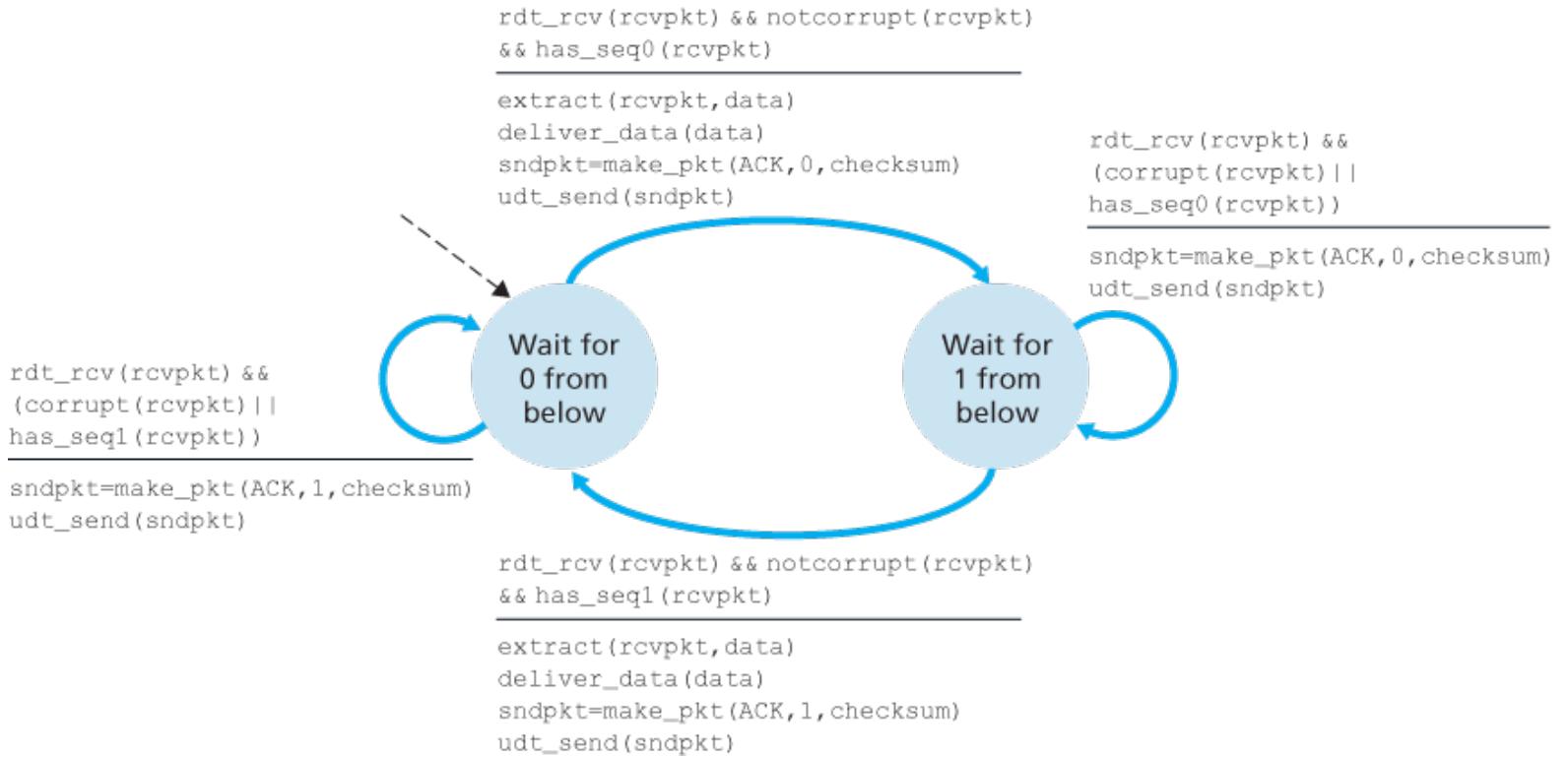


**Figure 3.13 *rdt2.2* sender**

already developed in *rdt2.2*—will allow us to answer the latter concern. Handling the first concern will require adding a new protocol mechanism.

There are many possible approaches toward dealing with packet loss (several more of which are explored in the exercises at the end of the chapter). Here, we'll put the burden of detecting and recovering from lost packets on the sender. Suppose that the sender transmits a data packet and either that packet, or the receiver's ACK of that packet, gets lost. In either case, no reply is forthcoming at the sender from the receiver. If the sender is willing to wait long enough so that it is *certain* that a packet has been lost, it can simply retransmit the data packet. You should convince yourself that this protocol does indeed work.

But how long must the sender wait to be certain that something has been lost? The sender must clearly wait at least as long as a round-trip delay between the sender and receiver (which may include buffering at intermediate routers) plus whatever amount of time is needed to process a packet at the receiver. In many networks, this worst-case maximum delay is very difficult even to estimate, much less know with certainty. Moreover, the protocol should ideally recover from packet loss as soon as possible; waiting for a worst-case delay could mean a long wait until error recovery.

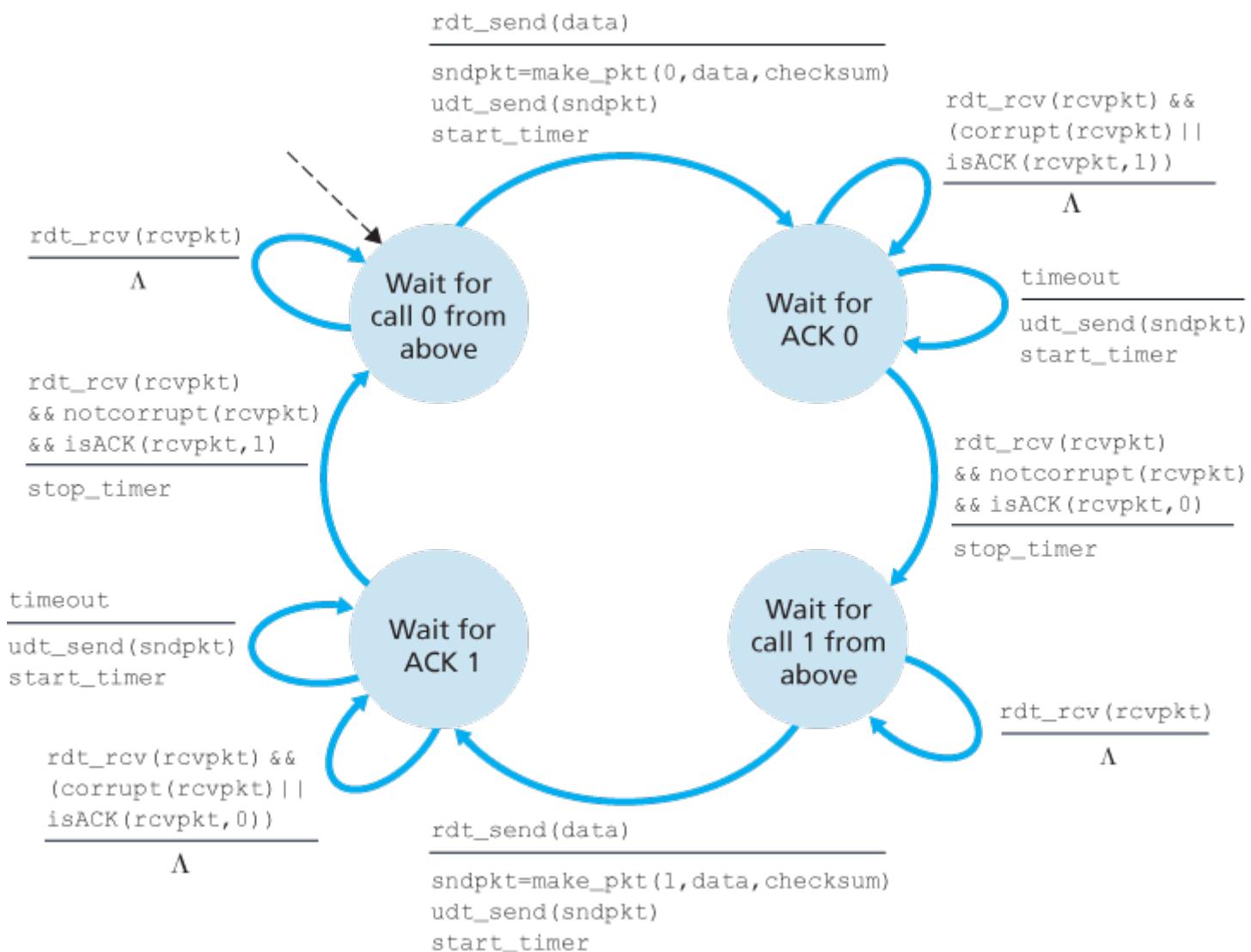


**Figure 3.14** *rdt2.2* receiver

is initiated. The approach thus adopted in practice is for the sender to judiciously choose a time value such that packet loss is likely, although not guaranteed, to have happened. If an ACK is not received within this time, the packet is retransmitted. Note that if a packet experiences a particularly large delay, the sender may retransmit the packet even though neither the data packet nor its ACK have been lost. This introduces the possibility of **duplicate data packets** in the sender-to-receiver channel. Happily, protocol *rdt2.2* already has enough functionality (that is, sequence numbers) to handle the case of duplicate packets.

From the sender's viewpoint, retransmission is a panacea. The sender does not know whether a data packet was lost, an ACK was lost, or if the packet or ACK was simply overly delayed. In all cases, the action is the same: retransmit. Implementing a time-based retransmission mechanism requires a **countdown timer** that can interrupt the sender after a given amount of time has expired. The sender will thus need to be able to (1) start the timer each time a packet (either a first-time packet or a retransmission) is sent, (2) respond to a timer interrupt (taking appropriate actions), and (3) stop the timer.

**Figure 3.15** shows the sender FSM for *rdt3.0*, a protocol that reliably transfers data over a channel that can corrupt or lose packets; in the homework problems, you'll be asked to provide the receiver FSM for *rdt3.0*. **Figure 3.16** shows how the protocol operates with no lost or delayed packets and how it handles lost data packets. In **Figure 3.16**, time moves forward from the top of the diagram toward the bottom of the



**Figure 3.15** *rdt3.0* sender

diagram; note that a receive time for a packet is necessarily later than the send time for a packet as a result of transmission and propagation delays. In **Figures 3.16(b)–(d)**, the send-side brackets indicate the times at which a timer is set and later times out. Several of the more subtle aspects of this protocol are explored in the exercises at the end of this chapter. Because packet sequence numbers alternate between 0 and 1, protocol *rdt3.0* is sometimes known as the **alternating-bit protocol**.

We have now assembled the key elements of a data transfer protocol. Checksums, sequence numbers, timers, and positive and negative acknowledgment packets each play a crucial and necessary role in the operation of the protocol. We now have a working reliable data transfer protocol!

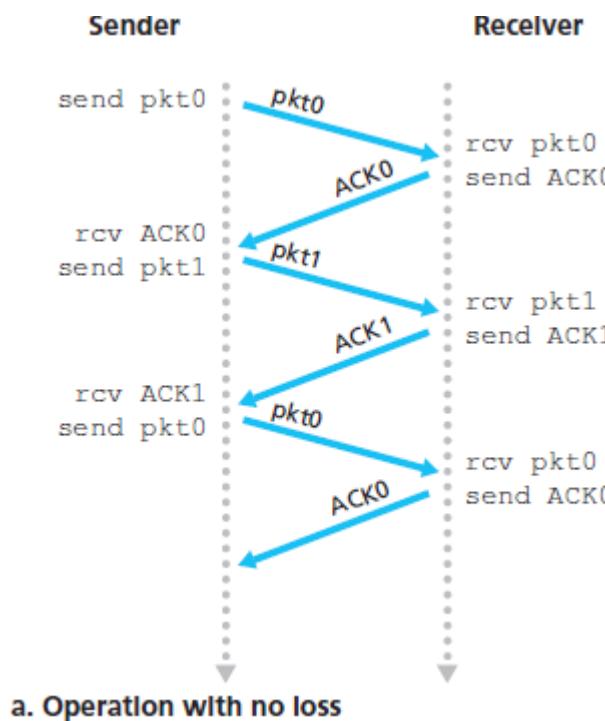


VideoNote

Developing a protocol and FSM representation for a simple application-layer protocol

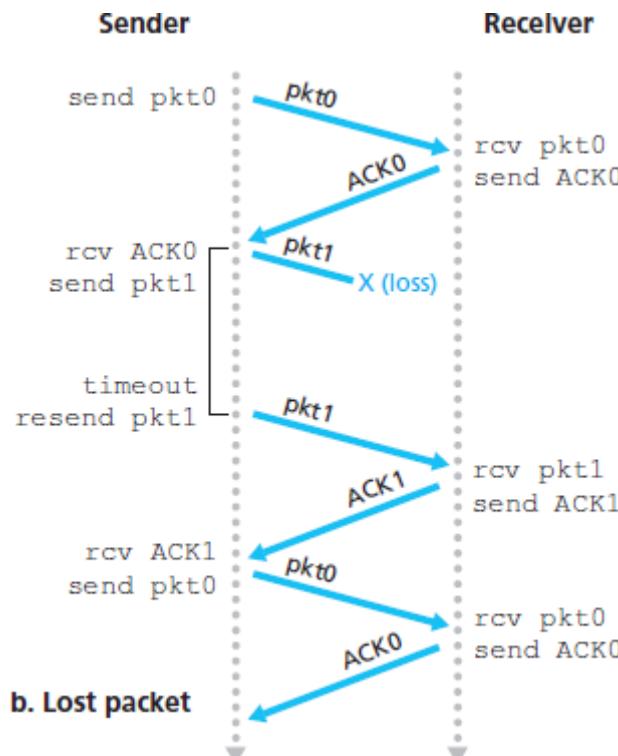
### 3.4.2 Pipelined Reliable Data Transfer Protocols

Protocol `rdt3.0` is a functionally correct protocol, but it is unlikely that anyone would be happy with its performance, particularly in today's high-speed networks. At the heart of `rdt3.0`'s performance problem is the fact that it is a stop-and-wait protocol.

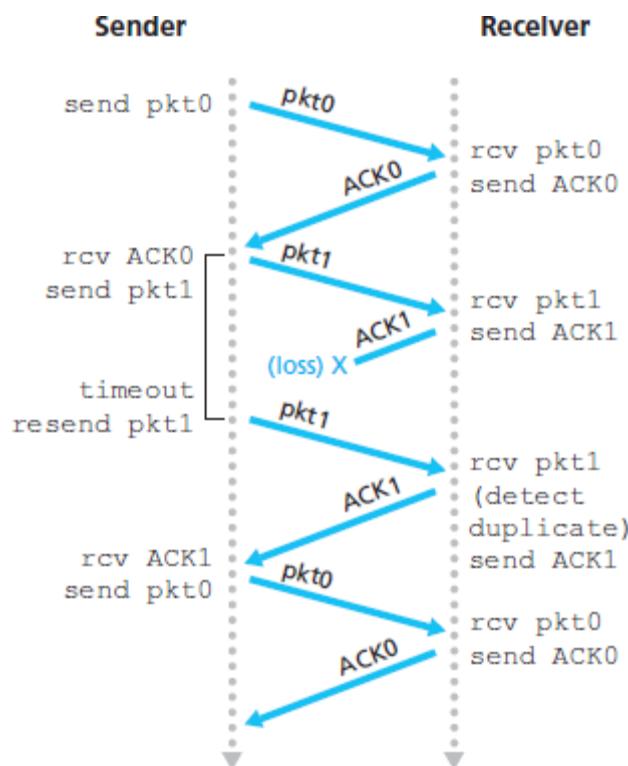


a. Operation with no loss

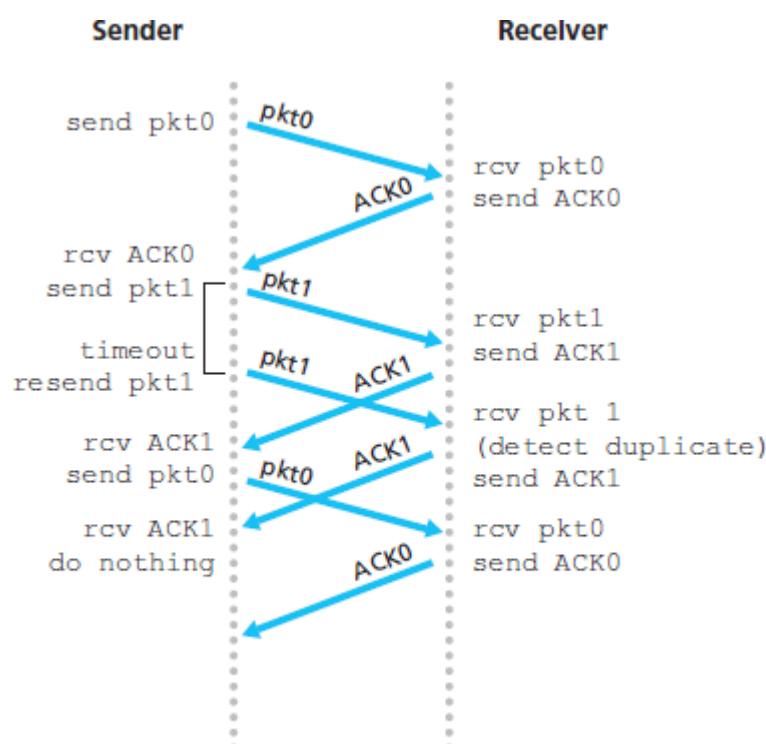
Figure 3.16 Operation of `rdt3.0`, the alternating-bit protocol



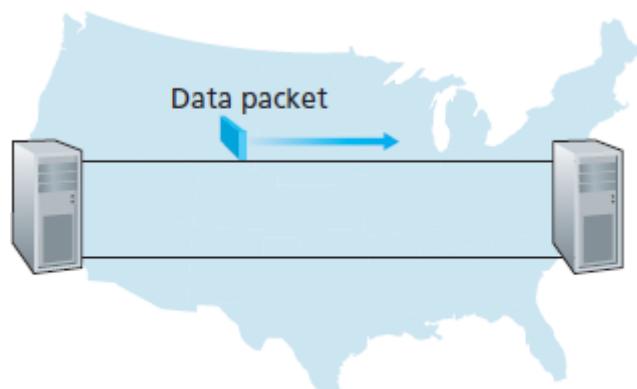
b. Lost packet



c. Lost ACK

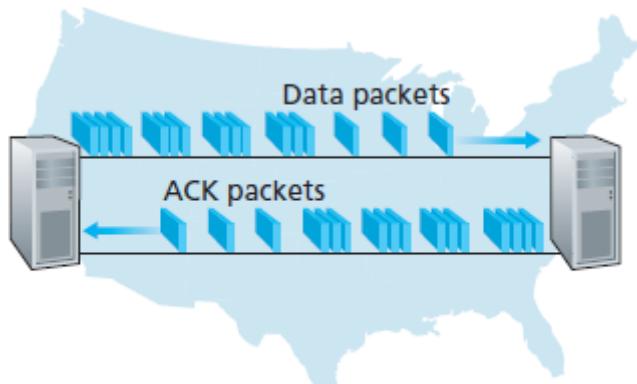


d. Premature timeout



a. A stop-and-wait protocol in operation

**Figure 3.17 Stop-and-wait versus pipelined protocol**



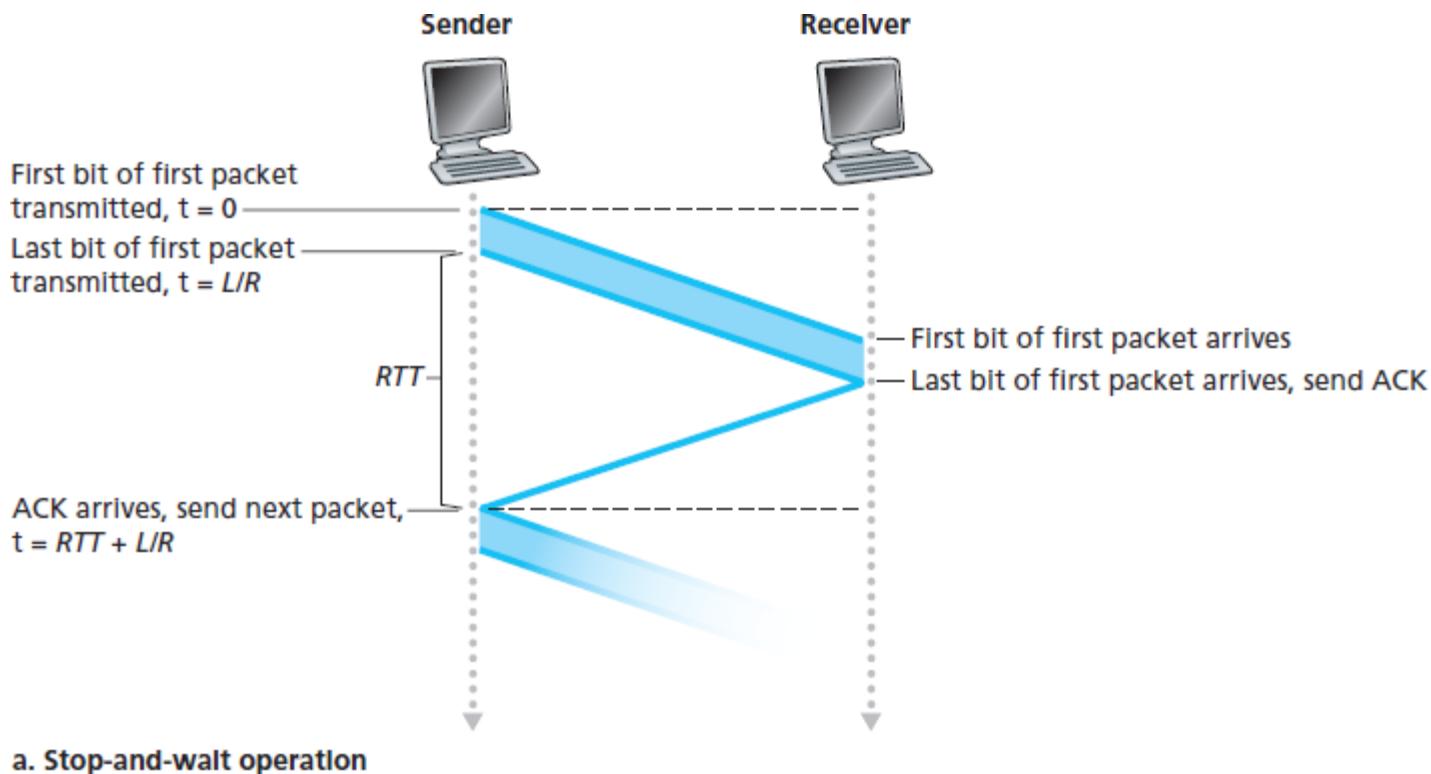
b. A pipelined protocol in operation

To appreciate the performance impact of this stop-and-wait behavior, consider an idealized case of two hosts, one located on the West Coast of the United States and the other located on the East Coast, as shown in [Figure 3.17](#). The speed-of-light round-trip propagation delay between these two end systems, RTT, is approximately 30 milliseconds. Suppose that they are connected by a channel with a transmission rate,  $R$ , of 1 Gbps ( $10^9$  bits per second). With a packet size,  $L$ , of 1,000 bytes (8,000 bits) per packet, including both header fields and data, the time needed to actually transmit the packet into the 1 Gbps link is

$$d_{\text{trans}} = L/R = 8000 \text{ bits/packet} / 10^9 \text{ bits/sec} = 8 \text{ microseconds}$$

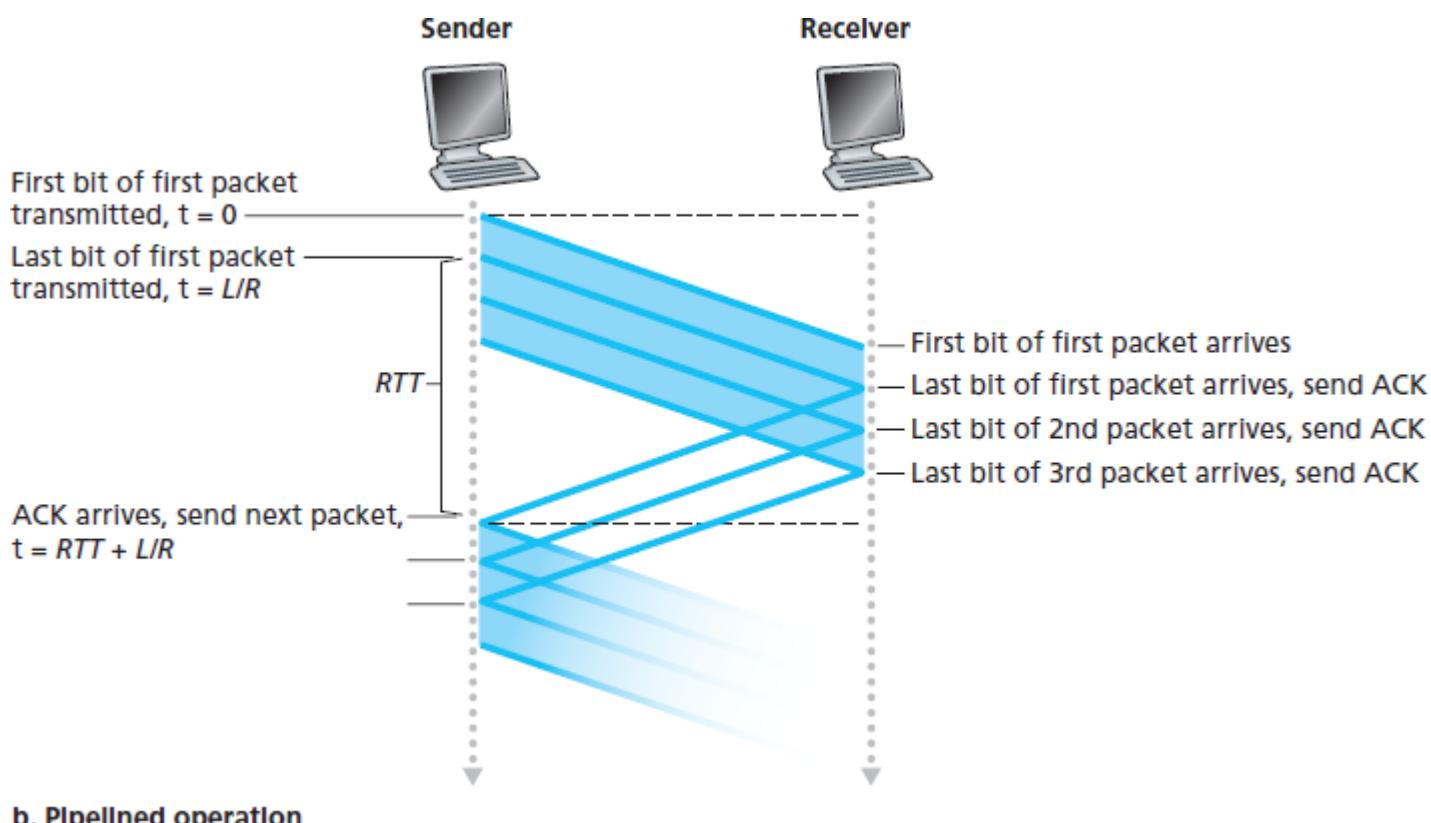
[Figure 3.18\(a\)](#) shows that with our stop-and-wait protocol, if the sender begins sending the packet at  $t=0$ , then at  $t=L/R=8$  microseconds, the last bit enters the channel at the sender side. The packet then makes its 15-msec cross-country journey, with the last bit of the packet emerging at the receiver at  $t=RTT/2+L/R= 15.008$  msec. Assuming for simplicity that ACK packets are extremely small (so that we can ignore their transmission time) and that the receiver can send an ACK as soon as the last bit of a data packet is received, the ACK emerges back at the sender at  $t=RTT+L/R=30.008$  msec. At this point, the sender can now transmit the next message. Thus, in 30.008 msec, the sender was sending for only 0.008 msec. If we define the **utilization** of the sender (or the channel) as the fraction of time the sender is actually busy sending bits into the channel, the analysis in [Figure 3.18\(a\)](#) shows that the stop-and-wait protocol has a rather dismal sender utilization,  $U_{\text{sender}}$ , of

$$U_{\text{sender}} = L/RTT + L/R = .008/30.008 = 0.00027$$



a. Stop-and-wait operation

Figure 3.18 Stop-and-wait and pipelined sending



b. Pipelined operation

That is, the sender was busy only 2.7 hundredths of one percent of the time! Viewed another way, the sender was able to send only 1,000 bytes in 30.008 milliseconds, an effective throughput of only 267 kbps—even though a 1 Gbps link was available! Imagine the unhappy network manager who just paid a fortune for a gigabit capacity link but manages to get a throughput of only 267 kilobits per second! This is a graphic example of how network protocols can limit the capabilities provided by the underlying network hardware. Also, we have neglected lower-layer protocol-processing times at the sender and receiver, as well as the processing and queuing delays that would occur at any intermediate routers.

between the sender and receiver. Including these effects would serve only to further increase the delay and further accentuate the poor performance.

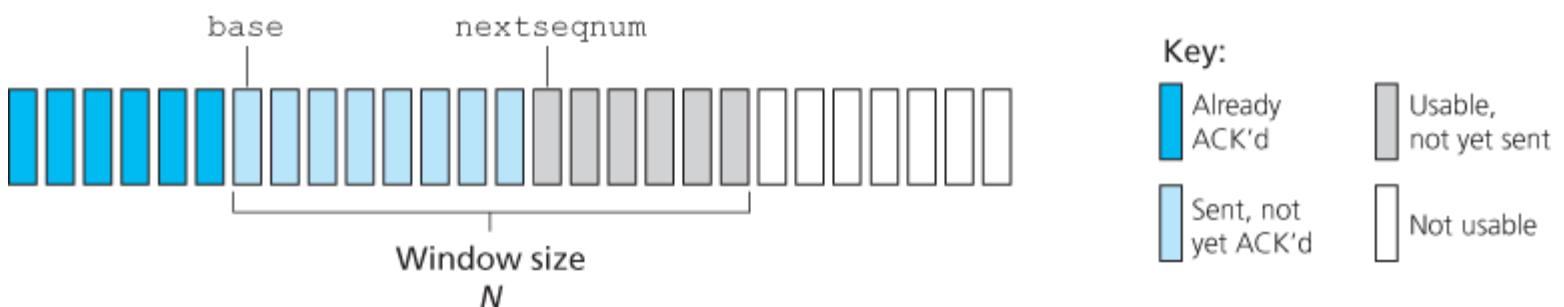
The solution to this particular performance problem is simple: Rather than operate in a stop-and-wait manner, the sender is allowed to send multiple packets without waiting for acknowledgments, as illustrated in [Figure 3.17\(b\)](#). [Figure 3.18\(b\)](#) shows that if the sender is allowed to transmit three packets before having to wait for acknowledgments, the utilization of the sender is essentially tripled. Since the many in-transit sender-to-receiver packets can be visualized as filling a pipeline, this technique is known as **pipelining**. Pipelining has the following consequences for reliable data transfer protocols:

- The range of sequence numbers must be increased, since each in-transit packet (not counting retransmissions) must have a unique sequence number and there may be multiple, in-transit, unacknowledged packets.
- The sender and receiver sides of the protocols may have to buffer more than one packet. Minimally, the sender will have to buffer packets that have been transmitted but not yet acknowledged. Buffering of correctly received packets may also be needed at the receiver, as discussed below.
- The range of sequence numbers needed and the buffering requirements will depend on the manner in which a data transfer protocol responds to lost, corrupted, and overly delayed packets. Two basic approaches toward pipelined error recovery can be identified: **Go-Back-N** and **selective repeat**.

### 3.4.3 Go-Back-N (GBN)

In a **Go-Back-N (GBN) protocol**, the sender is allowed to transmit multiple packets (when available) without waiting for an acknowledgment, but is constrained to have no more than some maximum allowable number,  $N$ , of unacknowledged packets in the pipeline. We describe the GBN protocol in some detail in this section. But before reading on, you are encouraged to play with the GBN applet (an awesome applet!) at the companion Web site.

[Figure 3.19](#) shows the sender's view of the range of sequence numbers in a GBN protocol. If we define *base* to be the sequence number of the oldest unacknowledged



[Figure 3.19](#) Sender's view of sequence numbers in Go-Back-N

packet and  $\text{nextseqnum}$  to be the smallest unused sequence number (that is, the sequence number of the next packet to be sent), then four intervals in the range of sequence numbers can be identified.

Sequence numbers in the interval  $[0, \text{base}-1]$  correspond to packets that have already been transmitted and acknowledged. The interval  $[\text{base}, \text{nextseqnum}-1]$  corresponds to packets that have been sent but not yet acknowledged. Sequence numbers in the interval  $[\text{nextseqnum}, \text{base}+\text{N}-1]$  can be used for packets that can be sent immediately, should data arrive from the upper layer. Finally, sequence numbers greater than or equal to  $\text{base}+\text{N}$  cannot be used until an unacknowledged packet currently in the pipeline (specifically, the packet with sequence number  $\text{base}$ ) has been acknowledged.

As suggested by [Figure 3.19](#), the range of permissible sequence numbers for transmitted but not yet acknowledged packets can be viewed as a window of size  $N$  over the range of sequence numbers. As the protocol operates, this window slides forward over the sequence number space. For this reason,  $N$  is often referred to as the **window size** and the GBN protocol itself as a **sliding-window protocol**. You might be wondering why we would even limit the number of outstanding, unacknowledged packets to a value of  $N$  in the first place. Why not allow an unlimited number of such packets? We'll see in [Section 3.5](#) that flow control is one reason to impose a limit on the sender. We'll examine another reason to do so in [Section 3.7](#), when we study TCP congestion control.

In practice, a packet's sequence number is carried in a fixed-length field in the packet header. If  $k$  is the number of bits in the packet sequence number field, the range of sequence numbers is thus  $[0, 2^k-1]$ . With a finite range of sequence numbers, all arithmetic involving sequence numbers must then be done using modulo  $2^k$  arithmetic. (That is, the sequence number space can be thought of as a ring of size  $2^k$ , where sequence number  $2^k-1$  is immediately followed by sequence number 0.) Recall that [\*rdt3.0\*](#) had a 1-bit sequence number and a range of sequence numbers of  $[0, 1]$ . Several of the problems at the end of this chapter explore the consequences of a finite range of sequence numbers. We will see in [Section 3.5](#) that TCP has a 32-bit sequence number field, where TCP sequence numbers count bytes in the byte stream rather than packets.

[Figures 3.20](#) and [3.21](#) give an extended FSM description of the sender and receiver sides of an ACK-based, NAK-free, GBN protocol. We refer to this FSM

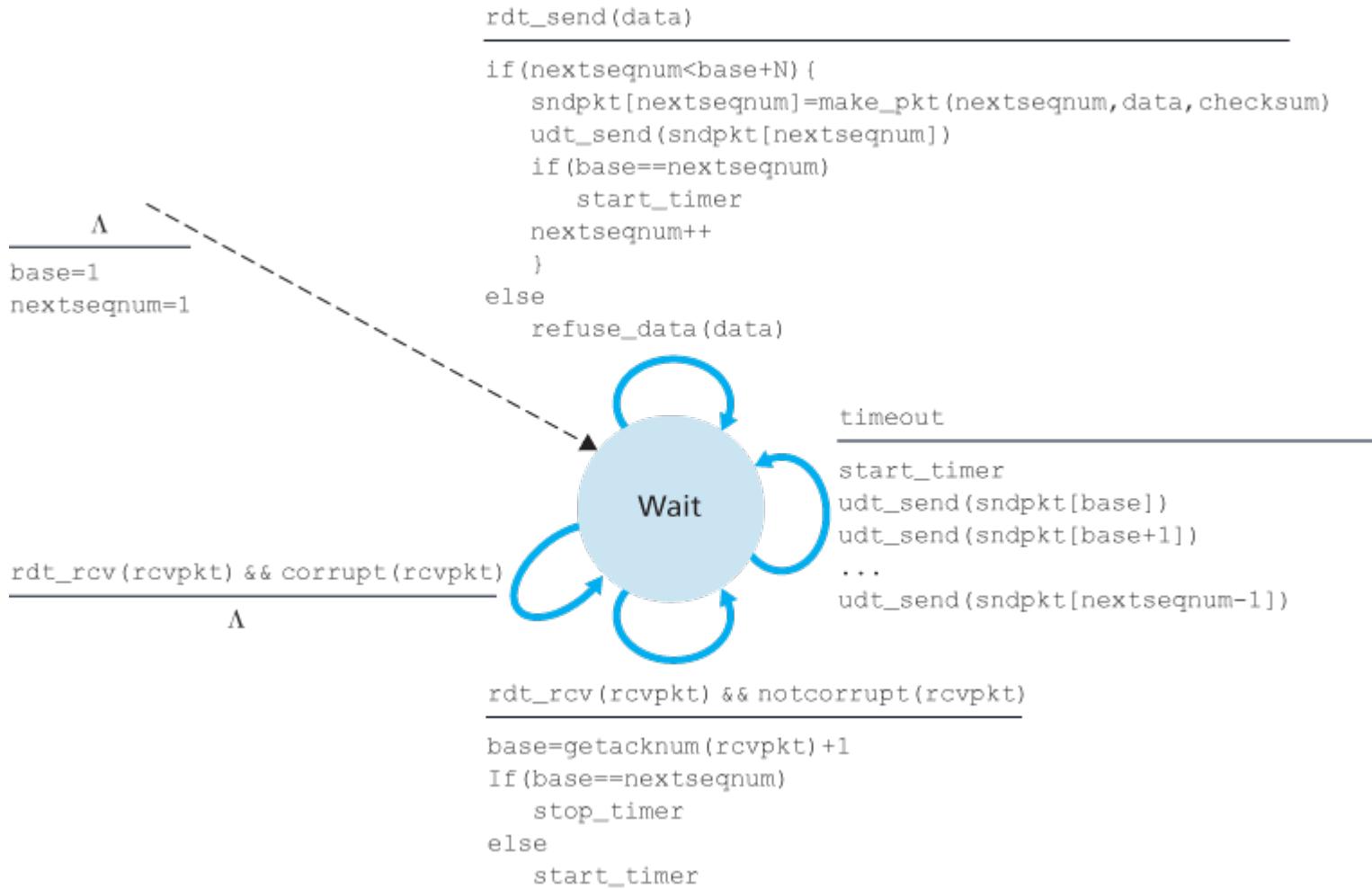


Figure 3.20 Extended FSM description of the GBN sender

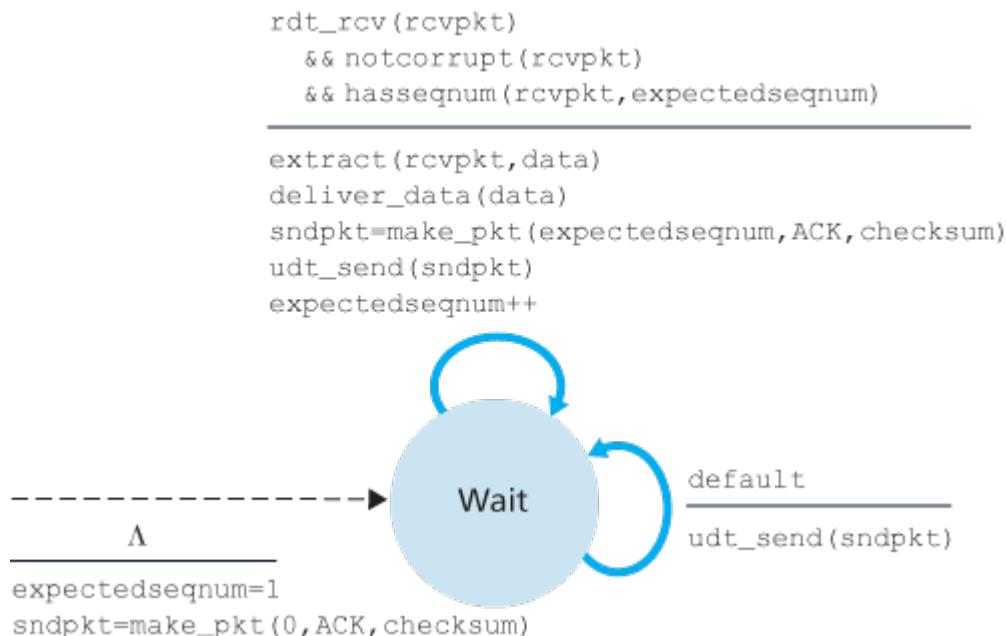


Figure 3.21 Extended FSM description of the GBN receiver

description as an *extended FSM* because we have added variables (similar to programming-language variables) for `base` and `nextseqnum`, and added operations on these variables and conditional actions involving these variables. Note that the extended FSM specification is now beginning to look somewhat like a programming-language specification. [Bochman 1984] provides an excellent survey of

additional extensions to FSM techniques as well as other programming-language-based techniques for specifying protocols.

The GBN sender must respond to three types of events:

- **Invocation from above.** When `rdt_send()` is called from above, the sender first checks to see if the window is full, that is, whether there are  $N$  outstanding, unacknowledged packets. If the window is not full, a packet is created and sent, and variables are appropriately updated. If the window is full, the sender simply returns the data back to the upper layer, an implicit indication that the window is full. The upper layer would presumably then have to try again later. In a real implementation, the sender would more likely have either buffered (but not immediately sent) this data, or would have a synchronization mechanism (for example, a semaphore or a flag) that would allow the upper layer to call `rdt_send()` only when the window is not full.
- **Receipt of an ACK.** In our GBN protocol, an acknowledgment for a packet with sequence number  $n$  will be taken to be a **cumulative acknowledgment**, indicating that all packets with a sequence number up to and including  $n$  have been correctly received at the receiver. We'll come back to this issue shortly when we examine the receiver side of GBN.
- **A timeout event.** The protocol's name, "Go-Back-N," is derived from the sender's behavior in the presence of lost or overly delayed packets. As in the stop-and-wait protocol, a timer will again be used to recover from lost data or acknowledgment packets. If a timeout occurs, the sender resends *all* packets that have been previously sent but that have not yet been acknowledged. Our sender in **Figure 3.20** uses only a single timer, which can be thought of as a timer for the oldest transmitted but not yet acknowledged packet. If an ACK is received but there are still additional transmitted but not yet acknowledged packets, the timer is restarted. If there are no outstanding, unacknowledged packets, the timer is stopped.

The receiver's actions in GBN are also simple. If a packet with sequence number  $n$  is received correctly and is in order (that is, the data last delivered to the upper layer came from a packet with sequence number  $n-1$ ), the receiver sends an ACK for packet  $n$  and delivers the data portion of the packet to the upper layer. In all other cases, the receiver discards the packet and resends an ACK for the most recently received in-order packet. Note that since packets are delivered one at a time to the upper layer, if packet  $k$  has been received and delivered, then all packets with a sequence number lower than  $k$  have also been delivered. Thus, the use of cumulative acknowledgments is a natural choice for GBN.

In our GBN protocol, the receiver discards out-of-order packets. Although it may seem silly and wasteful to discard a correctly received (but out-of-order) packet, there is some justification for doing so. Recall that the receiver must deliver data in order to the upper layer. Suppose now that packet  $n$  is expected, but packet  $n+1$  arrives. Because data must be delivered in order, the receiver *could* buffer (save) packet  $n+1$  and then deliver this packet to the upper layer after it had later received and delivered packet  $n$ . However, if packet  $n$  is lost, both it and packet  $n+1$  will eventually be retransmitted as a result of the

GBN retransmission rule at the sender. Thus, the receiver can simply discard packet  $n+1$ . The advantage of this approach is the simplicity of receiver buffering—the receiver need not buffer *any* out-of-order packets. Thus, while the sender must maintain the upper and lower bounds of its window and the position of *nextseqnum* within this window, the only piece of information the receiver need maintain is the sequence number of the next in-order packet. This value is held in the variable *expectedseqnum*, shown in the receiver FSM in [Figure 3.21](#). Of course, the disadvantage of throwing away a correctly received packet is that the subsequent retransmission of that packet might be lost or garbled and thus even more retransmissions would be required.

[Figure 3.22](#) shows the operation of the GBN protocol for the case of a window size of four packets. Because of this window size limitation, the sender sends packets 0 through 3 but then must wait for one or more of these packets to be acknowledged before proceeding. As each successive ACK (for example, *ACK0* and *ACK1*) is received, the window slides forward and the sender can transmit one new packet (pkt4 and pkt5, respectively). On the receiver side, packet 2 is lost and thus packets 3, 4, and 5 are found to be out of order and are discarded.

Before closing our discussion of GBN, it is worth noting that an implementation of this protocol in a protocol stack would likely have a structure similar to that of the extended FSM in [Figure 3.20](#). The implementation would also likely be in the form of various procedures that implement the actions to be taken in response to the various events that can occur. In such **event-based programming**, the various procedures are called (invoked) either by other procedures in the protocol stack, or as the result of an interrupt. In the sender, these events would be (1) a call from the upper-layer entity to invoke *rdt\_send()*, (2) a timer interrupt, and (3) a call from the lower layer to invoke *rdt\_rcv()* when a packet arrives. The programming exercises at the end of this chapter will give you a chance to actually implement these routines in a simulated, but realistic, network setting.

We note here that the GBN protocol incorporates almost all of the techniques that we will encounter when we study the reliable data transfer components of TCP in [Section 3.5](#). These techniques include the use of sequence numbers, cumulative acknowledgments, checksums, and a timeout/retransmit operation.

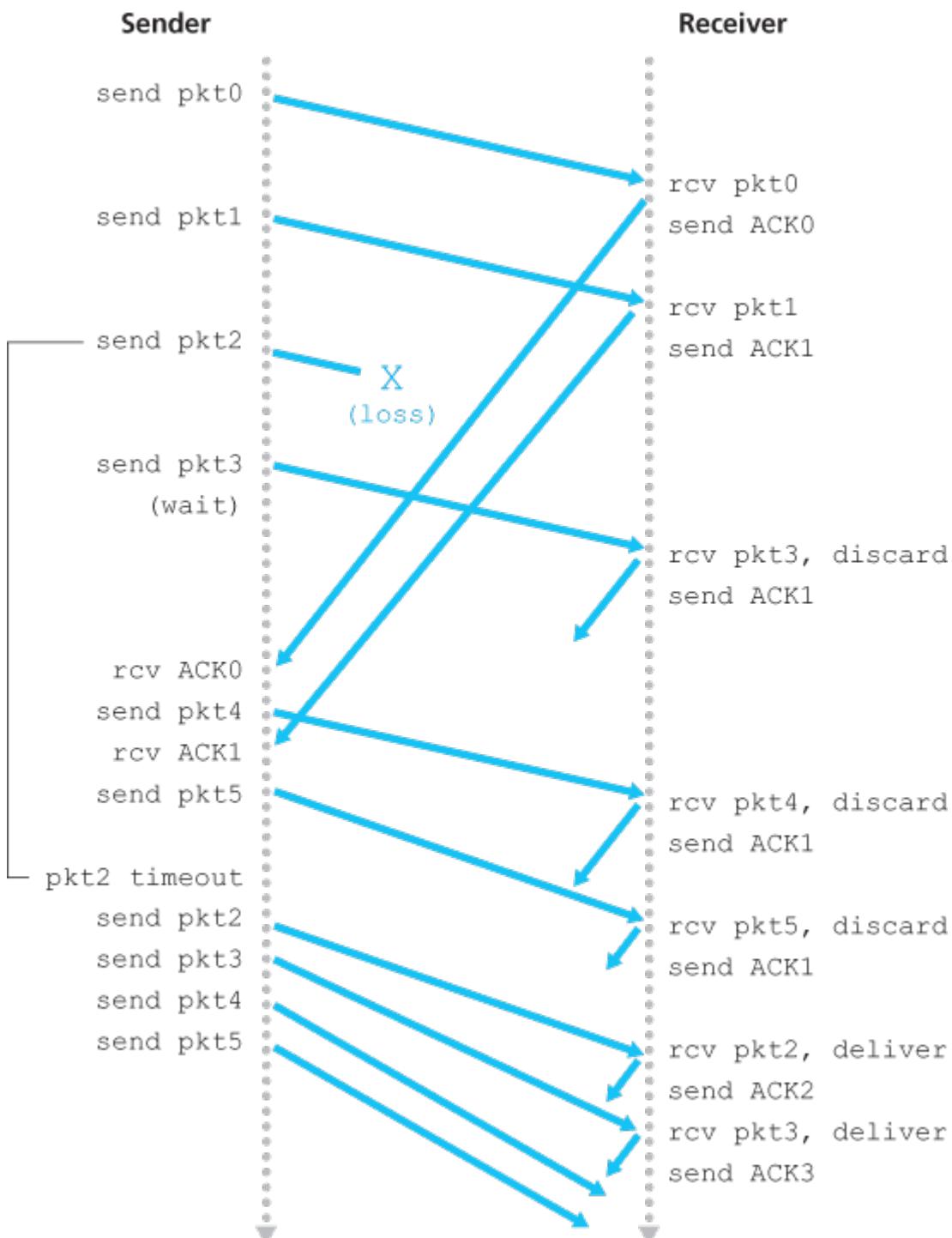


Figure 3.22 Go-Back-N in operation

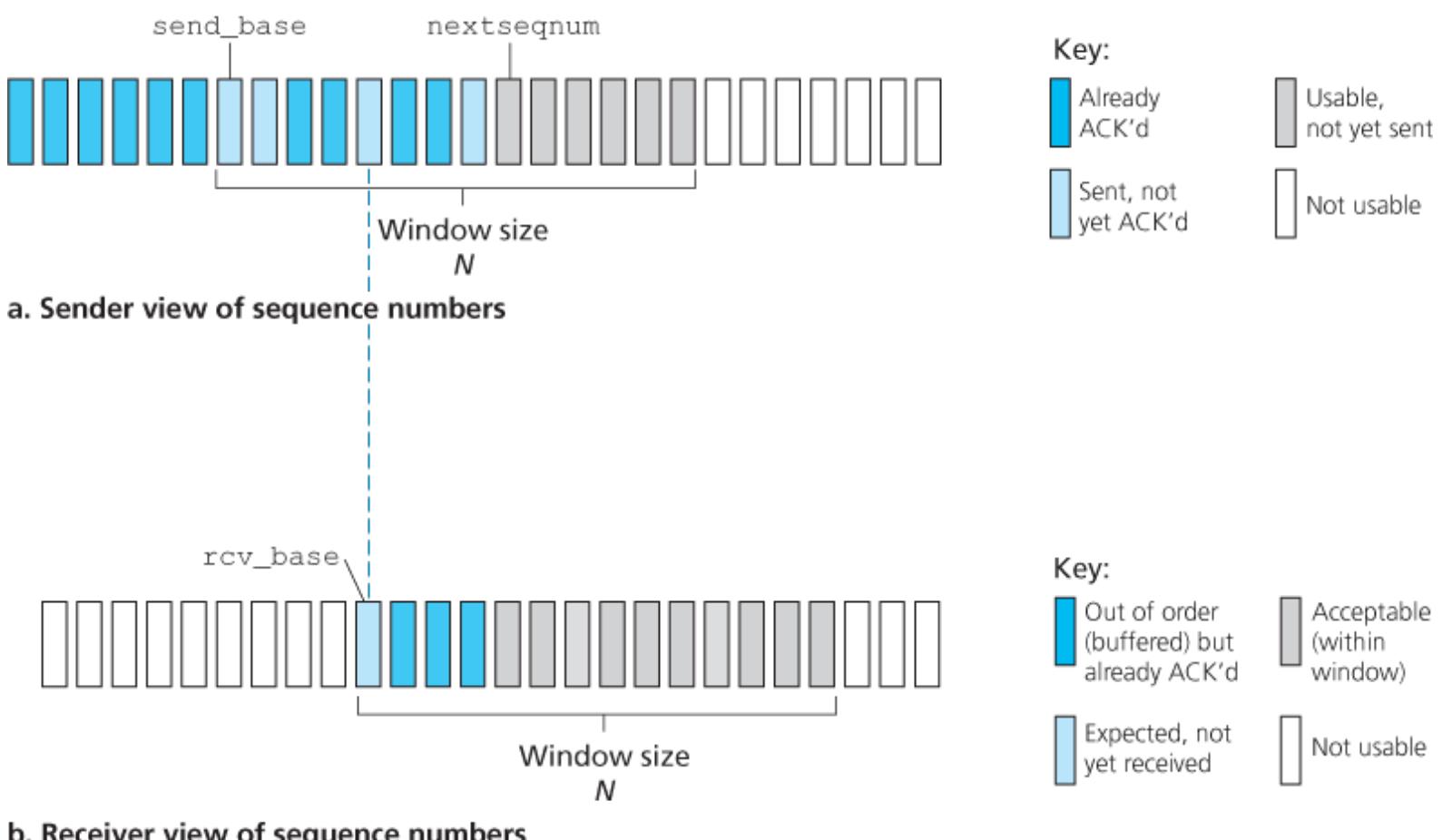
### 3.4.4 Selective Repeat (SR)

The GBN protocol allows the sender to potentially “fill the pipeline” in [Figure 3.17](#) with packets, thus avoiding the channel utilization problems we noted with stop-and-wait protocols. There are, however, scenarios in which GBN itself suffers from performance problems. In particular, when the window size and bandwidth-delay product are both large, many packets can be in the pipeline. A single packet error can thus cause GBN to retransmit a large number of packets, many unnecessarily. As the probability of channel errors increases, the pipeline can become filled with these unnecessary retransmissions. Imagine, in our message-dictation scenario, that if every time a word was garbled, the surrounding 1,000 words (for example, a window size of 1,000 words) had to be repeated. The dictation would be

slowed by all of the reiterated words.

As the name suggests, selective-repeat protocols avoid unnecessary retransmissions by having the sender retransmit only those packets that it suspects were received in error (that is, were lost or corrupted) at the receiver. This individual, as-needed, retransmission will require that the receiver *individually* acknowledge correctly received packets. A window size of  $N$  will again be used to limit the number of outstanding, unacknowledged packets in the pipeline. However, unlike GBN, the sender will have already received ACKs for some of the packets in the window. [Figure 3.23](#) shows the SR sender's view of the sequence number space. [Figure 3.24](#) details the various actions taken by the SR sender.

The SR receiver will acknowledge a correctly received packet whether or not it is in order. Out-of-order packets are buffered until any missing packets (that is, packets with lower sequence numbers) are received, at which point a batch of packets can be delivered in order to the upper layer. [Figure 3.25](#) itemizes the various actions taken by the SR receiver. [Figure 3.26](#) shows an example of SR operation in the presence of lost packets. Note that in [Figure 3.26](#), the receiver initially buffers packets 3, 4, and 5, and delivers them together with packet 2 to the upper layer when packet 2 is finally received.



**Figure 3.23 Selective-repeat (SR) sender and receiver views of sequence-number space**

1. *Data received from above.* When data is received from above, the SR sender checks the next available sequence number for the packet. If the sequence number is within the sender's window, the data is packetized and sent; otherwise it is either buffered or returned to the upper layer for later transmission, as in GBN.
2. *Timeout.* Timers are again used to protect against lost packets. However, each packet must now have its own logical timer, since only a single packet will be transmitted on timeout. A single hardware timer can be used to mimic the operation of multiple logical timers [Varghese 1997].
3. *ACK received.* If an ACK is received, the SR sender marks that packet as having been received, provided it is in the window. If the packet's sequence number is equal to `send_base`, the window base is moved forward to the unacknowledged packet with the smallest sequence number. If the window moves and there are untransmitted packets with sequence numbers that now fall within the window, these packets are transmitted.

**Figure 3.24 SR sender events and actions**

1. *Packet with sequence number in  $[rcv\_base, rcv\_base+N-1]$  is correctly received.* In this case, the received packet falls within the receiver's window and a selective ACK packet is returned to the sender. If the packet was not previously received, it is buffered. If this packet has a sequence number equal to the base of the receive window (`rcv_base` in Figure 3.22), then this packet, and any previously buffered and consecutively numbered (beginning with `rcv_base`) packets are delivered to the upper layer. The receive window is then moved forward by the number of packets delivered to the upper layer. As an example, consider Figure 3.26. When a packet with a sequence number of  $rcv\_base=2$  is received, it and packets 3, 4, and 5 can be delivered to the upper layer.
2. *Packet with sequence number in  $[rcv\_base-N, rcv\_base-1]$  is correctly received.* In this case, an ACK must be generated, even though this is a packet that the receiver has previously acknowledged.
3. *Otherwise.* Ignore the packet.

**Figure 3.25 SR receiver events and actions**

It is important to note that in Step 2 in [Figure 3.25](#), the receiver reacknowledges (rather than ignores) already received packets with certain sequence numbers *below* the current window base. You should convince yourself that this reacknowledgment is indeed needed. Given the sender and receiver sequence number spaces in [Figure 3.23](#), for example, if there is no ACK for packet `send_base` propagating from the

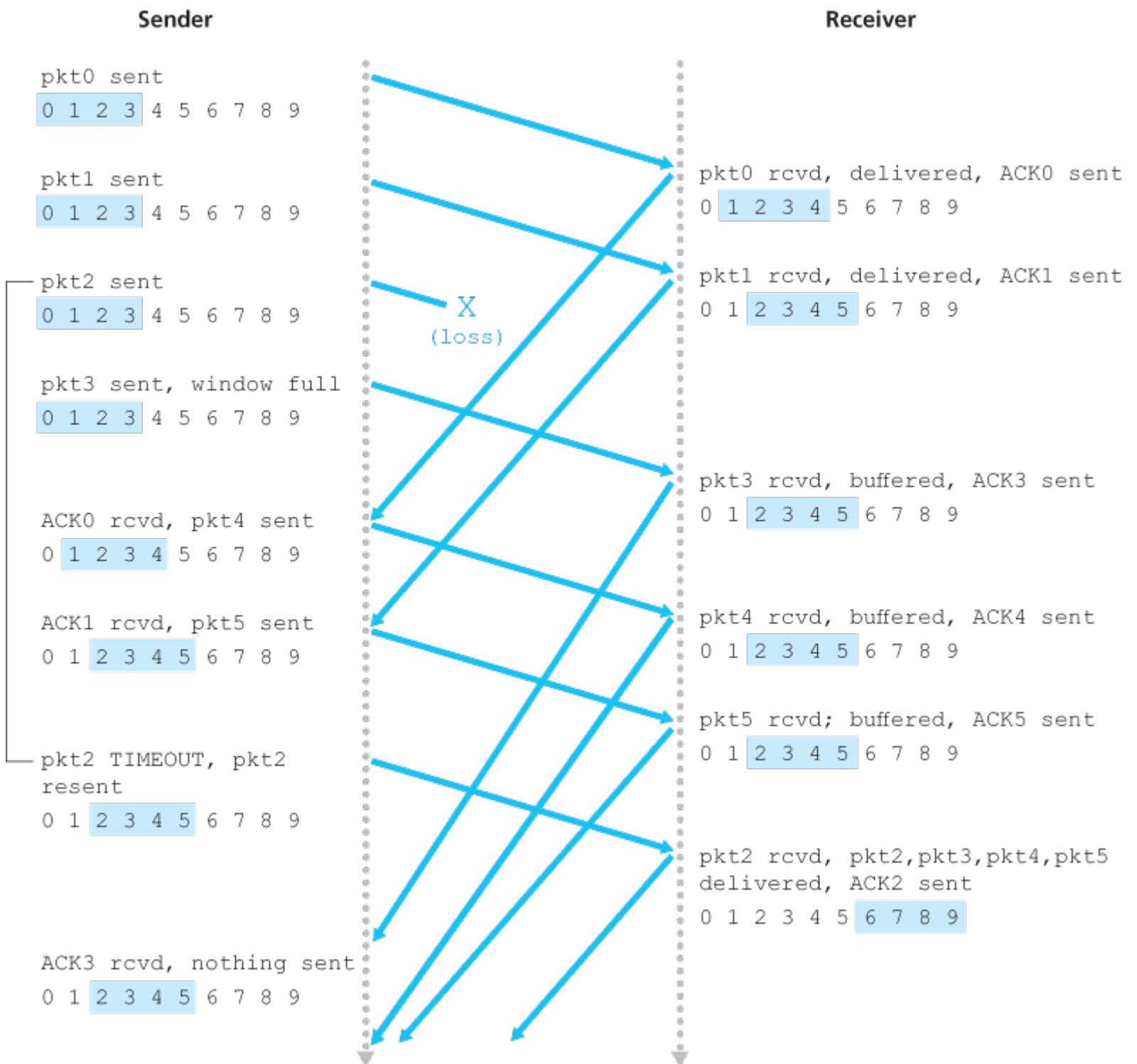


Figure 3.26 SR operation

receiver to the sender, the sender will eventually retransmit packet `send_base`, even though it is clear (to us, not the sender!) that the receiver has already received that packet. If the receiver were not to acknowledge this packet, the sender's window would never move forward! This example illustrates an important aspect of SR protocols (and many other protocols as well). The sender and receiver will not always have an identical view of what has been received correctly and what has not. For SR protocols, this means that the sender and receiver windows will not always coincide.

The lack of synchronization between sender and receiver windows has important consequences when we are faced with the reality of a finite range of sequence numbers. Consider what could happen, for example, with a finite range of four packet sequence numbers, 0, 1, 2, 3, and a window size of three.

Suppose packets 0 through 2 are transmitted and correctly received and acknowledged at the receiver. At this point, the receiver's window is over the fourth, fifth, and sixth packets, which have sequence numbers 3, 0, and 1, respectively. Now consider two scenarios. In the first scenario, shown in [Figure 3.27\(a\)](#), the ACKs for the first three packets are lost and the sender retransmits these packets. The receiver thus next receives a packet with sequence number 0—a copy of the first packet sent.

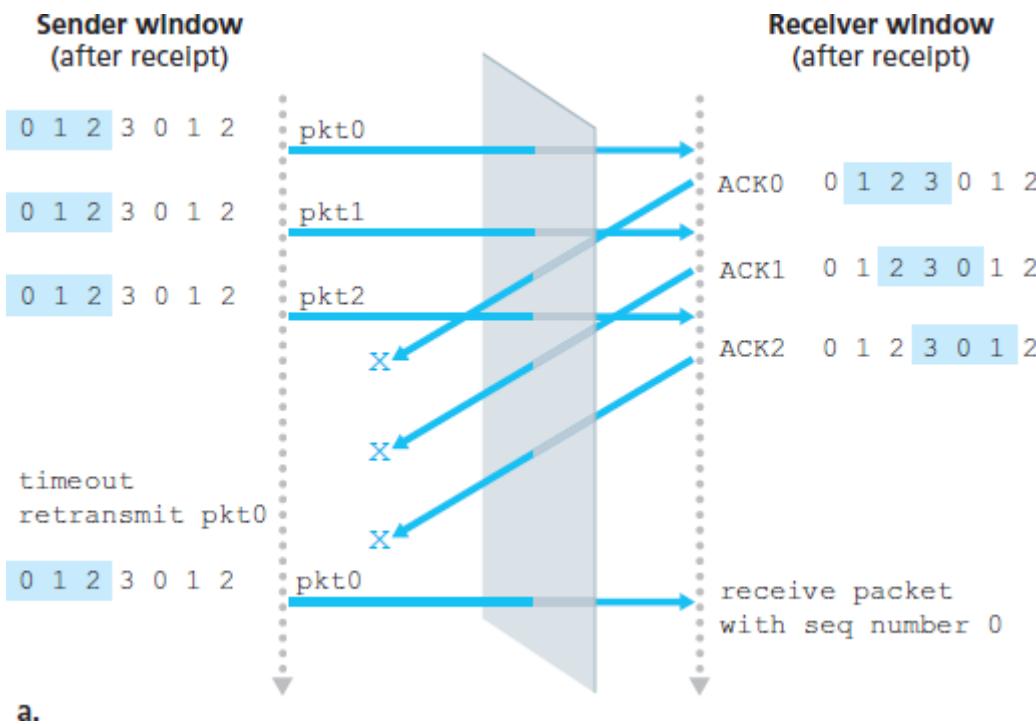
In the second scenario, shown in [Figure 3.27\(b\)](#), the ACKs for the first three packets are all delivered correctly. The sender thus moves its window forward and sends the fourth, fifth, and sixth packets, with sequence numbers 3, 0, and 1, respectively. The packet with sequence number 3 is lost, but the packet with sequence number 0 arrives—a packet containing *new* data.

Now consider the receiver's viewpoint in [Figure 3.27](#), which has a figurative curtain between the sender and the receiver, since the receiver cannot "see" the actions taken by the sender. All the receiver observes is the sequence of messages it receives from the channel and sends into the channel. As far as it is concerned, the two scenarios in [Figure 3.27](#) are *identical*. There is no way of distinguishing the retransmission of the first packet from an original transmission of the fifth packet. Clearly, a window size that is 1 less than the size of the sequence number space won't work. But how small must the window size be? A problem at the end of the chapter asks you to show that the window size must be less than or equal to half the size of the sequence number space for SR protocols.

At the companion Web site, you will find an applet that animates the operation of the SR protocol. Try performing the same experiments that you did with the GBN applet. Do the results agree with what you expect?

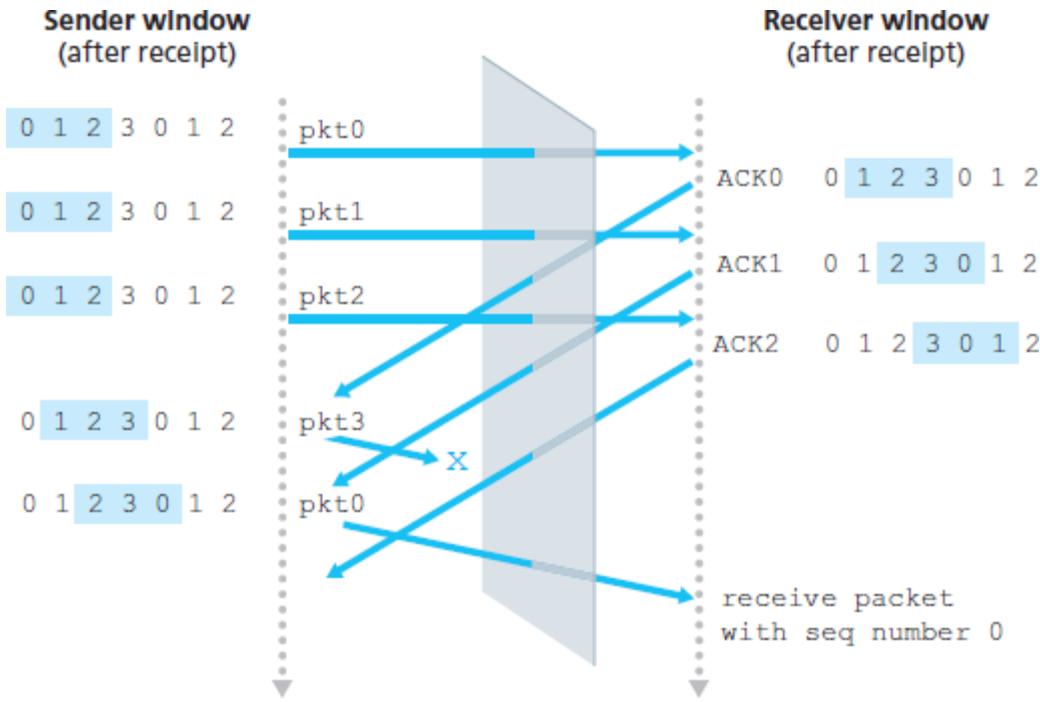
This completes our discussion of reliable data transfer protocols. We've covered a *lot* of ground and introduced numerous mechanisms that together provide for reliable data transfer. [Table 3.1](#) summarizes these mechanisms. Now that we have seen all of these mechanisms in operation and can see the "big picture," we encourage you to review this section again to see how these mechanisms were incrementally added to cover increasingly complex (and realistic) models of the channel connecting the sender and receiver, or to improve the performance of the protocols.

Let's conclude our discussion of reliable data transfer protocols by considering one remaining assumption in our underlying channel model. Recall that we have assumed that packets cannot be reordered within the channel between the sender and receiver. This is generally a reasonable assumption when the sender and receiver are connected by a single physical wire. However, when the "channel" connecting the two is a network, packet reordering can occur. One manifestation of packet reordering is that old copies of a packet with a sequence or acknowledgment



a.

Figure 3.27 SR receiver dilemma with too-large windows: A new packet or a retransmission?



b.

Table 3.1 Summary of reliable data transfer mechanisms and their use

Mechanism	Use, Comments
Checksum	Used to detect bit errors in a transmitted packet.
Timer	Used to timeout/retransmit a packet, possibly because the packet (or its ACK) was lost within the channel. Because timeouts can occur when a packet is delayed but not lost (premature timeout), or when a packet has been received by the receiver but the receiver-to-sender ACK has been lost, duplicate copies

	of a packet may be received by a receiver.
Sequence number	Used for sequential numbering of packets of data flowing from sender to receiver. Gaps in the sequence numbers of received packets allow the receiver to detect a lost packet. Packets with duplicate sequence numbers allow the receiver to detect duplicate copies of a packet.
Acknowledgment	Used by the receiver to tell the sender that a packet or set of packets has been received correctly. Acknowledgments will typically carry the sequence number of the packet or packets being acknowledged. Acknowledgments may be individual or cumulative, depending on the protocol.
Negative acknowledgment	Used by the receiver to tell the sender that a packet has not been received correctly. Negative acknowledgments will typically carry the sequence number of the packet that was not received correctly.
Window, pipelining	The sender may be restricted to sending only packets with sequence numbers that fall within a given range. By allowing multiple packets to be transmitted but not yet acknowledged, sender utilization can be increased over a stop-and-wait mode of operation. We'll see shortly that the window size may be set on the basis of the receiver's ability to receive and buffer messages, or the level of congestion in the network, or both.

number of  $x$  can appear, even though neither the sender's nor the receiver's window contains  $x$ . With packet reordering, the channel can be thought of as essentially buffering packets and spontaneously emitting these packets at *any* point in the future. Because sequence numbers may be reused, some care must be taken to guard against such duplicate packets. The approach taken in practice is to ensure that a sequence number is not reused until the sender is "sure" that any previously sent packets with sequence number  $x$  are no longer in the network. This is done by assuming that a packet cannot "live" in the network for longer than some fixed maximum amount of time. A maximum packet lifetime of approximately three minutes is assumed in the TCP extensions for high-speed networks [\[RFC 1323\]](#). [\[Sunshine 1978\]](#) describes a method for using sequence numbers such that reordering problems can be completely avoided.

## 3.5 Connection-Oriented Transport: TCP

Now that we have covered the underlying principles of reliable data transfer, let's turn to TCP—the Internet's transport-layer, connection-oriented, reliable transport protocol. In this section, we'll see that in order to provide reliable data transfer, TCP relies on many of the underlying principles discussed in the previous section, including error detection, retransmissions, cumulative acknowledgments, timers, and header fields for sequence and acknowledgment numbers. TCP is defined in RFC 793, RFC 1122, RFC 1323, RFC 2018, and RFC 2581.

### 3.5.1 The TCP Connection

TCP is said to be **connection-oriented** because before one application process can begin to send data to another, the two processes must first “handshake” with each other—that is, they must send some preliminary segments to each other to establish the parameters of the ensuing data transfer. As part of TCP connection establishment, both sides of the connection will initialize many TCP state variables (many of which will be discussed in this section and in [Section 3.7](#)) associated with the TCP connection.

The TCP “connection” is not an end-to-end TDM or FDM circuit as in a circuit-switched network. Instead, the “connection” is a logical one, with common state residing only in the TCPs in the two communicating end systems. Recall that because the TCP protocol runs only in the end systems and not in the intermediate network elements (routers and link-layer switches), the intermediate network elements do not maintain TCP connection state. In fact, the intermediate routers are completely oblivious to TCP connections; they see datagrams, not connections.

A TCP connection provides a **full-duplex service**: If there is a TCP connection between Process A on one host and Process B on another host, then application-layer data can flow from Process A to Process B at the same time as application-layer data flows from Process B to Process A. A TCP connection is also always **point-to-point**, that is, between a single sender and a single receiver. So-called “multicasting” (see the online supplementary materials for this text)—the transfer of data from one sender to many receivers in a single send operation—is not possible with TCP. With TCP, two hosts are company and three are a crowd!

Let's now take a look at how a TCP connection is established. Suppose a process running in one host wants to initiate a connection with another process in another host. Recall that the process that is

initiating the connection is called the *client process*, while the other process is called the *server process*. The client application process first informs the client transport layer that it wants to establish a connection

## CASE HISTORY

### *Vinton Cerf, Robert Kahn, and TCP/IP*

In the early 1970s, packet-switched networks began to proliferate, with the ARPAnet—the precursor of the Internet—being just one of many networks. Each of these networks had its own protocol. Two researchers, Vinton Cerf and Robert Kahn, recognized the importance of interconnecting these networks and invented a cross-network protocol called TCP/IP, which stands for Transmission Control Protocol/Internet Protocol. Although Cerf and Kahn began by seeing the protocol as a single entity, it was later split into its two parts, TCP and IP, which operated separately. Cerf and Kahn published a paper on TCP/IP in May 1974 in *IEEE Transactions on Communications Technology* [[Cerf 1974](#)].

The TCP/IP protocol, which is the bread and butter of today's Internet, was devised before PCs, workstations, smartphones, and tablets, before the proliferation of Ethernet, cable, and DSL, WiFi, and other access network technologies, and before the Web, social media, and streaming video. Cerf and Kahn saw the need for a networking protocol that, on the one hand, provides broad support for yet-to-be-defined applications and, on the other hand, allows arbitrary hosts and link-layer protocols to interoperate.

In 2004, Cerf and Kahn received the ACM's Turing Award, considered the “Nobel Prize of Computing” for “pioneering work on internetworking, including the design and implementation of the Internet's basic communications protocols, TCP/IP, and for inspired leadership in networking.”

to a process in the server. Recall from [Section 2.7.2](#), a Python client program does this by issuing the command

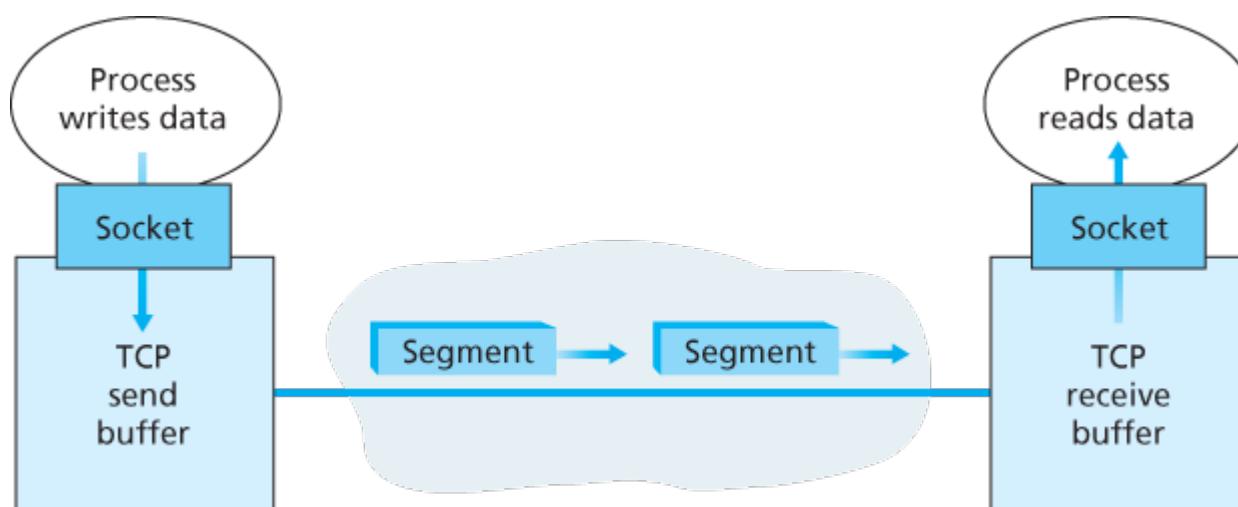
```
clientSocket.connect((serverName, serverPort))
```

where `serverName` is the name of the server and `serverPort` identifies the process on the server. TCP in the client then proceeds to establish a TCP connection with TCP in the server. At the end of this section we discuss in some detail the connection-establishment procedure. For now it suffices to know that the client first sends a special TCP segment; the server responds with a second special TCP segment; and finally the client responds again with a third special segment. The first two segments carry no payload, that is, no application-layer data; the third of these segments may carry a payload. Because

three segments are sent between the two hosts, this connection-establishment procedure is often referred to as a **three-way handshake**.

Once a TCP connection is established, the two application processes can send data to each other. Let's consider the sending of data from the client process to the server process. The client process passes a stream of data through the socket (the door of the process), as described in [Section 2.7](#). Once the data passes through the door, the data is in the hands of TCP running in the client. As shown in [Figure 3.28](#), TCP directs this data to the connection's **send buffer**, which is one of the buffers that is set aside during the initial three-way handshake. From time to time, TCP will grab chunks of data from the send buffer and pass the data to the network layer. Interestingly, the TCP specification [\[RFC 793\]](#) is very laid back about specifying when TCP should actually send buffered data, stating that TCP should "send that data in segments at its own convenience." The maximum amount of data that can be grabbed and placed in a segment is limited by the **maximum segment size (MSS)**. The MSS is typically set by first determining the length of the largest link-layer frame that can be sent by the local sending host (the so-called **maximum transmission unit, MTU**), and then setting the MSS to ensure that a TCP segment (when encapsulated in an IP datagram) plus the TCP/IP header length (typically 40 bytes) will fit into a single link-layer frame. Both Ethernet and PPP link-layer protocols have an MTU of 1,500 bytes. Thus a typical value of MSS is 1460 bytes. Approaches have also been proposed for discovering the path MTU—the largest link-layer frame that can be sent on all links from source to destination [\[RFC 1191\]](#)—and setting the MSS based on the path MTU value. Note that the MSS is the maximum amount of application-layer data in the segment, not the maximum size of the TCP segment including headers. (This terminology is confusing, but we have to live with it, as it is well entrenched.)

TCP pairs each chunk of client data with a TCP header, thereby forming **TCP segments**. The segments are passed down to the network layer, where they are separately encapsulated within network-layer IP datagrams. The IP datagrams are then sent into the network. When TCP receives a segment at the other end, the segment's data is placed in the TCP connection's receive buffer, as shown in [Figure 3.28](#). The application reads the stream of data from this buffer. Each side of the connection has



**Figure 3.28** TCP send and receive buffers

its own send buffer and its own receive buffer. (You can see the online flow-control applet at <http://www.awl.com/kurose-ross>, which provides an animation of the send and receive buffers.)

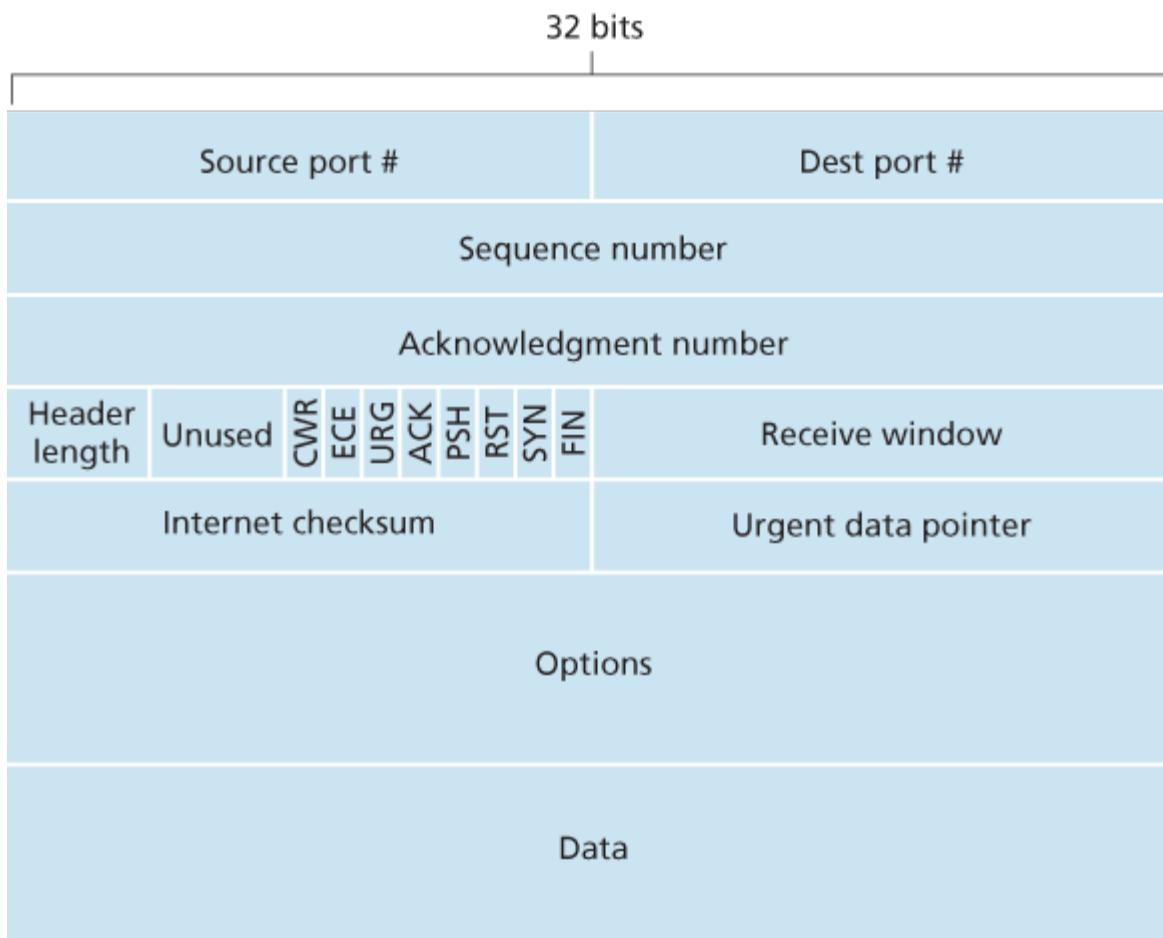
We see from this discussion that a TCP connection consists of buffers, variables, and a socket connection to a process in one host, and another set of buffers, variables, and a socket connection to a process in another host. As mentioned earlier, no buffers or variables are allocated to the connection in the network elements (routers, switches, and repeaters) between the hosts.

### 3.5.2 TCP Segment Structure

Having taken a brief look at the TCP connection, let's examine the TCP segment structure. The TCP segment consists of header fields and a data field. The data field contains a chunk of application data. As mentioned above, the MSS limits the maximum size of a segment's data field. When TCP sends a large file, such as an image as part of a Web page, it typically breaks the file into chunks of size MSS (except for the last chunk, which will often be less than the MSS). Interactive applications, however, often transmit data chunks that are smaller than the MSS; for example, with remote login applications like Telnet, the data field in the TCP segment is often only one byte. Because the TCP header is typically 20 bytes (12 bytes more than the UDP header), segments sent by Telnet may be only 21 bytes in length.

**Figure 3.29** shows the structure of the TCP segment. As with UDP, the header includes **source and destination port numbers**, which are used for multiplexing/demultiplexing data from/to upper-layer applications. Also, as with UDP, the header includes a **checksum field**. A TCP segment header also contains the following fields:

- The 32-bit **sequence number field** and the 32-bit **acknowledgment number field** are used by the TCP sender and receiver in implementing a reliable data transfer service, as discussed below.
- The 16-bit **receive window** field is used for flow control. We will see shortly that it is used to indicate the number of bytes that a receiver is willing to accept.
- The 4-bit **header length field** specifies the length of the TCP header in 32-bit words. The TCP header can be of variable length due to the TCP options field. (Typically, the options field is empty, so that the length of the typical TCP header is 20 bytes.)
- The optional and variable-length **options field** is used when a sender and receiver negotiate the maximum segment size (MSS) or as a window scaling factor for use in high-speed networks. A time-stamping option is also defined. See RFC 854 and RFC 1323 for additional details.
- The **flag field** contains 6 bits. The **ACK bit** is used to indicate that the value carried in the acknowledgment field is valid; that is, the segment contains an acknowledgment for a segment that has been successfully received. The **RST**,



**Figure 3.29** TCP segment structure

**SYN**, and **FIN** bits are used for connection setup and teardown, as we will discuss at the end of this section. The CWR and ECE bits are used in explicit congestion notification, as discussed in [Section 3.7.2](#). Setting the **PSH** bit indicates that the receiver should pass the data to the upper layer immediately. Finally, the **URG** bit is used to indicate that there is data in this segment that the sending-side upper-layer entity has marked as “urgent.” The location of the last byte of this urgent data is indicated by the 16-bit **urgent data pointer field**. TCP must inform the receiving-side upper-layer entity when urgent data exists and pass it a pointer to the end of the urgent data. (In practice, the PSH, URG, and the urgent data pointer are not used. However, we mention these fields for completeness.)

Our experience as teachers is that our students sometimes find discussion of packet formats rather dry and perhaps a bit boring. For a fun and fanciful look at TCP header fields, particularly if you love Legos™ as we do, see [\[Pomeranz 2010\]](#).

#### *Sequence Numbers and Acknowledgment Numbers*

Two of the most important fields in the TCP segment header are the sequence number field and the acknowledgment number field. These fields are a critical part of TCP’s reliable data transfer service. But before discussing how these fields are used to provide reliable data transfer, let us first explain what exactly TCP puts in these fields.

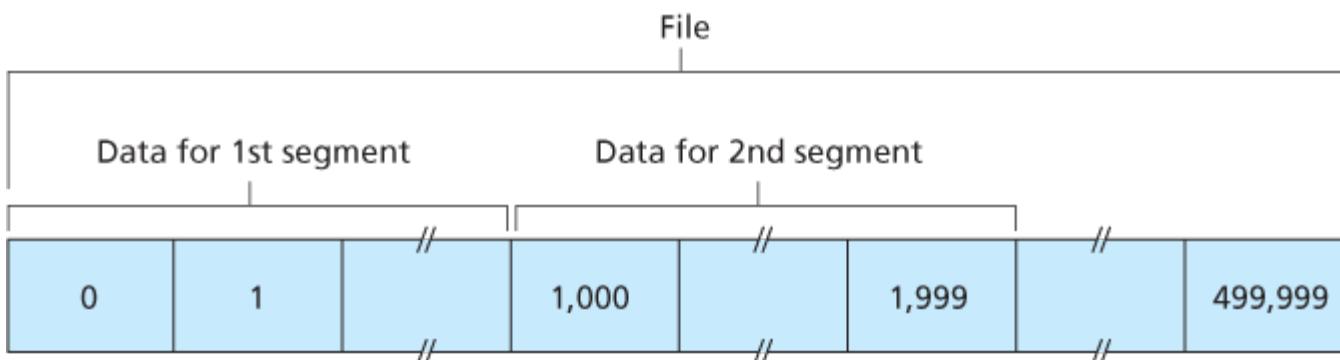


Figure 3.30 Dividing file data into TCP segments

TCP views data as an unstructured, but ordered, stream of bytes. TCP's use of sequence numbers reflects this view in that sequence numbers are over the stream of transmitted bytes and *not* over the series of transmitted segments. The **sequence number for a segment** is therefore the byte-stream number of the first byte in the segment. Let's look at an example. Suppose that a process in Host A wants to send a stream of data to a process in Host B over a TCP connection. The TCP in Host A will implicitly number each byte in the data stream. Suppose that the data stream consists of a file consisting of 500,000 bytes, that the MSS is 1,000 bytes, and that the first byte of the data stream is numbered 0. As shown in [Figure 3.30](#), TCP constructs 500 segments out of the data stream. The first segment gets assigned sequence number 0, the second segment gets assigned sequence number 1,000, the third segment gets assigned sequence number 2,000, and so on. Each sequence number is inserted in the sequence number field in the header of the appropriate TCP segment.

Now let's consider acknowledgment numbers. These are a little trickier than sequence numbers. Recall that TCP is full-duplex, so that Host A may be receiving data from Host B while it sends data to Host B (as part of the same TCP connection). Each of the segments that arrive from Host B has a sequence number for the data flowing from B to A. *The acknowledgment number that Host A puts in its segment is the sequence number of the next byte Host A is expecting from Host B.* It is good to look at a few examples to understand what is going on here. Suppose that Host A has received all bytes numbered 0 through 535 from B and suppose that it is about to send a segment to Host B. Host A is waiting for byte 536 and all the subsequent bytes in Host B's data stream. So Host A puts 536 in the acknowledgment number field of the segment it sends to B.

As another example, suppose that Host A has received one segment from Host B containing bytes 0 through 535 and another segment containing bytes 900 through 1,000. For some reason Host A has not yet received bytes 536 through 899. In this example, Host A is still waiting for byte 536 (and beyond) in order to re-create B's data stream. Thus, A's next segment to B will contain 536 in the acknowledgment number field. Because TCP only acknowledges bytes up to the first missing byte in the stream, TCP is said to provide **cumulative acknowledgments**.

This last example also brings up an important but subtle issue. Host A received the third segment (bytes 900 through 1,000) before receiving the second segment (bytes 536 through 899). Thus, the third segment arrived out of order. The subtle issue is: What does a host do when it receives out-of-order segments in a TCP connection? Interestingly, the TCP RFCs do not impose any rules here and leave the decision up to the programmers implementing a TCP implementation. There are basically two choices: either (1) the receiver immediately discards out-of-order segments (which, as we discussed earlier, can simplify receiver design), or (2) the receiver keeps the out-of-order bytes and waits for the missing bytes to fill in the gaps. Clearly, the latter choice is more efficient in terms of network bandwidth, and is the approach taken in practice.

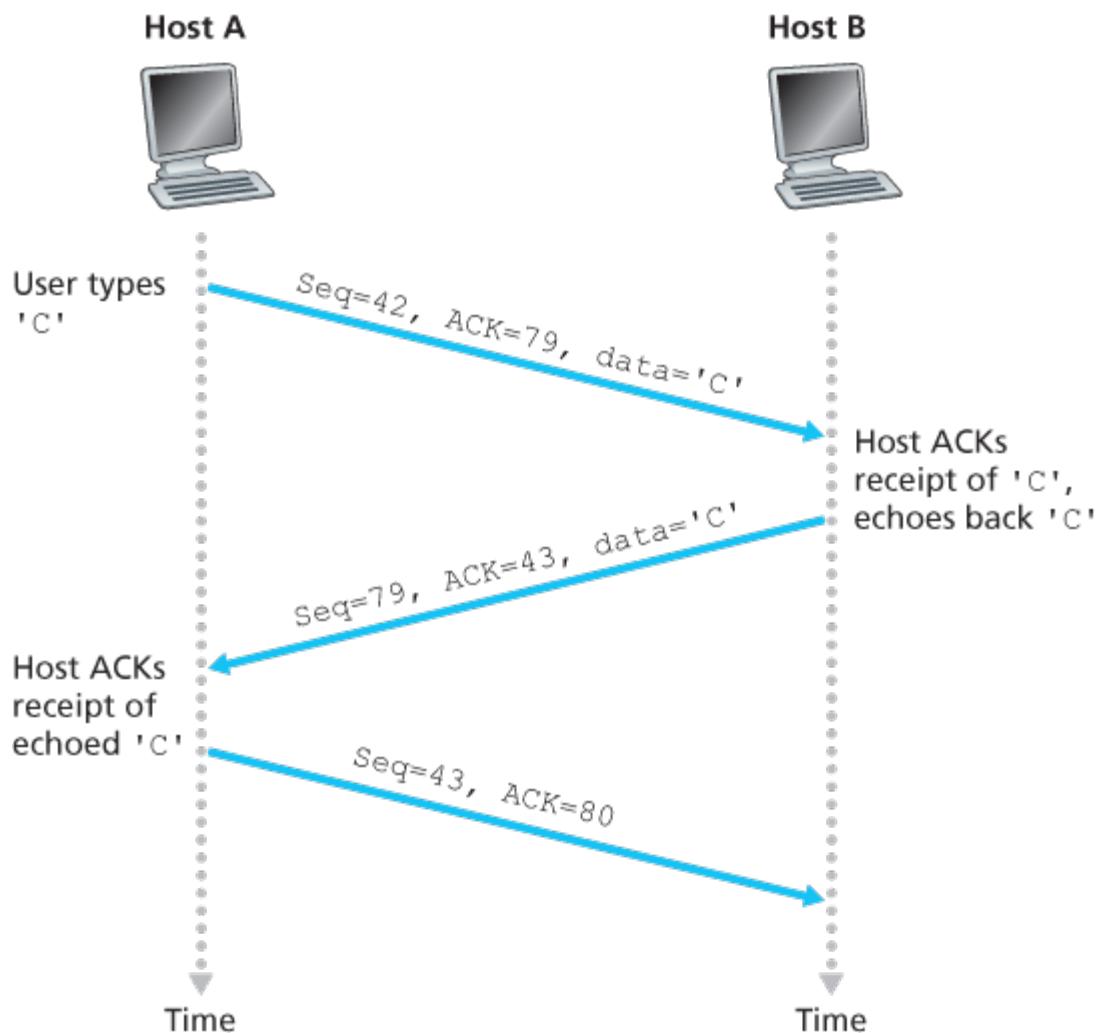
In [Figure 3.30](#), we assumed that the initial sequence number was zero. In truth, both sides of a TCP connection randomly choose an initial sequence number. This is done to minimize the possibility that a segment that is still present in the network from an earlier, already-terminated connection between two hosts is mistaken for a valid segment in a later connection between these same two hosts (which also happen to be using the same port numbers as the old connection) [\[Sunshine 1978\]](#).

#### *Telnet: A Case Study for Sequence and Acknowledgment Numbers*

Telnet, defined in RFC 854, is a popular application-layer protocol used for remote login. It runs over TCP and is designed to work between any pair of hosts. Unlike the bulk data transfer applications discussed in [Chapter 2](#), Telnet is an interactive application. We discuss a Telnet example here, as it nicely illustrates TCP sequence and acknowledgment numbers. We note that many users now prefer to use the SSH protocol rather than Telnet, since data sent in a Telnet connection (including passwords!) are not encrypted, making Telnet vulnerable to eavesdropping attacks (as discussed in [Section 8.7](#)).

Suppose Host A initiates a Telnet session with Host B. Because Host A initiates the session, it is labeled the client, and Host B is labeled the server. Each character typed by the user (at the client) will be sent to the remote host; the remote host will send back a copy of each character, which will be displayed on the Telnet user's screen. This "echo back" is used to ensure that characters seen by the Telnet user have already been received and processed at the remote site. Each character thus traverses the network twice between the time the user hits the key and the time the character is displayed on the user's monitor.

Now suppose the user types a single letter, 'C,' and then grabs a coffee. Let's examine the TCP segments that are sent between the client and server. As shown in [Figure 3.31](#), we suppose the starting sequence numbers are 42 and 79 for the client and server, respectively. Recall that the sequence number of a segment is the sequence number of the first byte in the data field. Thus, the first segment sent from the client will have sequence number 42; the first segment sent from the server will have sequence number 79. Recall that the acknowledgment number is the sequence



**Figure 3.31 Sequence and acknowledgment numbers for a simple Telnet application over TCP**

number of the next byte of data that the host is waiting for. After the TCP connection is established but before any data is sent, the client is waiting for byte 79 and the server is waiting for byte 42.

As shown in **Figure 3.31**, three segments are sent. The first segment is sent from the client to the server, containing the 1-byte ASCII representation of the letter ‘C’ in its data field. This first segment also has 42 in its sequence number field, as we just described. Also, because the client has not yet received any data from the server, this first segment will have 79 in its acknowledgment number field.

The second segment is sent from the server to the client. It serves a dual purpose. First it provides an acknowledgment of the data the server has received. By putting 43 in the acknowledgment field, the server is telling the client that it has successfully received everything up through byte 42 and is now waiting for bytes 43 onward. The second purpose of this segment is to echo back the letter ‘C.’ Thus, the second segment has the ASCII representation of ‘C’ in its data field. This second segment has the sequence number 79, the initial sequence number of the server-to-client data flow of this TCP connection, as this is the very first byte of data that the server is sending. Note that the acknowledgment for client-to-server data is carried in a segment carrying server-to-client data; this acknowledgment is said to be **piggybacked** on the server-to-client data segment.

The third segment is sent from the client to the server. Its sole purpose is to acknowledge the data it has received from the server. (Recall that the second segment contained data—the letter ‘C’—from the server to the client.) This segment has an empty data field (that is, the acknowledgment is not being piggybacked with any client-to-server data). The segment has 80 in the acknowledgment number field because the client has received the stream of bytes up through byte sequence number 79 and it is now waiting for bytes 80 onward. You might think it odd that this segment also has a sequence number since the segment contains no data. But because TCP has a sequence number field, the segment needs to have some sequence number.

### 3.5.3 Round-Trip Time Estimation and Timeout

TCP, like our *rdt* protocol in [Section 3.4](#), uses a timeout/retransmit mechanism to recover from lost segments. Although this is conceptually simple, many subtle issues arise when we implement a timeout/retransmit mechanism in an actual protocol such as TCP. Perhaps the most obvious question is the length of the timeout intervals. Clearly, the timeout should be larger than the connection’s round-trip time (RTT), that is, the time from when a segment is sent until it is acknowledged. Otherwise, unnecessary retransmissions would be sent. But how much larger? How should the RTT be estimated in the first place? Should a timer be associated with each and every unacknowledged segment? So many questions! Our discussion in this section is based on the TCP work in [\[Jacobson 1988\]](#) and the current IETF recommendations for managing TCP timers [\[RFC 6298\]](#).

#### *Estimating the Round-Trip Time*

Let’s begin our study of TCP timer management by considering how TCP estimates the round-trip time between sender and receiver. This is accomplished as follows. The sample RTT, denoted *SampleRTT*, for a segment is the amount of time between when the segment is sent (that is, passed to IP) and when an acknowledgment for the segment is received. Instead of measuring a *SampleRTT* for every transmitted segment, most TCP implementations take only one *SampleRTT* measurement at a time. That is, at any point in time, the *SampleRTT* is being estimated for only one of the transmitted but currently unacknowledged segments, leading to a new value of *SampleRTT* approximately once every RTT. Also, TCP never computes a *SampleRTT* for a segment that has been retransmitted; it only measures *SampleRTT* for segments that have been transmitted once [\[Karn 1987\]](#). (A problem at the end of the chapter asks you to consider why.)

Obviously, the *SampleRTT* values will fluctuate from segment to segment due to congestion in the routers and to the varying load on the end systems. Because of this fluctuation, any given *SampleRTT* value may be atypical. In order to estimate a typical RTT, it is therefore natural to take some sort of average of the *SampleRTT* values. TCP maintains an average, called *EstimatedRTT*, of the

*SampleRTT* values. Upon obtaining a new *SampleRTT*, TCP updates *EstimatedRTT* according to the following formula:

$$\text{EstimatedRTT} = (1 - \alpha) \cdot \text{EstimatedRTT} + \alpha \cdot \text{SampleRTT}$$

The formula above is written in the form of a programming-language statement—the new value of **EstimatedRTT** is a weighted combination of the previous value of **EstimatedRTT** and the new value for **SampleRTT**. The recommended value of  $\alpha$  is  $\alpha = 0.125$  (that is, 1/8) [RFC 6298], in which case the formula above becomes:

$$\text{EstimatedRTT} = 0.875 \cdot \text{EstimatedRTT} + 0.125 \cdot \text{SampleRTT}$$

Note that **EstimatedRTT** is a weighted average of the **SampleRTT** values. As discussed in a homework problem at the end of this chapter, this weighted average puts more weight on recent samples than on old samples. This is natural, as the more recent samples better reflect the current congestion in the network. In statistics, such an average is called an **exponential weighted moving average (EWMA)**.

The word “exponential” appears in EWMA because the weight of a given *SampleRTT* decays exponentially fast as the updates proceed. In the homework problems you will be asked to derive the exponential term in *EstimatedRTT*.

**Figure 3.32** shows the *SampleRTT* values and *EstimatedRTT* for a value of  $\alpha = 1/8$  for a TCP connection between *gaia.cs.umass.edu* (in Amherst, Massachusetts) to *fantasia.eurecom.fr* (in the south of France). Clearly, the variations in the *SampleRTT* are smoothed out in the computation of the *EstimatedRTT*.

In addition to having an estimate of the RTT, it is also valuable to have a measure of the variability of the RTT. [RFC 6298] defines the RTT variation, *DevRTT*, as an estimate of how much *SampleRTT* typically deviates from *EstimatedRTT*:

$$\text{DevRTT} = (1 - \beta) \cdot \text{DevRTT} + \beta \cdot |\text{SampleRTT} - \text{EstimatedRTT}|$$

Note that *DevRTT* is an EWMA of the difference between *SampleRTT* and *EstimatedRTT*. If the *SampleRTT* values have little fluctuation, then *DevRTT* will be small; on the other hand, if there is a lot of fluctuation, *DevRTT* will be large. The recommended value of  $\beta$  is 0.25.

## Setting and Managing the Retransmission Timeout Interval

Given values of *EstimatedRTT* and *DevRTT*, what value should be used for TCP's timeout interval? Clearly, the interval should be greater than or equal to

### PRINCIPLES IN PRACTICE

TCP provides reliable data transfer by using positive acknowledgments and timers in much the same way that we studied in [Section 3.4](#). TCP acknowledges data that has been received correctly, and it then retransmits segments when segments or their corresponding acknowledgments are thought to be lost or corrupted. Certain versions of TCP also have an implicit NAK mechanism—with TCP's fast retransmit mechanism, the receipt of three duplicate ACKs for a given segment serves as an implicit NAK for the following segment, triggering retransmission of that segment before timeout. TCP uses sequences of numbers to allow the receiver to identify lost or duplicate segments. Just as in the case of our reliable data transfer protocol, [\*rdt3.0\*](#), TCP cannot itself tell for certain if a segment, or its ACK, is lost, corrupted, or overly delayed. At the sender, TCP's response will be the same: retransmit the segment in question.

TCP also uses pipelining, allowing the sender to have multiple transmitted but yet-to-be-acknowledged segments outstanding at any given time. We saw earlier that pipelining can greatly improve a session's throughput when the ratio of the segment size to round-trip delay is small. The specific number of outstanding, unacknowledged segments that a sender can have is determined by TCP's flow-control and congestion-control mechanisms. TCP flow control is discussed at the end of this section; TCP congestion control is discussed in [Section 3.7](#). For the time being, we must simply be aware that the TCP sender uses pipelining.

*EstimatedRTT*, or unnecessary retransmissions would be sent. But the timeout interval should not be too much larger than *EstimatedRTT*; otherwise, when a segment is lost, TCP would not quickly retransmit the segment, leading to large data transfer delays. It is therefore desirable to set the timeout equal to the *EstimatedRTT* plus some margin. The margin should be large when there is a lot of fluctuation in the *SampleRTT* values; it should be small when there is little fluctuation. The value of *DevRTT* should thus come into play here. All of these considerations are taken into account in TCP's method for determining the retransmission timeout interval:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \cdot \text{DevRTT}$$

An initial *TimeoutInterval* value of 1 second is recommended [[RFC 6298](#)]. Also, when a timeout occurs, the value of *TimeoutInterval* is doubled to avoid a premature timeout occurring for a

subsequent segment that will soon be acknowledged. However, as soon as a segment is received and *EstimatedRTT* is updated, the *TimeoutInterval* is again computed using the formula above.

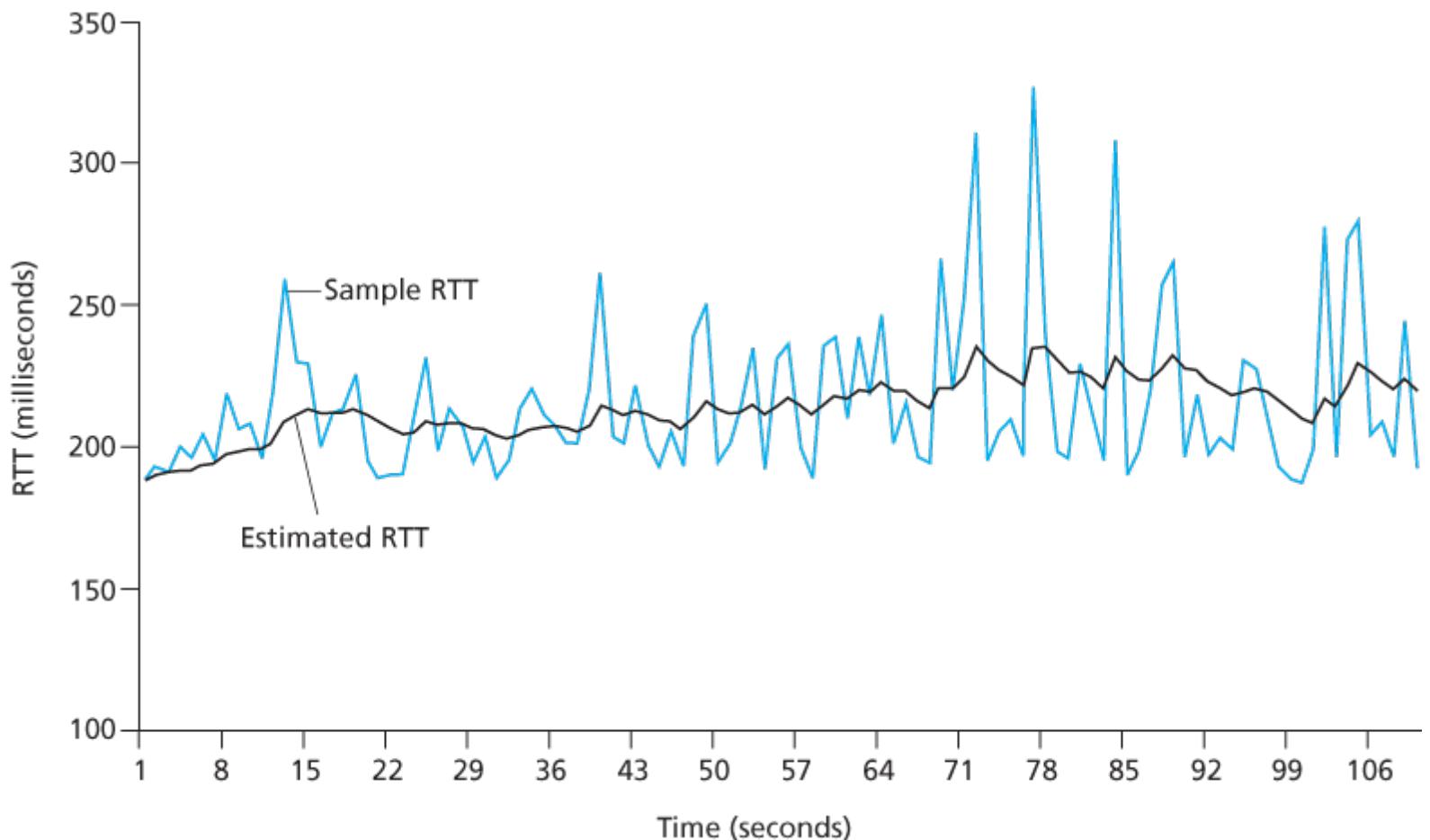


Figure 3.32 RTT samples and RTT estimates

### 3.5.4 Reliable Data Transfer

Recall that the Internet's network-layer service (IP service) is unreliable. IP does not guarantee datagram delivery, does not guarantee in-order delivery of datagrams, and does not guarantee the integrity of the data in the datagrams. With IP service, datagrams can overflow router buffers and never reach their destination, datagrams can arrive out of order, and bits in the datagram can get corrupted (flipped from 0 to 1 and vice versa). Because transport-layer segments are carried across the network by IP datagrams, transport-layer segments can suffer from these problems as well.

TCP creates a **reliable data transfer service** on top of IP's unreliable best-effort service. TCP's reliable data transfer service ensures that the data stream that a process reads out of its TCP receive buffer is uncorrupted, without gaps, without duplication, and in sequence; that is, the byte stream is exactly the same byte stream that was sent by the end system on the other side of the connection. How TCP provides a reliable data transfer involves many of the principles that we studied in [Section 3.4](#).

In our earlier development of reliable data transfer techniques, it was conceptually easiest to assume

that an individual timer is associated with each transmitted but not yet acknowledged segment. While this is great in theory, timer management can require considerable overhead. Thus, the recommended TCP timer management procedures [RFC 6298] use only a *single* retransmission timer, even if there are multiple transmitted but not yet acknowledged segments. The TCP protocol described in this section follows this single-timer recommendation.

We will discuss how TCP provides reliable data transfer in two incremental steps. We first present a highly simplified description of a TCP sender that uses only timeouts to recover from lost segments; we then present a more complete description that uses duplicate acknowledgments in addition to timeouts. In the ensuing discussion, we suppose that data is being sent in only one direction, from Host A to Host B, and that Host A is sending a large file.

**Figure 3.33** presents a highly simplified description of a TCP sender. We see that there are three major events related to data transmission and retransmission in the TCP sender: data received from application above; timer timeout; and ACK

```
/* Assume sender is not constrained by TCP flow or congestion control, that data from above is less
than MSS in size, and that data transfer is in one direction only. */

NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber

loop (forever) {
    switch(event)

        event: data received from application above
            create TCP segment with sequence number NextSeqNum
            if (timer currently not running)
                start timer
            pass segment to IP
            NextSeqNum=NextSeqNum+length(data)
            break;

        event: timer timeout
            retransmit not-yet-acknowledged segment with
                smallest sequence number
            start timer
            break;

        event: ACK received, with ACK field value of y
            if (y > SendBase) (
                SendBase=y
                if (there are currently any not-yet-acknowledged segments)
                    start timer
            )
            break;

    } /* end of loop forever */
```

**Figure 3.33 Simplified TCP sender**

receipt. Upon the occurrence of the first major event, TCP receives data from the application, encapsulates the data in a segment, and passes the segment to IP. Note that each segment includes a sequence number that is the byte-stream number of the first data byte in the segment, as described in [Section 3.5.2](#). Also note that if the timer is already not running for some other segment, TCP starts the timer when the segment is passed to IP. (It is helpful to think of the timer as being associated with the oldest unacknowledged segment.) The expiration interval for this timer is the *TimeoutInterval*, which is calculated from *EstimatedRTT* and *DevRTT*, as described in [Section 3.5.3](#).

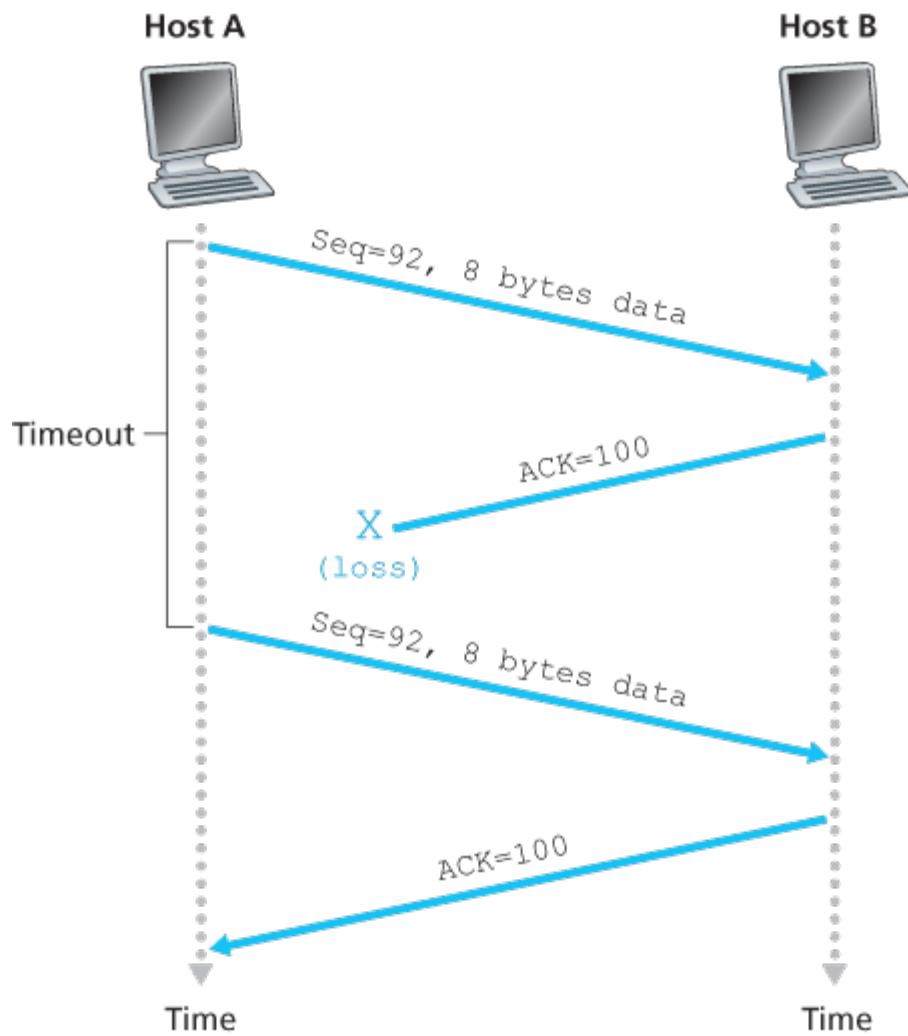
The second major event is the timeout. TCP responds to the timeout event by retransmitting the segment that caused the timeout. TCP then restarts the timer.

The third major event that must be handled by the TCP sender is the arrival of an acknowledgment segment (ACK) from the receiver (more specifically, a segment containing a valid ACK field value). On the occurrence of this event, TCP compares the ACK value *y* with its variable *SendBase*. The TCP state variable *SendBase* is the sequence number of the oldest unacknowledged byte. (Thus *SendBase-1* is the sequence number of the last byte that is known to have been received correctly and in order at the receiver.) As indicated earlier, TCP uses cumulative acknowledgments, so that *y* acknowledges the receipt of all bytes before byte number *y*. If *y* > *SendBase*, then the ACK is acknowledging one or more previously unacknowledged segments. Thus the sender updates its *SendBase* variable; it also restarts the timer if there currently are any not-yet-acknowledged segments.

### *A Few Interesting Scenarios*

We have just described a highly simplified version of how TCP provides reliable data transfer. But even this highly simplified version has many subtleties. To get a good feeling for how this protocol works, let's now walk through a few simple scenarios. [Figure 3.34](#) depicts the first scenario, in which Host A sends one segment to Host B. Suppose that this segment has sequence number 92 and contains 8 bytes of data. After sending this segment, Host A waits for a segment from B with acknowledgment number 100. Although the segment from A is received at B, the acknowledgment from B to A gets lost. In this case, the timeout event occurs, and Host A retransmits the same segment. Of course, when Host B receives the retransmission, it observes from the sequence number that the segment contains data that has already been received. Thus, TCP in Host B will discard the bytes in the retransmitted segment.

In a second scenario, shown in [Figure 3.35](#), Host A sends two segments back to back. The first segment has sequence number 92 and 8 bytes of data, and the second segment has sequence number 100 and 20 bytes of data. Suppose that both segments arrive intact at B, and B sends two separate acknowledgments for each of these segments. The first of these acknowledgments has acknowledgment number 100; the second has acknowledgment number 120. Suppose now that neither of the acknowledgments arrives at Host A before the timeout. When the timeout event occurs, Host



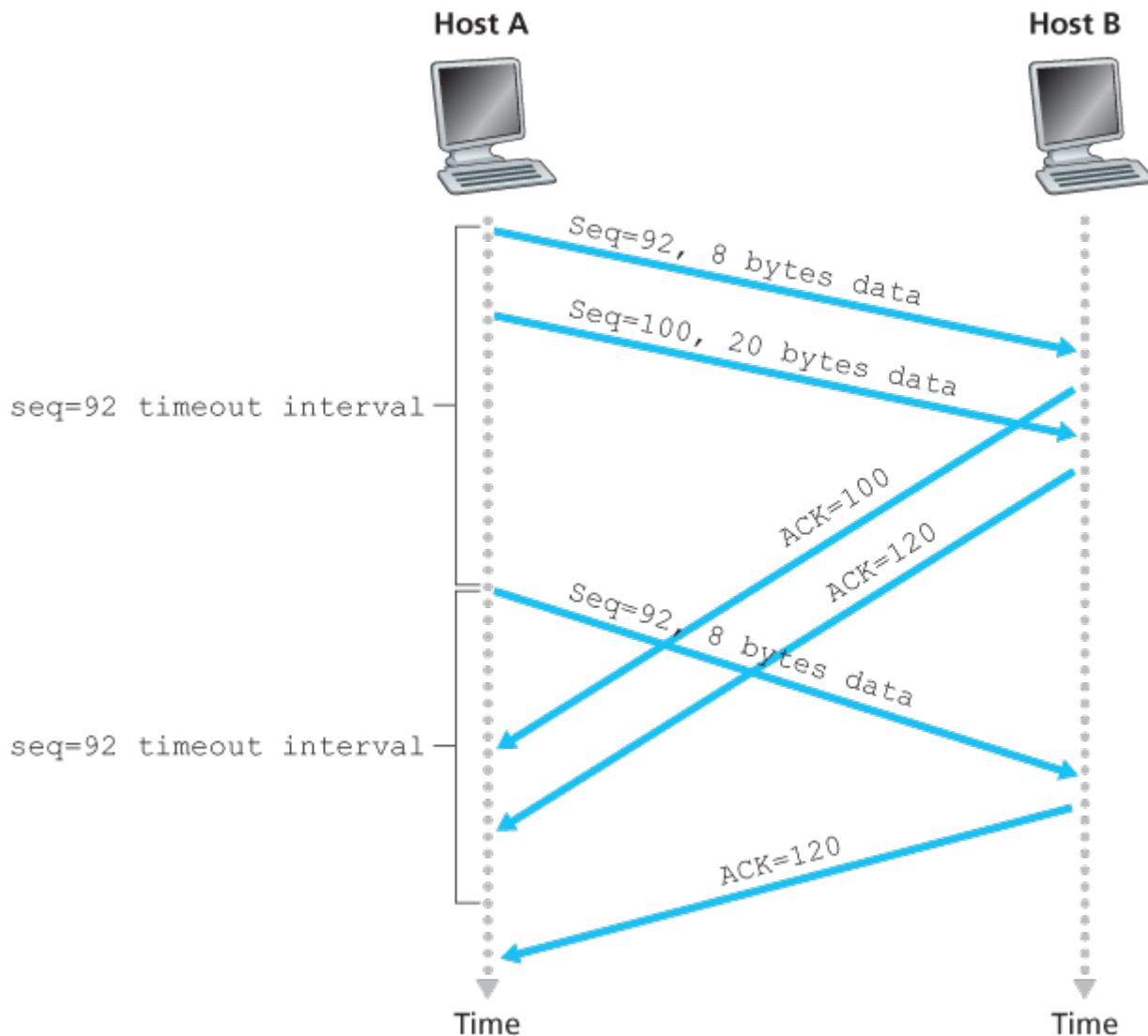
**Figure 3.34 Retransmission due to a lost acknowledgment**

A resends the first segment with sequence number 92 and restarts the timer. As long as the ACK for the second segment arrives before the new timeout, the second segment will not be retransmitted.

In a third and final scenario, suppose Host A sends the two segments, exactly as in the second example. The acknowledgment of the first segment is lost in the network, but just before the timeout event, Host A receives an acknowledgment with acknowledgment number 120. Host A therefore knows that Host B has received *everything* up through byte 119; so Host A does not resend either of the two segments. This scenario is illustrated in [Figure 3.36](#).

#### *Doubling the Timeout Interval*

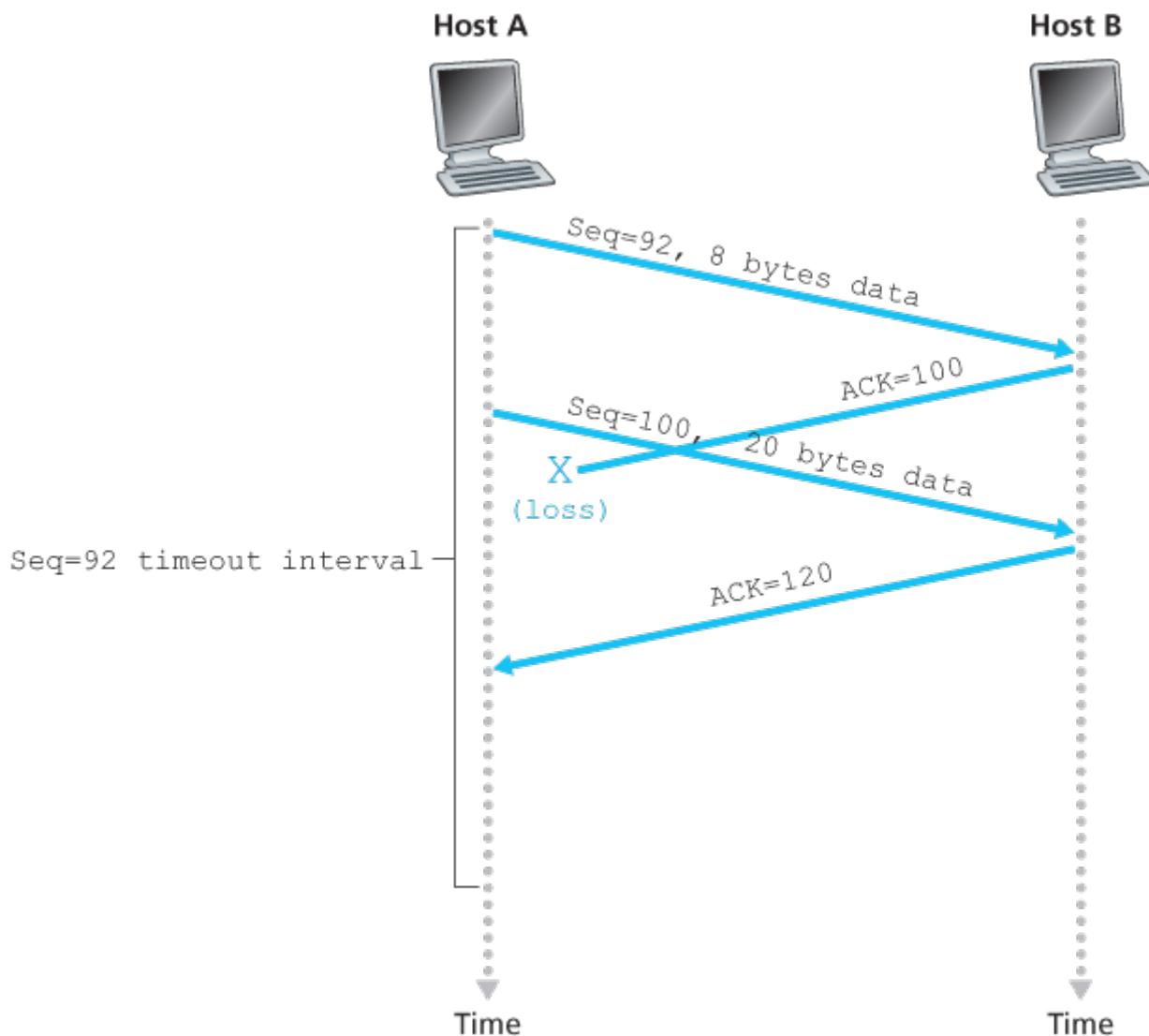
We now discuss a few modifications that most TCP implementations employ. The first concerns the length of the timeout interval after a timer expiration. In this modification, whenever the timeout event occurs, TCP retransmits the not-yet-acknowledged segment with the smallest sequence number, as described above. But each time TCP retransmits, it sets the next timeout interval to twice the previous value,



**Figure 3.35 Segment 100 not retransmitted**

rather than deriving it from the last *EstimatedRTT* and *DevRTT* (as described in [Section 3.5.3](#)). For example, suppose *TimeoutInterval* associated with the oldest not yet acknowledged segment is .75 sec when the timer first expires. TCP will then retransmit this segment and set the new expiration time to 1.5 sec. If the timer expires again 1.5 sec later, TCP will again retransmit this segment, now setting the expiration time to 3.0 sec. Thus the intervals grow exponentially after each retransmission. However, whenever the timer is started after either of the two other events (that is, data received from application above, and ACK received), the *TimeoutInterval* is derived from the most recent values of *EstimatedRTT* and *DevRTT*.

This modification provides a limited form of congestion control. (More comprehensive forms of TCP congestion control will be studied in [Section 3.7](#).) The timer expiration is most likely caused by congestion in the network, that is, too many packets arriving at one (or more) router queues in the path between the source and destination, causing packets to be dropped and/or long queuing delays. In times of congestion, if the sources continue to retransmit packets persistently, the congestion



**Figure 3.36 A cumulative acknowledgment avoids retransmission of the first segment**

may get worse. Instead, TCP acts more politely, with each sender retransmitting after longer and longer intervals. We will see that a similar idea is used by Ethernet when we study CSMA/CD in [Chapter 6](#).

#### *Fast Retransmit*

One of the problems with timeout-triggered retransmissions is that the timeout period can be relatively long. When a segment is lost, this long timeout period forces the sender to delay resending the lost packet, thereby increasing the end-to-end delay. Fortunately, the sender can often detect packet loss well before the timeout event occurs by noting so-called duplicate ACKs. A **duplicate ACK** is an ACK that reacknowledges a segment for which the sender has already received an earlier acknowledgment. To understand the sender's response to a duplicate ACK, we must look at why the receiver sends a duplicate ACK in the first place. [Table 3.2](#) summarizes the TCP receiver's ACK generation policy [\[RFC 5681\]](#). When a TCP receiver receives

**Table 3.2 TCP ACK Generation Recommendation [RFC 5681]**

Event	TCP Receiver Action

Arrival of in-order segment with expected sequence number. All data up to expected sequence number already acknowledged.	Delayed ACK. Wait up to 500 msec for arrival of another in-order segment. If next in-order segment does not arrive in this interval, send an ACK.
Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK transmission.	One Immediately send single cumulative ACK, ACKing both in-order segments.
Arrival of out-of-order segment with higher-than-expected sequence number. Gap detected.	Immediately send duplicate ACK, indicating sequence number of next expected byte (which is the lower end of the gap).
Arrival of segment that partially or completely fills in gap in received data.	Immediately send ACK, provided that segment starts at the lower end of gap.

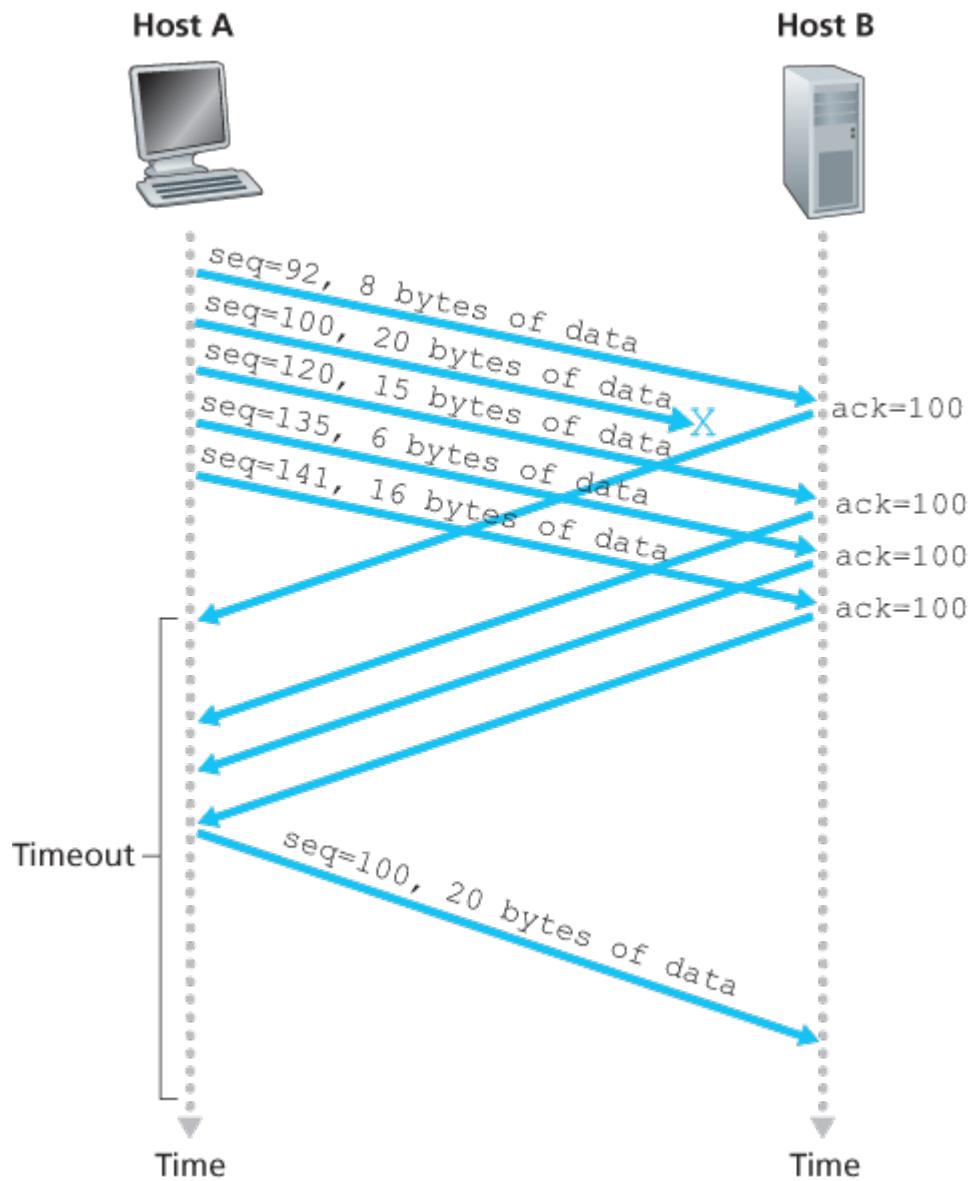
a segment with a sequence number that is larger than the next, expected, in-order sequence number, it detects a gap in the data stream—that is, a missing segment. This gap could be the result of lost or reordered segments within the network. Since TCP does not use negative acknowledgments, the receiver cannot send an explicit negative acknowledgment back to the sender. Instead, it simply reacknowledges (that is, generates a duplicate ACK for) the last in-order byte of data it has received.

(Note that [Table 3.2](#) allows for the case that the receiver does not discard out-of-order segments.)

Because a sender often sends a large number of segments back to back, if one segment is lost, there will likely be many back-to-back duplicate ACKs. If the TCP sender receives three duplicate ACKs for the same data, it takes this as an indication that the segment following the segment that has been ACKed three times has been lost. (In the homework problems, we consider the question of why the sender waits for three duplicate ACKs, rather than just a single duplicate ACK.) In the case that three duplicate ACKs are received, the TCP sender performs a [fast retransmit \[RFC 5681\]](#), retransmitting the missing segment *before* that segment's timer expires. This is shown in [Figure 3.37](#), where the second segment is lost, then retransmitted before its timer expires. For TCP with fast retransmit, the following code snippet replaces the ACK received event in [Figure 3.33](#):

```
event: ACK received, with ACK field value of y
    if (y > SendBase) {
        SendBase=y
        if (there are currently any not yet
            acknowledged segments)
            start timer
```

```
}
```



**Figure 3.37** Fast retransmit: retransmitting the missing segment before the segment's timer expires

```

else /* a duplicate ACK for already ACKed
      segment */
      increment number of duplicate ACKs
      received for y
      if (number of duplicate ACKS received
          for y==3)
          /* TCP fast retransmit */
          resend segment with sequence number y
      }
break;
}
```

We noted earlier that many subtle issues arise when a timeout/retransmit mechanism is implemented in an actual protocol such as TCP. The procedures above, which have evolved as a result of more than 20 years of experience with TCP timers, should convince you that this is indeed the case!

### Go-Back-N or Selective Repeat?

Let us close our study of TCP's error-recovery mechanism by considering the following question: Is TCP a GBN or an SR protocol? Recall that TCP acknowledgments are cumulative and correctly received but out-of-order segments are not individually ACKed by the receiver. Consequently, as shown in [Figure 3.33](#) (see also [Figure 3.19](#)), the TCP sender need only maintain the smallest sequence number of a transmitted but unacknowledged byte (`SendBase`) and the sequence number of the next byte to be sent (`NextSeqNum`). In this sense, TCP looks a lot like a GBN-style protocol. But there are some striking differences between TCP and Go-Back-N. Many TCP implementations will buffer correctly received but out-of-order segments [[Stevens 1994](#)]. Consider also what happens when the sender sends a sequence of segments  $1, 2, \dots, N$ , and all of the segments arrive in order without error at the receiver. Further suppose that the acknowledgment for packet  $n < N$  gets lost, but the remaining  $N-1$  acknowledgments arrive at the sender before their respective timeouts. In this example, GBN would retransmit not only packet  $n$ , but also all of the subsequent packets  $n+1, n+2, \dots, N$ . TCP, on the other hand, would retransmit at most one segment, namely, segment  $n$ . Moreover, TCP would not even retransmit segment  $n$  if the acknowledgment for segment  $n+1$  arrived before the timeout for segment  $n$ .

A proposed modification to TCP, the so-called [selective acknowledgment \[RFC 2018\]](#), allows a TCP receiver to acknowledge out-of-order segments selectively rather than just cumulatively acknowledging the last correctly received, in-order segment. When combined with selective retransmission—skipping the retransmission of segments that have already been selectively acknowledged by the receiver—TCP looks a lot like our generic SR protocol. Thus, TCP's error-recovery mechanism is probably best categorized as a hybrid of GBN and SR protocols.

### 3.5.5 Flow Control

Recall that the hosts on each side of a TCP connection set aside a receive buffer for the connection. When the TCP connection receives bytes that are correct and in sequence, it places the data in the receive buffer. The associated application process will read data from this buffer, but not necessarily at the instant the data arrives. Indeed, the receiving application may be busy with some other task and may not even attempt to read the data until long after it has arrived. If the application is relatively slow at reading the data, the sender can very easily overflow the connection's receive buffer by sending too much data too quickly.

TCP provides a **flow-control service** to its applications to eliminate the possibility of the sender overflowing the receiver's buffer. Flow control is thus a speed-matching service—matching the rate at which the sender is sending against the rate at which the receiving application is reading. As noted earlier, a TCP sender can also be throttled due to congestion within the IP network; this form of sender control is referred to as **congestion control**, a topic we will explore in detail in [Sections 3.6](#) and [3.7](#). Even though the actions taken by flow and congestion control are similar (the throttling of the sender), they are obviously taken for very different reasons. Unfortunately, many authors use the terms interchangeably, and the savvy reader would be wise to distinguish between them. Let's now discuss how TCP provides its flow-control service. In order to see the forest for the trees, we suppose throughout this section that the TCP implementation is such that the TCP receiver discards out-of-order segments.

TCP provides flow control by having the *sender* maintain a variable called the **receive window**. Informally, the receive window is used to give the sender an idea of how much free buffer space is available at the receiver. Because TCP is full-duplex, the sender at each side of the connection maintains a distinct receive window. Let's investigate the receive window in the context of a file transfer. Suppose that Host A is sending a large file to Host B over a TCP connection. Host B allocates a receive buffer to this connection; denote its size by *RcvBuffer*. From time to time, the application process in Host B reads from the buffer. Define the following variables:

- *LastByteRead*: the number of the last byte in the data stream read from the buffer by the application process in B
- *LastByteRcvd*: the number of the last byte in the data stream that has arrived from the network and has been placed in the receive buffer at B

Because TCP is not permitted to overflow the allocated buffer, we must have

$$LastByteRcvd - LastByteRead \leq RcvBuffer$$

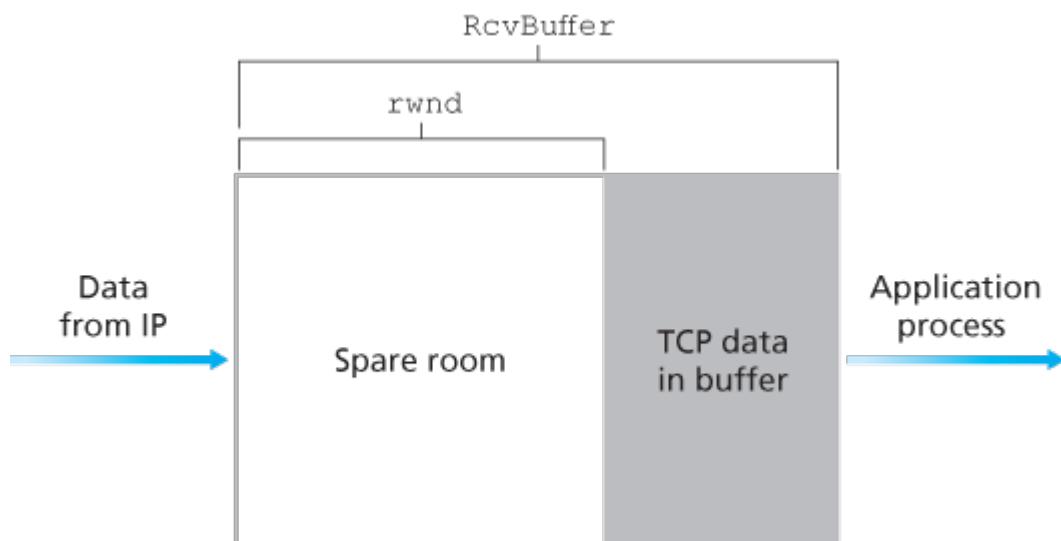
The receive window, denoted *rwnd* is set to the amount of spare room in the buffer:

$$rwnd = RcvBuffer - [LastByteRcvd - LastByteRead]$$

Because the spare room changes with time, *rwnd* is dynamic. The variable *rwnd* is illustrated in [Figure 3.38](#).

How does the connection use the variable `rwnd` to provide the flow-control service? Host B tells Host A how much spare room it has in the connection buffer by placing its current value of `rwnd` in the receive window field of every segment it sends to A. Initially, Host B sets `rwnd = RcvBuffer`. Note that to pull this off, Host B must keep track of several connection-specific variables.

Host A in turn keeps track of two variables, `LastByteSent` and `LastByteAcked`, which have obvious meanings. Note that the difference between these two variables, `LastByteSent - LastByteAcked`, is the amount of unacknowledged data that A has sent into the connection. By keeping the amount of unacknowledged data less than the value of `rwnd`, Host A is assured that it is not



**Figure 3.38 The receive window (`rwnd`) and the receive buffer (`RcvBuffer`)**

overflowing the receive buffer at Host B. Thus, Host A makes sure throughout the connection's life that

$$LastByteSent - LastByteAcked \leq rwnd$$

There is one minor technical problem with this scheme. To see this, suppose Host B's receive buffer becomes full so that `rwnd = 0`. After advertising `rwnd = 0` to Host A, also suppose that B has *nothing* to send to A. Now consider what happens. As the application process at B empties the buffer, TCP does not send new segments with new `rwnd` values to Host A; indeed, TCP sends a segment to Host A only if it has data to send or if it has an acknowledgment to send. Therefore, Host A is never informed that some space has opened up in Host B's receive buffer—Host A is blocked and can transmit no more data! To solve this problem, the TCP specification requires Host A to continue to send segments with one data byte when B's receive window is zero. These segments will be acknowledged by the receiver. Eventually the buffer will begin to empty and the acknowledgments will contain a nonzero `rwnd` value.

The online site at <http://www.awl.com/kurose-ross> for this book provides an interactive Java applet that illustrates the operation of the TCP receive window.

Having described TCP's flow-control service, we briefly mention here that UDP does not provide flow control and consequently, segments may be lost at the receiver due to buffer overflow. For example, consider sending a series of UDP segments from a process on Host A to a process on Host B. For a typical UDP implementation, UDP will append the segments in a finite-sized buffer that "precedes" the corresponding socket (that is, the door to the process). The process reads one entire segment at a time from the buffer. If the process does not read the segments fast enough from the buffer, the buffer will overflow and segments will get dropped.

### 3.5.6 TCP Connection Management

In this subsection we take a closer look at how a TCP connection is established and torn down. Although this topic may not seem particularly thrilling, it is important because TCP connection establishment can significantly add to perceived delays (for example, when surfing the Web). Furthermore, many of the most common network attacks—including the incredibly popular SYN flood attack—exploit vulnerabilities in TCP connection management. Let's first take a look at how a TCP connection is established. Suppose a process running in one host (client) wants to initiate a connection with another process in another host (server). The client application process first informs the client TCP that it wants to establish a connection to a process in the server. The TCP in the client then proceeds to establish a TCP connection with the TCP in the server in the following manner:

- **Step 1.** The client-side TCP first sends a special TCP segment to the server-side TCP. This special segment contains no application-layer data. But one of the flag bits in the segment's header (see [Figure 3.29](#)), the SYN bit, is set to 1. For this reason, this special segment is referred to as a SYN segment. In addition, the client randomly chooses an initial sequence number (`client_isn`) and puts this number in the sequence number field of the initial TCP SYN segment. This segment is encapsulated within an IP datagram and sent to the server. There has been considerable interest in properly randomizing the choice of the `client_isn` in order to avoid certain security attacks [\[CERT 2001–09\]](#).
- **Step 2.** Once the IP datagram containing the TCP SYN segment arrives at the server host (assuming it does arrive!), the server extracts the TCP SYN segment from the datagram, allocates the TCP buffers and variables to the connection, and sends a connection-granted segment to the client TCP. (We'll see in [Chapter 8](#) that the allocation of these buffers and variables before completing the third step of the three-way handshake makes TCP vulnerable to a denial-of-service attack known as SYN flooding.) This connection-granted segment also contains no application-layer data. However, it does contain three important pieces of information in the segment header. First, the SYN bit is set to 1. Second, the acknowledgment field of the TCP segment header is set to

`client_isn+1`. Finally, the server chooses its own initial sequence number (`server_isn`) and puts this value in the sequence number field of the TCP segment header. This connection-granted segment is saying, in effect, “I received your SYN packet to start a connection with your initial sequence number, `client_isn`. I agree to establish this connection. My own initial sequence number is `server_isn`.” The connection-granted segment is referred to as a **SYNACK segment**.

- **Step 3.** Upon receiving the SYNACK segment, the client also allocates buffers and variables to the connection. The client host then sends the server yet another segment; this last segment acknowledges the server’s connection-granted segment (the client does so by putting the value `server_isn+1` in the acknowledgment field of the TCP segment header). The SYN bit is set to zero, since the connection is established. This third stage of the three-way handshake may carry client-to-server data in the segment payload.

Once these three steps have been completed, the client and server hosts can send segments containing data to each other. In each of these future segments, the SYN bit will be set to zero. Note that in order to establish the connection, three packets are sent between the two hosts, as illustrated in

**Figure 3.39.** For this reason, this connection-establishment procedure is often referred to as a **three-way handshake**. Several aspects of the TCP three-way handshake are explored in the homework problems (Why are initial sequence numbers needed? Why is a three-way handshake, as opposed to a two-way handshake, needed?). It’s interesting to note that a rock climber and a belayer (who is stationed below the rock climber and whose job it is to handle the climber’s safety rope) use a three-way-handshake communication protocol that is identical to TCP’s to ensure that both sides are ready before the climber begins ascent.

All good things must come to an end, and the same is true with a TCP connection. Either of the two processes participating in a TCP connection can end the connection. When a connection ends, the “resources” (that is, the buffers and variables)

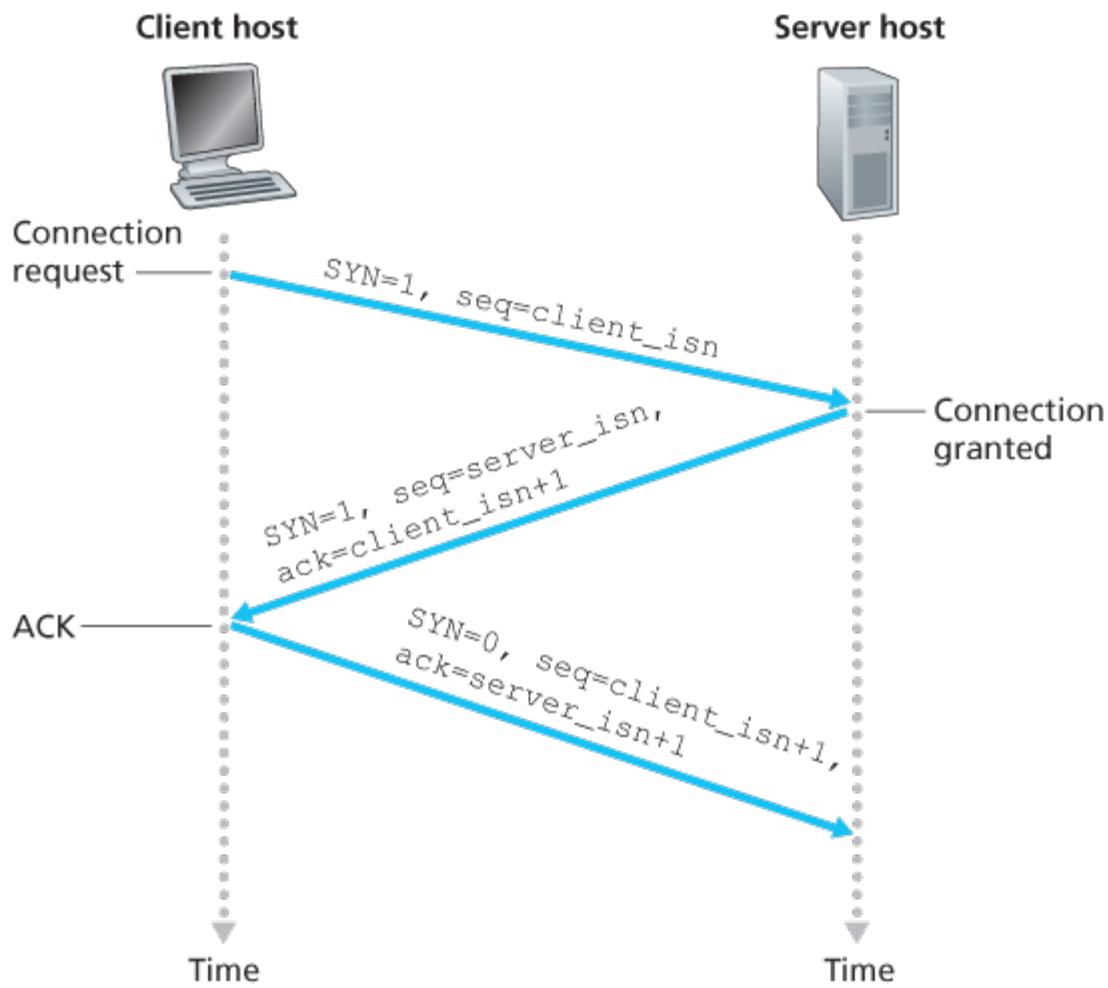
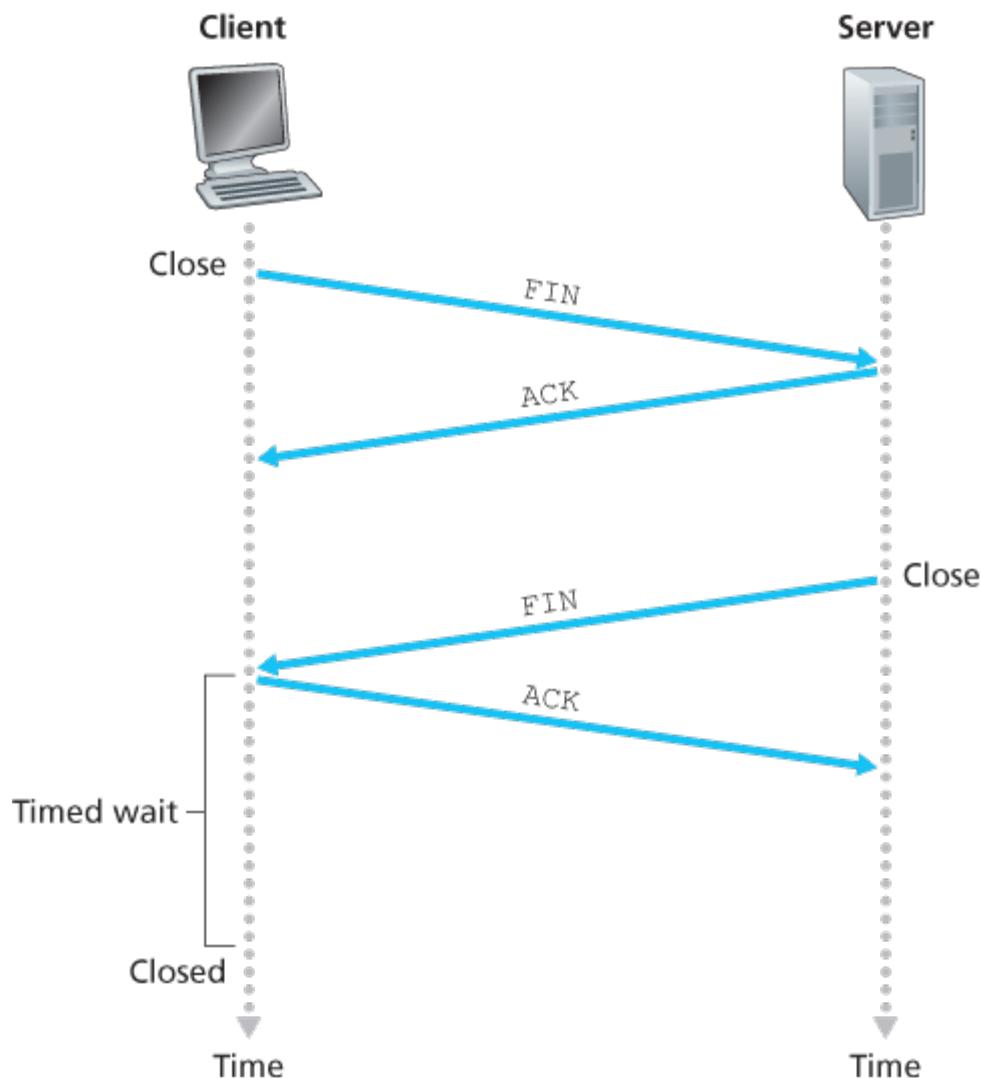


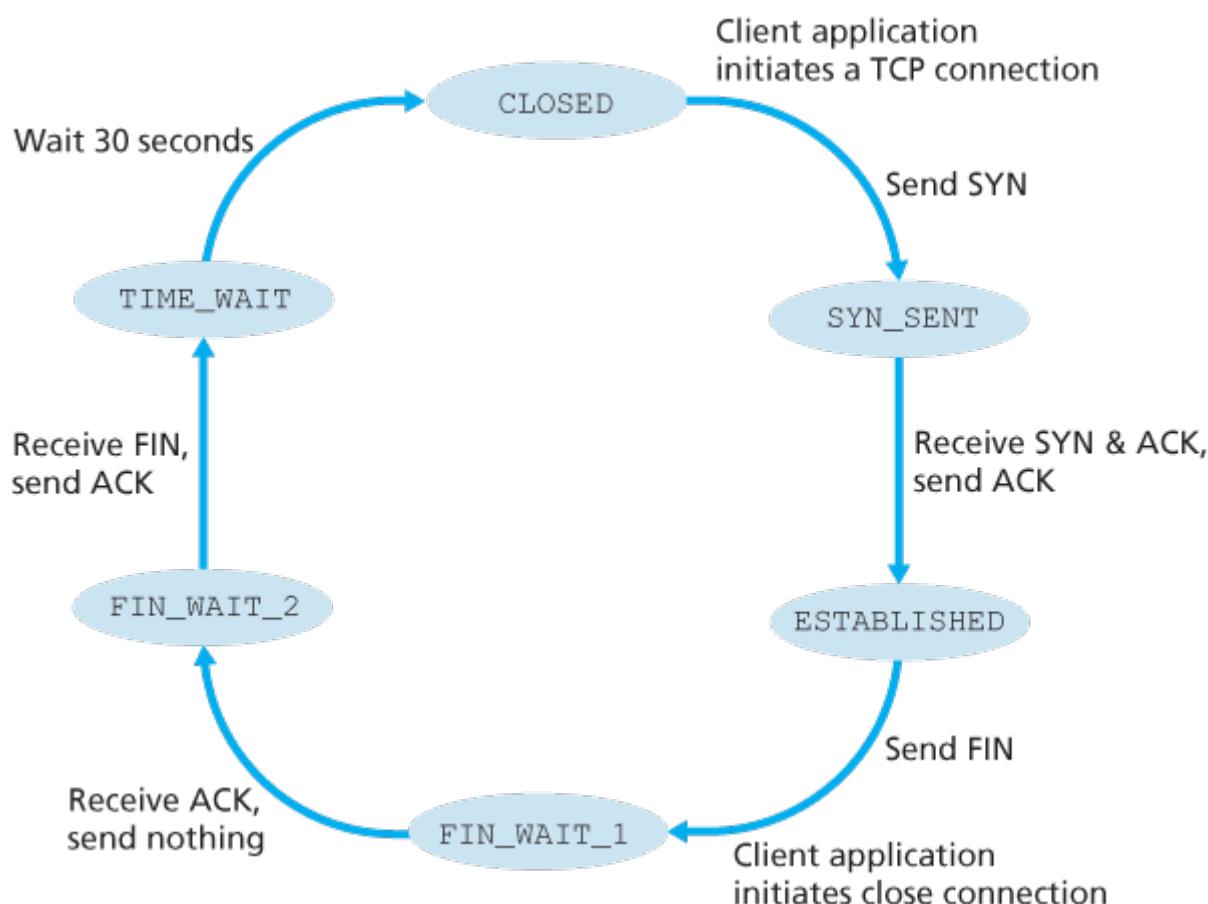
Figure 3.39 TCP three-way handshake: segment exchange



**Figure 3.40 Closing a TCP connection**

in the hosts are deallocated. As an example, suppose the client decides to close the connection, as shown in **Figure 3.40**. The client application process issues a close command. This causes the client TCP to send a special TCP segment to the server process. This special segment has a flag bit in the segment's header, the FIN bit (see **Figure 3.29**), set to 1. When the server receives this segment, it sends the client an acknowledgment segment in return. The server then sends its own shutdown segment, which has the FIN bit set to 1. Finally, the client acknowledges the server's shutdown segment. At this point, all the resources in the two hosts are now deallocated.

During the life of a TCP connection, the TCP protocol running in each host makes transitions through various **TCP states**. **Figure 3.41** illustrates a typical sequence of TCP states that are visited by the *client* TCP. The client TCP begins in the CLOSED state. The application on the client side initiates a new TCP connection (by creating a `Socket` object in our Java examples as in the Python examples from **Chapter 2**). This causes TCP in the client to send a SYN segment to TCP in the server. After having sent the SYN segment, the client TCP enters the SYN\_SENT state. While in the SYN\_SENT state, the client TCP waits for a segment from the server TCP that includes an acknowledgment for the client's previous segment and

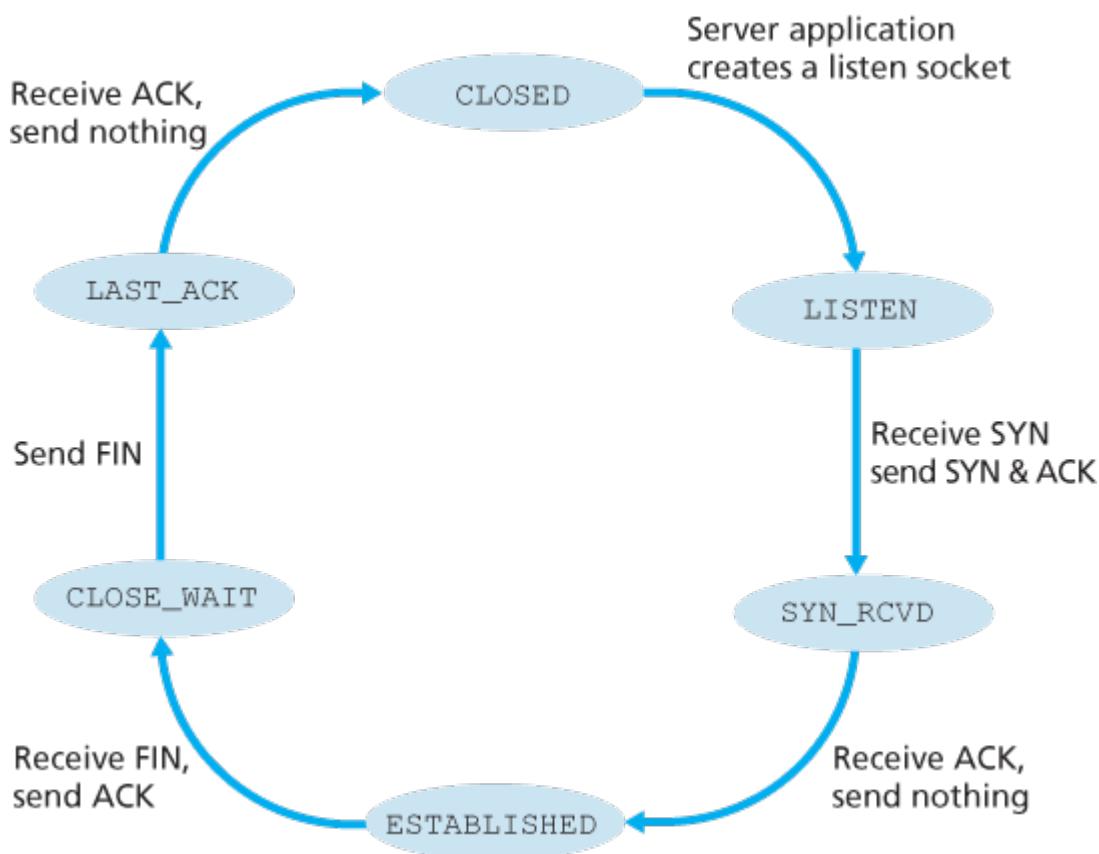


**Figure 3.41 A typical sequence of TCP states visited by a client TCP**

has the SYN bit set to 1. Having received such a segment, the client TCP enters the ESTABLISHED state. While in the ESTABLISHED state, the TCP client can send and receive TCP segments containing payload (that is, application-generated) data.

Suppose that the client application decides it wants to close the connection. (Note that the server could also choose to close the connection.) This causes the client TCP to send a TCP segment with the FIN bit set to 1 and to enter the FIN\_WAIT\_1 state. While in the FIN\_WAIT\_1 state, the client TCP waits for a TCP segment from the server with an acknowledgment. When it receives this segment, the client TCP enters the FIN\_WAIT\_2 state. While in the FIN\_WAIT\_2 state, the client waits for another segment from the server with the FIN bit set to 1; after receiving this segment, the client TCP acknowledges the server's segment and enters the TIME\_WAIT state. The TIME\_WAIT state lets the TCP client resend the final acknowledgment in case the ACK is lost. The time spent in the TIME\_WAIT state is implementation-dependent, but typical values are 30 seconds, 1 minute, and 2 minutes. After the wait, the connection formally closes and all resources on the client side (including port numbers) are released.

**Figure 3.42** illustrates the series of states typically visited by the server-side TCP, assuming the client begins connection teardown. The transitions are self-explanatory. In these two state-transition diagrams, we have only shown how a TCP connection is normally established and shut down. We have not described what happens in certain pathological scenarios, for example, when both sides of a connection want to initiate or shut down at the same time. If you are interested in learning about



**Figure 3.42 A typical sequence of TCP states visited by a server-side TCP**

this and other advanced issues concerning TCP, you are encouraged to see Stevens' comprehensive book [[Stevens 1994](#)].

Our discussion above has assumed that both the client and server are prepared to communicate, i.e., that the server is listening on the port to which the client sends its SYN segment. Let's consider what happens when a host receives a TCP segment whose port numbers or source IP address do not match with any of the ongoing sockets in the host. For example, suppose a host receives a TCP SYN packet with destination port 80, but the host is not accepting connections on port 80 (that is, it is not running a Web server on port 80). Then the host will send a special reset segment to the source. This TCP segment has the RST flag bit (see [Section 3.5.2](#)) set to 1. Thus, when a host sends a reset segment, it is telling the source "I don't have a socket for that segment. Please do not resend the segment." When a host receives a UDP packet whose destination port number doesn't match with an ongoing UDP socket, the host sends a special ICMP datagram, as discussed in [Chapter 5](#).

Now that we have a good understanding of TCP connection management, let's revisit the nmap port-scanning tool and examine more closely how it works. To explore a specific TCP port, say port 6789, on a target host, nmap will send a TCP SYN segment with destination port 6789 to that host. There are three possible outcomes:

- **The source host receives a TCP SYNACK segment from the target host.** Since this means that an application is running with TCP port 6789 on the target post, nmap returns "open."

#### FOCUS ON SECURITY

##### *The Syn Flood Attack*

We've seen in our discussion of TCP's three-way handshake that a server allocates and initializes connection variables and buffers in response to a received SYN. The server then sends a SYNACK in response, and awaits an ACK segment from the client. If the client does not send an ACK to complete the third step of this 3-way handshake, eventually (often after a minute or more) the server will terminate the half-open connection and reclaim the allocated resources.

This TCP connection management protocol sets the stage for a classic Denial of Service (DoS) attack known as the **SYN flood attack**. In this attack, the attacker(s) send a large number of TCP SYN segments, without completing the third handshake step. With this deluge of SYN segments, the server's connection resources become exhausted as they are allocated (but never used!) for half-open connections; legitimate clients are then denied service. Such SYN flooding attacks were among the first documented DoS attacks [[CERT SYN 1996](#)]. Fortunately, an effective defense known as **SYN cookies** [[RFC 4987](#)] are now deployed in most major operating systems. SYN cookies work as follows:

- When the server receives a SYN segment, it does not know if the segment is coming

from a legitimate user or is part of a SYN flood attack. So, instead of creating a half-open TCP connection for this SYN, the server creates an initial TCP sequence number that is a complicated function (hash function) of source and destination IP addresses and port numbers of the SYN segment, as well as a secret number only known to the server. This carefully crafted initial sequence number is the so-called “cookie.” The server then sends the client a SYNACK packet with this special initial sequence number. *Importantly, the server does not remember the cookie or any other state information corresponding to the SYN.*

- A legitimate client will return an ACK segment. When the server receives this ACK, it must verify that the ACK corresponds to some SYN sent earlier. But how is this done if the server maintains no memory about SYN segments? As you may have guessed, it is done with the cookie. Recall that for a legitimate ACK, the value in the acknowledgment field is equal to the initial sequence number in the SYNACK (the cookie value in this case) plus one (see [Figure 3.39](#)). The server can then run the same hash function using the source and destination IP address and port numbers in the SYNACK (which are the same as in the original SYN) and the secret number. If the result of the function plus one is the same as the acknowledgment (cookie) value in the client’s SYNACK, the server concludes that the ACK corresponds to an earlier SYN segment and is hence valid. The server then creates a fully open connection along with a socket.
- On the other hand, if the client does not return an ACK segment, then the original SYN has done no harm at the server, since the server hasn’t yet allocated any resources in response to the original bogus SYN.

- **The source host receives a TCP RST segment from the target host.** This means that the SYN segment reached the target host, but the target host is not running an application with TCP port 6789. But the attacker at least knows that the segments destined to the host at port 6789 are not blocked by any firewall on the path between source and target hosts. (Firewalls are discussed in [Chapter 8](#).)
- **The source receives nothing.** This likely means that the SYN segment was blocked by an intervening firewall and never reached the target host.

Nmap is a powerful tool that can “case the joint” not only for open TCP ports, but also for open UDP ports, for firewalls and their configurations, and even for the versions of applications and operating systems. Most of this is done by manipulating TCP connection-management segments [\[Skoudis 2006\]](#). You can download nmap from [www.nmap.org](http://www.nmap.org).

This completes our introduction to error control and flow control in TCP. In [Section 3.7](#) we’ll return to TCP and look at TCP congestion control in some depth. Before doing so, however, we first step back and examine congestion-control issues in a broader context.



## 3.6 Principles of Congestion Control

In the previous sections, we examined both the general principles and specific TCP mechanisms used to provide for a reliable data transfer service in the face of packet loss. We mentioned earlier that, in practice, such loss typically results from the overflowing of router buffers as the network becomes congested. Packet retransmission thus treats a symptom of network congestion (the loss of a specific transport-layer segment) but does not treat the cause of network congestion—too many sources attempting to send data at too high a rate. To treat the cause of network congestion, mechanisms are needed to throttle senders in the face of network congestion.

In this section, we consider the problem of congestion control in a general context, seeking to understand why congestion is a bad thing, how network congestion is manifested in the performance received by upper-layer applications, and various approaches that can be taken to avoid, or react to, network congestion. This more general study of congestion control is appropriate since, as with reliable data transfer, it is high on our “top-ten” list of fundamentally important problems in networking. The following section contains a detailed study of TCP’s congestion-control algorithm.

### 3.6.1 The Causes and the Costs of Congestion

Let’s begin our general study of congestion control by examining three increasingly complex scenarios in which congestion occurs. In each case, we’ll look at why congestion occurs in the first place and at the cost of congestion (in terms of resources not fully utilized and poor performance received by the end systems). We’ll not (yet) focus on how to react to, or avoid, congestion but rather focus on the simpler issue of understanding what happens as hosts increase their transmission rate and the network becomes congested.

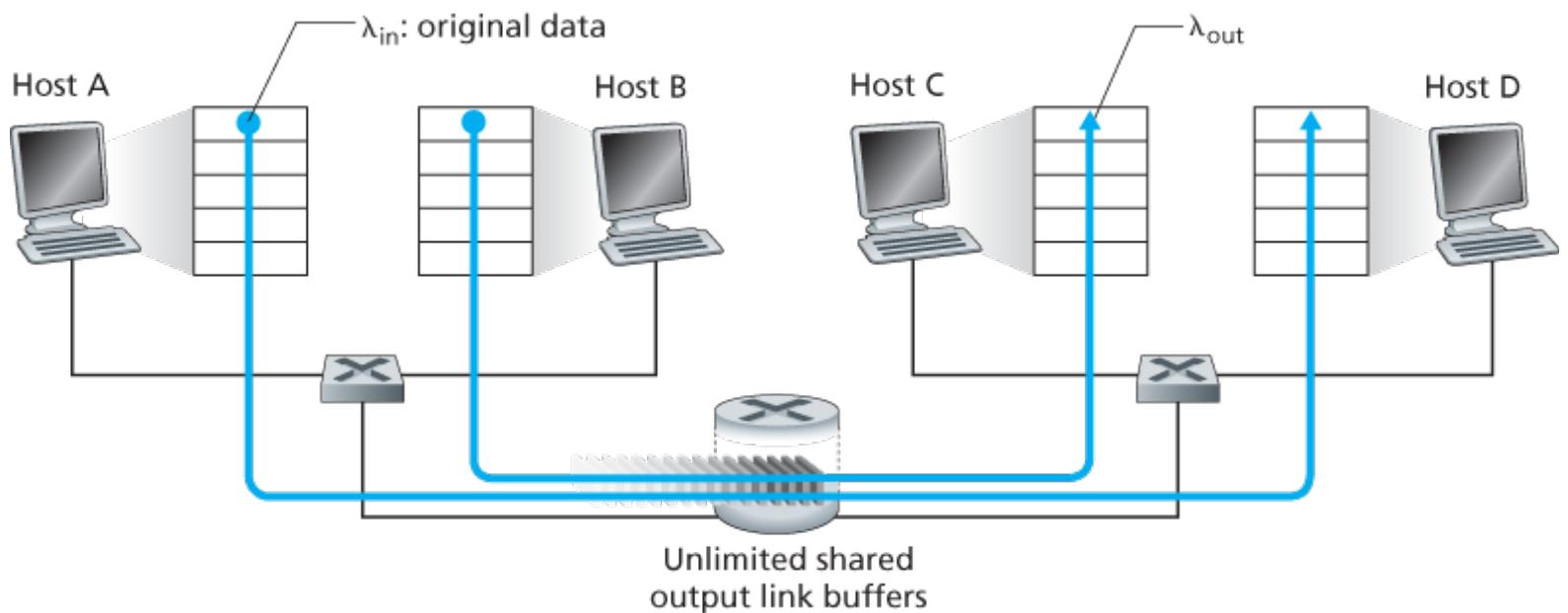
#### *Scenario 1: Two Senders, a Router with Infinite Buffers*

We begin by considering perhaps the simplest congestion scenario possible: Two hosts (A and B) each have a connection that shares a single hop between source and destination, as shown in [Figure 3.43](#).

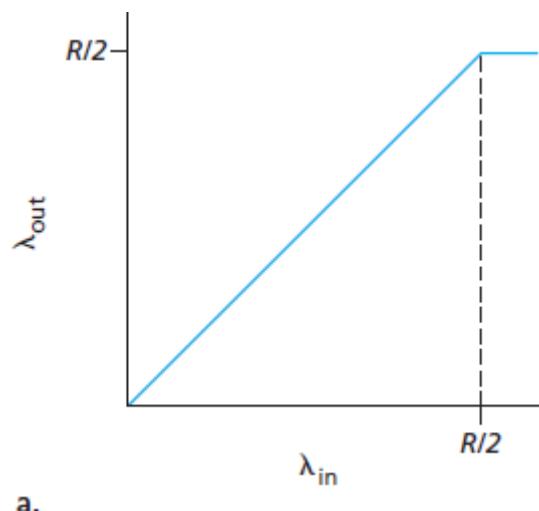
Let’s assume that the application in Host A is sending data into the connection (for example, passing data to the transport-level protocol via a socket) at an average rate of  $\lambda_{in}$  bytes/sec. These data are original in the sense that each unit of data is sent into the socket only once. The underlying transport-level protocol is a simple one. Data is encapsulated and sent; no error recovery (for example,

retransmission), flow control, or congestion control is performed. Ignoring the additional overhead due to adding transport- and lower-layer header information, the rate at which Host A offers traffic to the router in this first scenario is thus  $\lambda_{in}$  bytes/sec. Host B operates in a similar manner, and we assume for simplicity that it too is sending at a rate of  $\lambda_{in}$  bytes/sec. Packets from Hosts A and B pass through a router and over a shared outgoing link of capacity  $R$ . The router has buffers that allow it to store incoming packets when the packet-arrival rate exceeds the outgoing link's capacity. In this first scenario, we assume that the router has an infinite amount of buffer space.

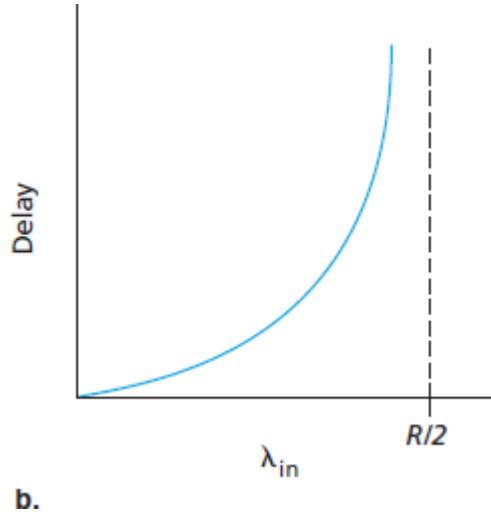
**Figure 3.44** plots the performance of Host A's connection under this first scenario. The left graph plots the **per-connection throughput** (number of bytes per



**Figure 3.43 Congestion scenario 1: Two connections sharing a single hop with infinite buffers**



**Figure 3.44 Congestion scenario 1: Throughput and delay as a function of host sending rate**

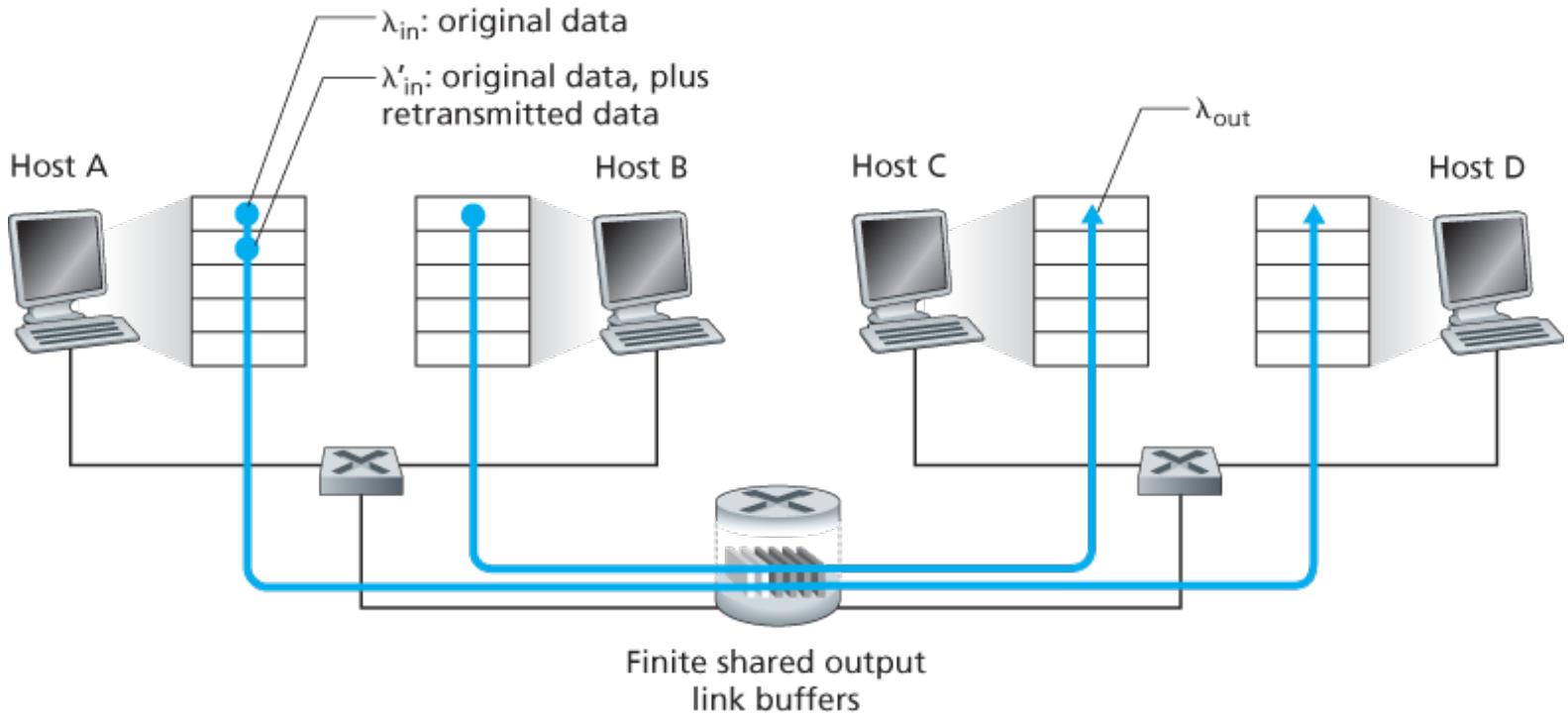


second at the receiver) as a function of the connection-sending rate. For a sending rate between 0 and  $R/2$ , the throughput at the receiver equals the sender's sending rate—everything sent by the sender is received at the receiver with a finite delay. When the sending rate is above  $R/2$ , however, the throughput is only  $R/2$ . This upper limit on throughput is a consequence of the sharing of link capacity between two connections. The link simply cannot deliver packets to a receiver at a steady-state rate that exceeds  $R/2$ . No matter how high Hosts A and B set their sending rates, they will each never see a throughput higher than  $R/2$ .

Achieving a per-connection throughput of  $R/2$  might actually appear to be a good thing, because the link is fully utilized in delivering packets to their destinations. The right-hand graph in [Figure 3.44](#), however, shows the consequence of operating near link capacity. As the sending rate approaches  $R/2$  (from the left), the average delay becomes larger and larger. When the sending rate exceeds  $R/2$ , the average number of queued packets in the router is unbounded, and the average delay between source and destination becomes infinite (assuming that the connections operate at these sending rates for an infinite period of time and there is an infinite amount of buffering available). Thus, while operating at an aggregate throughput of near  $R$  may be ideal from a throughput standpoint, it is far from ideal from a delay standpoint. *Even in this (extremely) idealized scenario, we've already found one cost of a congested network—large queuing delays are experienced as the packet-arrival rate nears the link capacity.*

### Scenario 2: Two Senders and a Router with Finite Buffers

Let's now slightly modify scenario 1 in the following two ways (see [Figure 3.45](#)). First, the amount of router buffering is assumed to be finite. A consequence of this real-world assumption is that packets will be dropped when arriving to an already-full buffer. Second, we assume that each connection is reliable. If a packet containing

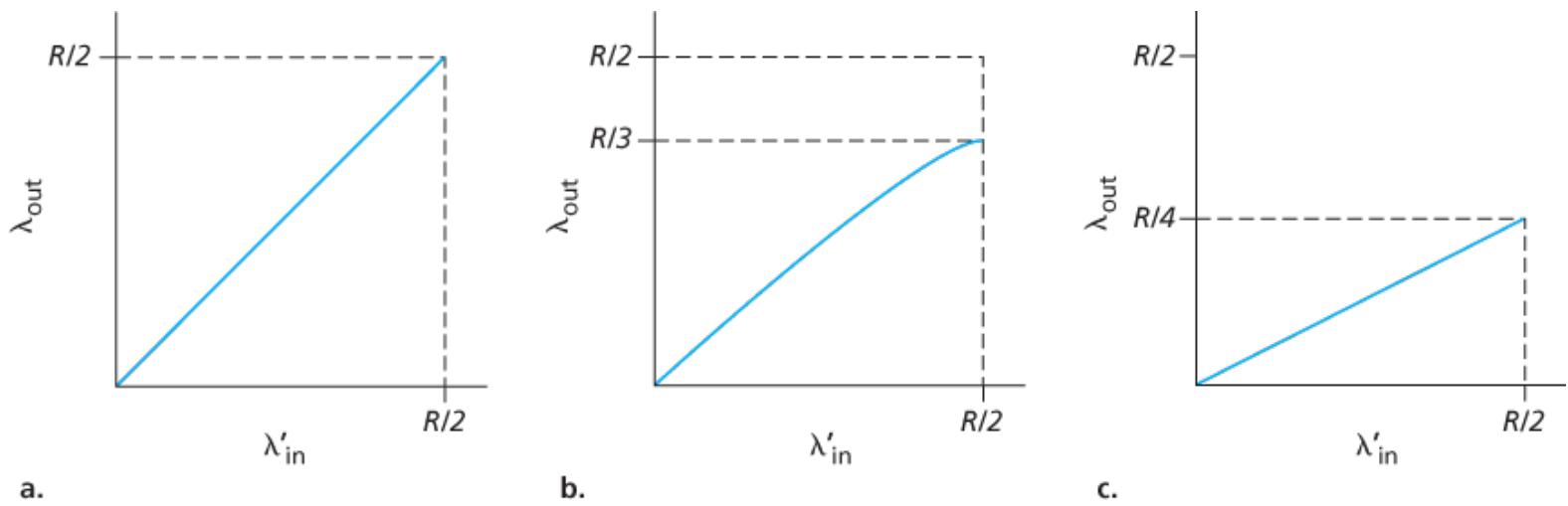


**Figure 3.45 Scenario 2: Two hosts (with retransmissions) and a router with finite buffers**

a transport-level segment is dropped at the router, the sender will eventually retransmit it. Because packets can be retransmitted, we must now be more careful with our use of the term *sending rate*. Specifically, let us again denote the rate at which the application sends original data into the socket by  $\lambda_{in}$  bytes/sec. The rate at which the transport layer sends segments (containing original data *and* retransmitted data) into the network will be denoted  $\lambda'_in$  bytes/sec.  $\lambda'_in$  is sometimes referred to as the **offered load** to the network.

The performance realized under scenario 2 will now depend strongly on how retransmission is performed. First, consider the unrealistic case that Host A is able to somehow (magically!) determine whether or not a buffer is free in the router and thus sends a packet only when a buffer is free. In this case, no loss would occur,  $\lambda_{in}$  would be equal to  $\lambda'_in$ , and the throughput of the connection would be equal to  $\lambda_{in}$ . This case is shown in [Figure 3.46\(a\)](#). From a throughput standpoint, performance is ideal—everything that is sent is received. Note that the average host sending rate cannot exceed  $R/2$  under this scenario, since packet loss is assumed never to occur.

Consider next the slightly more realistic case that the sender retransmits only when a packet is known for certain to be lost. (Again, this assumption is a bit of a stretch. However, it is possible that the sending host might set its timeout large enough to be virtually assured that a packet that has not been acknowledged has been lost.) In this case, the performance might look something like that shown in [Figure 3.46\(b\)](#). To appreciate what is happening here, consider the case that the offered load,  $\lambda'_in$  (the rate of original data transmission plus retransmissions), equals  $R/2$ . According to [Figure 3.46\(b\)](#), at this value of the offered load, the rate at which data



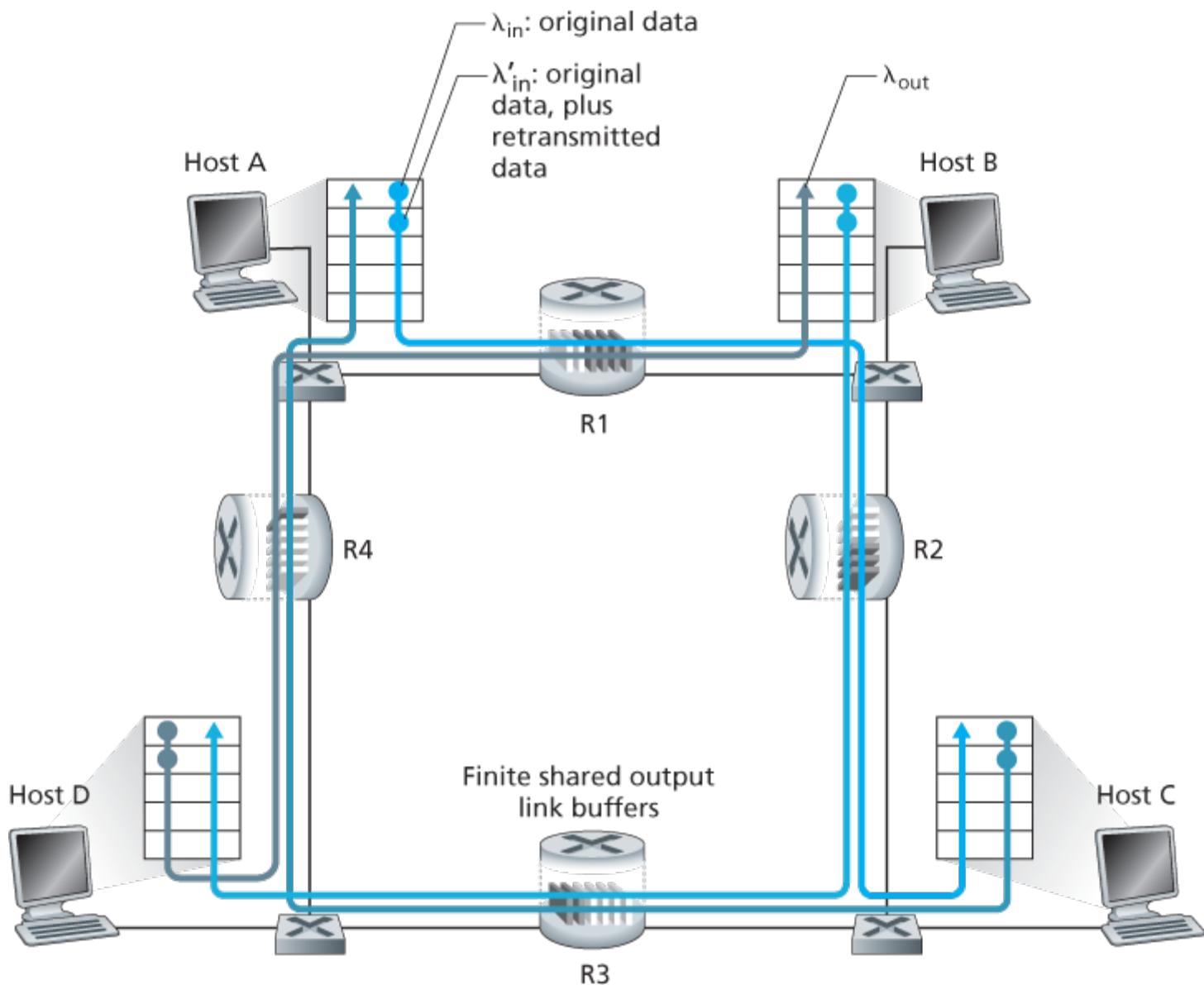
**Figure 3.46 Scenario 2 performance with finite buffers**

are delivered to the receiver application is  $R/3$ . Thus, out of the  $0.5R$  units of data transmitted,  $0.333R$  bytes/sec (on average) are original data and  $0.166R$  bytes/sec (on average) are retransmitted data. We see here another cost of a congested network—the sender must perform retransmissions in order to compensate for dropped (lost) packets due to buffer overflow.

Finally, let us consider the case that the sender may time out prematurely and retransmit a packet that has been delayed in the queue but not yet lost. In this case, both the original data packet and the retransmission may reach the receiver. Of course, the receiver needs but one copy of this packet and will discard the retransmission. In this case, the work done by the router in forwarding the retransmitted copy of the original packet was wasted, as the receiver will have already received the original copy of this packet. The router would have better used the link transmission capacity to send a different packet instead. Here then is yet another cost of a congested network—unnecessary retransmissions by the sender in the face of large delays may cause a router to use its link bandwidth to forward unnecessary copies of a packet. **Figure 3.46 (c)** shows the throughput versus offered load when each packet is assumed to be forwarded (on average) twice by the router. Since each packet is forwarded twice, the throughput will have an asymptotic value of  $R/4$  as the offered load approaches  $R/2$ .

### Scenario 3: Four Senders, Routers with Finite Buffers, and Multihop Paths

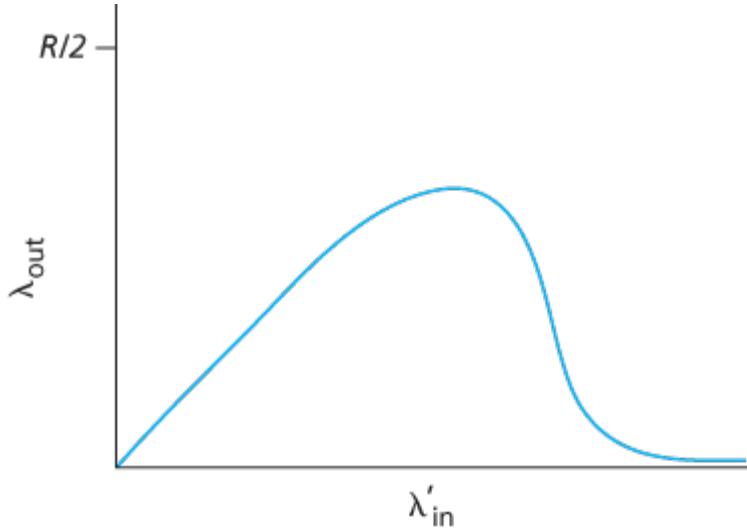
In our final congestion scenario, four hosts transmit packets, each over overlapping two-hop paths, as shown in **Figure 3.47**. We again assume that each host uses a timeout/retransmission mechanism to implement a reliable data transfer service, that all hosts have the same value of  $\lambda_{\text{in}}$ , and that all router links have capacity  $R$  bytes/sec.



**Figure 3.47 Four senders, routers with finite buffers, and multihop paths**

Let's consider the connection from Host A to Host C, passing through routers R1 and R2. The A–C connection shares router R1 with the D–B connection and shares router R2 with the B–D connection. For extremely small values of  $\lambda_{in}$ , buffer overflows are rare (as in congestion scenarios 1 and 2), and the throughput approximately equals the offered load. For slightly larger values of  $\lambda_{in}$ , the corresponding throughput is also larger, since more original data is being transmitted into the network and delivered to the destination, and overflows are still rare. Thus, for small values of  $\lambda_{in}$ , an increase in  $\lambda_{in}$  results in an increase in  $\lambda_{out}$ .

Having considered the case of extremely low traffic, let's next examine the case that  $\lambda_{in}$  (and hence  $\lambda'_{in}$ ) is extremely large. Consider router R2. The A–C traffic arriving to router R2 (which arrives at R2 after being forwarded from R1) can have an arrival rate at R2 that is at most  $R$ , the capacity of the link from R1 to R2, regardless of the value of  $\lambda_{in}$ . If  $\lambda'_{in}$  is extremely large for all connections (including the



**Figure 3.48 Scenario 3 performance with finite buffers and multihop paths**

B–D connection), then the arrival rate of B–D traffic at R2 can be much larger than that of the A–C traffic. Because the A–C and B–D traffic must compete at router R2 for the limited amount of buffer space, the amount of A–C traffic that successfully gets through R2 (that is, is not lost due to buffer overflow) becomes smaller and smaller as the offered load from B–D gets larger and larger. In the limit, as the offered load approaches infinity, an empty buffer at R2 is immediately filled by a B–D packet, and the throughput of the A–C connection at R2 goes to zero. This, in turn, *implies that the A–C end-to-end throughput goes to zero* in the limit of heavy traffic. These considerations give rise to the offered load versus throughput tradeoff shown in [Figure 3.48](#).

The reason for the eventual decrease in throughput with increasing offered load is evident when one considers the amount of wasted work done by the network. In the high-traffic scenario outlined above, whenever a packet is dropped at a second-hop router, the work done by the first-hop router in forwarding a packet to the second-hop router ends up being “wasted.” The network would have been equally well off (more accurately, equally bad off) if the first router had simply discarded that packet and remained idle. More to the point, the transmission capacity used at the first router to forward the packet to the second router could have been much more profitably used to transmit a different packet. (For example, when selecting a packet for transmission, it might be better for a router to give priority to packets that have already traversed some number of upstream routers.) So here we see yet another cost of dropping a packet due to congestion—when a packet is dropped along a path, the transmission capacity that was used at each of the upstream links to forward that packet to the point at which it is dropped ends up having been wasted.

### 3.6.2 Approaches to Congestion Control

In [Section 3.7](#), we'll examine TCP's specific approach to congestion control in great detail. Here, we identify the two broad approaches to congestion control that are taken in practice and discuss specific

network architectures and congestion-control protocols embodying these approaches.

At the highest level, we can distinguish among congestion-control approaches by whether the network layer provides explicit assistance to the transport layer for congestion-control purposes:

- **End-to-end congestion control.** In an end-to-end approach to congestion control, the network layer provides no explicit support to the transport layer for congestion-control purposes. Even the presence of network congestion must be inferred by the end systems based only on observed network behavior (for example, packet loss and delay). We'll see shortly in [Section 3.7.1](#) that TCP takes this end-to-end approach toward congestion control, since the IP layer is not required to provide feedback to hosts regarding network congestion. TCP segment loss (as indicated by a timeout or the receipt of three duplicate acknowledgments) is taken as an indication of network congestion, and TCP decreases its window size accordingly. We'll also see a more recent proposal for TCP congestion control that uses increasing round-trip segment delay as an indicator of increased network congestion
- **Network-assisted congestion control.** With network-assisted congestion control, routers provide explicit feedback to the sender and/or receiver regarding the congestion state of the network. This feedback may be as simple as a single bit indicating congestion at a link – an approach taken in the early IBM SNA [[Schwartz 1982](#)], DEC DECnet [[Jain 1989](#); [Ramakrishnan 1990](#)] architectures, and ATM [[Black 1995](#)] network architectures. More sophisticated feedback is also possible. For example, in [ATM Available Bite Rate \(ABR\)](#) congestion control, a router informs the sender of the maximum host sending rate it (the router) can support on an outgoing link. As noted above, the Internet-default versions of IP and TCP adopt an end-to-end approach towards congestion control. We'll see, however, in [Section 3.7.2](#) that, more recently, IP and TCP may also optionally implement network-assisted congestion control.

For network-assisted congestion control, congestion information is typically fed back from the network to the sender in one of two ways, as shown in [Figure 3.49](#). Direct feedback may be sent from a network router to the sender. This form of notification typically takes the form of a choke packet (essentially saying, "I'm congested!"). The second and more common form of notification occurs when a router marks/updates a field in a packet flowing from sender to receiver to indicate congestion. Upon receipt of a marked packet, the receiver then notifies the sender of the congestion indication. This latter form of notification takes a full round-trip time.

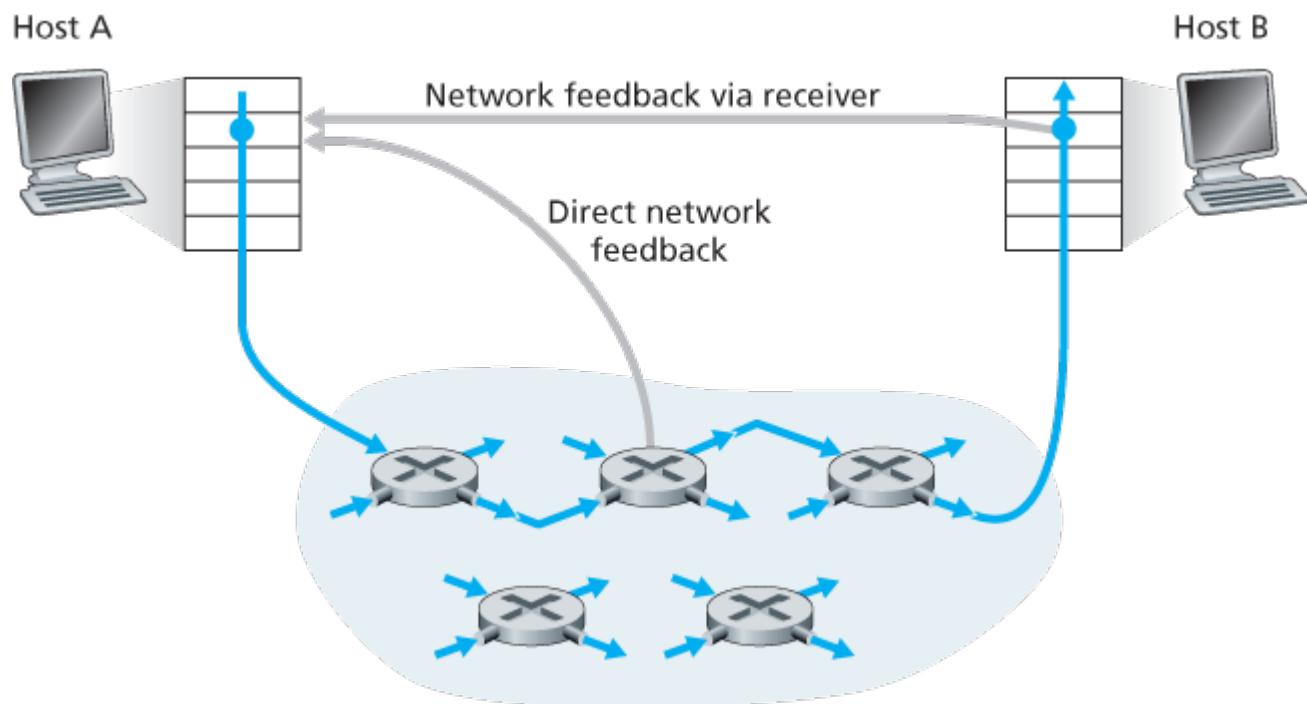


Figure 3.49 Two feedback pathways for network-indicated congestion information

## 3.7 TCP Congestion Control

In this section we return to our study of TCP. As we learned in [Section 3.5](#), TCP provides a reliable transport service between two processes running on different hosts. Another key component of TCP is its congestion-control mechanism. As indicated in the previous section, TCP must use end-to-end congestion control rather than network-assisted congestion control, since the IP layer provides no explicit feedback to the end systems regarding network congestion.

The approach taken by TCP is to have each sender limit the rate at which it sends traffic into its connection as a function of perceived network congestion. If a TCP sender perceives that there is little congestion on the path between itself and the destination, then the TCP sender increases its send rate; if the sender perceives that there is congestion along the path, then the sender reduces its send rate. But this approach raises three questions. First, how does a TCP sender limit the rate at which it sends traffic into its connection? Second, how does a TCP sender perceive that there is congestion on the path between itself and the destination? And third, what algorithm should the sender use to change its send rate as a function of perceived end-to-end congestion?

Let's first examine how a TCP sender limits the rate at which it sends traffic into its connection. In [Section 3.5](#) we saw that each side of a TCP connection consists of a receive buffer, a send buffer, and several variables (*LastByteRead*, *rwnd*, and so on). The TCP congestion-control mechanism operating at the sender keeps track of an additional variable, the **congestion window**. The congestion window, denoted *cwnd*, imposes a constraint on the rate at which a TCP sender can send traffic into the network. Specifically, the amount of unacknowledged data at a sender may not exceed the minimum of *cwnd* and *rwnd*, that is:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \min\{\text{cwnd}, \text{rwnd}\}$$

In order to focus on congestion control (as opposed to flow control), let us henceforth assume that the TCP receive buffer is so large that the receive-window constraint can be ignored; thus, the amount of unacknowledged data at the sender is solely limited by *cwnd*. We will also assume that the sender always has data to send, i.e., that all segments in the congestion window are sent.

The constraint above limits the amount of unacknowledged data at the sender and therefore indirectly limits the sender's send rate. To see this, consider a connection for which loss and packet transmission delays are negligible. Then, roughly, at the beginning of every RTT, the constraint permits the sender to

send  $cwnd$  bytes of data into the connection; at the end of the RTT the sender receives acknowledgments for the data. *Thus the sender's send rate is roughly  $cwnd/RTT$  bytes/sec. By adjusting the value of  $cwnd$ , the sender can therefore adjust the rate at which it sends data into its connection.*

Let's next consider how a TCP sender perceives that there is congestion on the path between itself and the destination. Let us define a "loss event" at a TCP sender as the occurrence of either a timeout or the receipt of three duplicate ACKs from the receiver. (Recall our discussion in [Section 3.5.4](#) of the timeout event in [Figure 3.33](#) and the subsequent modification to include fast retransmit on receipt of three duplicate ACKs.) When there is excessive congestion, then one (or more) router buffers along the path overflows, causing a datagram (containing a TCP segment) to be dropped. The dropped datagram, in turn, results in a loss event at the sender—either a timeout or the receipt of three duplicate ACKs—which is taken by the sender to be an indication of congestion on the sender-to-receiver path.

Having considered how congestion is detected, let's next consider the more optimistic case when the network is congestion-free, that is, when a loss event doesn't occur. In this case, acknowledgments for previously unacknowledged segments will be received at the TCP sender. As we'll see, TCP will take the arrival of these acknowledgments as an indication that all is well—that segments being transmitted into the network are being successfully delivered to the destination—and will use acknowledgments to increase its congestion window size (and hence its transmission rate). Note that if acknowledgments arrive at a relatively slow rate (e.g., if the end-end path has high delay or contains a low-bandwidth link), then the congestion window will be increased at a relatively slow rate. On the other hand, if acknowledgments arrive at a high rate, then the congestion window will be increased more quickly. Because TCP uses acknowledgments to trigger (or clock) its increase in congestion window size, TCP is said to be **self-clocking**.

Given the *mechanism* of adjusting the value of  $cwnd$  to control the sending rate, the critical question remains: *How should a TCP sender determine the rate at which it should send?* If TCP senders collectively send too fast, they can congest the network, leading to the type of congestion collapse that we saw in [Figure 3.48](#). Indeed, the version of TCP that we'll study shortly was developed in response to observed Internet congestion collapse [[Jacobson 1988](#)] under earlier versions of TCP. However, if TCP senders are too cautious and send too slowly, they could under utilize the bandwidth in the network; that is, the TCP senders could send at a higher rate without congesting the network. How then do the TCP senders determine their sending rates such that they don't congest the network but at the same time make use of all the available bandwidth? Are TCP senders explicitly coordinated, or is there a distributed approach in which the TCP senders can set their sending rates based only on local information? TCP answers these questions using the following guiding principles:

- **A lost segment implies congestion, and hence, the TCP sender's rate should be decreased when a segment is lost.** Recall from our discussion in [Section 3.5.4](#), that a timeout event or the

receipt of four acknowledgments for a given segment (one original ACK and then three duplicate ACKs) is interpreted as an implicit “loss event” indication of the segment following the quadruply ACKed segment, triggering a retransmission of the lost segment. From a congestion-control standpoint, the question is how the TCP sender should decrease its congestion window size, and hence its sending rate, in response to this inferred loss event.

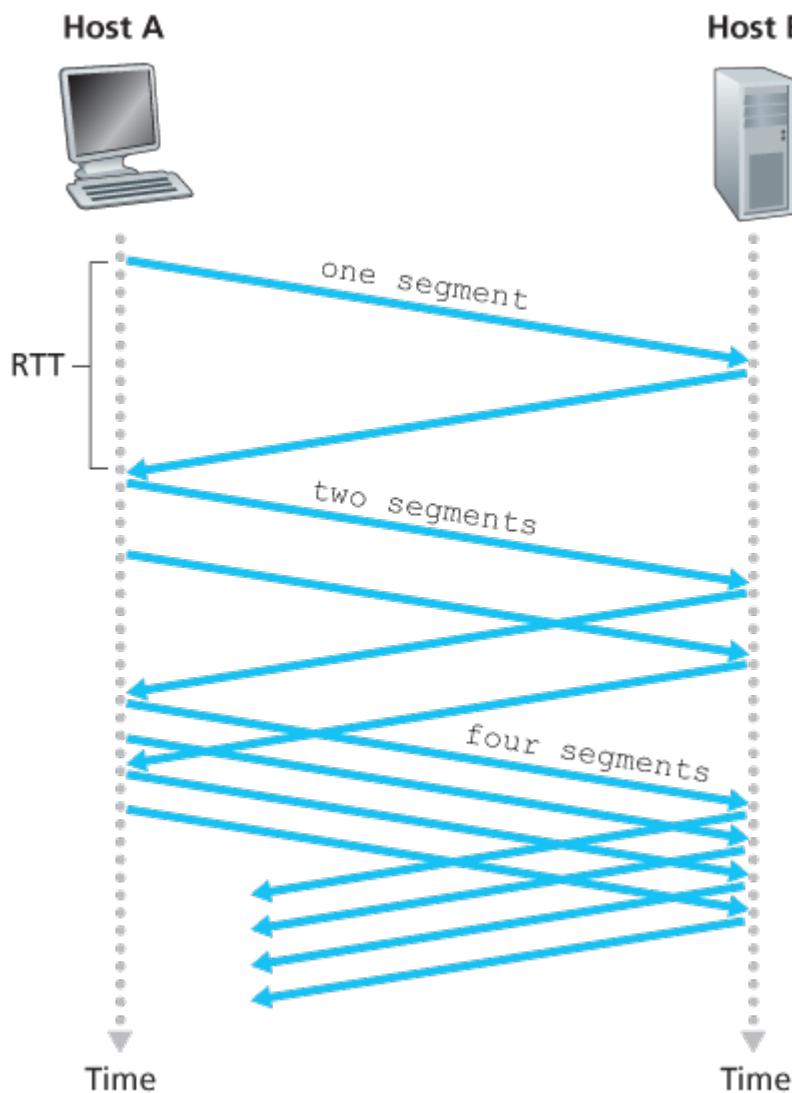
- **An acknowledged segment indicates that the network is delivering the sender’s segments to the receiver, and hence, the sender’s rate can be increased when an ACK arrives for a previously unacknowledged segment.** The arrival of acknowledgments is taken as an implicit indication that all is well—segments are being successfully delivered from sender to receiver, and the network is thus not congested. The congestion window size can thus be increased.
- **Bandwidth probing.** Given ACKs indicating a congestion-free source-to-destination path and loss events indicating a congested path, TCP’s strategy for adjusting its transmission rate is to increase its rate in response to arriving ACKs until a loss event occurs, at which point, the transmission rate is decreased. The TCP sender thus increases its transmission rate to probe for the rate that at which congestion onset begins, backs off from that rate, and then begins probing again to see if the congestion onset rate has changed. The TCP sender’s behavior is perhaps analogous to the child who requests (and gets) more and more goodies until finally he/she is finally told “No!”, backs off a bit, but then begins making requests again shortly afterwards. Note that there is no explicit signaling of congestion state by the network—ACKs and loss events serve as implicit signals—and that each TCP sender acts on local information asynchronously from other TCP senders.

Given this overview of TCP congestion control, we’re now in a position to consider the details of the celebrated **TCP congestion-control algorithm**, which was first described in [\[Jacobson 1988\]](#) and is standardized in [\[RFC 5681\]](#). The algorithm has three major components: (1) slow start, (2) congestion avoidance, and (3) fast recovery. Slow start and congestion avoidance are mandatory components of TCP, differing in how they increase the size of *cwnd* in response to received ACKs. We’ll see shortly that slow start increases the size of *cwnd* more rapidly (despite its name!) than congestion avoidance. Fast recovery is recommended, but not required, for TCP senders.

### *Slow Start*

When a TCP connection begins, the value of *cwnd* is typically initialized to a small value of 1 MSS [\[RFC 3390\]](#), resulting in an initial sending rate of roughly MSS/RTT. For example, if MSS = 500 bytes and RTT = 200 msec, the resulting initial sending rate is only about 20 kbps. Since the available bandwidth to the TCP sender may be much larger than MSS/RTT, the TCP sender would like to find the amount of available bandwidth quickly. Thus, in the **slow-start** state, the value of *cwnd* begins at 1 MSS and increases by 1 MSS every time a transmitted segment is first acknowledged. In the example of

[Figure 3.50](#), TCP sends the first segment into the network



**Figure 3.50 TCP slow start**

and waits for an acknowledgment. When this acknowledgment arrives, the TCP sender increases the congestion window by one MSS and sends out two maximum-sized segments. These segments are then acknowledged, with the sender increasing the congestion window by 1 MSS for each of the acknowledged segments, giving a congestion window of 4 MSS, and so on. This process results in a doubling of the sending rate every RTT. Thus, the TCP send rate starts slow but grows exponentially during the slow start phase.

But when should this exponential growth end? Slow start provides several answers to this question. First, if there is a loss event (i.e., congestion) indicated by a timeout, the TCP sender sets the value of *cwnd* to 1 and begins the slow start process anew. It also sets the value of a second state variable, *ssthresh* (shorthand for “slow start threshold”) to  $cwnd/2$ —half of the value of the congestion window value when congestion was detected. The second way in which slow start may end is directly tied to the value of *ssthresh*. Since *ssthresh* is half the value of *cwnd* when congestion was last detected, it might be a bit reckless to keep doubling *cwnd* when it reaches or surpasses the value of *ssthresh*. Thus, when the value of *cwnd* equals *ssthresh*, slow start ends and TCP transitions into congestion avoidance mode. As we’ll see, TCP increases *cwnd* more cautiously when in congestion-avoidance mode. The final way in which slow start can end is if three duplicate ACKs are

detected, in which case TCP performs a fast retransmit (see [Section 3.5.4](#)) and enters the fast recovery state, as discussed below. TCP’s behavior in slow start is summarized in the FSM description of TCP congestion control in [Figure 3.51](#). The slow-start algorithm traces its roots to [\[Jacobson 1988\]](#); an approach similar to slow start was also proposed independently in [\[Jain 1986\]](#).

### Congestion Avoidance

On entry to the congestion-avoidance state, the value of *cwnd* is approximately half its value when congestion was last encountered—congestion could be just around the corner! Thus, rather than doubling the value of *cwnd* every RTT, TCP adopts a more conservative approach and increases the value of *cwnd* by just a single MSS every RTT [\[RFC 5681\]](#). This can be accomplished in several ways. A common approach is for the TCP sender to increase *cwnd* by MSS bytes ( $\text{MSS}/\text{cwnd}$ ) whenever a new acknowledgment arrives. For example, if MSS is 1,460 bytes and *cwnd* is 14,600 bytes, then 10 segments are being sent within an RTT. Each arriving ACK (assuming one ACK per segment) increases the congestion window size by 1/10 MSS, and thus, the value of the congestion window will have increased by one MSS after ACKs when all 10 segments have been received.

But when should congestion avoidance’s linear increase (of 1 MSS per RTT) end? TCP’s congestion-avoidance algorithm behaves the same when a timeout occurs. As in the case of slow start: The value of *cwnd* is set to 1 MSS, and the value of *ssthresh* is updated to half the value of *cwnd* when the loss event occurred. Recall, however, that a loss event also can be triggered by a triple duplicate ACK event.

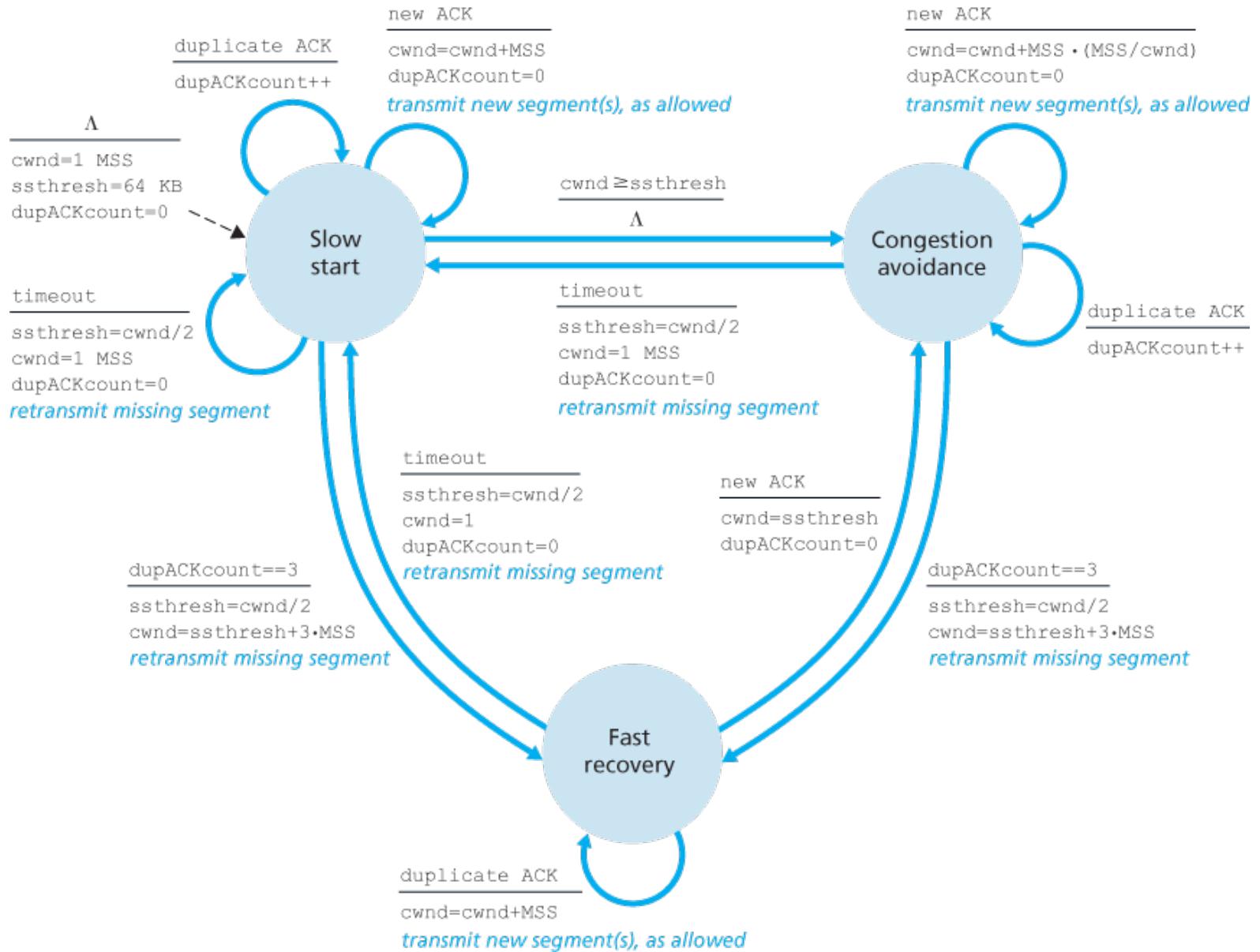


Figure 3.51 FSM description of TCP congestion control

In this case, the network is continuing to deliver segments from sender to receiver (as indicated by the receipt of duplicate ACKs). So TCP's behavior to this type of loss event should be less drastic than with a timeout-indicated loss: TCP halves the value of  $cwnd$  (adding in 3 MSS for good measure to account for the triple duplicate ACKs received) and records the value of  $ssthresh$  to be half the value of  $cwnd$  when the triple duplicate ACKs were received. The fast-recovery state is then entered.

### Fast Recovery

In fast recovery, the value of  $cwnd$  is increased by 1 MSS for every duplicate ACK received for the missing segment that caused TCP to enter the fast-recovery state. Eventually, when an ACK arrives for the missing segment, TCP enters the



VideoNote

## PRINCIPLES IN PRACTICE

### TCP SPLITTING: OPTIMIZING THE PERFORMANCE OF CLOUD SERVICES

For cloud services such as search, e-mail, and social networks, it is desirable to provide a high-level of responsiveness, ideally giving users the illusion that the services are running within their own end systems (including their smartphones). This can be a major challenge, as users are often located far away from the data centers responsible for serving the dynamic content associated with the cloud services. Indeed, if the end system is far from a data center, then the RTT will be large, potentially leading to poor response time performance due to TCP slow start.

As a case study, consider the delay in receiving a response for a search query. Typically, the server requires three TCP windows during slow start to deliver the response [Pathak 2010]. Thus the time from when an end system initiates a TCP connection until the time when it receives the last packet of the response is roughly  $4 \cdot \text{RTT}$  (one RTT to set up the TCP connection plus three RTTs for the three windows of data) plus the processing time in the data center. These RTT delays can lead to a noticeable delay in returning search results for a significant fraction of queries. Moreover, there can be significant packet loss in access networks, leading to TCP retransmissions and even larger delays.

One way to mitigate this problem and improve user-perceived performance is to (1) deploy front-end servers closer to the users, and (2) utilize **TCP splitting** by breaking the TCP connection at the front-end server. With TCP splitting, the client establishes a TCP connection to the nearby front-end, and the front-end maintains a persistent TCP connection to the data center with a very large TCP congestion window [Tariq 2008, Pathak 2010, Chen 2011]. With this approach, the response time roughly becomes  $4 \cdot \text{RTT}_{\text{FE}} + \text{RTT}_{\text{BE}} +$  processing time, where  $\text{RTT}_{\text{FE}}$  is the round-trip time between client and front-end server, and  $\text{RTT}_{\text{BE}}$  is the round-trip time between the front-end server and the data center (back-end server). If the front-end server is close to client, then this response time approximately becomes RTT plus processing time, since  $\text{RTT}_{\text{FE}}$  is negligibly small and  $\text{RTT}_{\text{BE}}$  is approximately RTT. In summary, TCP splitting can reduce the networking delay roughly from  $4 \cdot \text{RTT}$  to RTT, significantly improving user-perceived performance, particularly for users who are far from the nearest data center. TCP splitting also helps reduce TCP retransmission delays caused by losses in access networks. Google and Akamai have made extensive use of their CDN servers in access networks (recall our discussion in **Section 2.6**) to perform TCP splitting for the cloud services they support [Chen 2011].

congestion-avoidance state after deflating  $cwnd$ . If a timeout event occurs, fast recovery transitions to the slow-start state after performing the same actions as in slow start and congestion avoidance: The value of  $cwnd$  is set to 1 MSS, and the value of  $ssthresh$  is set to half the value of  $cwnd$  when the loss event occurred.

Fast recovery is a recommended, but not required, component of TCP [RFC 5681]. It is interesting that an early version of TCP, known as **TCP Tahoe**, unconditionally cut its congestion window to 1 MSS and entered the slow-start phase after either a timeout-indicated or triple-duplicate-ACK-indicated loss event. The newer version of TCP, **TCP Reno**, incorporated fast recovery.

**Figure 3.52** illustrates the evolution of TCP's congestion window for both Reno and Tahoe. In this figure, the threshold is initially equal to 8 MSS. For the first eight transmission rounds, Tahoe and Reno take identical actions. The congestion window climbs exponentially fast during slow start and hits the threshold at the fourth round of transmission. The congestion window then climbs linearly until a triple duplicate- ACK event occurs, just after transmission round 8. Note that the congestion window is 12·MSS when this loss event occurs. The value of  $ssthresh$  is then set to  $0.5 \cdot cwnd = 6 \cdot MSS$ . Under TCP Reno, the congestion window is set to  $cwnd = 9 \cdot MSS$  and then grows linearly. Under TCP Tahoe, the congestion window is set to 1 MSS and grows exponentially until it reaches the value of  $ssthresh$ , at which point it grows linearly.

**Figure 3.51** presents the complete FSM description of TCP's congestion-control algorithms—slow start, congestion avoidance, and fast recovery. The figure also indicates where transmission of new segments or retransmitted segments can occur. Although it is important to distinguish between TCP error control/retransmission and TCP congestion control, it's also important to appreciate how these two aspects of TCP are inextricably linked.

### *TCP Congestion Control: Retrospective*

Having delved into the details of slow start, congestion avoidance, and fast recovery, it's worthwhile to now step back and view the forest from the trees. Ignoring the

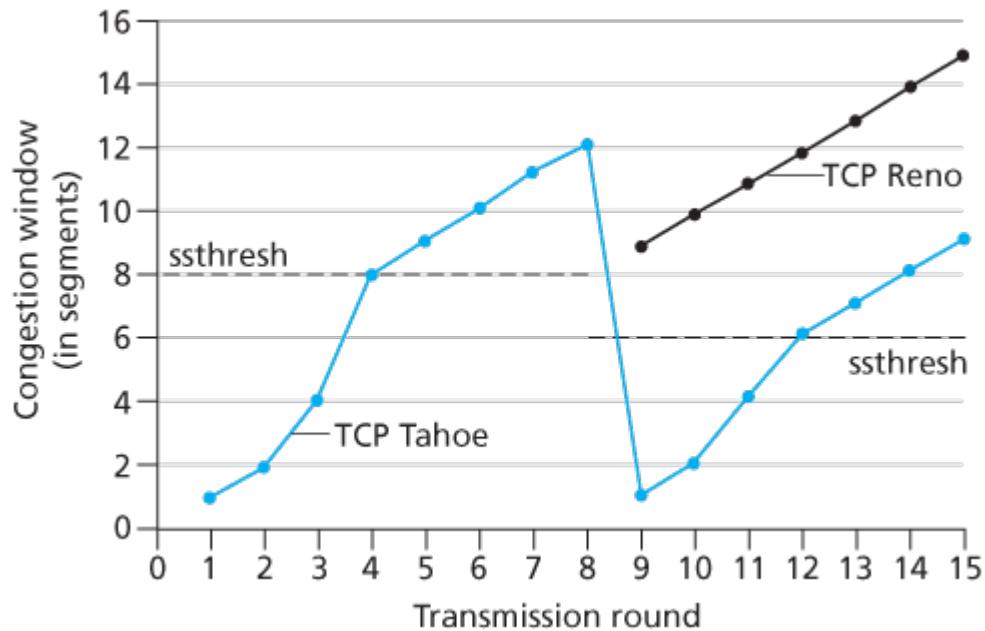


Figure 3.52 Evolution of TCP’s congestion window (Tahoe and Reno)

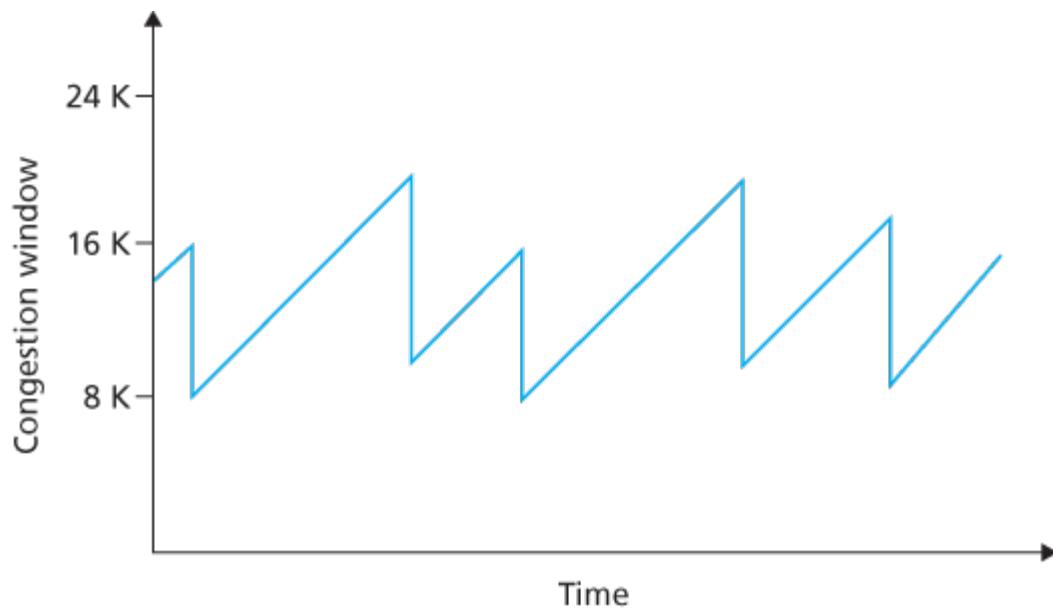


Figure 3.53 Additive-increase, multiplicative-decrease congestion control

initial slow-start period when a connection begins and assuming that losses are indicated by triple duplicate ACKs rather than timeouts, TCP’s congestion control consists of linear (additive) increase in *cwnd* of 1 MSS per RTT and then a halving (multiplicative decrease) of *cwnd* on a triple duplicate-ACK event. For this reason, TCP congestion control is often referred to as an **additive-increase, multiplicative-decrease (AIMD)** form of congestion control. AIMD congestion control gives rise to the “saw tooth” behavior shown in **Figure 3.53**, which also nicely illustrates our earlier intuition of TCP “probing” for bandwidth—TCP linearly increases its congestion window size (and hence its transmission rate) until a triple duplicate-ACK event occurs. It then decreases its congestion window size by a factor of two but then again begins increasing it linearly, probing to see if there is additional available bandwidth.

As noted previously, many TCP implementations use the Reno algorithm [[Padhye 2001](#)]. Many variations of the Reno algorithm have been proposed [[RFC 3782](#); [RFC 2018](#)]. The TCP Vegas algorithm [[Brakmo 1995](#); [Ahn 1995](#)] attempts to avoid congestion while maintaining good throughput. The basic idea of Vegas is to (1) detect congestion in the routers between source and destination *before* packet loss occurs, and (2) lower the rate linearly when this imminent packet loss is detected. Imminent packet loss is predicted by observing the RTT. The longer the RTT of the packets, the greater the congestion in the routers. As of late 2015, the Ubuntu Linux implementation of TCP provided slowstart, congestion avoidance, fast recovery, fast retransmit, and SACK, by default; alternative congestion control algorithms, such as TCP Vegas and BIC [[Xu 2004](#)], are also provided. For a survey of the many flavors of TCP, see [[Afanasyev 2010](#)].

TCP's AIMD algorithm was developed based on a tremendous amount of engineering insight and experimentation with congestion control in operational networks. Ten years after TCP's development, theoretical analyses showed that TCP's congestion-control algorithm serves as a distributed asynchronous-optimization algorithm that results in several important aspects of user and network performance being simultaneously optimized [[Kelly 1998](#)]. A rich theory of congestion control has since been developed [[Srikant 2004](#)].

#### *Macroscopic Description of TCP Throughput*

Given the saw-toothed behavior of TCP, it's natural to consider what the average throughput (that is, the average rate) of a long-lived TCP connection might be. In this analysis we'll ignore the slow-start phases that occur after timeout events. (These phases are typically very short, since the sender grows out of the phase exponentially fast.) During a particular round-trip interval, the rate at which TCP sends data is a function of the congestion window and the current *RTT*. When the window size is  $w$  bytes and the current round-trip time is  $RTT$  seconds, then TCP's transmission rate is roughly  $w/RTT$ . TCP then probes for additional bandwidth by increasing  $w$  by 1 MSS each  $RTT$  until a loss event occurs. Denote by  $W$  the value of  $w$  when a loss event occurs. Assuming that  $RTT$  and  $W$  are approximately constant over the duration of the connection, the TCP transmission rate ranges from  $W/(2 \cdot RTT)$  to  $W/RTT$ .

These assumptions lead to a highly simplified macroscopic model for the steady-state behavior of TCP. The network drops a packet from the connection when the rate increases to  $W/RTT$ ; the rate is then cut in half and then increases by  $MSS/RTT$  every  $RTT$  until it again reaches  $W/RTT$ . This process repeats itself over and over again. Because TCP's throughput (that is, rate) increases linearly between the two extreme values, we have

$$\text{average throughput of a connection} = 0.75 \cdot W/RTT$$

Using this highly idealized model for the steady-state dynamics of TCP, we can also derive an interesting expression that relates a connection's loss rate to its available bandwidth [[Mahdavi 1997](#)].

This derivation is outlined in the homework problems. A more sophisticated model that has been found empirically to agree with measured data is [\[Padhye 2000\]](#).

### TCP Over High-Bandwidth Paths

It is important to realize that TCP congestion control has evolved over the years and indeed continues to evolve. For a summary of current TCP variants and discussion of TCP evolution, see [\[Floyd 2001, RFC 5681, Afanasyev 2010\]](#). What was good for the Internet when the bulk of the TCP connections carried SMTP, FTP, and Telnet traffic is not necessarily good for today's HTTP-dominated Internet or for a future Internet with services that are still undreamed of.

The need for continued evolution of TCP can be illustrated by considering the high-speed TCP connections that are needed for grid- and cloud-computing applications. For example, consider a TCP connection with 1,500-byte segments and a 100 ms *RTT*, and suppose we want to send data through this connection at 10 Gbps. Following [\[RFC 3649\]](#), we note that using the TCP throughput formula above, in order to achieve a 10 Gbps throughput, the average congestion window size would need to be 83,333 segments. That's a *lot* of segments, leading us to be rather concerned that one of these 83,333 in-flight segments might be lost. What would happen in the case of a loss? Or, put another way, what fraction of the transmitted segments could be lost that would allow the TCP congestion-control algorithm specified in [Figure 3.51](#) still to achieve the desired 10 Gbps rate? In the homework questions for this chapter, you are led through the derivation of a formula relating the throughput of a TCP connection as a function of the loss rate ( $L$ ), the round-trip time ( $RTT$ ), and the maximum segment size (MSS):

$$\text{average throughput of a connection} = 1.22 \cdot \text{MSS} \cdot \text{RTT}$$

Using this formula, we can see that in order to achieve a throughput of 10 Gbps, today's TCP congestion-control algorithm can only tolerate a segment loss probability of  $2 \cdot 10^{-10}$  (or equivalently, one loss event for every 5,000,000,000 segments)—a very low rate. This observation has led a number of researchers to investigate new versions of TCP that are specifically designed for such high-speed environments; see [\[Jin 2004; Kelly 2003; Ha 2008; RFC 7323\]](#) for discussions of these efforts.

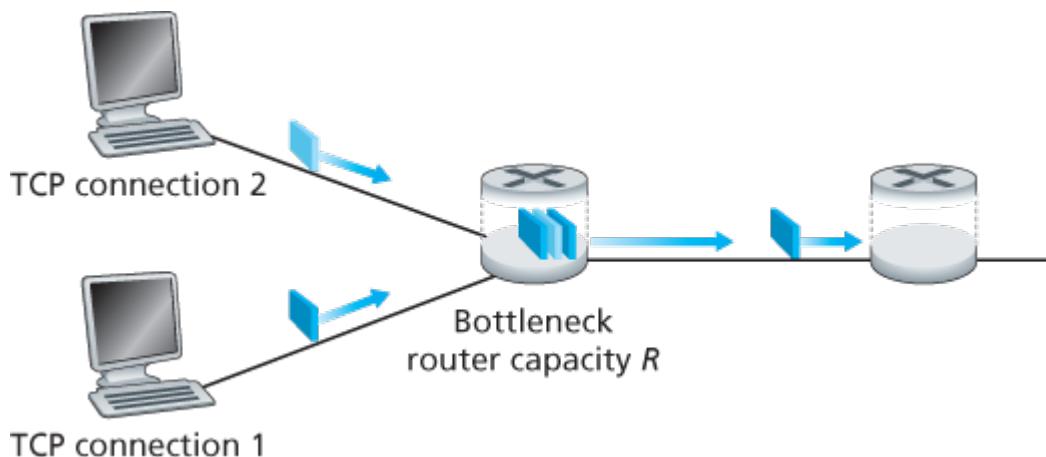
#### 3.7.1 Fairness

Consider  $K$  TCP connections, each with a different end-to-end path, but all passing through a bottleneck link with transmission rate  $R$  bps. (By *bottleneck link*, we mean that for each connection, all the other links along the connection's path are not congested and have abundant transmission capacity as compared with the transmission capacity of the bottleneck link.) Suppose each connection is transferring a large file and there is no UDP traffic passing through the bottleneck link. A congestion-control mechanism is said to be *fair* if the average transmission rate of each connection is approximately  $R/K$ :

that is, each connection gets an equal share of the link bandwidth.

Is TCP's AIMD algorithm fair, particularly given that different TCP connections may start at different times and thus may have different window sizes at a given point in time? [Chiu 1989] provides an elegant and intuitive explanation of why TCP congestion control converges to provide an equal share of a bottleneck link's bandwidth among competing TCP connections.

Let's consider the simple case of two TCP connections sharing a single link with transmission rate  $R$ , as shown in [Figure 3.54](#). Assume that the two connections



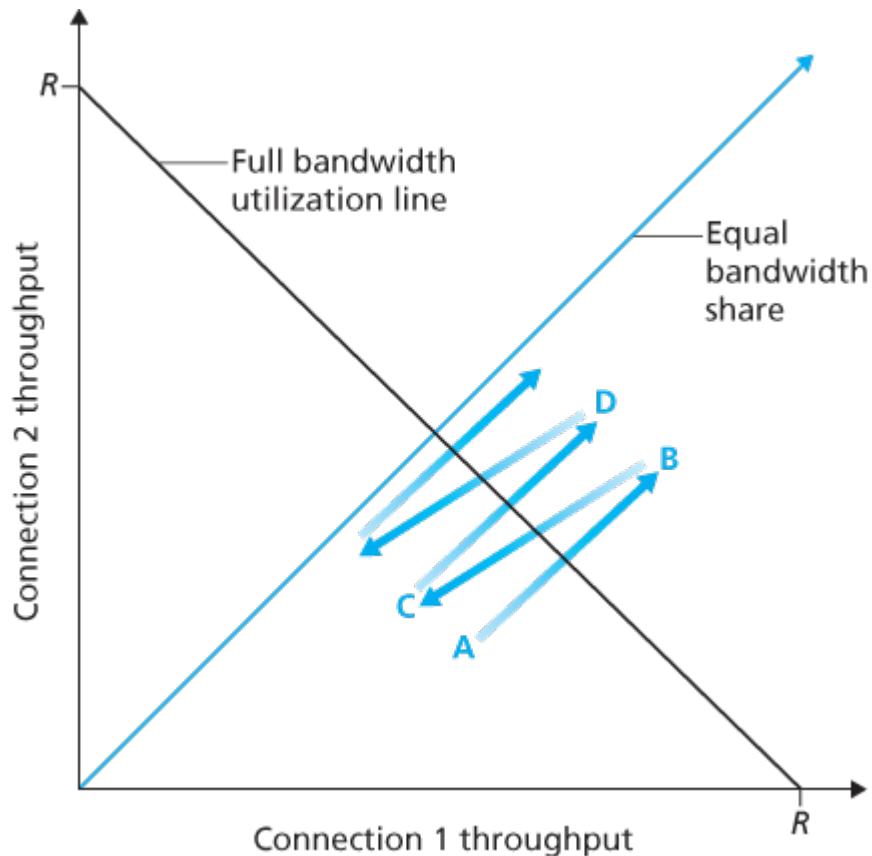
**Figure 3.54** Two TCP connections sharing a single bottleneck link

have the same MSS and RTT (so that if they have the same congestion window size, then they have the same throughput), that they have a large amount of data to send, and that no other TCP connections or UDP datagrams traverse this shared link. Also, ignore the slow-start phase of TCP and assume the TCP connections are operating in CA mode (AIMD) at all times.

[Figure 3.55](#) plots the throughput realized by the two TCP connections. If TCP is to share the link bandwidth equally between the two connections, then the realized throughput should fall along the 45-degree arrow (equal bandwidth share) emanating from the origin. Ideally, the sum of the two throughputs should equal  $R$ . (Certainly, each connection receiving an equal, but zero, share of the link capacity is not a desirable situation!) So the goal should be to have the achieved throughputs fall somewhere near the intersection of the equal bandwidth share line and the full bandwidth utilization line in [Figure 3.55](#).

Suppose that the TCP window sizes are such that at a given point in time, connections 1 and 2 realize throughputs indicated by point A in [Figure 3.55](#). Because the amount of link bandwidth jointly consumed by the two connections is less than  $R$ , no loss will occur, and both connections will increase their window by 1 MSS per RTT as a result of TCP's congestion-avoidance algorithm. Thus, the joint throughput of the two connections proceeds along a 45-degree line (equal increase for both

connections) starting from point A. Eventually, the link bandwidth jointly consumed by the two connections will be greater than  $R$ , and eventually packet loss will occur. Suppose that connections 1 and 2 experience packet loss when they realize throughputs indicated by point B. Connections 1 and 2 then decrease their windows by a factor of two. The resulting throughputs realized are thus at point C, halfway along a vector starting at B and ending at the origin. Because the joint bandwidth use is less than  $R$  at point C, the two connections again increase their throughputs along a 45-degree line starting from C. Eventually, loss will again occur, for example, at point D, and the two connections again decrease their window sizes by a factor of two, and so on. You should convince yourself that the bandwidth realized by the two connections eventually fluctuates along the equal bandwidth share line. You should also convince



**Figure 3.55 Throughput realized by TCP connections 1 and 2**

yourself that the two connections will converge to this behavior regardless of where they are in the two-dimensional space! Although a number of idealized assumptions lie behind this scenario, it still provides an intuitive feel for why TCP results in an equal sharing of bandwidth among connections.

In our idealized scenario, we assumed that only TCP connections traverse the bottleneck link, that the connections have the same RTT value, and that only a single TCP connection is associated with a host-destination pair. In practice, these conditions are typically not met, and client-server applications can thus obtain very unequal portions of link bandwidth. In particular, it has been shown that when multiple connections share a common bottleneck, those sessions with a smaller RTT are able to grab the available bandwidth at that link more quickly as it becomes free (that is, open their congestion windows faster) and thus will enjoy higher throughput than those connections with larger RTTs [Lakshman]

1997].

### Fairness and UDP

We have just seen how TCP congestion control regulates an application's transmission rate via the congestion window mechanism. Many multimedia applications, such as Internet phone and video conferencing, often do not run over TCP for this very reason—they do not want their transmission rate throttled, even if the network is very congested. Instead, these applications prefer to run over UDP, which does not have built-in congestion control. When running over UDP, applications can pump their audio and video into the network at a constant rate and occasionally lose packets, rather than reduce their rates to “fair” levels at times of congestion and not lose any packets. From the perspective of TCP, the multimedia applications running over UDP are not being fair—they do not cooperate with the other connections nor adjust their transmission rates appropriately. Because TCP congestion control will decrease its transmission rate in the face of increasing congestion (loss), while UDP sources need not, it is possible for UDP sources to crowd out TCP traffic. An area of research today is thus the development of congestion-control mechanisms for the Internet that prevent UDP traffic from bringing the Internet's throughput to a grinding halt [[Floyd 1999](#); [Floyd 2000](#); [Kohler 2006](#); [RFC 4340](#)].

### Fairness and Parallel TCP Connections

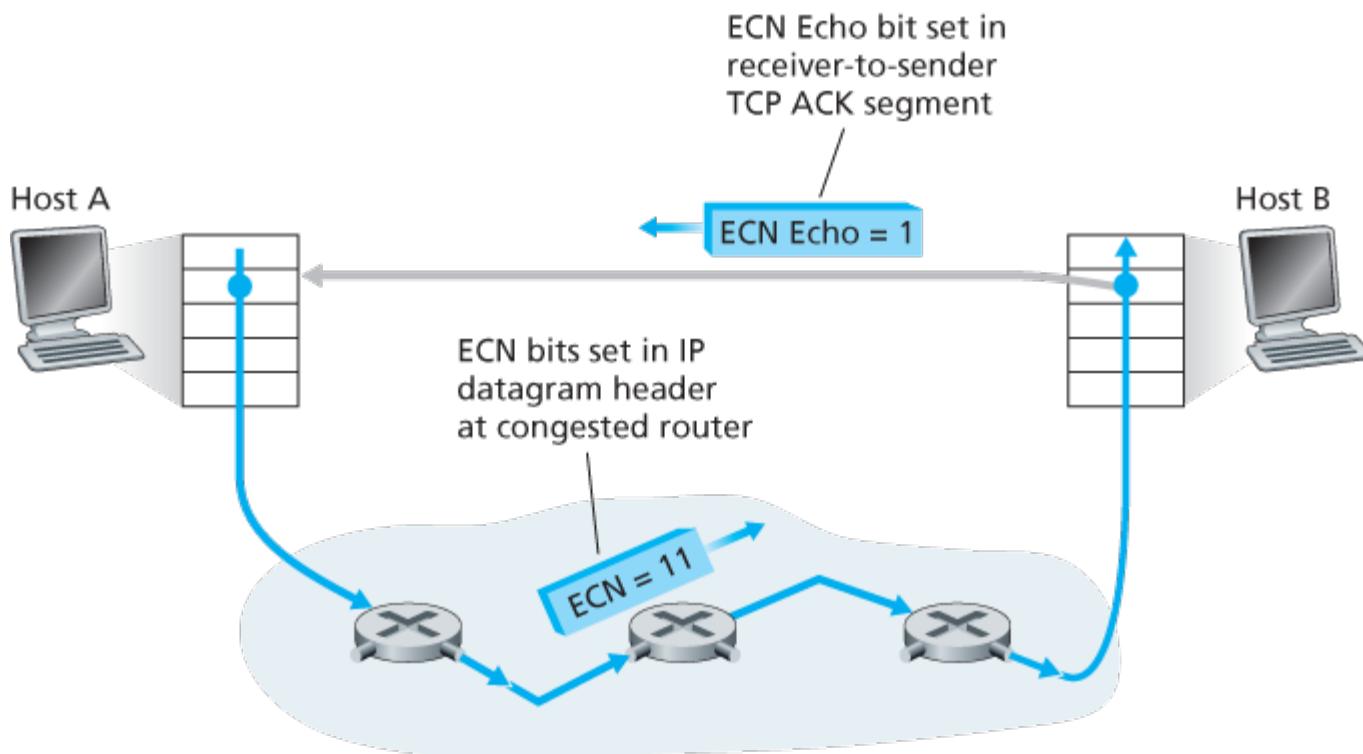
But even if we could force UDP traffic to behave fairly, the fairness problem would still not be completely solved. This is because there is nothing to stop a TCP-based application from using multiple parallel connections. For example, Web browsers often use multiple parallel TCP connections to transfer the multiple objects within a Web page. (The exact number of multiple connections is configurable in most browsers.) When an application uses multiple parallel connections, it gets a larger fraction of the bandwidth in a congested link. As an example, consider a link of rate  $R$  supporting nine ongoing client-server applications, with each of the applications using one TCP connection. If a new application comes along and also uses one TCP connection, then each application gets approximately the same transmission rate of  $R/10$ . But if this new application instead uses 11 parallel TCP connections, then the new application gets an unfair allocation of more than  $R/2$ . Because Web traffic is so pervasive in the Internet, multiple parallel connections are not uncommon.

## 3.7.2 Explicit Congestion Notification (ECN): Network-assisted Congestion Control

Since the initial standardization of slow start and congestion avoidance in the late 1980's [[RFC 1122](#)], TCP has implemented the form of end-end congestion control that we studied in [Section 3.7.1](#): a TCP sender receives no explicit congestion indications from the network layer, and instead infers congestion through observed packet loss. More recently, extensions to both IP and TCP [[RFC 3168](#)] have been proposed, implemented, and deployed that allow the network to explicitly signal congestion to a TCP

sender and receiver. This form of network-assisted congestion control is known as **Explicit Congestion Notification**. As shown in **Figure 3.56**, the TCP and IP protocols are involved.

At the network layer, two bits (with four possible values, overall) in the Type of Service field of the IP datagram header (which we'll discuss in **Section 4.3**) are used for ECN. One setting of the ECN bits is used by a router to indicate that it (the



**Figure 3.56 Explicit Congestion Notification: network-assisted congestion control**

router) is experiencing congestion. This congestion indication is then carried in the marked IP datagram to the destination host, which then informs the sending host, as shown in **Figure 3.56**. RFC 3168 does not provide a definition of when a router is congested; that decision is a configuration choice made possible by the router vendor, and decided by the network operator. However, RFC 3168 does recommend that an ECN congestion indication be set only in the face of persistent congestion. A second setting of the ECN bits is used by the sending host to inform routers that the sender and receiver are ECN-capable, and thus capable of taking action in response to ECN-indicated network congestion.

As shown in **Figure 3.56**, when the TCP in the receiving host receives an ECN congestion indication via a received datagram, the TCP in the receiving host informs the TCP in the sending host of the congestion indication by setting the ECE (Explicit Congestion Notification Echo) bit (see **Figure 3.29**) in a receiver-to-sender TCP ACK segment. The TCP sender, in turn, reacts to an ACK with an ECE congestion indication by halving the congestion window, as it would react to a lost segment using fast retransmit, and sets the CWR (Congestion Window Reduced) bit in the header of the next transmitted TCP sender-to-receiver segment.

Other transport-layer protocols besides TCP may also make use of network-layer-signaled ECN. The Datagram Congestion Control Protocol (DCCP) [\[RFC 4340\]](#) provides a low-overhead, congestion-controlled UDP-like unreliable service that utilizes ECN. DCTCP (Data Center TCP) [\[Alizadeh 2010\]](#), a version of TCP designed specifically for data center networks, also makes use of ECN.

## 3.8 Summary

We began this chapter by studying the services that a transport-layer protocol can provide to network applications. At one extreme, the transport-layer protocol can be very simple and offer a no-frills service to applications, providing only a multiplexing/demultiplexing function for communicating processes. The Internet's UDP protocol is an example of such a no-frills transport-layer protocol. At the other extreme, a transport-layer protocol can provide a variety of guarantees to applications, such as reliable delivery of data, delay guarantees, and bandwidth guarantees. Nevertheless, the services that a transport protocol can provide are often constrained by the service model of the underlying network-layer protocol. If the network-layer protocol cannot provide delay or bandwidth guarantees to transport-layer segments, then the transport-layer protocol cannot provide delay or bandwidth guarantees for the messages sent between processes.

We learned in [Section 3.4](#) that a transport-layer protocol can provide reliable data transfer even if the underlying network layer is unreliable. We saw that providing reliable data transfer has many subtle points, but that the task can be accomplished by carefully combining acknowledgments, timers, retransmissions, and sequence numbers.

Although we covered reliable data transfer in this chapter, we should keep in mind that reliable data transfer can be provided by link-, network-, transport-, or application-layer protocols. Any of the upper four layers of the protocol stack can implement acknowledgments, timers, retransmissions, and sequence numbers and provide reliable data transfer to the layer above. In fact, over the years, engineers and computer scientists have independently designed and implemented link-, network-, transport-, and application-layer protocols that provide reliable data transfer (although many of these protocols have quietly disappeared).

In [Section 3.5](#), we took a close look at TCP, the Internet's connection-oriented and reliable transport-layer protocol. We learned that TCP is complex, involving connection management, flow control, and round-trip time estimation, as well as reliable data transfer. In fact, TCP is actually more complex than our description—we intentionally did not discuss a variety of TCP patches, fixes, and improvements that are widely implemented in various versions of TCP. All of this complexity, however, is hidden from the network application. If a client on one host wants to send data reliably to a server on another host, it simply opens a TCP socket to the server and pumps data into that socket. The client-server application is blissfully unaware of TCP's complexity.

In [Section 3.6](#), we examined congestion control from a broad perspective, and in [Section 3.7](#), we showed how TCP implements congestion control. We learned that congestion control is imperative for

the well-being of the network. Without congestion control, a network can easily become gridlocked, with little or no data being transported end-to-end. In [Section 3.7](#) we learned that TCP implements an end-to-end congestion-control mechanism that additively increases its transmission rate when the TCP connection’s path is judged to be congestion-free, and multiplicatively decreases its transmission rate when loss occurs. This mechanism also strives to give each TCP connection passing through a congested link an equal share of the link bandwidth. We also examined in some depth the impact of TCP connection establishment and slow start on latency. We observed that in many important scenarios, connection establishment and slow start significantly contribute to end-to-end delay. We emphasize once more that while TCP congestion control has evolved over the years, it remains an area of intensive research and will likely continue to evolve in the upcoming years.

Our discussion of specific Internet transport protocols in this chapter has focused on UDP and TCP—the two “work horses” of the Internet transport layer. However, two decades of experience with these two protocols has identified circumstances in which neither is ideally suited. Researchers have thus been busy developing additional transport-layer protocols, several of which are now IETF proposed standards.

The Datagram Congestion Control Protocol (DCCP) [\[RFC 4340\]](#) provides a low-overhead, message-oriented, UDP-like unreliable service, but with an application-selected form of congestion control that is compatible with TCP. If reliable or semi-reliable data transfer is needed by an application, then this would be performed within the application itself, perhaps using the mechanisms we have studied in [Section 3.4](#). DCCP is envisioned for use in applications such as streaming media (see [Chapter 9](#)) that can exploit the tradeoff between timeliness and reliability of data delivery, but that want to be responsive to network congestion.

Google’s QUIC (Quick UDP Internet Connections) protocol [[Iyengar 2016](#)], implemented in Google’s Chromium browser, provides reliability via retransmission as well as error correction, fast-connection setup, and a rate-based congestion control algorithm that aims to be TCP friendly—all implemented as an application-level protocol on top of UDP. In early 2015, Google reported that roughly half of all requests from Chrome to Google servers are served over QUIC.

DCTCP (Data Center TCP) [\[Alizadeh 2010\]](#) is a version of TCP designed specifically for data center networks, and uses ECN to better support the mix of short- and long-lived flows that characterize data center workloads.

The Stream Control Transmission Protocol (SCTP) [\[RFC 4960, RFC 3286\]](#) is a reliable, message-oriented protocol that allows several different application-level “streams” to be multiplexed through a single SCTP connection (an approach known as “multi-streaming”). From a reliability standpoint, the different streams within the connection are handled separately, so that packet loss in one stream does not affect the delivery of data in other streams. QUIC provides similar multi-stream semantics. SCTP

also allows data to be transferred over two outgoing paths when a host is connected to two or more networks, optional delivery of out-of-order data, and a number of other features. SCTP's flow- and congestion-control algorithms are essentially the same as in TCP.

The TCP-Friendly Rate Control (TFRC) protocol [[RFC 5348](#)] is a congestion-control protocol rather than a full-fledged transport-layer protocol. It specifies a congestion-control mechanism that could be used in another transport protocol such as DCCP (indeed one of the two application-selectable protocols available in DCCP is TFRC). The goal of TFRC is to smooth out the “saw tooth” behavior (see [Figure 3.53](#)) in TCP congestion control, while maintaining a long-term sending rate that is “reasonably” close to that of TCP. With a smoother sending rate than TCP, TFRC is well-suited for multimedia applications such as IP telephony or streaming media where such a smooth rate is important. TFRC is an “equation-based” protocol that uses the measured packet loss rate as input to an equation [[Padhye 2000](#)] that estimates what TCP's throughput would be if a TCP session experiences that loss rate. This rate is then taken as TFRC's target sending rate.

Only the future will tell whether DCCP, SCTP, QUIC, or TFRC will see widespread deployment. While these protocols clearly provide enhanced capabilities over TCP and UDP, TCP and UDP have proven themselves “good enough” over the years. Whether “better” wins out over “good enough” will depend on a complex mix of technical, social, and business considerations.

In [Chapter 1](#), we said that a computer network can be partitioned into the “network edge” and the “network core.” The network edge covers everything that happens in the end systems. Having now covered the application layer and the transport layer, our discussion of the network edge is complete. It is time to explore the network core! This journey begins in the next two chapters, where we'll study the network layer, and continues into [Chapter 6](#), where we'll study the link layer.

# Homework Problems and Questions

## Chapter 3 Review Questions

### SECTIONS 3.1–3.3

R1. Suppose the network layer provides the following service. The network layer in the source host accepts a segment of maximum size 1,200 bytes and a destination host address from the transport layer. The network layer then guarantees to deliver the segment to the transport layer at the destination host. Suppose many network application processes can be running at the destination host.

- a. Design the simplest possible transport-layer protocol that will get application data to the desired process at the destination host. Assume the operating system in the destination host has assigned a 4-byte port number to each running application process.
- b. Modify this protocol so that it provides a “return address” to the destination process.
- c. In your protocols, does the transport layer “have to do anything” in the core of the computer network?

R2. Consider a planet where everyone belongs to a family of six, every family lives in its own house, each house has a unique address, and each person in a given house has a unique name. Suppose this planet has a mail service that delivers letters from source house to destination house. The mail service requires that (1) the letter be in an envelope, and that (2) the address of the destination house (and nothing more) be clearly written on the envelope. Suppose each family has a delegate family member who collects and distributes letters for the other family members. The letters do not necessarily provide any indication of the recipients of the letters.

- a. Using the solution to Problem R1 above as inspiration, describe a protocol that the delegates can use to deliver letters from a sending family member to a receiving family member.
- b. In your protocol, does the mail service ever have to open the envelope and examine the letter in order to provide its service?

R3. Consider a TCP connection between Host A and Host B. Suppose that the TCP segments traveling from Host A to Host B have source port number  $x$  and destination port number  $y$ . What are the source and destination port numbers for the segments traveling from Host B to Host A?

R4. Describe why an application developer might choose to run an application over UDP rather than TCP.

R5. Why is it that voice and video traffic is often sent over TCP rather than UDP in today's Internet? (*Hint:* The answer we are looking for has nothing to do with TCP's congestion-control mechanism.)

R6. Is it possible for an application to enjoy reliable data transfer even when the application runs over UDP? If so, how?

R7. Suppose a process in Host C has a UDP socket with port number 6789. Suppose both Host A and Host B each send a UDP segment to Host C with destination port number 6789. Will both of these segments be directed to the same socket at Host C? If so, how will the process at Host C know that these two segments originated from two different hosts?

R8. Suppose that a Web server runs in Host C on port 80. Suppose this Web server uses persistent connections, and is currently receiving requests from two different Hosts, A and B. Are all of the requests being sent through the same socket at Host C? If they are being passed through different sockets, do both of the sockets have port 80? Discuss and explain.

#### SECTION 3.4

R9. In our *rdt* protocols, why did we need to introduce sequence numbers?

R10. In our *rdt* protocols, why did we need to introduce timers?

R11. Suppose that the roundtrip delay between sender and receiver is constant and known to the sender. Would a timer still be necessary in protocol *rdt 3.0*, assuming that packets can be lost? Explain.

R12. Visit the Go-Back-N Java applet at the companion Web site.

- a. Have the source send five packets, and then pause the animation before any of the five packets reach the destination. Then kill the first packet and resume the animation. Describe what happens.
- b. Repeat the experiment, but now let the first packet reach the destination and kill the first acknowledgment. Describe again what happens.
- c. Finally, try sending six packets. What happens?

R13. Repeat R12, but now with the Selective Repeat Java applet. How are Selective Repeat and Go-Back-N different?

#### SECTION 3.5

R14. True or false?

- a. Host A is sending Host B a large file over a TCP connection. Assume Host B has no data to send Host A. Host B will not send acknowledgments to Host A because Host B cannot piggyback the acknowledgments on data.

- b. The size of the TCP *rwnd* never changes throughout the duration of the connection.
- c. Suppose Host A is sending Host B a large file over a TCP connection. The number of unacknowledged bytes that A sends cannot exceed the size of the receive buffer.
- d. Suppose Host A is sending a large file to Host B over a TCP connection. If the sequence number for a segment of this connection is  $m$ , then the sequence number for the subsequent segment will necessarily be  $m+1$ .
- e. The TCP segment has a field in its header for *rwnd*.
- f. Suppose that the last *SampleRTT* in a TCP connection is equal to 1 sec. The current value of *TimeoutInterval* for the connection will necessarily be  $\geq 1$  sec.
- g. Suppose Host A sends one segment with sequence number 38 and 4 bytes of data over a TCP connection to Host B. In this same segment the acknowledgment number is necessarily 42.

R15. Suppose Host A sends two TCP segments back to back to Host B over a TCP connection. The first segment has sequence number 90; the second has sequence number 110.

- a. How much data is in the first segment?
- b. Suppose that the first segment is lost but the second segment arrives at B. In the acknowledgment that Host B sends to Host A, what will be the acknowledgment number?

R16. Consider the Telnet example discussed in [Section 3.5](#). A few seconds after the user types the letter ‘C,’ the user types the letter ‘R.’ After typing the letter ‘R,’ how many segments are sent, and what is put in the sequence number and acknowledgment fields of the segments?

## SECTION 3.7

R17. Suppose two TCP connections are present over some bottleneck link of rate  $R$  bps. Both connections have a huge file to send (in the same direction over the bottleneck link). The transmissions of the files start at the same time. What transmission rate would TCP like to give to each of the connections?

R18. True or false? Consider congestion control in TCP. When the timer expires at the sender, the value of *ssthresh* is set to one half of its previous value.

R19. In the discussion of TCP splitting in the sidebar in [Section 3.7](#), it was claimed that the response time with TCP splitting is approximately  $4 \cdot \text{RTT}_{\text{FE}} + \text{RTT}_{\text{BE}} + \text{processing time}$ . Justify this claim.

## Problems

P1. Suppose Client A initiates a Telnet session with Server S. At about the same time, Client B

also initiates a Telnet session with Server S. Provide possible source and destination port numbers for

- a. The segments sent from A to S.
- b. The segments sent from B to S.
- c. The segments sent from S to A.
- d. The segments sent from S to B.
- e. If A and B are different hosts, is it possible that the source port number in the segments from A to S is the same as that from B to S?
- f. How about if they are the same host?

P2. Consider [Figure 3.5](#). What are the source and destination port values in the segments flowing from the server back to the clients' processes? What are the IP addresses in the network-layer datagrams carrying the transport-layer segments?

P3. UDP and TCP use 1s complement for their checksums. Suppose you have the following three 8-bit bytes: 01010011, 01100110, 01110100. What is the 1s complement of the sum of these 8-bit bytes? (Note that although UDP and TCP use 16-bit words in computing the checksum, for this problem you are being asked to consider 8-bit sums.) Show all work. Why is it that UDP takes the 1s complement of the sum; that is, why not just use the sum? With the 1s complement scheme, how does the receiver detect errors? Is it possible that a 1-bit error will go undetected? How about a 2-bit error?

P4.

- a. Suppose you have the following 2 bytes: 01011100 and 01100101. What is the 1s complement of the sum of these 2 bytes?
- b. Suppose you have the following 2 bytes: 11011010 and 01100101. What is the 1s complement of the sum of these 2 bytes?
- c. For the bytes in part (a), give an example where one bit is flipped in each of the 2 bytes and yet the 1s complement doesn't change.

P5. Suppose that the UDP receiver computes the Internet checksum for the received UDP segment and finds that it matches the value carried in the checksum field. Can the receiver be absolutely certain that no bit errors have occurred? Explain.

P6. Consider our motivation for correcting protocol [rdt2.1](#). Show that the receiver, shown in [Figure 3.57](#), when operating with the sender shown in [Figure 3.11](#), can lead the sender and receiver to enter into a deadlock state, where each is waiting for an event that will never occur.

P7. In protocol [rdt3.0](#), the ACK packets flowing from the receiver to the sender do not have sequence numbers (although they do have an ACK field that contains the sequence number of the packet they are acknowledging). Why is it that our ACK packets do not require sequence numbers?



Figure 3.57 An incorrect receiver for protocol *rdt 2.1*

P8. Draw the FSM for the receiver side of protocol *rdt3.0*.

P9. Give a trace of the operation of protocol *rdt3.0* when data packets and acknowledgment packets are garbled. Your trace should be similar to that used in [Figure 3.16](#).

P10. Consider a channel that can lose packets but has a maximum delay that is known. Modify protocol *rdt2.1* to include sender timeout and retransmit. Informally argue why your protocol can communicate correctly over this channel.

P11. Consider the *rdt2.2* receiver in [Figure 3.14](#), and the creation of a new packet in the self-transition (i.e., the transition from the state back to itself) in the Wait-for-0-from-below and the Wait-for-1-from-below states: *sndpkt=make\_pkt(ACK, 1, checksum)* and *sndpkt=make\_pkt(ACK, 0, checksum)*. Would the protocol work correctly if this action were removed from the self-transition in the Wait-for-1-from-below state? Justify your answer.

What if this event were removed from the self-transition in the Wait-for-0-from-below state? [Hint: In this latter case, consider what would happen if the first sender-to-receiver packet were corrupted.]

P12. The sender side of *rdt3.0* simply ignores (that is, takes no action on) all received packets that are either in error or have the wrong value in the *acknum* field of an acknowledgment packet. Suppose that in such circumstances, *rdt3.0* were simply to retransmit the current data packet. Would the protocol still work? (Hint: Consider what would happen if there were only bit errors; there are no packet losses but premature timeouts can occur. Consider how many times the *n*th packet is sent, in the limit as *n* approaches infinity.)

P13. Consider the *rdt 3.0* protocol. Draw a diagram showing that if the network connection between the sender and receiver can reorder messages (that is, that two messages propagating in the medium between the sender and receiver can be reordered), then the alternating-bit protocol will not work correctly (make sure you clearly identify the sense in which it will not work correctly). Your diagram should have the sender on the left and the receiver on the right, with the time axis running down the page, showing data (D) and acknowledgment (A) message exchange. Make sure you indicate the sequence number associated with any data or acknowledgment segment.

P14. Consider a reliable data transfer protocol that uses only negative acknowledgments. Suppose the sender sends data only infrequently. Would a NAK-only protocol be preferable to a protocol that uses ACKs? Why? Now suppose the sender has a lot of data to send and the end-to-end connection experiences few losses. In this second case, would a NAK-only protocol be preferable to a protocol that uses ACKs? Why?

P15. Consider the cross-country example shown in [Figure 3.17](#). How big would the window size have to be for the channel utilization to be greater than 98 percent? Suppose that the size of a packet is 1,500 bytes, including both header fields and data.

P16. Suppose an application uses *rdt 3.0* as its transport layer protocol. As the stop-and-wait protocol has very low channel utilization (shown in the cross-country example), the designers of this application let the receiver keep sending back a number (more than two) of alternating ACK 0 and ACK 1 even if the corresponding data have not arrived at the receiver. Would this application design increase the channel utilization? Why? Are there any potential problems with this approach? Explain.

P17. Consider two network entities, A and B, which are connected by a perfect bi-directional channel (i.e., any message sent will be received correctly; the channel will not corrupt, lose, or re-order packets). A and B are to deliver data messages to each other in an alternating manner: First, A must deliver a message to B, then B must deliver a message to A, then A must deliver a message to B and so on. If an entity is in a state where it should not attempt to deliver a message to the other side, and there is an event like *rdt\_send(data)* call from above that attempts to pass data down for transmission to the other side, this call from above can simply be ignored with a call to *rdt\_unable\_to\_send(data)*, which informs the higher layer that it is currently not able to send data. [Note: This simplifying assumption is made so you don't have to worry about buffering data.]

Draw a FSM specification for this protocol (one FSM for A, and one FSM for B!). Note that you do not have to worry about a reliability mechanism here; the main point of this question is to create a FSM specification that reflects the synchronized behavior of the two entities. You should use the following events and actions that have the same meaning as protocol rdt1.0 in [Figure 3.9](#): *rdt\_send(data)*, *packet = make\_pkt(data)*, *udt\_send(packet)*, *rdt\_rcv(packet)*, *extract(packet, data)*, *deliver\_data(data)*. Make sure your protocol reflects the strict alternation of sending between A and B. Also, make sure to indicate the initial states for A and B in your FSM descriptions.

P18. In the generic SR protocol that we studied in [Section 3.4.4](#), the sender transmits a message as soon as it is available (if it is in the window) without waiting for an acknowledgment. Suppose now that we want an SR protocol that sends messages two at a time. That is, the sender will send a pair of messages and will send the next pair of messages only when it knows that both messages in the first pair have been received correctly.

Suppose that the channel may lose messages but will not corrupt or reorder messages. Design an error-control protocol for the unidirectional reliable transfer of messages. Give an FSM description of the sender and receiver. Describe the format of the packets sent between sender and receiver, and vice versa. If you use any procedure calls other than those in [Section 3.4](#) (for example, `udt_send()`, `start_timer()`, `rdt_rcv()`, and so on), clearly state their actions. Give an example (a timeline trace of sender and receiver) showing how your protocol recovers from a lost packet.

P19. Consider a scenario in which Host A wants to simultaneously send packets to Hosts B and C. A is connected to B and C via a broadcast channel—a packet sent by A is carried by the channel to both B and C. Suppose that the broadcast channel connecting A, B, and C can independently lose and corrupt packets (and so, for example, a packet sent from A might be correctly received by B, but not by C). Design a stop-and-wait-like error-control protocol for reliably transferring packets from A to B and C, such that A will not get new data from the upper layer until it knows that both B and C have correctly received the current packet. Give FSM descriptions of A and C. (*Hint:* The FSM for B should be essentially the same as for C.) Also, give a description of the packet format(s) used.

P20. Consider a scenario in which Host A and Host B want to send messages to Host C. Hosts A and C are connected by a channel that can lose and corrupt (but not reorder) messages. Hosts B and C are connected by another channel (independent of the channel connecting A and C) with the same properties. The transport layer at Host C should alternate in delivering messages from A and B to the layer above (that is, it should first deliver the data from a packet from A, then the data from a packet from B, and so on). Design a stop-and-wait-like error-control protocol for reliably transferring packets from A and B to C, with alternating delivery at C as described above. Give FSM descriptions of A and C. (*Hint:* The FSM for B should be essentially the same as for A.) Also, give a description of the packet format(s) used.

P21. Suppose we have two network entities, A and B. B has a supply of data messages that will be sent to A according to the following conventions. When A gets a request from the layer above to get the next data (D) message from B, A must send a request (R) message to B on the A-to-B channel. Only when B receives an R message can it send a data (D) message back to A on the B-to-A channel. A should deliver exactly one copy of each D message to the layer above. R messages can be lost (but not corrupted) in the A-to-B channel; D messages, once sent, are always delivered correctly. The delay along both channels is unknown and variable.

Design (give an FSM description of) a protocol that incorporates the appropriate mechanisms to compensate for the loss-prone A-to-B channel and implements message passing to the layer above at entity A, as discussed above. Use only those mechanisms that are absolutely

necessary.

P22. Consider the GBN protocol with a sender window size of 4 and a sequence number range of 1,024. Suppose that at time  $t$ , the next in-order packet that the receiver is expecting has a sequence number of  $k$ . Assume that the medium does not reorder messages. Answer the following questions:

- a. What are the possible sets of sequence numbers inside the sender's window at time  $t$ ? Justify your answer.
- b. What are all possible values of the ACK field in all possible messages currently propagating back to the sender at time  $t$ ? Justify your answer.

P23. Consider the GBN and SR protocols. Suppose the sequence number space is of size  $k$ . What is the largest allowable sender window that will avoid the occurrence of problems such as that in [Figure 3.27](#) for each of these protocols?

P24. Answer true or false to the following questions and briefly justify your answer:

- a. With the SR protocol, it is possible for the sender to receive an ACK for a packet that falls outside of its current window.
- b. With GBN, it is possible for the sender to receive an ACK for a packet that falls outside of its current window.
- c. The alternating-bit protocol is the same as the SR protocol with a sender and receiver window size of 1.
- d. The alternating-bit protocol is the same as the GBN protocol with a sender and receiver window size of 1.

P25. We have said that an application may choose UDP for a transport protocol because UDP offers finer application control (than TCP) of what data is sent in a segment and when.

- a. Why does an application have more control of what data is sent in a segment?
- b. Why does an application have more control on when the segment is sent?

P26. Consider transferring an enormous file of  $L$  bytes from Host A to Host B. Assume an MSS of 536 bytes.

- a. What is the maximum value of  $L$  such that TCP sequence numbers are not exhausted? Recall that the TCP sequence number field has 4 bytes.
- b. For the  $L$  you obtain in (a), find how long it takes to transmit the file. Assume that a total of 66 bytes of transport, network, and data-link header are added to each segment before the resulting packet is sent out over a 155 Mbps link. Ignore flow control and congestion control so A can pump out the segments back to back and continuously.

P27. Host A and B are communicating over a TCP connection, and Host B has already received from A all bytes up through byte 126. Suppose Host A then sends two segments to Host B back-to-back. The first and second segments contain 80 and 40 bytes of data, respectively. In the first

segment, the sequence number is 127, the source port number is 302, and the destination port number is 80. Host B sends an acknowledgment whenever it receives a segment from Host A.

- a. In the second segment sent from Host A to B, what are the sequence number, source port number, and destination port number?
- b. If the first segment arrives before the second segment, in the acknowledgment of the first arriving segment, what is the acknowledgment number, the source port number, and the destination port number?
- c. If the second segment arrives before the first segment, in the acknowledgment of the first arriving segment, what is the acknowledgment number?
- d. Suppose the two segments sent by A arrive in order at B. The first acknowledgment is lost and the second acknowledgment arrives after the first timeout interval. Draw a timing diagram, showing these segments and all other segments and acknowledgments sent. (Assume there is no additional packet loss.) For each segment in your figure, provide the sequence number and the number of bytes of data; for each acknowledgment that you add, provide the acknowledgment number.

P28. Host A and B are directly connected with a 100 Mbps link. There is one TCP connection between the two hosts, and Host A is sending to Host B an enormous file over this connection. Host A can send its application data into its TCP socket at a rate as high as 120 Mbps but Host B can read out of its TCP receive buffer at a maximum rate of 50 Mbps. Describe the effect of TCP flow control.

P29. SYN cookies were discussed in [Section 3.5.6](#).

- a. Why is it necessary for the server to use a special initial sequence number in the SYNACK?
- b. Suppose an attacker knows that a target host uses SYN cookies. Can the attacker create half-open or fully open connections by simply sending an ACK packet to the target? Why or why not?
- c. Suppose an attacker collects a large amount of initial sequence numbers sent by the server. Can the attacker cause the server to create many fully open connections by sending ACKs with those initial sequence numbers? Why?

P30. Consider the network shown in Scenario 2 in [Section 3.6.1](#). Suppose both sending hosts A and B have some fixed timeout values.

- a. Argue that increasing the size of the finite buffer of the router might possibly decrease the throughput ( $\lambda_{out}$ ).
- b. Now suppose both hosts dynamically adjust their timeout values (like what TCP does) based on the buffering delay at the router. Would increasing the buffer size help to increase the throughput? Why?

P31. Suppose that the five measured *SampleRTT* values (see [Section 3.5.3](#)) are 106 ms, 120

ms, 140 ms, 90 ms, and 115 ms. Compute the  $\text{EstimatedRTT}$  after each of these SampleRTT values is obtained, using a value of  $\alpha=0.125$  and assuming that the value of  $\text{EstimatedRTT}$  was 100 ms just before the first of these five samples were obtained. Compute also the  $\text{DevRTT}$  after each sample is obtained, assuming a value of  $\beta=0.25$  and assuming the value of  $\text{DevRTT}$  was 5 ms just before the first of these five samples was obtained. Last, compute the TCP  $\text{TimeoutInterval}$  after each of these samples is obtained.

P32. Consider the TCP procedure for estimating RTT. Suppose that  $\alpha=0.1$ . Let  $\text{SampleRTT}_1$  be the most recent sample RTT, let  $\text{SampleRTT}_2$  be the next most recent sample RTT, and so on.

- a. For a given TCP connection, suppose four acknowledgments have been returned with corresponding sample RTTs:  $\text{SampleRTT}_4$ ,  $\text{SampleRTT}_3$ ,  $\text{SampleRTT}_2$ , and  $\text{SampleRTT}_1$ . Express  $\text{EstimatedRTT}$  in terms of the four sample RTTs.
- b. Generalize your formula for  $n$  sample RTTs.
- c. For the formula in part (b) let  $n$  approach infinity. Comment on why this averaging procedure is called an exponential moving average.

P33. In [Section 3.5.3](#), we discussed TCP's estimation of RTT. Why do you think TCP avoids measuring the  $\text{SampleRTT}$  for retransmitted segments?

P34. What is the relationship between the variable  $\text{SendBase}$  in [Section 3.5.4](#) and the variable  $\text{LastByteRcvd}$  in [Section 3.5.5](#)?

P35. What is the relationship between the variable  $\text{LastByteRcvd}$  in [Section 3.5.5](#) and the variable  $y$  in [Section 3.5.4](#)?

P36. In [Section 3.5.4](#), we saw that TCP waits until it has received three duplicate ACKs before performing a fast retransmit. Why do you think the TCP designers chose not to perform a fast retransmit after the first duplicate ACK for a segment is received?

P37. Compare GBN, SR, and TCP (no delayed ACK). Assume that the timeout values for all three protocols are sufficiently long such that 5 consecutive data segments and their corresponding ACKs can be received (if not lost in the channel) by the receiving host (Host B) and the sending host (Host A) respectively. Suppose Host A sends 5 data segments to Host B, and the 2nd segment (sent from A) is lost. In the end, all 5 data segments have been correctly received by Host B.

- a. How many segments has Host A sent in total and how many ACKs has Host B sent in total? What are their sequence numbers? Answer this question for all three protocols.
- b. If the timeout values for all three protocol are much longer than 5 RTT, then which protocol successfully delivers all five data segments in shortest time interval?

P38. In our description of TCP in [Figure 3.53](#), the value of the threshold,  $\text{ssthresh}$ , is set as  $\text{ssthresh}=\text{cwnd}/2$  in several places and  $\text{ssthresh}$  value is referred to as being set to half the window size when a loss event occurred. Must the rate at which the sender is sending when the loss event occurred be approximately equal to  $\text{cwnd}$  segments per RTT? Explain your

answer. If your answer is no, can you suggest a different manner in which  $ssthresh$  should be set?

P39. Consider [Figure 3.46\(b\)](#). If  $\lambda'_{in}$  increases beyond  $R/2$ , can  $\lambda_{out}$  increase beyond  $R/3$ ?

Explain. Now consider [Figure 3.46\(c\)](#). If  $\lambda'_{in}$  increases beyond  $R/2$ , can  $\lambda_{out}$  increase beyond  $R/4$  under the assumption that a packet will be forwarded twice on average from the router to the receiver? Explain.

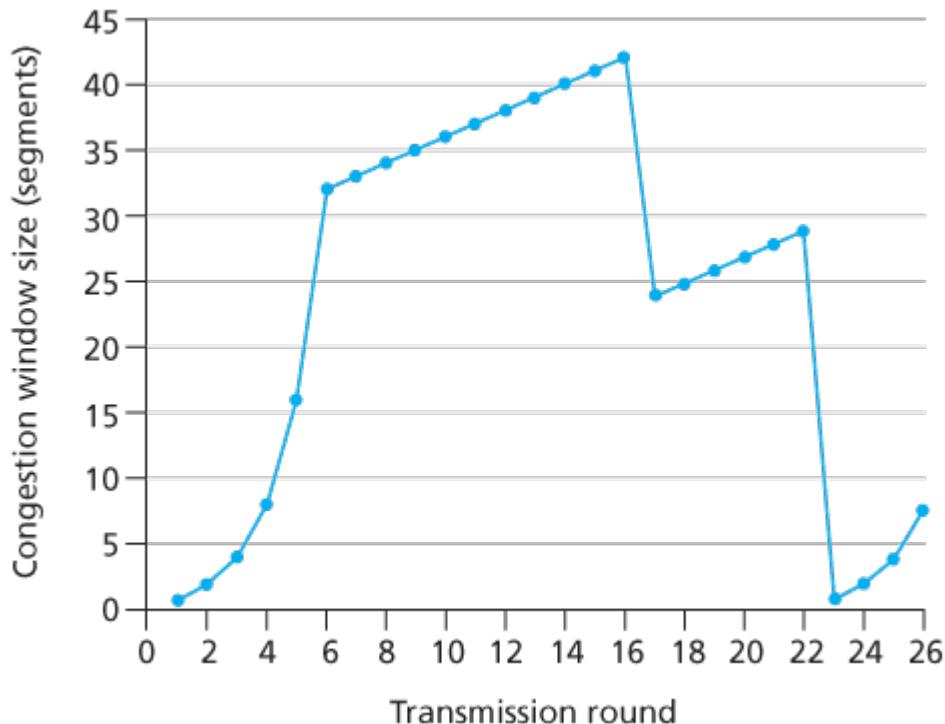
P40. Consider [Figure 3.58](#). Assuming TCP Reno is the protocol experiencing the behavior shown above, answer the following questions. In all cases, you should provide a short discussion justifying your answer.



VideoNote

Examining the behavior of TCP

- a. Identify the intervals of time when TCP slow start is operating.
- b. Identify the intervals of time when TCP congestion avoidance is operating.
- c. After the 16th transmission round, is segment loss detected by a triple duplicate ACK or by a timeout?
- d. After the 22nd transmission round, is segment loss detected by a triple duplicate ACK or by a timeout?



**Figure 3.58 TCP window size as a function of time**

- e. What is the initial value of  $ssthresh$  at the first transmission round?
- f. What is the value of  $ssthresh$  at the 18th transmission round?
- g. What is the value of  $ssthresh$  at the 24th transmission round?
- h. During what transmission round is the 70th segment sent?
- i. Assuming a packet loss is detected after the 26th round by the receipt of a triple duplicate ACK, what will be the values of the congestion window size and of  $ssthresh$ ?
- j. Suppose TCP Tahoe is used (instead of TCP Reno), and assume that triple duplicate ACKs are received at the 16th round. What are the  $ssthresh$  and the congestion window size at the 19th round?
- k. Again suppose TCP Tahoe is used, and there is a timeout event at 22nd round. How many packets have been sent out from 17th round till 22nd round, inclusive?

P41. Refer to [Figure 3.55](#), which illustrates the convergence of TCP's AIMD algorithm. Suppose that instead of a multiplicative decrease, TCP decreased the window size by a constant amount. Would the resulting AIAD algorithm converge to an equal share algorithm? Justify your answer using a diagram similar to [Figure 3.55](#).

P42. In [Section 3.5.4](#), we discussed the doubling of the timeout interval after a timeout event. This mechanism is a form of congestion control. Why does TCP need a window-based congestion-control mechanism (as studied in [Section 3.7](#)) in addition to this doubling-timeout-interval mechanism?

P43. Host A is sending an enormous file to Host B over a TCP connection. Over this connection there is never any packet loss and the timers never expire. Denote the transmission rate of the link connecting Host A to the Internet by  $R$  bps. Suppose that the process in Host A is capable of sending data into its TCP socket at a rate  $S$  bps, where  $S=10 \cdot R$ . Further suppose that the TCP receive buffer is large enough to hold the entire file, and the send buffer can hold only one percent of the file. What would prevent the process in Host A from continuously passing data to its TCP socket at rate  $S$  bps? TCP flow control? TCP congestion control? Or something else? Elaborate.

P44. Consider sending a large file from a host to another over a TCP connection that has no loss.

- a. Suppose TCP uses AIMD for its congestion control without slow start. Assuming  $cwnd$  increases by 1 MSS every time a batch of ACKs is received and assuming approximately constant round-trip times, how long does it take for  $cwnd$  increase from 6 MSS to 12 MSS (assuming no loss events)?
- b. What is the average throughput (in terms of MSS and RTT) for this connection up through time=6 RTT?

P45. Recall the macroscopic description of TCP throughput. In the period of time from when the

connection's rate varies from  $W/(2 \cdot RTT)$  to  $W/RTT$ , only one packet is lost (at the very end of the period).

- a. Show that the loss rate (fraction of packets lost) is equal to

$$L = \text{loss rate} = 138W^2 + 34W$$

- b. Use the result above to show that if a connection has loss rate  $L$ , then its average rate is approximately given by

$$\approx 1.22 \cdot MSS \cdot RTT$$

P46. Consider that only a single TCP (Reno) connection uses one 10Mbps link which does not buffer any data. Suppose that this link is the only congested link between the sending and receiving hosts. Assume that the TCP sender has a huge file to send to the receiver, and the receiver's receive buffer is much larger than the congestion window. We also make the following assumptions: each TCP segment size is 1,500 bytes; the two-way propagation delay of this connection is 150 msec; and this TCP connection is always in congestion avoidance phase, that is, ignore slow start.

- a. What is the maximum window size (in segments) that this TCP connection can achieve?
- b. What is the average window size (in segments) and average throughput (in bps) of this TCP connection?
- c. How long would it take for this TCP connection to reach its maximum window again after recovering from a packet loss?

P47. Consider the scenario described in the previous problem. Suppose that the 10Mbps link can buffer a finite number of segments. Argue that in order for the link to always be busy sending data, we would like to choose a buffer size that is at least the product of the link speed  $C$  and the two-way propagation delay between the sender and the receiver.

P48. Repeat Problem 46, but replacing the 10 Mbps link with a 10 Gbps link. Note that in your answer to part c, you will realize that it takes a very long time for the congestion window size to reach its maximum window size after recovering from a packet loss. Sketch a solution to solve this problem.

P49. Let  $T$  (measured by RTT) denote the time interval that a TCP connection takes to increase its congestion window size from  $W/2$  to  $W$ , where  $W$  is the maximum congestion window size. Argue that  $T$  is a function of TCP's average throughput.

P50. Consider a simplified TCP's AIMD algorithm where the congestion window size is measured in number of segments, not in bytes. In additive increase, the congestion window size increases by one segment in each RTT. In multiplicative decrease, the congestion window size decreases by half (if the result is not an integer, round down to the nearest integer). Suppose that two TCP connections,  $C_1$  and  $C_2$ , share a single congested link of speed 30 segments per second. Assume that both  $C_1$  and  $C_2$  are in the congestion avoidance phase. Connection  $C_1$ 's RTT is 50 msec and connection  $C_2$ 's RTT is 100 msec. Assume that when the data rate in the

link exceeds the link's speed, all TCP connections experience data segment loss.

- a. If both  $C_1$  and  $C_2$  at time  $t_0$  have a congestion window of 10 segments, what are their congestion window sizes after 1000 msec?
- b. In the long run, will these two connections get the same share of the bandwidth of the congested link? Explain.

P51. Consider the network described in the previous problem. Now suppose that the two TCP connections,  $C_1$  and  $C_2$ , have the same RTT of 100 msec. Suppose that at time  $t_0$ ,  $C_1$ 's congestion window size is 15 segments but  $C_2$ 's congestion window size is 10 segments.

- a. What are their congestion window sizes after 2200 msec?
- b. In the long run, will these two connections get about the same share of the bandwidth of the congested link?
- c. We say that two connections are synchronized, if both connections reach their maximum window sizes at the same time and reach their minimum window sizes at the same time. In the long run, will these two connections get synchronized eventually? If so, what are their maximum window sizes?
- d. Will this synchronization help to improve the utilization of the shared link? Why? Sketch some idea to break this synchronization.

P52. Consider a modification to TCP's congestion control algorithm. Instead of additive increase, we can use multiplicative increase. A TCP sender increases its window size by a small positive constant  $\alpha$  ( $0 < \alpha < 1$ ) whenever it receives a valid ACK. Find the functional relationship between loss rate  $L$  and maximum congestion window  $W$ . Argue that for this modified TCP, regardless of TCP's average throughput, a TCP connection always spends the same amount of time to increase its congestion window size from  $W/2$  to  $W$ .

P53. In our discussion of TCP futures in [Section 3.7](#), we noted that to achieve a throughput of 10 Gbps, TCP could only tolerate a segment loss probability of  $2 \cdot 10^{-10}$  (or equivalently, one loss event for every 5,000,000,000 segments). Show the derivation for the values of  $2 \cdot 10^{-10}$  (1 out of 5,000,000) for the RTT and MSS values given in [Section 3.7](#). If TCP needed to support a 100 Gbps connection, what would the tolerable loss be?

P54. In our discussion of TCP congestion control in [Section 3.7](#), we implicitly assumed that the TCP sender always had data to send. Consider now the case that the TCP sender sends a large amount of data and then goes idle (since it has no more data to send) at  $t_1$ . TCP remains idle for a relatively long period of time and then wants to send more data at  $t_2$ . What are the advantages and disadvantages of having TCP use the `cwnd` and `ssthresh` values from  $t_1$  when starting to send data at  $t_2$ ? What alternative would you recommend? Why?

P55. In this problem we investigate whether either UDP or TCP provides a degree of end-point authentication.

- a. Consider a server that receives a request within a UDP packet and responds to that request within a UDP packet (for example, as done by a DNS server). If a client with IP

- address X spoofs its address with address Y, where will the server send its response?
- b. Suppose a server receives a SYN with IP source address Y, and after responding with a SYNACK, receives an ACK with IP source address Y with the correct acknowledgment number. Assuming the server chooses a random initial sequence number and there is no “man-in-the-middle,” can the server be certain that the client is indeed at Y (and not at some other address X that is spoofing Y)?

P56. In this problem, we consider the delay introduced by the TCP slow-start phase. Consider a client and a Web server directly connected by one link of rate  $R$ . Suppose the client wants to retrieve an object whose size is exactly equal to  $15 S$ , where  $S$  is the maximum segment size (MSS). Denote the round-trip time between client and server as RTT (assumed to be constant). Ignoring protocol headers, determine the time to retrieve the object (including TCP connection establishment) when

- a.  $4 S/R > S/R + RTT > 2S/R$
- b.  $S/R + RTT > 4 S/R$
- c.  $S/R > RTT$ .

## Programming Assignments

### *Implementing a Reliable Transport Protocol*

In this laboratory programming assignment, you will be writing the sending and receiving transport-level code for implementing a simple reliable data transfer protocol. There are two versions of this lab, the alternating-bit-protocol version and the GBN version. This lab should be fun—your implementation will differ very little from what would be required in a real-world situation.

Since you probably don’t have standalone machines (with an OS that you can modify), your code will have to execute in a simulated hardware/software environment. However, the programming interface provided to your routines—the code that would call your entities from above and from below—is very close to what is done in an actual UNIX environment. (Indeed, the software interfaces described in this programming assignment are much more realistic than the infinite loop senders and receivers that many texts describe.) Stopping and starting timers are also simulated, and timer interrupts will cause your timer handling routine to be activated.

The full lab assignment, as well as code you will need to compile with your own code, are available at this book’s Web site: [www.pearsonhighered.com/cs-resources](http://www.pearsonhighered.com/cs-resources).

### *Wireshark Lab: Exploring TCP*

In this lab, you'll use your Web browser to access a file from a Web server. As in earlier Wireshark labs, you'll use Wireshark to capture the packets arriving at your computer. Unlike earlier labs, you'll *also* be able to download a Wireshark-readable packet trace from the Web server from which you downloaded the file. In this server trace, you'll find the packets that were generated by your own access of the Web server. You'll analyze the client- and server-side traces to explore aspects of TCP. In particular, you'll evaluate the performance of the TCP connection between your computer and the Web server. You'll trace TCP's window behavior, and infer packet loss, retransmission, flow control and congestion control behavior, and estimated roundtrip time.

As is the case with all Wireshark labs, the full description of this lab is available at this book's Web site, [www.pearsonhighered.com/cs-resources](http://www.pearsonhighered.com/cs-resources).

### *Wireshark Lab: Exploring UDP*

In this short lab, you'll do a packet capture and analysis of your favorite application that uses UDP (for example, DNS or a multimedia application such as Skype). As we learned in [Section 3.3](#), UDP is a simple, no-frills transport protocol. In this lab, you'll investigate the header fields in the UDP segment as well as the checksum calculation.

As is the case with all Wireshark labs, the full description of this lab is available at this book's Web site, [www.pearsonhighered.com/cs-resources](http://www.pearsonhighered.com/cs-resources).

#### AN INTERVIEW WITH...

##### **Van Jacobson**

Van Jacobson works at Google and was previously a Research Fellow at PARC. Prior to that, he was co-founder and Chief Scientist of Packet Design. Before that, he was Chief Scientist at Cisco. Before joining Cisco, he was head of the Network Research Group at Lawrence Berkeley National Laboratory and taught at UC Berkeley and Stanford. Van received the ACM SIGCOMM Award in 2001 for outstanding lifetime contribution to the field of communication networks and the IEEE Kobayashi Award in 2002 for "contributing to the understanding of network congestion and developing congestion control mechanisms that enabled the successful scaling of the Internet". He was elected to the U.S. National Academy of Engineering in 2004.



Please describe one or two of the most exciting projects you have worked on during your career.

What were the biggest challenges?

School teaches us lots of ways to find answers. In every interesting problem I've worked on, the challenge has been finding the right question. When Mike Karels and I started looking at TCP congestion, we spent months staring at protocol and packet traces asking "Why is it failing?". One day in Mike's office, one of us said "The reason I can't figure out why it fails is because I don't understand how it ever worked to begin with." That turned out to be the right question and it forced us to figure out the "ack clocking" that makes TCP work. After that, the rest was easy.

More generally, where do you see the future of networking and the Internet?

For most people, the Web is the Internet. Networking geeks smile politely since we know the Web is an application running over the Internet but what if they're right? The Internet is about enabling conversations between pairs of hosts. The Web is about distributed information production and consumption. "Information propagation" is a very general view of communication of which "pairwise conversation" is a tiny subset. We need to move into the larger tent.

Networking today deals with broadcast media (radios, PONs, etc.) by pretending it's a point-to-point wire. That's massively inefficient. Terabits-per-second of data are being exchanged all over the World via thumb drives or smart phones but we don't know how to treat that as "networking". ISPs are busily setting up caches and CDNs to scalably distribute video and audio. Caching is a necessary part of the solution but there's no part of today's networking—from Information, Queuing or Traffic Theory down to the Internet protocol specs—that tells us how to engineer and deploy it. I think and hope that over the next few years, networking will evolve to embrace the much larger vision of communication that underlies the Web.

What people inspired you professionally?

When I was in grad school, Richard Feynman visited and gave a colloquium. He talked about a piece of Quantum theory that I'd been struggling with all semester and his explanation was so simple and lucid that what had been incomprehensible gibberish to me became obvious and inevitable. That ability to see and convey the simplicity that underlies our complex world seems to me a rare and wonderful gift.

What are your recommendations for students who want careers in computer science and networking?

It's a wonderful field—computers and networking have probably had more impact on society than any invention since the book. Networking is fundamentally about connecting stuff, and studying it helps you make intellectual connections: Ant foraging & Bee dances demonstrate protocol design better than RFCs, traffic jams or people leaving a packed stadium are the essence of congestion, and students finding flights back to school in a post-Thanksgiving blizzard are the core of dynamic routing. If you're interested in lots of stuff and want to have an impact, it's hard to imagine a better field.