Computer Networking

A Top-Down Approach

sixth edition

KUROSE | ROSS

COMPUTER NETWORKING

SIXTH EDITION

A Top-Down Approach

JAMES F. KUROSE

University of Massachusetts, Amherst

KEITH W. ROSS

Polytechnic Institute of NYU

PEARSON

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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 2 of 128

Vice President and Editorial Director, ECS: Marcia Horton
Editor in Chief: Michael Hirsch
Editorial Assistant: Emma Snider
Vice President Marketing: Patrice Jones
Marketing Manager: Yez Alayan
Marketing Coordinator: Kathryn Ferranti
Vice President and Director of Production: Vince O'Brien
Managing Editor: Jeff Holcomb
Senior Production Project Manager: Marilyn Lloyd
Manufacturing Manager: Nick Sklitsis
Operations Specialist: Lisa McDowell
Art Director, Cover: Anthony Gemmellaro
Art Coordinator: Janet Theurer/ Theurer Briggs Design
Art Studio: Patrice Rossi Calkin/ Rossi Illustration and Design
Cover Designer: Liz Harasymcuk
Text Designer: Joyce Cosentino Wells
Cover Image: ©Fancy/Alamy
Media Editor: Dan Sandin
Full-Service Vendor: PreMediaGlobal
Senior Project Manager: Andrea Stefanowicz
Printer/Binder: Edwards Brothers
Cover Printer: Lehigh-Phoenix Color

This book was composed in Quark. Basal font is Times. Display font is Berkeley.

Copyright © 2013, 2010, 2008, 2005, 2003 by Pearson Education, Inc., publishing as Addison-Wesley. 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. To obtain permission(s) to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to 201-236-3290.

Many of the designations by manufacturers and sellers 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
Kurose, James F.
Computer networking : a top-down approach / James F. Kurose, Keith W. Ross.—6th ed.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-13-285620-1
ISBN-10: 0-13-285620-4
1. Internet. 2. Computer networks. I. Ross, Keith W., 1956- II. Title.
TK5105.875.I57K88 2012
004.6—dc23
2011048215

10 9 8 7 6 5 4 3 2 1

PEARSON

ISBN-10: 0-13-285620-4
ISBN-13: 978-0-13-285620-1

About the Authors

Jim Kurose

Jim Kurose is a Distinguished University Professor of Computer Science at the University of Massachusetts, Amherst.

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 been the recipient of a GE Fellowship, an IBM Faculty Development Award, and a Lilly Teaching Fellowship.

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

Keith Ross

Keith Ross is the Leonard J. Shustek Chair Professor and Head of the Computer Science Department at Polytechnic Institute of NYU. Before joining NYU-Poly in 2003, he was a professor at the University of Pennsylvania (13 years) and a professor at Eurecom Institute (5 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 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 security and privacy, social networks, peer-to-peer networking, Internet measurement, video streaming, content distribution networks, and stochastic modeling. He is 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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 4 of 128

This page intentionally left blank

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

This page intentionally left blank

Preface

Welcome to the sixth edition of *Computer Networking: A Top-Down Approach.* Since the publication of the first edition 12 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 Sixth 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 sixth 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 sixth edition has been revised and updated substantially:

- The Companion Web site has been significantly expanded and enriched to include VideoNotes and interactive exercises, as discussed later in this Preface.
- In Chapter 1, the treatment of access networks has been modernized, and the description of the Internet ISP ecosystem has been substantially revised, accounting for the recent emergence of content provider networks, such as Google's. The presentation of packet switching and circuit switching has also been reorganized, providing a more topical rather than historical orientation.
- In Chapter 2, Python has replaced Java for the presentation of socket programming. While still explicitly exposing the key ideas behind the socket API, Python code is easier to understand for the novice programmer. Moreover, unlike Java, Python provides access to raw sockets, enabling students to build a larger variety of network applications. Java-based socket programming labs have been replaced with corresponding Python labs, and a new Python-based ICMP Ping lab has been added. As always, when material is retired from the book, such as Java-based socket programming material, it remains available on the book's Companion Web site (see following text).
- In Chapter 3, the presentation of one of the reliable data transfer protocols has been simplified and a new sidebar on TCP splitting, commonly used to optimize the performance of cloud services, has been added.
- In Chapter 4, the section on router architectures has been significantly updated, reflecting recent developments and practices in the field. Several new integrative sidebars involving DNS, BGP, and OSPF are included.

- Chapter 5 has been reorganized and streamlined, accounting for the ubiquity of switched Ethernet in local area networks and the consequent increased use of Ethernet in point-to-point scenarios. Also, a new section on data center networking has been added.
- Chapter 6 has been updated to reflect recent advances in wireless networks, particularly cellular data networks and 4G services and architecture.
- Chapter 7, which focuses on multimedia networking, has gone through a major revision. The chapter now includes an in-depth discussion of streaming video, including adaptive streaming, and an entirely new and modernized discussion of CDNs. A newly added section describes the Netflix, YouTube, and Kankan video streaming systems. The material that has been removed to make way for these new topics is still available on the Companion Web site.
- Chapter 8 now contains an expanded discussion on endpoint authentication.
- Significant new material involving end-of-chapter problems has been added. As with all previous editions, homework problems have been revised, added, and removed.

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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 9 of 128

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 12 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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 10 of 128

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. Two fundamentally important network-layer issues are determining "good" paths between two routers and interconnecting a large number of heterogeneous networks. 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 Web Site

Each new copy of this textbook includes six months of access to a Companion Web site for all book readers at http://www.pearsonhighered.com/kurose-ross, which includes:

- *Interactive learning material.* An important new component of the sixth edition is the significantly expanded online and interactive learning material. The book's Companion Web site now contains VideoNotes—video presentations of

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 11 of 128

important topics thoughout the book done by the authors, as well as walk-throughs of solutions to problems similar to those at the end of the chapter. We've also added Interactive Exercises that can 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. 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 ATM networks and the RTSP protocol for multimedia. Material that appeared in earlier editions of the text is still of interest, and 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.

Pedagogical Features

We have each been teaching computer networking for more than 20 years. Together, we bring more than 50 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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 12 of 128

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 or send e-mail to computing@aw.com 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 sixth 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 five 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 5 in sequence, following our top-down philosophy. Each of these five chapters leverages material from the preceding chapters. After completing the first five chapters, the instructor has quite a bit of flexibility. There are no interdependencies among the last four chapters, so they can be taught in any order. However, each of the last four chapters depends on the material in the first five chapters. Many instructors first teach the first five chapters and then teach one of the last four chapters for "dessert."

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 13 of 128

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 four 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 and ross@poly.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 fifth 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)

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 14 of 128

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)
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)

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 15 of 128

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 Sate 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 Rodrigez (Telefonica)
Sumit Roy (University of Washington)
Avi Rubin (Johns Hopkins University)
Dan Rubenstein (Columbia University)
Douglas Salane (John Jay College)
Despina Saparilla (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)
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)
Cliff C. Zou (University of Central Florida)

We also want to thank the entire Addison-Wesley team—in particular, Michael Hirsch, Marilyn Lloyd, and Emma Snider—who have done an absolutely outstanding job on this sixth 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 book, and to Andrea Stefanowicz and her team at PreMediaGlobal for their wonderful production work on this edition. Finally, a most special thanks go to Michael Hirsch, our editor at Addison-Wesley, and Susan Hartman, our former editor at Addison-Wesley. 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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 17 of 128

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 22
- 1.3.1 Packet Switching 22
- 1.3.2 Circuit Switching 27
- 1.3.3 A Network of Networks 32

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 42
- 1.4.4 Throughput in Computer Networks 44

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 60
- 1.7.1 The Development of Packet Switching: 1961–1972 60
- 1.7.2 Proprietary Networks and Internetworking: 1972–1980 62
- 1.7.3 A Proliferation of Networks: 1980–1990 63
- 1.7.4 The Internet Explosion: The 1990s 64
- 1.7.5 The New Millennium 65

1.8 Summary 66

Homework Problems and Questions 68

Wireshark Lab 78

Interview: Leonard Kleinrock 80

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 18 of 128

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 91
- 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
- 2.2.6 The Conditional GET 114

2.3 File Transfer: FTP 116
- 2.3.1 FTP Commands and Replies 118

2.4 Electronic Mail in the Internet 118
- 2.4.1 SMTP 121
- 2.4.2 Comparison with HTTP 124
- 2.4.3 Mail Message Format 125
- 2.4.4 Mail Access Protocols 125

2.5 DNS—The Internet's Directory Service 130
- 2.5.1 Services Provided by DNS 131
- 2.5.2 Overview of How DNS Works 133
- 2.5.3 DNS Records and Messages 139

2.6 Peer-to-Peer Applications 144
- 2.6.1 P2P File Distribution 145
- 2.6.2 Distributed Hash Tables (DHTs) 151

2.7 Socket Programming: Creating Network Applications 156
- 2.7.1 Socket Programming with UDP 157
- 2.7.2 Socket Programming with TCP 163

2.8 Summary 168

Homework Problems and Questions 169

Socket Programming Assignments 179

Wireshark Labs: HTTP, DNS 181

Interview: Marc Andreessen 182

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 19 of 128

Chapter 3 Transport Layer 185

3.1 Introduction and Transport-Layer Services 186
3.1.1 Relationship Between Transport and Network Layers 186
3.1.2 Overview of the Transport Layer in the Internet 189
3.2 Multiplexing and Demultiplexing 191
3.3 Connectionless Transport: UDP 198
3.3.1 UDP Segment Structure 202
3.3.2 UDP Checksum 202
3.4 Principles of Reliable Data Transfer 204
3.4.1 Building a Reliable Data Transfer Protocol 206
3.4.2 Pipelined Reliable Data Transfer Protocols 215
3.4.3 Go-Back-N (GBN) 218
3.4.4 Selective Repeat (SR) 223
3.5 Connection-Oriented Transport: TCP 230
3.5.1 The TCP Connection 231
3.5.2 TCP Segment Structure 233
3.5.3 Round-Trip Time Estimation and Timeout 238
3.5.4 Reliable Data Transfer 242
3.5.5 Flow Control 250
3.5.6 TCP Connection Management 252
3.6 Principles of Congestion Control 259
3.6.1 The Causes and the Costs of Congestion 259
3.6.2 Approaches to Congestion Control 265
3.6.3 Network-Assisted Congestion-Control Example: ATM ABR Congestion Control 266
3.7 TCP Congestion Control 269
3.7.1 Fairness 279
3.8 Summary 283
Homework Problems and Questions 285
Programming Assignments 300
Wireshark Labs: TCP, UDP 301
Interview: Van Jacobson 302

Chapter 4 The Network Layer 305

4.1 Introduction 306
4.1.1 Forwarding and Routing 308
4.1.2 Network Service Models 310
4.2 Virtual Circuit and Datagram Networks 313
4.2.1 Virtual-Circuit Networks 314
4.2.2 Datagram Networks 317
4.2.3 Origins of VC and Datagram Networks 319

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 20 of 128

4.3 What's Inside a Router? 320
4.3.1 Input Processing 322
4.3.2 Switching 324
4.3.3 Output Processing 326
4.3.4 Where Does Queuing Occur? 327
4.3.5 The Routing Control Plane 331
4.4 The Internet Protocol (IP): Forwarding and Addressing in the Internet 331
4.4.1 Datagram Format 332
4.4.2 IPv4 Addressing 338
4.4.3 Internet Control Message Protocol (ICMP) 353
4.4.4 IPv6 356
4.4.5 A Brief Foray into IP Security 362
4.5 Routing Algorithms 363
4.5.1 The Link-State (LS) Routing Algorithm 366
4.5.2 The Distance-Vector (DV) Routing Algorithm 371
4.5.3 Hierarchical Routing 379
4.6 Routing in the Internet 383
4.6.1 Intra-AS Routing in the Internet: RIP 384
4.6.2 Intra-AS Routing in the Internet: OSPF 388
4.6.3 Inter-AS Routing: BGP 390
4.7 Broadcast and Multicast Routing 399
4.7.1 Broadcast Routing Algorithms 400
4.7.2 Multicast 405
4.8 Summary 412
Homework Problems and Questions 413
Programming Assignments 429
Wireshark Labs: IP, ICMP 430
Interview: Vinton G. Cerf 431

Chapter 5 The Link Layer: Links, Access Networks, and LANs 433

5.1 Introduction to the Link Layer 434
5.1.1 The Services Provided by the Link Layer 436
5.1.2 Where Is the Link Layer Implemented? 437
5.2 Error-Detection and -Correction Techniques 438
5.2.1 Parity Checks 440
5.2.2 Checksumming Methods 442
5.2.3 Cyclic Redundancy Check (CRC) 443
5.3 Multiple Access Links and Protocols 445
5.3.1 Channel Partitioning Protocols 448
5.3.2 Random Access Protocols 449
5.3.3 Taking-Turns Protocols 459
5.3.4 DOCSIS: The Link-Layer Protocol for Cable Internet Access 460

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 21 of 128

5.4 Switched Local Area Networks 461
5.4.1 Link-Layer Addressing and ARP 462
5.4.2 Ethernet 469
5.4.3 Link-Layer Switches 476
5.4.4 Virtual Local Area Networks (VLANs) 482
5.5 Link Virtualization: A Network as a Link Layer 486
5.5.1 Multiprotocol Label Switching (MPLS) 487
5.6 Data Center Networking 490
5.7 Retrospective: A Day in the Life of a Web Page Request 495
5.7.1 Getting Started: DHCP, UDP, IP, and Ethernet 495
5.7.2 Still Getting Started: DNS and ARP 497
5.7.3 Still Getting Started: Intra-Domain Routing to the DNS Server 498
5.7.4 Web Client-Server Interaction: TCP and HTTP 499
5.8 Summary 500
Homework Problems and Questions 502
Wireshark Labs: Ethernet and ARP, DHCP 510
Interview: Simon S. Lam 511

Chapter 6 Wireless and Mobile Networks 513
6.1 Introduction 514
6.2 Wireless Links and Network Characteristics 519
6.2.1 CDMA 522
6.3 WiFi: 802.11 Wireless LANs 526
6.3.1 The 802.11 Architecture 527
6.3.2 The 802.11 MAC Protocol 531
6.3.3 The IEEE 802.11 Frame 537
6.3.4 Mobility in the Same IP Subnet 541
6.3.5 Advanced Features in 802.11 542
6.3.6 Personal Area Networks: Bluetooth and Zigbee 544
6.4 Cellular Internet Access 546
6.4.1 An Overview of Cellular Network Architecture 547
6.4.2 3G Cellular Data Networks: Extending the Internet to Cellular Subscribers 550
6.4.3 On to 4G: LTE 553
6.5 Mobility Management: Principles 555
6.5.1 Addressing 557
6.5.2 Routing to a Mobile Node 559
6.6 Mobile IP 564
6.7 Managing Mobility in Cellular Networks 570
6.7.1 Routing Calls to a Mobile User 571
6.7.2 Handoffs in GSM 572

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 22 of 128

6.8 Wireless and Mobility: Impact on Higher-Layer Protocols 575
6.9 Summary 578
Homework Problems and Questions 578
Wireshark Lab: IEEE 802.11 (WiFi) 583
Interview: Deborah Estrin 584

Chapter 7 Multimedia Networking 587
7.1 Multimedia Networking Applications 588
7.1.1 Properties of Video 588
7.1.2 Properties of Audio 590
7.1.3 Types of Multimedia Network Applications 591
7.2 Streaming Stored Video 593
7.2.1 UDP Streaming 595
7.2.2 HTTP Streaming 596
7.2.3 Adaptive Streaming and DASH 600
7.2.4 Content Distribution Networks 602
7.2.5 Case Studies: Netflix, YouTube, and Kankan 608
7.3 Voice-over-IP 612
7.3.1 Limitations of the Best-Effort IP Service 612
7.3.2 Removing Jitter at the Receiver for Audio 614
7.3.3 Recovering from Packet Loss 617
7.3.4 Case Study: VoIP with Skype 620
7.4 Protocols for Real-Time Conversational Applications 623
7.4.1 RTP 624
7.4.2 SIP 627
7.5 Network Support for Multimedia 632
7.5.1 Dimensioning Best-Effort Networks 634
7.5.2 Providing Multiple Classes of Service 636
7.5.3 Diffserv 648
7.5.4 Per-Connection Quality-of-Service (QoS) Guarantees: Resource Reservation and Call Admission 652
7.6 Summary 655
Homework Problems and Questions 656
Programming Assignment 666
Interview: Henning Schulzrinne 668

Chapter 8 Security in Computer Networks 671
8.1 What Is Network Security? 672
8.2 Principles of Cryptography 675
8.2.1 Symmetric Key Cryptography 676
8.2.2 Public Key Encryption 683

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 23 of 128

8.3 Message Integrity and Digital Signatures 688
8.3.1 Cryptographic Hash Functions 689
8.3.2 Message Authentication Code 691
8.3.3 Digital Signatures 693
8.4 End-Point Authentication 700
8.4.1 Authentication Protocol *ap1.0* 700
8.4.2 Authentication Protocol *ap2.0* 701
8.4.3 Authentication Protocol *ap3.0* 702
8.4.4 Authentication Protocol *ap3.1* 703
8.4.5 Authentication Protocol *ap4.0* 703
8.5 Securing E-Mail 705
8.5.1 Secure E-Mail 706
8.5.2 PGP 710
8.6 Securing TCP Connections: SSL 711
8.6.1 The Big Picture 713
8.6.2 A More Complete Picture 716
8.7 Network-Layer Security: IPsec and Virtual Private Networks 718
8.7.1 IPsec and Virtual Private Networks (VPNs) 718
8.7.2 The AH and ESP Protocols 720
8.7.3 Security Associations 720
8.7.4 The IPsec Datagram 721
8.7.5 IKE: Key Management in IPsec 725
8.8 Securing Wireless LANs 726
8.8.1 Wired Equivalent Privacy (WEP) 726
8.8.2 IEEE 802.11i 728
8.9 Operational Security: Firewalls and Intrusion Detection Systems 731
8.9.1 Firewalls 731
8.9.2 Intrusion Detection Systems 739
8.10 Summary 742
Homework Problems and Questions 744
Wireshark Lab: SSL 752
IPsec Lab 752
Interview: Steven M. Bellovin 753

Chapter 9 Network Management 755
9.1 What Is Network Management? 756
9.2 The Infrastructure for Network Management 760
9.3 The Internet-Standard Management Framework 764
9.3.1 Structure of Management Information: SMI 766
9.3.2 Management Information Base: MIB 770

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 24 of 128

9.3.3 SNMP Protocol Operations and Transport Mappings 772
9.3.4 Security and Administration 775
9.4 ASN.1 778
9.5 Conclusion 783
Homework Problems and Questions 783
Interview: Jennifer Rexford 786

References 789
Index 823

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 25 of 128

COMPUTER NETWORKING

SIXTH EDITION

A Top-Down Approach

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 26 of 128

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 protocols that support these applications. 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 email, 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. Since 2000, we have seen an explosion of popular voice and video applications, including: voice-over-IP (VoIP) and video conferencing over IP such as Skype; user-generated video distribution such as YouTube; and movies on demand such as Netflix. During this same period we have also seen the immergence of highly engaging multi-player online games, including Second Life and World of Warcraft. And most recently, we have seen the emergence of a new generation of social networking applications, such as Facebook and Twitter, which have created engaging human networks on top of the Internet's network of routers and communication links. Clearly, there has been no slowing down of new

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 27 of 128

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, and peer-to-peer (P2P) file distribution (Chapter 8 focuses on multimedia applications, including streaming video and VoIP). We then cover network application development, over both TCP and UDP. In particular, we study the socket API 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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 28 of 128

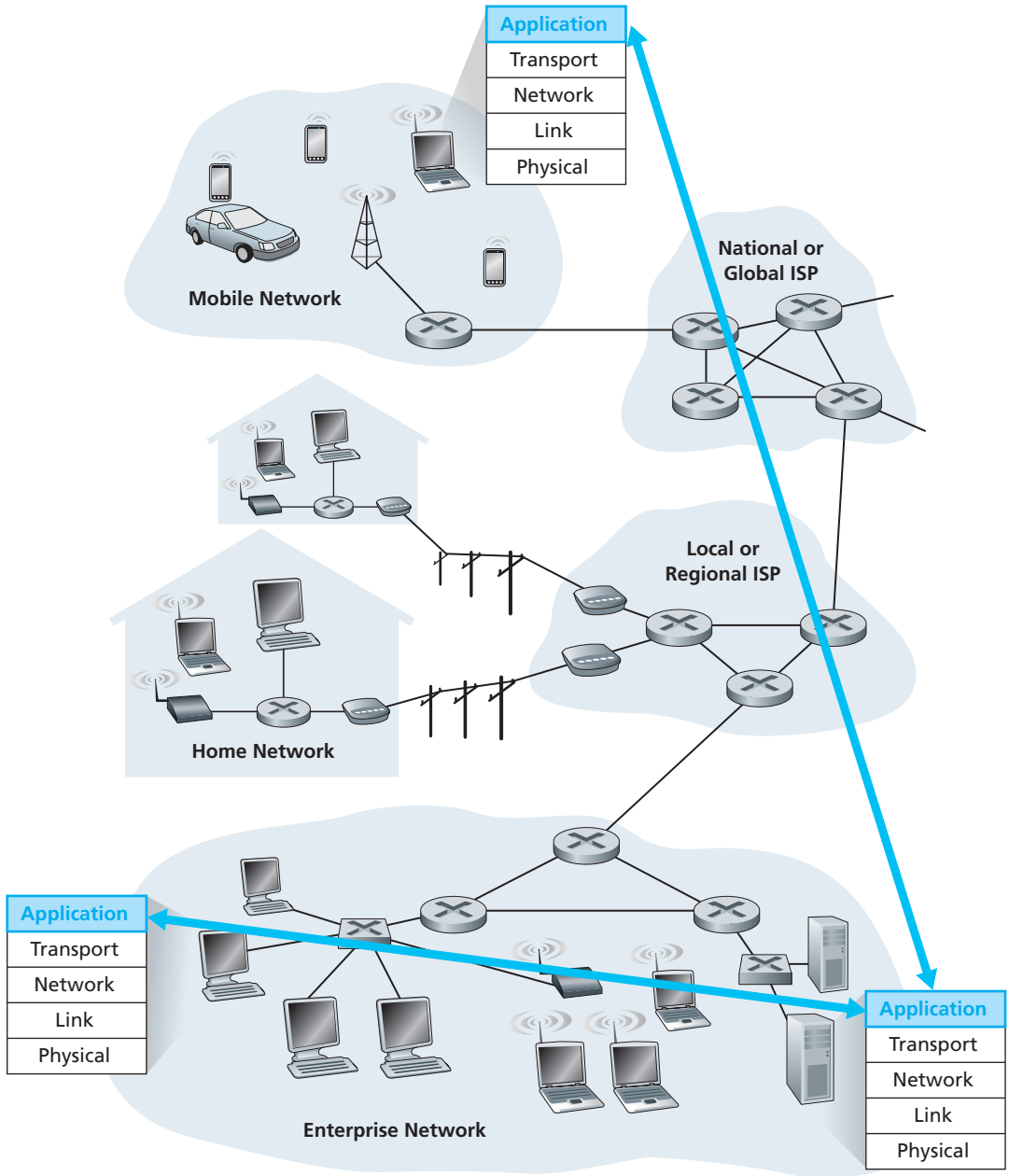


Figure 2.1 ♦ Communication for a network application takes place between end systems at the application layer

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 29 of 128

2.1.1 Network Application Architectures

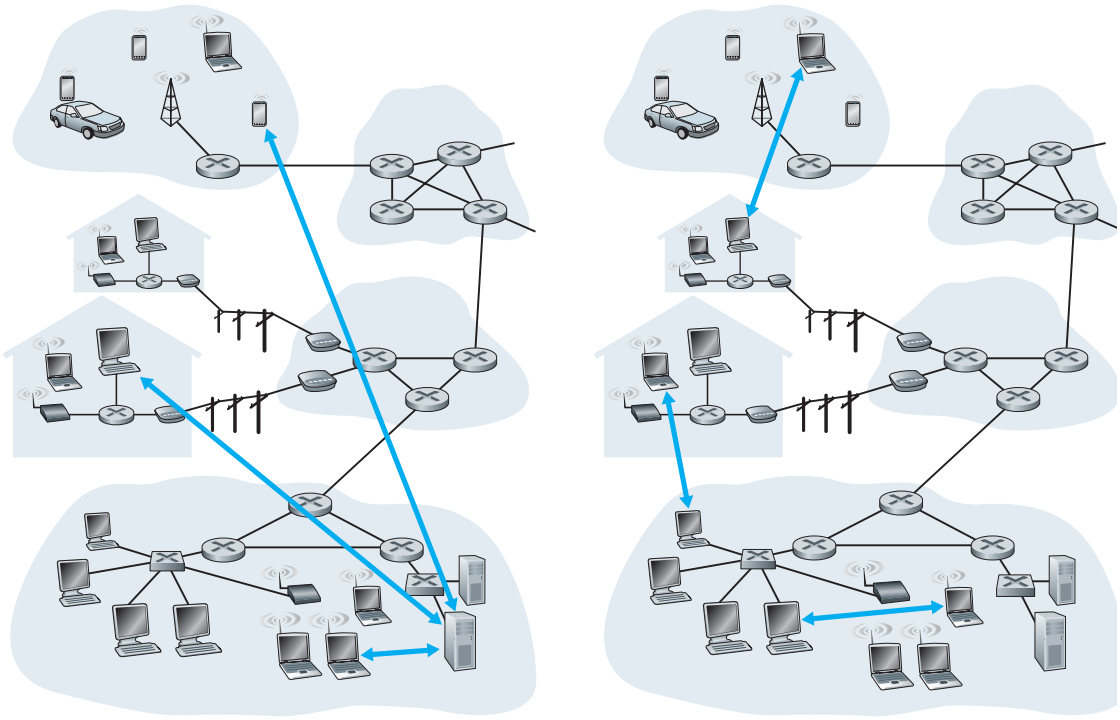
Before diving into software coding, you should have a broad architectural plan for your application. Keep in mind that an application's architecture is distinctly different from the network architecture (e.g., the five-layer Internet architecture discussed in Chapter 1). From the application developer's 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 and Bing), Internet commerce (e.g., Amazon and e-Bay), Web-based email (e.g., Gmail and Yahoo Mail), social networking (e.g., Facebook and Twitter)—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 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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 30 of 128



a. Client-server architecture

b. Peer-to-peer architecture

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

download acceleration (e.g., Xunlei), Internet Telephony (e.g., Skype), and IPTV (e.g., Kankan and PPstream). 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, future P2P applications face three major challenges:

1. *ISP Friendly.* Most residential ISPs (including DSL and cable ISPs) have been dimensioned for "asymmetrical" bandwidth usage, that is, for much more

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 31 of 128

downstream than upstream traffic. But P2P video streaming and file distribution applications shift upstream traffic from servers to residential ISPs, thereby putting significant stress on the ISPs. Future P2P applications need to be designed so that they are friendly to ISPs [Xie 2008].

2. *Security.* Because of their highly distributed and open nature, P2P applications can be a challenge to secure [Doucer 2002; Yu 2006; Liang 2006; Naoumov 2006; Dhungel 2008; LeBlond 2011].
3. *Incentives.* The success of future P2P applications also depends on convincing users to volunteer bandwidth, storage, and computation resources to the applications, which is the challenge of incentive design [Feldman 2005; Piatek 2008; Aperjis 2008; Liu 2010].

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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 32 of 128

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),

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 33 of 128

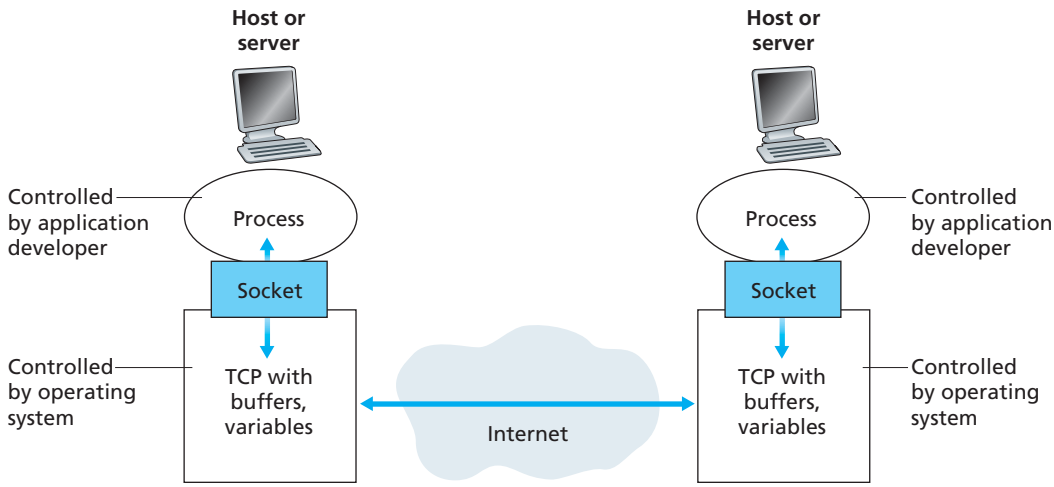


Figure 2.3 ♦ Application processes, sockets, and underlying transport protocol

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. 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 http://www.iana.org. We'll examine port numbers in detail in Chapter 3.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 34 of 128

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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 35 of 128

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 7, [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,

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 36 of 128

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.

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–1Mbps 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
Instant messaging	No loss	Elastic	Yes and no

Figure 2.4 ♦ Requirements of selected network applications

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 37 of 128

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 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.

FOCUS ON SECURITY

SECURING TCP

Neither TCP nor UDP provide 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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 38 of 128

- *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 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 7. 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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 39 of 128

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

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.

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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 40 of 128

- 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, file transfer, electronic mail, directory service, 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. After covering the Web, we briefly examine FTP, because it provides a nice contrast to HTTP. 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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 41 of 128

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. Finally, we discuss in this chapter several P2P applications, focusing on file sharing applications, and distributed lookup services. In Chapter 7, we'll cover multimedia applications, including streaming video and voice-over-IP.

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 Web sites. Graphics stimulate our senses. Forms, JavaScript, Java applets, and many other devices enable us to interact with pages and sites. And the Web serves as a platform for many killer applications emerging after 2003, including YouTube, Gmail, and Facebook.

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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 42 of 128

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

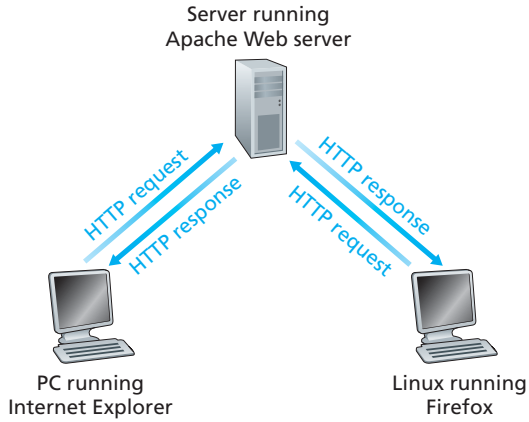


Figure 2.6 ♦ HTTP request-response behavior

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 43 of 128

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 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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 44 of 128

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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 45 of 128

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

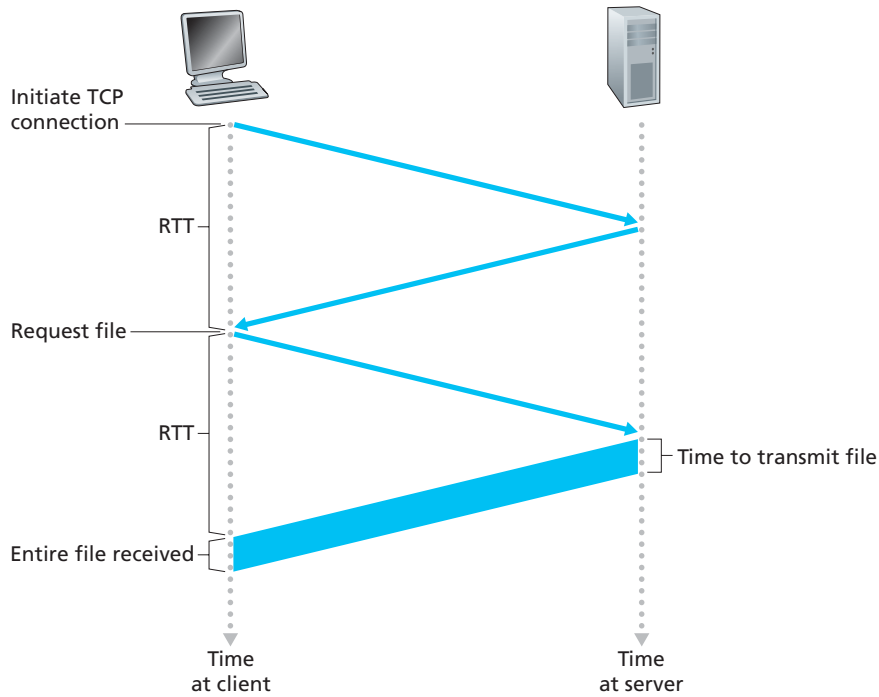


Figure 2.7 ♦ Back-of-the-envelope calculation for the time needed to request and receive an HTML file

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 46 of 128

the three-way handshake (the acknowledgment) into the TCP connection. Once the request message arrives at 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 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. 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].

2.2.3 HTTP Message Format

The HTTP specifications [RFC 1945; RFC 2616] 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
```

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 47 of 128

```
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:` 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:` 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:` 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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 48 of 128

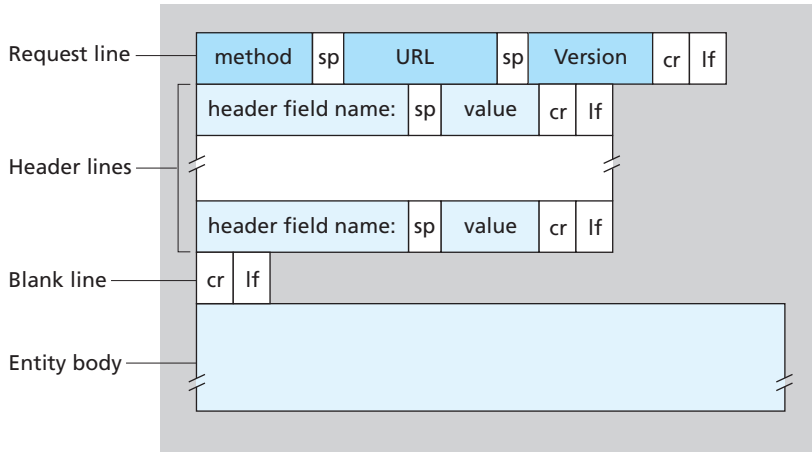


Figure 2.8 ♦ General format of an HTTP request message

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`. 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
```

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 49 of 128

```
Date: Tue, 09 Aug 2011 15:44:04 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 09 Aug 2011 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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 50 of 128

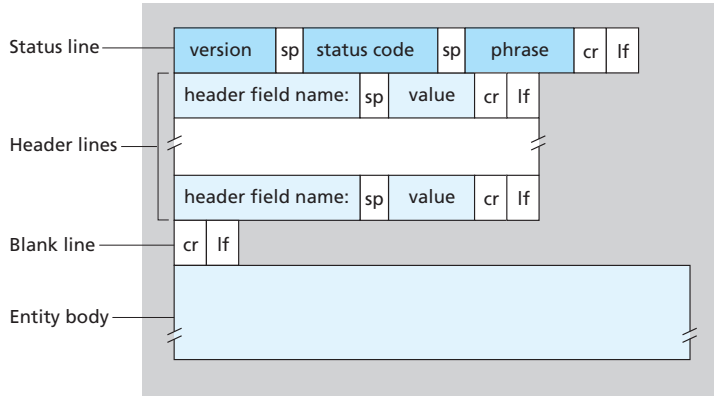


Figure 2.9 ♦ General format of an HTTP response message

- `400 Bad Request:` This is a generic error code indicating that the request could not be understood by the server.
- `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 cis.poly.edu 80

GET /~ross/ HTTP/1.1
Host: cis.poly.edu
```

(Press the carriage return twice after typing the last line.) This opens a TCP connection to port 80 of the host `cis.poly.edu` and then sends the HTTP request message. You should see a response message that includes the base HTML file of Professor Ross's homepage. If you'd rather just see the HTTP message lines and not receive the object itself, replace `GET` with `HEAD`. Finally, replace `/~ross/` with `/~banana/` and see what kind of response message you get.

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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 51 of 128

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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 52 of 128

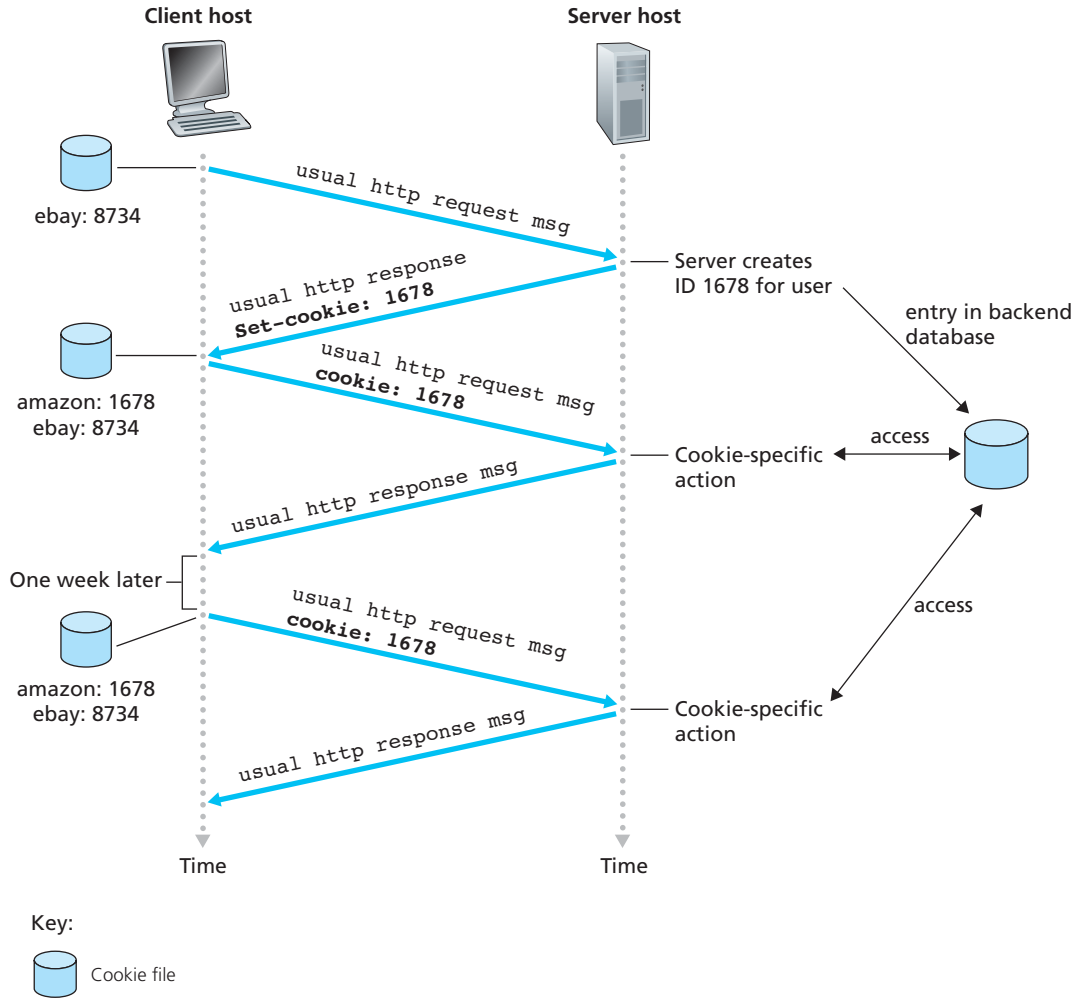


Figure 2.10 ♦ Keeping user state with cookies

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

```
Cookie: 1678
```

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 53 of 128

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 2012] 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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 54 of 128

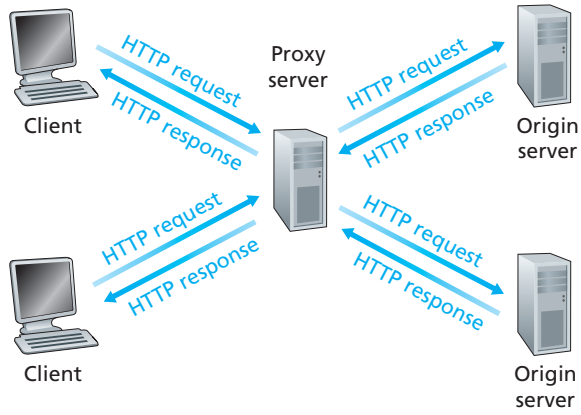


Figure 2.11 ♦ Clients requesting objects through a Web cache

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.
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 AOL) 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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 55 of 128

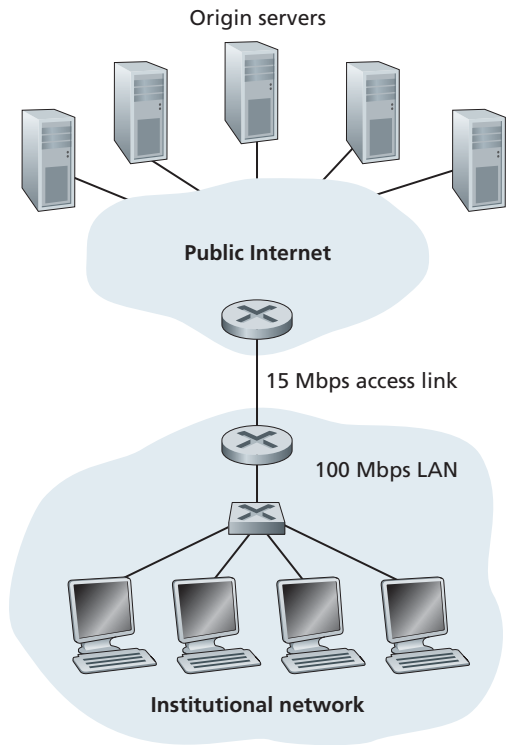


Figure 2.12 ♦ Bottleneck between an institutional network and the Internet

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."

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 56 of 128

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 to upgrade its link to the Internet. The institution does, of course, have to purchase

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 57 of 128

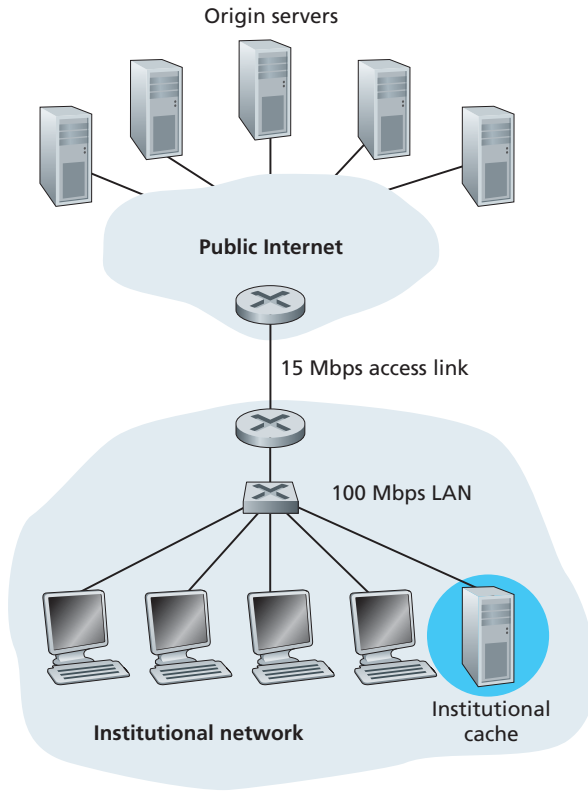


Figure 2.13 ♦ Adding a cache to the institutional network

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 Microsoft). We will discuss CDNs in more detail in Chapter 7.

2.2.6 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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 58 of 128

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, 8 Oct 2011 15:39:29
Server: Apache/1.3.0 (Unix)
Last-Modified: Wed, 7 Sep 2011 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, 7 Sep 2011 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 7 Sep 2011 09:23:24. Then, fourth, the Web server sends a response message to the cache:

```
HTTP/1.1 304 Not Modified
Date: Sat, 15 Oct 2011 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 File Transfer: FTP

In a typical FTP session, the user is sitting in front of one host (the local host) and wants to transfer files to or from a remote host. In order for the user to access the remote account, the user must provide a user identification and a password. After providing this authorization information, the user can transfer files from the local file system to the remote file system and vice versa. As shown in Figure 2.14, the user interacts with FTP through an FTP user agent. The user first provides the hostname of the remote host, causing the FTP client process in the local host to establish a TCP connection with the FTP server process in the remote host. The user then provides the user identification and password, which are sent over the TCP connection as part of FTP commands. Once the server has authorized the user, the user copies one or more files stored in the local file system into the remote file system (or vice versa).

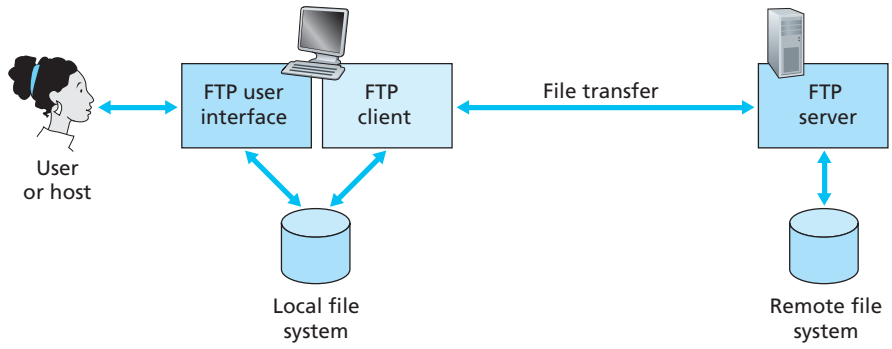


Figure 2.14 ♦ FTP moves files between local and remote file systems

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 60 of 128

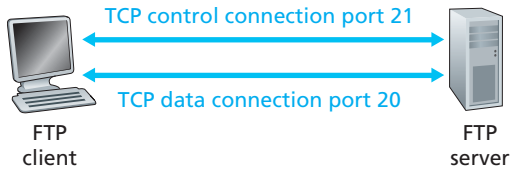


Figure 2.15 ♦ Control and data connections

HTTP and FTP are both file transfer protocols and have many common characteristics; for example, they both run on top of TCP. However, the two application-layer protocols have some important differences. The most striking difference is that FTP uses two parallel TCP connections to transfer a file, a **control connection** and a **data connection**. The control connection is used for sending control information between the two hosts—information such as user identification, password, commands to change remote directory, and commands to "put" and "get" files. The data connection is used to actually send a file. Because FTP uses a separate control connection, FTP is said to send its control information **out-of-band**. HTTP, as you recall, sends request and response header lines into the same TCP connection that carries the transferred file itself. For this reason, HTTP is said to send its control information **in-band**. In the next section, we'll see that SMTP, the main protocol for electronic mail, also sends control information in-band. The FTP control and data connections are illustrated in Figure 2.15.

When a user starts an FTP session with a remote host, the client side of FTP (user) first initiates a control TCP connection with the server side (remote host) on server port number 21. The client side of FTP sends the user identification and password over this control connection. The client side of FTP also sends, over the control connection, commands to change the remote directory. When the server side receives a command for a file transfer over the control connection (either to, or from, the remote host), the server side initiates a TCP data connection to the client side. FTP sends exactly one file over the data connection and then closes the data connection. If, during the same session, the user wants to transfer another file, FTP opens another data connection. Thus, with FTP, the control connection remains open throughout the duration of the user session, but a new data connection is created for each file transferred within a session (that is, the data connections are non-persistent).

Throughout a session, the FTP server must maintain **state** about the user. In particular, the server must associate the control connection with a specific user account, and the server must keep track of the user's current directory as the user wanders about the remote directory tree. Keeping track of this state information for each ongoing user session significantly constrains the total number of sessions that FTP can maintain simultaneously. Recall that HTTP, on the other hand, is stateless—it does not have to keep track of any user state.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 61 of 128

2.3.1 FTP Commands and Replies

We end this section with a brief discussion of some of the more common FTP commands and replies. The commands, from client to server, and replies, from server to client, are sent across the control connection in 7-bit ASCII format. Thus, like HTTP commands, FTP commands are readable by people. In order to delineate successive commands, a carriage return and line feed end each command. Each command consists of four uppercase ASCII characters, some with optional arguments. Some of the more common commands are given below:

- `USER username:` Used to send the user identification to the server.
- `PASS password:` Used to send the user password to the server.
- `LIST:` Used to ask the server to send back a list of all the files in the current remote directory. The list of files is sent over a (new and non-persistent) data connection rather than the control TCP connection.
- `RETR filename:` Used to retrieve (that is, get) a file from the current directory of the remote host. This command causes the remote host to initiate a data connection and to send the requested file over the data connection.
- `STOR filename:` Used to store (that is, put) a file into the current directory of the remote host.

There is typically a one-to-one correspondence between the command that the user issues and the FTP command sent across the control connection. Each command is followed by a reply, sent from server to client. The replies are three-digit numbers, with an optional message following the number. This is similar in structure to the status code and phrase in the status line of the HTTP response message. Some typical replies, along with their possible messages, are as follows:

- `331 Username OK, password required`
- `125 Data connection already open; transfer starting`
- `425 Can't open data connection`
- `452 Error writing file`

Readers who are interested in learning about the other FTP commands and replies are encouraged to read RFC 959.

2.4 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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 62 of 128

become more and 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 2.16 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

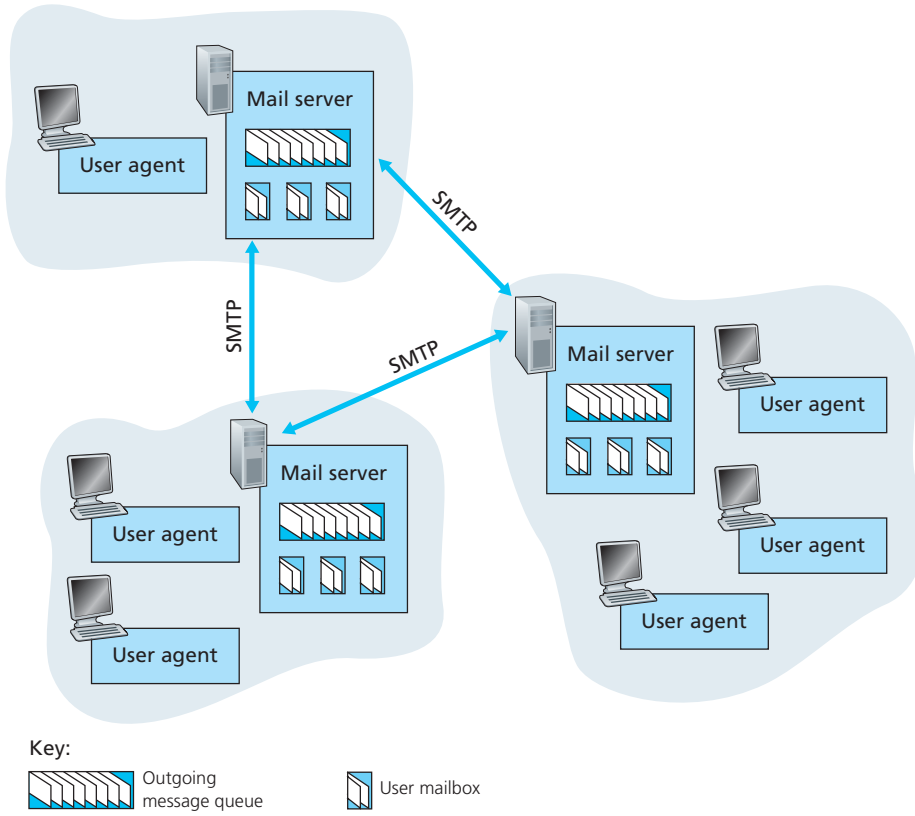


Figure 2.16 ♦ A high-level view of the Internet e-mail system

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 63 of 128

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 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.

CASE HISTORY

WEB E-MAIL

In December 1995, just a few years after the Web was "invented," Sabeer Bhatia and Jack Smith visited the Internet venture capitalist Draper Fisher Jurvetson and proposed developing a free Web-based e-mail system. The idea was to give a free e-mail account to anyone who wanted one, and to make the accounts accessible from the Web. In exchange for 15 percent of the company, Draper Fisher Jurvetson financed Bhatia and Smith, who formed a company called Hotmail. With three full-time people and 14 part-time people who worked for stock options, they were able to develop and launch the service in July 1996. Within a month after launch, they had 100,000 subscribers. In December 1997, less than 18 months after launching the service, Hotmail had over 12 million subscribers and was acquired by Microsoft, reportedly for $400 million. The success of Hotmail is often attributed to its "first-mover advantage" and to the intrinsic "viral marketing" of e-mail. (Perhaps some of the students reading this book will be among the new entrepreneurs who conceive and develop first-mover Internet services with inherent viral marketing.)

Web e-mail continues to thrive, becoming more sophisticated and powerful every year. One of the most popular services today is Google's gmail, which offers gigabytes of free storage, advanced spam filtering and virus detection, e-mail encryption (using SSL), mail fetching from third-party e-mail services, and a search-oriented interface. Asynchronous messaging within social networks, such as Facebook, has also become popular in recent years.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 64 of 128

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.

2.4.1 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`), 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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 65 of 128

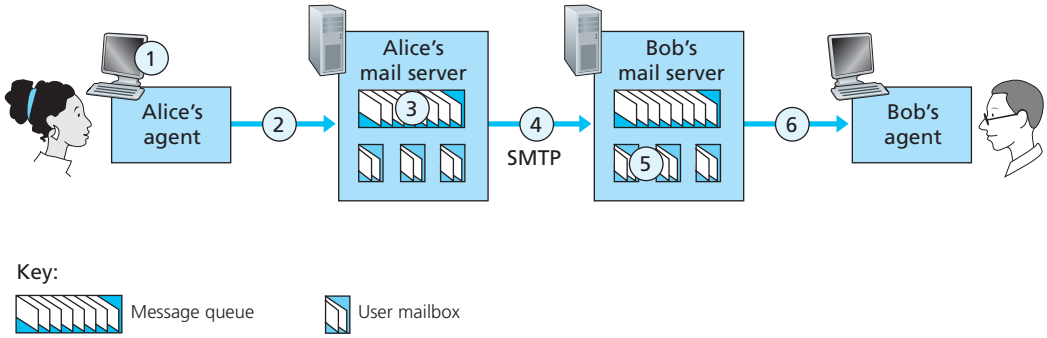


Figure 2.17 ♦ Alice sends a message to Bob

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 2.17.

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 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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 66 of 128

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` and the hostname of the server is `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.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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 67 of 128

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.

2.4.2 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. Internet mail places all of the message's objects into one message.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 68 of 128

2.4.3 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.

2.4.4 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,

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 69 of 128

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 2.18, 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 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

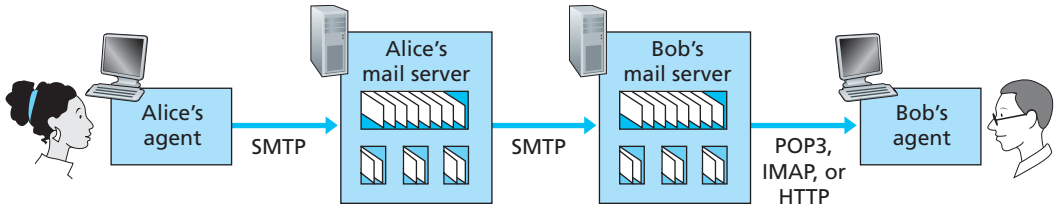


Figure 2.18 ♦ E-mail protocols and their communicating entities

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 70 of 128

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 2.18 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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 72 of 128

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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 73 of 128

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.

2.5 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 `cnn.com`, `www.yahoo.com`, `gaia.cs.umass.edu`, and `cis.poly.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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 74 of 128

2.5.1 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 2012]. The DNS protocol runs over UDP and uses port 53.

DNS is commonly employed by other application-layer protocols—including HTTP, SMTP, and FTP—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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 75 of 128

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.

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 Hotmail, Bob's e-mail address might be as simple as `bob@hotmail.com`. However, the hostname of the Hotmail mail server is more complicated and much less mnemonic than simply `hotmail.com` (for example, the canonical hostname might be something like `relay1.west-coast.hotmail.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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 76 of 128

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 Chapter 7).

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 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].

2.5.2 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).

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 77 of 128

- **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 2.19. 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 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

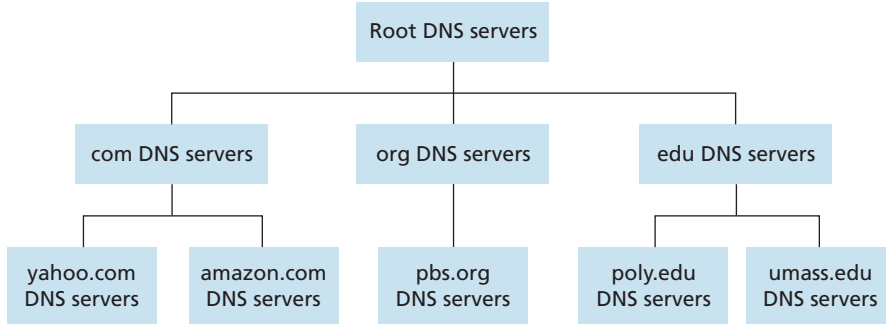


Figure 2.19 ♦ Portion of the hierarchy of DNS servers

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 78 of 128

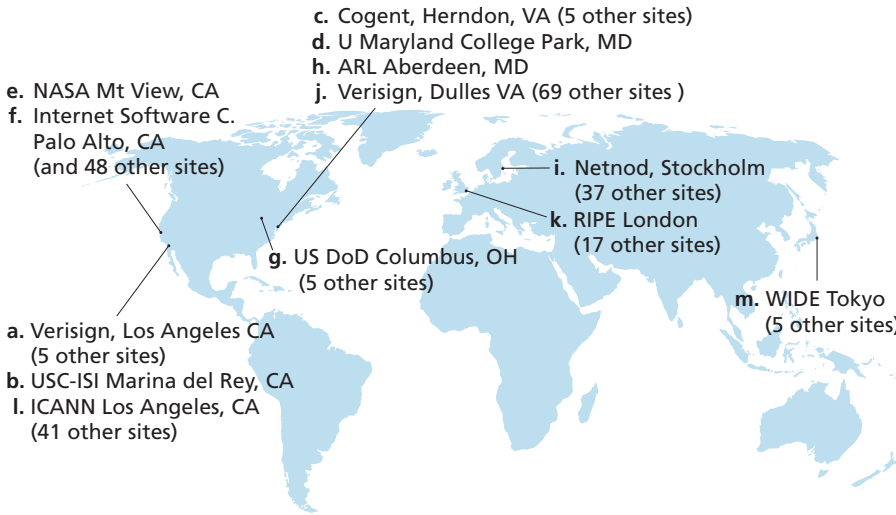


Figure 2.20 ♦ DNS root servers in 2012 (name, organization, location)

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.** In the Internet there are 13 root DNS servers (labeled A through M), most of which are located in North America. An October 2006 map of the root DNS servers is shown in Figure 2.20; a list of the current root DNS servers is available via [Root-servers 2012]. Although we have referred to each of the 13 root DNS servers as if it were a single server, each "server" is actually a network of replicated servers, for both security and reliability purposes. All together, there are 247 root servers as of fall 2011.
- **Top-level domain (TLD) servers.** These servers are responsible for 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. 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. See [IANA TLD 2012] for a list of all top-level domains.
- **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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 79 of 128

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 2.19. 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 university, an academic department, an employee's company, or a residential 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 `cis.poly.edu` desires the IP address of `gaia.cs.umass.edu`. Also suppose that Polytechnic's local DNS server is called `dns.poly.edu` and that an authoritative DNS server for `gaia.cs.umass.edu` is called `dns.umass.edu`. As shown in Figure 2.21, the host `cis.poly.edu` first sends a DNS query message to its local DNS server, `dns.poly.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 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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 80 of 128

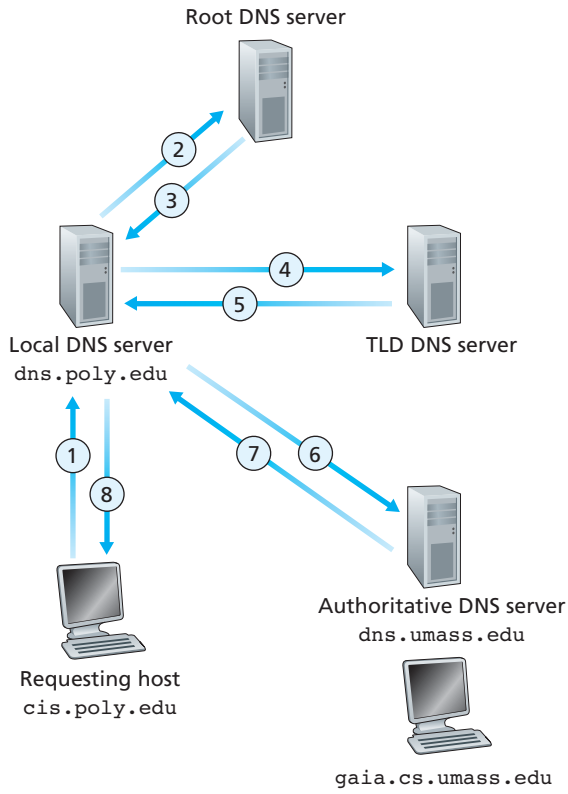


Figure 2.21 ♦ Interaction of the various DNS servers

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.poly.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.poly.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 2.21 makes use of both **recursive queries** and **iterative queries**. The query sent from `cis.poly.edu` to `dns.poly.edu` is a recursive query, since the query asks `dns.poly.edu` to obtain the mapping on its

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 81 of 128

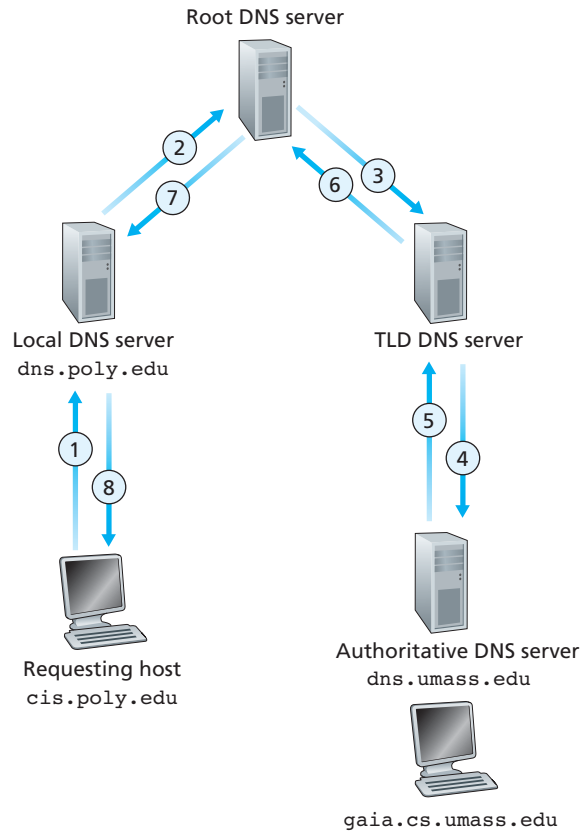


Figure 2.22 ♦ Recursive queries in DNS

behalf. But the subsequent three queries are iterative since all of the replies are directly returned to `dns.poly.edu`. In theory, any DNS query can be iterative or recursive. For example, Figure 2.22 shows a DNS query chain for which all of the queries are recursive. In practice, the queries typically follow the pattern in Figure 2.21: 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 ricocheting around

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 82 of 128

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 2.21, each time the local DNS server `dns.poly.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.poly.edu` queries `dns.poly.edu` for the IP address for the hostname `cnn.com`. Furthermore, suppose that a few hours later, another Polytechnic University host, say, `kiwi.poly.fr`, also queries `dns.poly.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 (this often happens).

2.5.3 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 [Abitz 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 2.23.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 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,

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 84 of 128

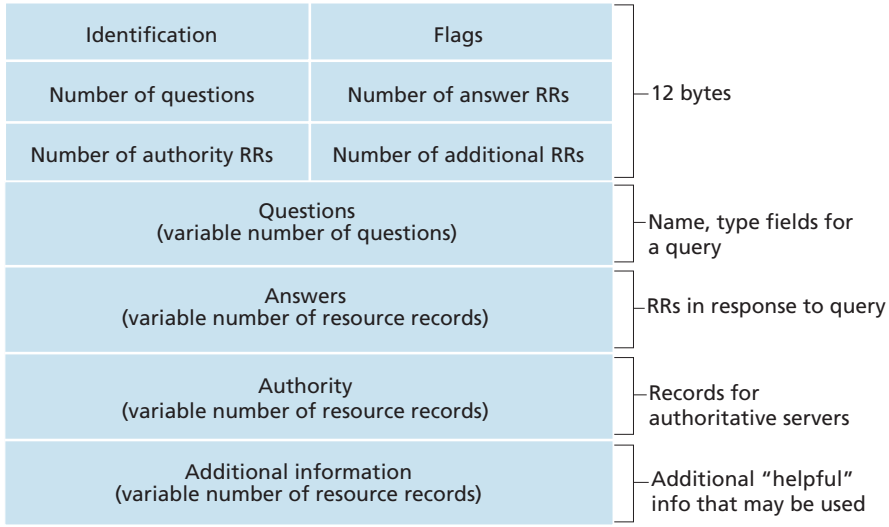


Figure 2.23 ♦ DNS message format

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**

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 85 of 128

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` 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`, `net`, and `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` 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`, `dns2.networkutopia.com`, `212.212.212.1`, 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`, the registrar would insert the following two resource records into the DNS system:

```
(networkutopia.com, dns1.networkutopia.com, NS)

(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` and the Type MX resource record for your mail server `mail.networkutopia.com` are entered into your authoritative DNS servers. (Until recently, the contents of each DNS server were configured statically,

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 86 of 128

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 servers. (ICMP messages are discussed in Chapter 4. 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].

Another important DNS attack is not an attack on the DNS service per se, but instead exploits the DNS infrastructure to launch a DDoS attack against a targeted host (for example, your university's mail server). In this attack, the attacker sends DNS queries to many authoritative DNS servers, with each query having the spoofed source address of the targeted host. The DNS servers then send their replies directly to the targeted host. If the queries can be crafted in such a way that a response is much larger

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 87 of 128

FOCUS ON SECURITY

(in bytes) than a query (so-called amplification), then the attacker can potentially overwhelm the target without having to generate much of its own traffic. Such reflection attacks exploiting DNS have had limited success to date [Mirkovic 2005].

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. There have been successful reflector attacks; however, these attacks can be (and are being) addressed by appropriate configuration of DNS servers.

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`. 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`. 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!

2.6 Peer-to-Peer Applications

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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 88 of 128

The peers are not owned by a service provider, but are instead desktops and laptops controlled by users.

In this section we'll examine two different applications that are particularly well-suited for P2P designs. The first is file distribution, where the application distributes a file from a single source to a large number of peers. File distribution is a nice place to start our investigation of P2P, as it clearly exposes the self-scalability of P2P architectures. As a specific example for file distribution, we'll describe the popular BitTorrent system. The second P2P application we'll examine is a database distributed over a large community of peers. For this application, we'll explore the concept of a Distributed Hash Table (DHT).

2.6.1 P2P File Distribution

We begin our foray into P2P by considering a very natural 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 2012, 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 2.24, 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 ith peer's access link by u_i, and the download rate of the ith 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 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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 89 of 128

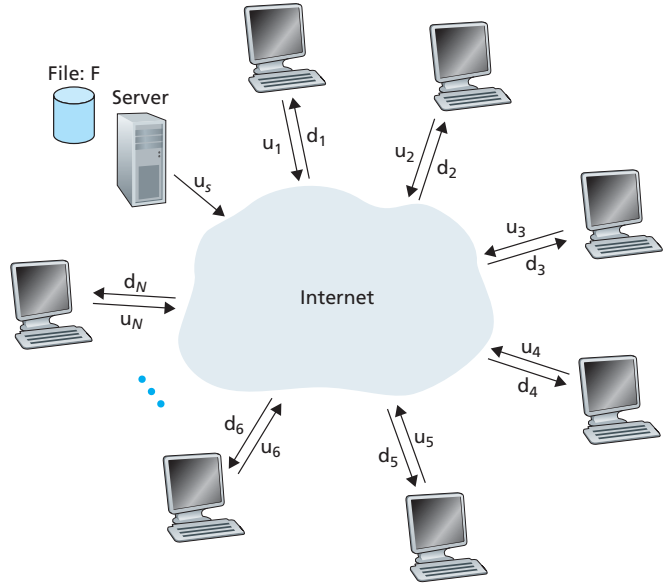


Figure 2.24 ♦ An illustrative file distribution problem

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, $d_{\min} = \min\{d_1, d_p, \ldots, d_N\}$. 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

$$D_{cs} \geq \max\left\{\frac{NF}{u_s}, \frac{F}{d_{min}}\right\}.$$

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 90 of 128

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,

$$D_{cs} = \max\left\{\frac{NF}{u_s}, \frac{F}{d_{min}}\right\} \tag{2.1}$$

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, $u_{\text{total}} = u_s + u_1 + \ldots + u_N$. 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_{total}. Thus, the minimum distribution time is also at least $NF/(u_s + u_1 + \ldots + u_N)$.

Putting these three observations together, we obtain the minimum distribution time for P2P, denoted by D_{P2P}.

$$D_{\text{P2P}} \geq \max\left\{\frac{F}{u_s}, \frac{F}{d_{min}}, \frac{NF}{u_s + \sum_{i=1}^{N} u_i}\right\} \tag{2.2}$$

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 91 of 128

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,

$$D_{\text{P2P}} = \max\left\{\frac{F}{u_s}, \frac{F}{d_{min}}, \frac{NF}{u_s + \sum_{i=1}^{N} u_i}\right\} \tag{2.3}$$

Figure 2.25 compares the minimum distribution time for the client-server and P2P architectures assuming that all peers have the same upload rate u. In Figure 2.25, we have set $F/u = 1$ hour, $u_s = 10u$, and $d_{\min} \geq u_s$. Thus, a peer can transmit the entire file in one hour, the server transmission rate is 10 times the peer upload rate, and (for simplicity) the peer download rates are set large enough so as not to have an effect. We see from Figure 2.25 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.

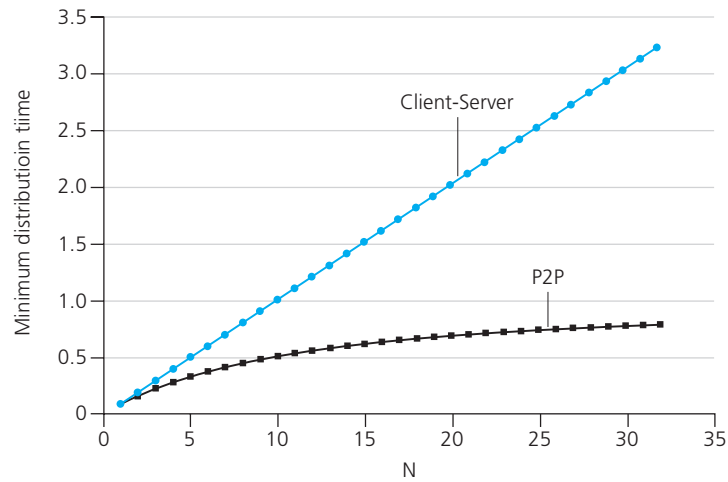


Figure 2.25 ♦ Distribution time for P2P and client-server architectures

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 92 of 128

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*. 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 2.26, 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 2.26, 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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 93 of 128

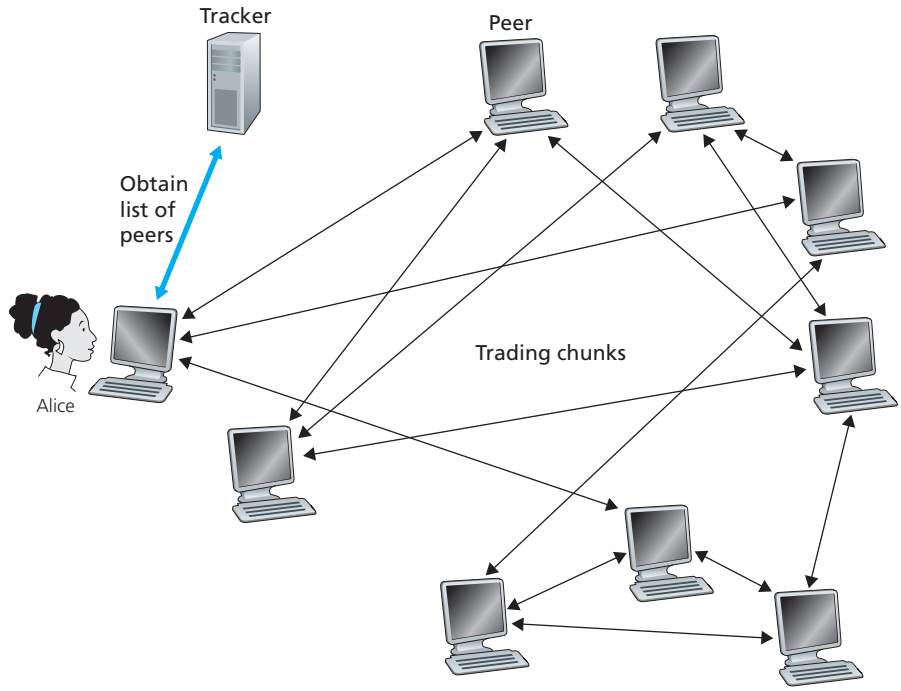


Figure 2.26 ♦ File distribution with BitTorrent

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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 94 of 128

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].

Interesting variants of the BitTorrent protocol are proposed [Guo 2005; Piatek 2007]. Also, many of the P2P live streaming applications, such as PPLive and ppstream, have been inspired by BitTorrent [Hei 2007].

2.6.2 Distributed Hash Tables (DHTs)

In this section, we will consider how to implement a simple database in a P2P network. Let's begin by describing a centralized version of this simple database, which will simply contain (key, value) pairs. For example, the keys could be social security numbers and the values could be the corresponding human names; in this case, an example key-value pair is (156-45-7081, Johnny Wu). Or the keys could be content names (e.g., names of movies, albums, and software), and the value could be the IP address at which the content is stored; in this case, an example key-value pair is (Led Zeppelin IV, 128.17.123.38). We query the database with a key. If there are one or more key-value pairs in the database that match the query key, the database returns the corresponding values. So, for example, if the database stores social security numbers and their corresponding human names, we can query with a specific social security number, and the database returns the name of the human who has that social security number. Or, if the database stores content names and their corresponding IP addresses, we can query with a specific content name, and the database returns the IP addresses that store the specific content.

Building such a database is straightforward with a client-server architecture that stores all the (key, value) pairs in one central server. So in this section, we'll instead consider how to build a distributed, P2P version of this database that will store the (key, value) pairs over millions of peers. In the P2P system, each peer will only hold a small subset of the totality of the (key, value) pairs. We'll allow any peer to query the distributed database with a particular key. The distributed database will then locate the peers that have the corresponding (key, value) pairs and return the key-value pairs to the querying peer. Any peer will also be allowed to insert new key-value pairs into the database. Such a distributed database is referred to as a **distributed hash table (DHT)**.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 95 of 128

VideoNote
Walking through distributed hash tables

Before describing how we can create a DHT, let's first describe a specific example DHT service in the context of P2P file sharing. In this case, a key is the content name and the value is the IP address of a peer that has a copy of the content. So, if Bob and Charlie each have a copy of the latest Linux distribution, then the DHT database will include the following two key-value pairs: (Linux, IP_{Bob}) and (Linux, $IP_{Charlie}$). More specifically, since the DHT database is distributed over the peers, some peer, say Dave, will be responsible for the key "Linux" and will have the corresponding key-value pairs. Now suppose Alice wants to obtain a copy of Linux. Clearly, she first needs to know which peers have a copy of Linux before she can begin to download it. To this end, she queries the DHT with "Linux" as the key. The DHT then determines that the peer Dave is responsible for the key "Linux." The DHT then contacts peer Dave, obtains from Dave the key-value pairs (Linux, IP_{Bob}) and (Linux, $IP_{Charlie}$), and passes them on to Alice. Alice can then download the latest Linux distribution from either IP_{Bob} or $IP_{Charlie}$.

Now let's return to the general problem of designing a DHT for general key-value pairs. One naïve approach to building a DHT is to randomly scatter the (key, value) pairs across all the peers and have each peer maintain a list of the IP addresses of all participating peers. In this design, the querying peer sends its query to all other peers, and the peers containing the (key, value) pairs that match the key can respond with their matching pairs. Such an approach is completely unscalable, of course, as it would require each peer to not only know about all other peers (possibly millions of such peers!) but even worse, have each query sent to *all* peers.

We now describe an elegant approach to designing a DHT. To this end, let's first assign an identifier to each peer, where each identifier is an integer in the range $[0, 2^n - 1]$ for some fixed n. Note that each such identifier can be expressed by an n-bit representation. Let's also require each key to be an integer in the same range. The astute reader may have observed that the example keys described a little earlier (social security numbers and content names) are not integers. To create integers out of such keys, we will use a hash function that maps each key (e.g., social security number) to an integer in the range $[0, 2^n - 1]$. A hash function is a many-to-one function for which two different inputs can have the same output (same integer), but the likelihood of the having the same output is extremely small. (Readers who are unfamiliar with hash functions may want to visit Chapter 7, in which hash functions are discussed in some detail.) The hash function is assumed to be available to all peers in the system. Henceforth, when we refer to the "key," we are referring to the hash of the original key. So, for example, if the original key is "Led Zeppelin IV," the key used in the DHT will be the integer that equals the hash of "Led Zeppelin IV." As you may have guessed, this is why "Hash" is used in the term "Distributed Hash Function."

Let's now consider the problem of storing the (key, value) pairs in the DHT. The central issue here is defining a rule for assigning keys to peers. Given that each peer has an integer identifier and that each key is also an integer in the same range, a natural approach is to assign each (key, value) pair to the peer whose identifier is the *closest* to the key. To implement such a scheme, we'll need to define what is meant by "closest," for which many conventions are possible. For convenience, let's define the

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 96 of 128

closest peer as the *closest successor of the key*. To gain some insight here, let's take a look at a specific example. Suppose $n = 4$ so that all the peer and key identifiers are in the range [0, 15]. Further suppose that there are eight peers in the system with identifiers 1, 3, 4, 5, 8, 10, 12, and 15. Finally, suppose we want to store the (key, value) pair (11, Johnny Wu) in one of the eight peers. But in which peer? Using our closest convention, since peer 12 is the closest successor for key 11, we therefore store the pair (11, Johnny Wu) in the peer 12. [To complete our definition of closest, if the key is exactly equal to one of the peer identifiers, we store the (key, value) pair in that matching peer; and if the key is larger than all the peer identifiers, we use a modulo-2^n convention, storing the (key, value) pair in the peer with the smallest identifier.]

Now suppose a peer, Alice, wants to insert a (key, value) pair into the DHT. Conceptually, this is straightforward: She first determines the peer whose identifier is closest to the key; she then sends a message to that peer, instructing it to store the (key, value) pair. But how does Alice determine the peer that is closest to the key? If Alice were to keep track of all the peers in the system (peer IDs and corresponding IP addresses), she could locally determine the closest peer. But such an approach requires e*ach* peer to keep track of *all* other peers in the DHT—which is completely impractical for a large-scale system with millions of peers.

Circular DHT

To address this problem of scale, let's now consider organizing the peers into a circle. In this circular arrangement, each peer only keeps track of its immediate successor and immediate predecessor (modulo 2^n). An example of such a circle is shown in Figure 2.27(a). In this example, n is again 4 and there are the same eight

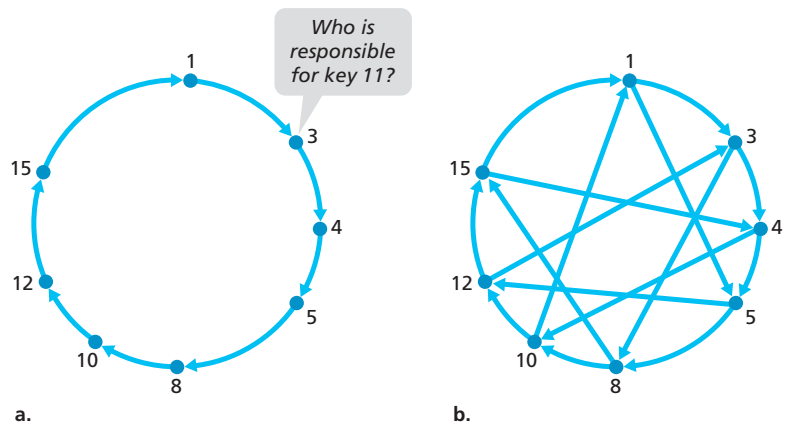


Figure 2.27 ♦ (a) A circular DHT. Peer 3 wants to determine who is responsible for key 11. (b) A circular DHT with shortcuts

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 97 of 128

peers from the previous example. Each peer is only aware of its immediate successor and predecessor; for example, peer 5 knows the IP address and identifier for peers 8 and 4 but does not necessarily know anything about any other peers that may be in the DHT. This circular arrangement of the peers is a special case of an **overlay network**. In an overlay network, the peers form an abstract logical network which resides above the "underlay" computer network consisting of physical links, routers, and hosts. The links in an overlay network are not physical links, but are simply virtual liaisons between pairs of peers. In the overlay in Figure 2.27(a), there are eight peers and eight overlay links; in the overlay in Figure 2.27(b) there are eight peers and 16 overlay links. A single overlay link typically uses many physical links and physical routers in the underlay network.

Using the circular overlay in Figure 2.27(a), now suppose that peer 3 wants to determine which peer in the DHT is responsible for key 11. Using the circular overlay, the origin peer (peer 3) creates a message saying "Who is responsible for key 11?" and sends this message clockwise around the circle. Whenever a peer receives such a message, because it knows the identifier of its successor and predecessor, it can determine whether it is responsible for (that is, closest to) the key in question. If a peer is not responsible for the key, it simply sends the message to its successor. So, for example, when peer 4 receives the message asking about key 11, it determines that it is not responsible for the key (because its successor is closer to the key), so it just passes the message along to peer 5. This process continues until the message arrives at peer 12, who determines that it is the closest peer to key 11. At this point, peer 12 can send a message back to the querying peer, peer 3, indicating that it is responsible for key 11.

The circular DHT provides a very elegant solution for reducing the amount of overlay information each peer must manage. In particular, each peer needs only to be aware of two peers, its immediate successor and its immediate predecessor. But this solution introduces yet a new problem. Although each peer is only aware of two neighboring peers, to find the node responsible for a key (in the worst case), all N nodes in the DHT will have to forward a message around the circle; $N/2$ messages are sent on average.

Thus, in designing a DHT, there is tradeoff between the number of neighbors each peer has to track and the number of messages that the DHT needs to send to resolve a single query. On one hand, if each peer tracks all other peers (mesh overlay), then only one message is sent per query, but each peer has to keep track of N peers. On the other hand, with a circular DHT, each peer is only aware of two peers, but $N/2$ messages are sent on average for each query. Fortunately, we can refine our designs of DHTs so that the number of neighbors per peer as well as the number of messages per query is kept to an acceptable size. One such refinement is to use the circular overlay as a foundation, but add "shortcuts" so that each peer not only keeps track of its immediate successor and predecessor, but also of a relatively small number of shortcut peers scattered about the circle. An example of such a circular DHT with some shortcuts is shown in Figure 2.27(b). Shortcuts are used to expedite the routing of query messages. Specifically, when a peer receives a message that is querying for a key, it forwards the

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 98 of 128

message to the neighbor (successor neighbor or one of the shortcut neighbors) which is the closet to the key. Thus, in Figure 2.27(b), when peer 4 receives the message asking about key 11, it determines that the closet peer to the key (among its neighbors) is its shortcut neighbor 10 and then forwards the message directly to peer 10. Clearly, shortcuts can significantly reduce the number of messages used to process a query.

The next natural question is "How many shortcut neighbors should a peer have, and which peers should be these shortcut neighbors? This question has received significant attention in the research community [Balakrishnan 2003; Androutsellis-Theotokis 2004]. Importantly, it has been shown that the DHT can be designed so that both the number of neighbors per peer as well as the number of messages per query is O(log N), where N is the number of peers. Such designs strike a satisfactory compromise between the extreme solutions of using mesh and circular overlay topologies.

Peer Churn

In P2P systems, a peer can come or go without warning. Thus, when designing a DHT, we also must be concerned about maintaining the DHT overlay in the presence of such peer churn. To get a big-picture understanding of how this could be accomplished, let's once again consider the circular DHT in Figure 2.27(a). To handle peer churn, we will now require each peer to track (that is, know the IP address of) its first and second successors; for example, peer 4 now tracks both peer 5 and peer 8. We also require each peer to periodically verify that its two successors are alive (for example, by periodically sending ping messages to them and asking for responses). Let's now consider how the DHT is maintained when a peer abruptly leaves. For example, suppose peer 5 in Figure 2.27(a) abruptly leaves. In this case, the two peers preceding the departed peer (4 and 3) learn that 5 has departed, since it no longer responds to ping messages. Peers 4 and 3 thus need to update their successor state information. Let's consider how peer 4 updates its state:

1. Peer 4 replaces its first successor (peer 5) with its second successor (peer 8).
2. Peer 4 then asks its new first successor (peer 8) for the identifier and IP address of its immediate successor (peer 10). Peer 4 then makes peer 10 its second successor.

In the homework problems, you will be asked to determine how peer 3 updates its overlay routing information.

Having briefly addressed what has to be done when a peer leaves, let's now consider what happens when a peer wants to join the DHT. Let's say a peer with identifier 13 wants to join the DHT, and at the time of joining, it only knows about peer 1's existence in the DHT. Peer 13 would first send peer 1 a message, saying "what will be 13's predecessor and successor?" This message gets forwarded through the DHT until it reaches peer 12, who realizes that it will be 13's predecessor and that its current successor, peer 15, will become 13's successor. Next, peer 12 sends this predecessor and successor information to peer 13. Peer 13 can now join

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 99 of 128

the DHT by making peer 15 its successor and by notifying peer 12 that it should change its immediate successor to 13.

DHTs have been finding widespread use in practice. For example, BitTorrent uses the Kademlia DHT to create a distributed tracker. In the BitTorrent, the key is the torrent identifier and the value is the IP addresses of all the peers currently participating in the torrent [Falkner 2007, Neglia 2007]. In this manner, by querying the DHT with a torrent identifier, a newly arriving BitTorrent peer can determine the peer that is responsible for the identifier (that is, for tracking the peers in the torrent). After having found that peer, the arriving peer can query it for a list of other peers in the torrent.

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 FTP protocol, described in Section 2.3 and explicitly defined in RFC 959; similarly, the server program could be an implementation of the FTP server protocol, also explicitly defined in RFC 959. 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 Firefox 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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 100 of 128

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. 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 Web site 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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 101 of 128

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 2.28 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 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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 102 of 128

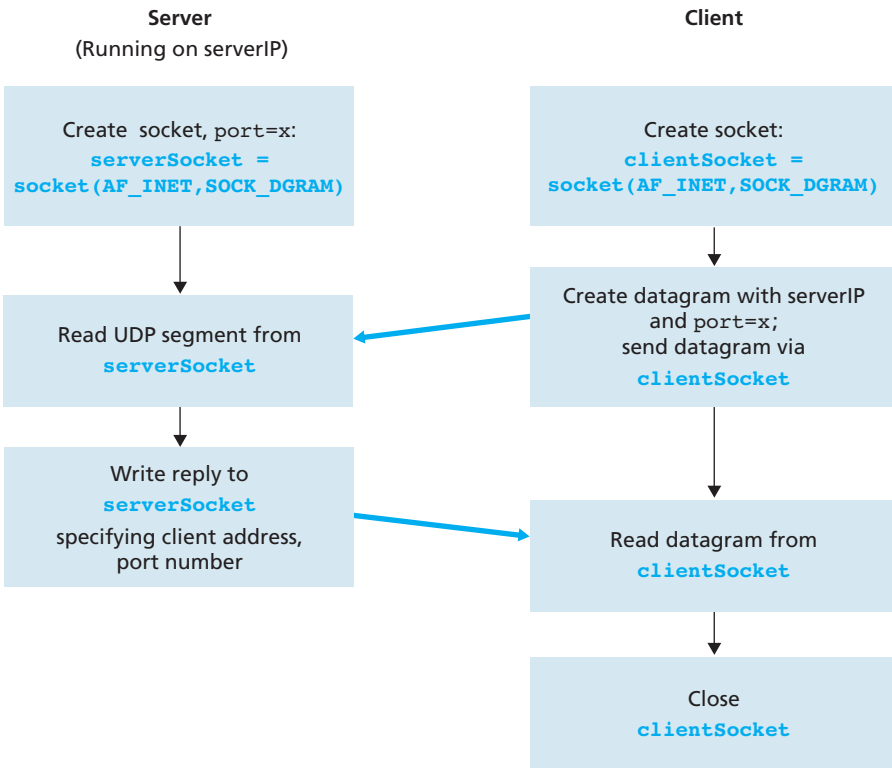


Figure 2.28 ♦ The client-server application using UDP

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(socket.AF_INET, socket.SOCK_DGRAM)
message = raw_input('Input lowercase sentence:')
clientSocket.sendto(message,(serverName, serverPort))
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print modifiedMessage
clientSocket.close()
```

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 103 of 128

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 string `serverName` to 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(socket.AF_INET, socket.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 data:" 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,(serverName, serverPort))
```

In the above line, 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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 104 of 128

```
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
```

This line prints out modifiedMessage on the user's display. 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 1:
        message, clientAddress = serverSocket.recvfrom(2048)
        modifiedMessage = message.upper()
        serverSocket.sendto(modifiedMessage, 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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 105 of 128

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.upper()
```

This line is the heart of our simple application. It takes the line sent by the client and uses the method `upper()` to capitalize it.

```
serverSocket.sendto(modifiedMessage, clientAddress)
```

This last line attaches the client's address (IP address and port number) to the capitalized message, 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 install and compile UDPClient.py in one host and UDPServer.py in 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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 106 of 128

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 2.29, 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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 107 of 128

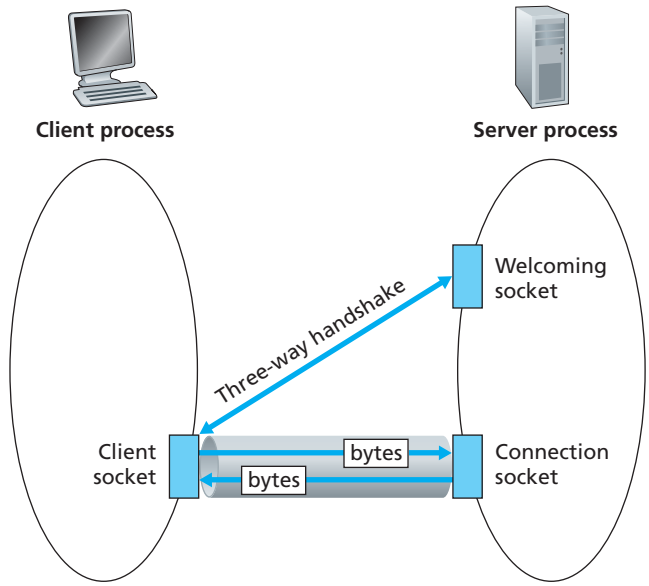


Figure 2.29 ♦ The `TCPServer` process has two sockets

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 2.30 highlights the main socket-related activity of the client and server that communicate over the TCP transport service.

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)
modifiedSentence = clientSocket.recv(1024)
print 'From Server:', modifiedSentence
clientSocket.close()
```

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 108 of 128

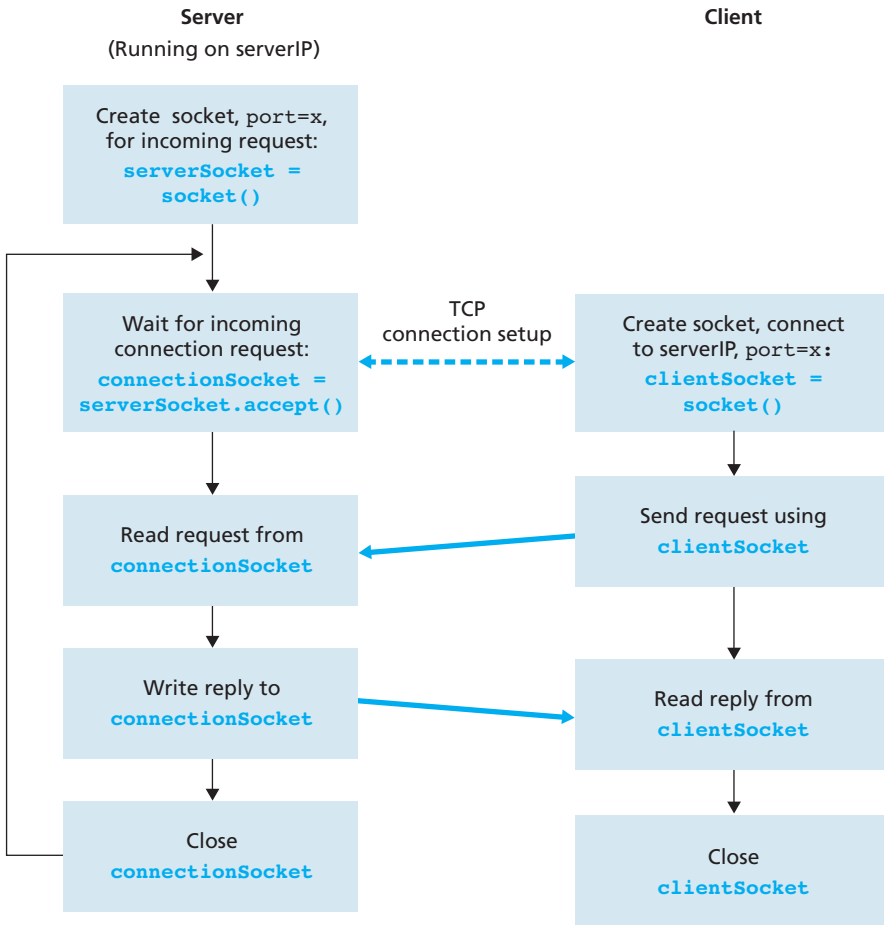


Figure 2.30 ♦ The client-server application using TCP

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 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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 109 of 128

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)
```

The above line sends the string `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).

TCPServer.py

Now let's take a look at the server program.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 110 of 128

```
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 1:
   connectionSocket, addr = serverSocket.accept()
   sentence = connectionSocket.recv(1024)
   capitalizedSentence = sentence.upper()
   connectionSocket.send(capitalizedSentence)
   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 not are not only guaranteed to arrive at the other side but also guaranteed arrive in order.

```
connectionSocket.close()
```

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 111 of 128

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 Chapters 2, 4, and 7. 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, FTP, 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 also learned about the increasingly prevalent P2P architecture and how it is used in many applications. 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, barebones 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, FTP, 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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 112 of 128

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?

SECTIONS 2.2–2.5

R10. What is meant by a handshaking protocol?

R11. Why do HTTP, FTP, 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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 113 of 128

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. Why is it said that FTP sends control information "out-of-band"?

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 emails, and examine the header of a message sent from a user with an .edu email 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 2.6

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?

R24. Consider a DHT with a mesh overlay topology (that is, every peer tracks all peers in the system). What are the advantages and disadvantages of such a design? What are the advantages and disadvantages of a circular DHT (with no shortcuts)?

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 114 of 128

R25. List at least four different applications that are naturally suitable for P2P architectures. (*Hint:* File distribution and instant messaging are two.)

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` and `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. Read RFC 959 for FTP. List all of the client commands that are supported by the RFC.

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 italized 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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 115 of 128

```
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 Dec2005 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-ST550A Spring 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?

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 116 of 128

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 $RTT_1, \ldots, RTT_n$. 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 Δ is the average time required to send an object over the access link and β 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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 117 of 128

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 email (modified from a real spam email). Assuming only the originator of this spam email is malacious and all other hosts are honest, identify the malacious host that has generated this spam email.

```
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
```

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 118 of 128

```
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 2.21, a DNS server higher 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, yahoo.com, or 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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 120 of 128

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 $u_s/N \le d_{\min}$. Specify a distribution scheme that has a distribution time of NF/u_s.

b. Suppose that $u_s/N \ge d_{\min}$. 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 $u_s \le (u_s + u_1 + \ldots + u_N)/N$. Specify a distribution scheme that has a distribution time of F/u_s.

b. Suppose that $u_s \ge (u_s + u_1 + \ldots + u_N)/N$. Specify a distribution scheme that has a distribution time of $NF/(u_s + u_1 + \ldots + u_N)$.

c. Conclude that the minimum distribution time is in general given by $\max\{F/u_s, NF/(u_s + u_1 + \ldots + u_N)\}$.

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. In the circular DHT example in Section 2.6.2, suppose that peer 3 learns that peer 5 has left. How does peer 3 update its successor state information? Which peer is now its first successor? Its second successor?

VideoNote
Walking through distributed hash tables

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 121 of 128

P28. In the circular DHT example in Section 2.6.2, suppose that a new peer 6 wants to join the DHT and peer 6 initially only knows peer 15's IP address. What steps are taken?

P29. Because an integer in $[0, 2^n - 1]$ can be expressed as an n-bit binary number in a DHT, each key can be expressed as $k = (k_0, k_1, \ldots, k_{n-1})$, and each peer identifier can be expressed $p = (p_0, p_1, \ldots, p_{n-1})$. Let's now define the XOR distance between a key k and peer p as

$$d(k, p) = \sum_{j=0}^{n-1} |k_j - p_j| 2^j$$

Describe how this metric can be used to assign (key, value) pairs to peers. (To learn about how to build an efficient DHT using this natural metric, see [Maymounkov 2002] in which the Kademlia DHT is described.)

P30. As DHTs are overlay networks, they may not necessarily match the underlay physical network well in the sense that two neighboring peers might be physically very far away; for example, one peer could be in Asia and its neighbor could be in North America. If we randomly and uniformly assign identifiers to newly joined peers, would this assignment scheme cause such a mismatch? Explain. And how would such a mismatch affect the DHT's performance?

P31. 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?

P32. 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?

P33. 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?

P34 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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 122 of 128

advantage and one disadvantage of byte-oriented API versus having the API explicitly recognize and preserve application-defined message boundaries?

P35. 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.

P36. Many BitTorrent clients use DHTs to create a distributed tracker. For these DHTs, what is the "key" and what is the "value"?

Socket Programming Assignments

The companion Web site 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 4. The sixth assignment employs multimedia protocols and is summarized at the end of Chapter 7. 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 http://www.awl.com/kurose-ross.

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 Web site, 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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 123 of 128

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 4). 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 Web site). 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 email 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 email 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 Web site provides the skeleton code for your client. Your job is to complete the code and test your client by sending email 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 Web site 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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 124 of 128

Wireshark Lab: HTTP

VideoNote
Using Wireshark to investigate the HTTP protocol

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, http://www.awl.com/kurose-ross.

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, http://www.awl.com/kurose-ross.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 125 of 128

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

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 126 of 128

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.

Intent IQ, LLC
Ex. 2005, IPR2024-00419
Page 127 of 128

This page intentionally left blank