
SECOND EDITION

COMPUTER NETWORKS

A Systems Approach

The Morgan Kaufmann Series in Networking

Series Editor, David Clark

Computer Networks: A Systems Approach, 2e

Larry L. Peterson and Bruce S. Davie

High-Performance Communication Networks, 2e

Jean Walrand and Pravin Varaiya

Internetworking Multimedia

Jon Crowcroft, Mark Handley, and Ian Wakeman

Understanding Networked Applications: A First Course

David G. Messerschmitt

Integrated Management of Networked Systems: Concepts, Architectures, and their Operational Application

Heinz-Gerd Hegering, Sebastian Abeck, and Bernhard Neumair

Virtual Private Networks: Making the Right Connection

Dennis Fowler

Networked Applications: A Guide to the New Computing Infrastructure

David G. Messerschmitt

Modern Cable Television Technology: Video, Voice, and Data Communications

Walter Ciciora, James Farmer, and David Large

Switching in IP Networks: IP Switching, Tag Switching, and Related Technologies

Bruce S. Davie, Paul Doolan, and Yakov Rekhter

Wide Area Network Design: Concepts and Tools for Optimization

Robert S. Cahn

Optical Networks: A Practical Perspective

Rajiv Ramaswami and Kumar Sivarajan

Practical Computer Network Analysis and Design

James D. McCabe

Frame Relay Applications: Business and Technology Case Studies

James P. Cavanagh

For a list of forthcoming titles, please visit our Web site at
<http://www.mkp.com/publish/mann/networking.htm>

SECOND EDITION

Larry L. Peterson & Bruce S. Davie

COMPUTER NETWORKS

A Systems Approach



MORGAN KAUFMANN PUBLISHERS

AN IMPRINT OF ACADEMIC PRESS

A Harcourt Science and Technology Company

SAN FRANCISCO SAN DIEGO NEW YORK BOSTON
LONDON SYDNEY TOKYO

Senior Editor Jennifer Mann
Director of Production and Manufacturing Yonie Overton
Production Editor Cheri Palmer
Editorial Assistant Karyn Johnson
Cover and Text Design Ross Carron Design
Cover Image Alain Choynet / ImageBank
Normandy Bridge at Night, Normandy, France
Composition/Illustration Windfall Software, using ZzT_EX
Copyeditor Ken DellaPenta
Proofreader Jennifer McClain
Indexer Steve Rath
Printer Courier Corporation

General Library System
University of Wisconsin - Madisc
728 State Street
Madison, WI 53706-1494
U.S.A.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances where Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

ACADEMIC PRESS

A Harcourt Science and Technology Company
525 B Street, Suite 1900, San Diego, CA 92101-4495, USA
<http://www.academicpress.com>

Academic Press

Harcourt Place, 32 Jamestown Road, London, NW1 7BY United Kingdom
<http://www.hbuk.co.uk/ap/>

Morgan Kaufmann Publishers

340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA
<http://www.mkp.com>

© 1996, 2000 by Academic Press

All rights reserved

Published 1996. Second Edition 2000

Printed in the United States of America

04 03 02 5 4

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

Library of Congress Cataloging-in-Publication Data

Peterson, Larry L.

Computer networks : a systems approach / Larry L. Peterson & Bruce S. Davie. — 2nd ed.

p. cm.

Includes bibliographical references.

ISBN 1-55860-514-2 (cloth). — ISBN 1-55860-577-0 (paper)

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

TK5105.5.P479 2000

004.6'5—dc21

99-36400

CIP

This book is printed on acid-free paper.

CONTENTS

Foreword	vii
Foreword to the First Edition	ix
Preface	xix
 1 Foundation	
Problem: Building a Network	2
1.1 Requirements	4
1.1.1 Connectivity	4
1.1.2 Cost-Effective Resource Sharing	8
1.1.3 Support for Common Services	12
1.1.4 Performance -105	20
1.2 Network Architecture	29
1.2.1 Layering and Protocols	30
1.2.2 OSI Architecture	36
1.2.3 Internet Architecture	38
1.3 Implementing Network Software	41
1.3.1 Application Programming Interface (Sockets)	42
1.3.2 Example Application	45
1.3.3 Protocol Implementation Issues	48
1.4 Summary	56
Open Issue: Ubiquitous Networking	57
Further Reading	58
Exercises	60
 2 Direct Link Networks	
Problem: Physically Connecting Hosts	68
2.1 Hardware Building Blocks	70
2.1.1 Nodes	70
2.1.2 Links	71
2.2 Encoding (NRZ, NRZI, Manchester, 4B/5B)	80

2.3 Framing	84
2.3.1 Byte-Oriented Protocols (BISYNC, PPP, DDCMP)	84
2.3.2 Bit-Oriented Protocols (HDLC)	87
2.3.3 Clock-Based Framing (SONET)	89
2.4 Error Detection	92
2.4.1 Two-Dimensional Parity	94
2.4.2 Internet Checksum Algorithm	94
2.4.3 Cyclic Redundancy Check	96
2.5 Reliable Transmission	101
2.5.1 Stop-and-Wait	103
2.5.2 Sliding Window	105
2.5.3 Concurrent Logical Channels	115
2.6 Ethernet (802.3)	116
2.6.1 Physical Properties	117
2.6.2 Access Protocol	119
2.6.3 Experience with Ethernet	124
2.7 Token Rings (802.5, FDDI)	125
2.7.1 Physical Properties	126
2.7.2 Token Ring Media Access Control	127
2.7.3 Token Ring Maintenance	129
2.7.4 Frame Format	131
2.7.5 FDDI	132
2.8 Wireless (802.11)	136
2.8.1 Physical Properties	136
2.8.2 Collision Avoidance	137
2.8.3 Distribution System	139
2.8.4 Frame Format	141
2.9 Network Adaptors	142
2.9.1 Components	142
2.9.2 View from the Host	143
~~2.9.3~~ Device Drivers	148
2.9.3 2.9.4 Memory Bottleneck	151
2.10 Summary	153
Open Issue: Does It Belong in Hardware?	154
Further Reading	155
Exercises	156

3 Packet Switching	
Problem: Not All Networks Are Directly Connected	170
3.1 Switching and Forwarding	172
3.1.1 Datagrams	174
3.1.2 Virtual Circuit Switching	176
3.1.3 Source Routing	183
~~3.1.4~~ Implementation and Performance	185
3.2 Bridges and LAN Switches	186
3.2.1 Learning Bridges	187
3.2.2 Spanning Tree Algorithm	190
3.2.3 Broadcast and Multicast	195
3.2.4 Limitations of Bridges	196
3.3 Cell Switching (ATM)	198
3.3.1 Cells	198
3.3.2 Segmentation and Reassembly	203
3.3.3 Virtual Paths	209
3.3.4 Physical Layers for ATM	210
3.3.5 ATM in the LAN	211
3.4 Switching Hardware	215
~~3.4.1~~ Design Goals	216
3.4.2 Ports and Fabrics	219
~~3.4.3~~ Crossbar Switches	221
~~3.4.4~~ Shared-Media Switches	226
~~3.4.5~~ Self-Routing Fabrics	227
3.5 Summary	232
Open Issue: The Future of ATM	233
Further Reading	234
Exercises	235
4 Internetworking	
Problem: There Is More Than One Network	246
4.1 Simple Internetworking (IP)	248
4.1.1 What Is an Internetwork?	248
4.1.2 Service Model	250
4.1.3 Global Addresses	262
4.1.4 Datagram Forwarding in IP	264
4.1.5 Address Translation (ARP)	268

4.1.6	Host Configuration (DHCP)	273
4.1.7	Error Reporting (ICMP)	276
4.1.8	Virtual Networks and Tunnels	277
4.2	Routing	280
4.2.1	Network as a Graph	282
4.2.2	Distance Vector (RIP)	284
4.2.3	Link State (OSPF)	292
4.2.4	Metrics	301
4.2.5	Routing for Mobile Hosts	304
4.3	Global Internet	309
4.3.1	Subnetting	311
4.3.2	Classless Routing (CIDR)	316
4.3.3	Interdomain Routing (BGP)	319
4.3.4	Routing Areas	326
4.3.5	IP version 6 (IPv6)	328
4.4	Multicast	340
4.4.1	Link-State Multicast	341
4.4.2	Distance-Vector Multicast	344
4.4.3	Protocol Independent Multicast (PIM)	346
4.5	Summary	350
	Open Issue: IP, ATM, and MPLS	351
	Further Reading	352
	Exercises	354
5	End-to-End Protocols	
	Problem: Getting Processes to Communicate	368
5.1	Simple Demultiplexer (UDP)	370
5.2	Reliable Byte Stream (TCP)	371
5.2.1	End-to-End Issues	373
5.2.2	Segment Format	375
5.2.3	Connection Establishment and Termination	378
5.2.4	Sliding Window Revisited	383
5.2.5	Adaptive Retransmission	389
5.2.6	Record Boundaries	393
5.2.7	TCP Extensions	394
5.2.8	Alternative Design Choices	395
5.3	Remote Procedure Call	397

5.3.1 Bulk Transfer (BLAST)	400
5.3.2 Request/Reply (CHAN)	407
5.3.3 Dispatcher (SELECT)	417
5.3.4 Putting It All Together (SunRPC, DCE)	420
5.4 Performance	426
5.4.1 Experimental Method	427
5.4.2 Latency	428
5.4.3 Throughput	430
5.5 Summary	430
Open Issue: Application-Specific Protocols	431
Further Reading	432
Exercises	433
 6 Congestion Control and Resource Allocation	
Problem: Allocating Resources	446
6.1 Issues in Resource Allocation	448
6.1.1 Network Model	448
6.1.2 Taxonomy	452
6.1.3 Evaluation Criteria	454
6.2 Queuing Disciplines	457
6.2.1 FIFO	458
6.2.2 Fair Queuing	459
6.3 TCP Congestion Control	464
6.3.1 Additive Increase/Multiplicative Decrease	465
6.3.2 Slow Start	468
6.3.3 Fast Retransmit and Fast Recovery	472
6.4 Congestion-Avoidance Mechanisms	475
6.4.1 DECbit	475
6.4.2 Random Early Detection (RED)	476
6.4.3 Source-Based Congestion Avoidance	482
6.5 Quality of Service	488
6.5.1 Application Requirements	489
6.5.2 Integrated Services (RSVP)	494
6.5.3 Differentiated Services	505
6.5.4 ATM Quality of Service	509
6.6 Summary	513
Open Issue: Inside versus Outside the Network	513

Further Reading	514
Exercises	516
7 End-to-End Data	
Problem: What Do We Do with the Data?	528
7.1 Presentation Formatting	530
7.1.1 Taxonomy	531
7.1.2 Examples (XDR, ASN.1, NDR)	535
<i>7.1.3 - XML ?</i>	
7.2 Data Compression	539
7.2.1 Lossless Compression Algorithms	541
7.2.2 Image Compression (JPEG)	543
7.2.3 Video Compression (MPEG)	548
7.2.4 Transmitting MPEG over a Network	553
7.2.5 Audio Compression (MP3)	557
7.3 Summary	559
Open Issue: Computer Networks Meet Consumer Electronics	559
Further Reading	560
Exercises	561
8 Network Security	
Problem: Securing the Data	568
8.1 Cryptographic Algorithms	570
8.1.1 Requirements	572
8.1.2 The Data Encryption Standard (DES)	573
8.1.3 RSA	578
8.1.4 Message Digest 5 (MD5)	581
8.1.5 Implementation and Performance	584
8.2 Security Mechanisms	584
8.2.1 Authentication Protocols	585
8.2.2 Message Integrity Protocols	588
8.2.3 Public Key Distribution (X.509)	591
8.3 Example Systems	594
8.3.1 Privacy Enhanced Mail (PEM)	594
<i>8.3.2 - SSL ?</i>	
8.3.2 Pretty Good Privacy (PGP)	599
8.3.3 Transport Layer Security (TLS, SSL, HTTPS)	601
8.3.4 IP Security (IPSEC)	605
8.4 Firewalls	608
8.4.1 Filter-Based Firewalls	609

8.4.2 Proxy-Based Firewalls	610
8.4.3 Limitations	612
8.5 Summary	613
Open Issue: Denial-of-Service Attacks	614
Further Reading	615
Exercises	615
 9 Applications	
Problem: Applications Need Their Own Protocols	622
9.1 Name Service (DNS)	624
9.1.1 Domain Hierarchy	625
9.1.2 Name Servers	626
9.1.3 Name Resolution	630
9.2 Traditional Applications	633
9.2.1 Electronic Mail (SMTP, MIME)	634
9.2.2 World Wide Web (HTTP)	640
9.2.3 Network Management (SNMP)	646
9.3 Multimedia Applications	649
9.3.1 Requirements	651
9.3.2 Real-time Transport Protocol (RTP)	653
9.3.3 Session Control and Call Control (H.323)	660
9.4 Summary	662
Open Issue: Programming the Network	663
Further Reading	664
Exercises	665
 Glossary	673
References	699
Index	717
About the Authors	749

9.4 - Overlay Networks?

PROBLEM

Applications Need Their Own Protocols

Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.

—Winston Churchill

We started this book by talking about application programs—everything from Web browsers to videoconferencing tools—that people want to run over computer networks. In the intervening chapters, we have developed, one layer at a time, the networking infrastructure needed to make such applications possible. We have now come full circle, back to network

applications. These applications are part network protocol (in the sense that they exchange messages with their peers on other machines) and part traditional application program (in the sense that they interact with the windowing system, the file system, and ultimately, the user).

This chapter explores some of the most popular network applications available today, with a focus on their protocols. What you will quickly recognize is that a protocol is a protocol, no matter what layer it runs at. Said another way, the best way to prepare yourself to write network applications is to first understand how to design good network protocols.

The first example we look at—a distributed name service—also happens to be the first application implemented on a network. Although it technically qualifies as a network application—it is, in effect, a distributed database built on top of the underlying transport protocols—it is not an application that users normally invoke explicitly. Nevertheless, it is an application that all other applications depend upon. This is because the name server is used to translate host names into host addresses; the existence of such an application allows the users of other applications to refer to remote hosts by name rather than by address. In other words, a name server is usually used by other applications, rather than by humans.

We then turn our attention to describing a variety of familiar, and not so familiar, network applications. These range from exchanging email and surfing the Web, to managing a set of network elements, to emerging audio

and video applications like vic and vat. This list is by no means exhaustive, but it does serve to illustrate the trick in designing application-level protocols, which is to augment the underlying transport services of TCP and UDP so as to provide the precise communication service required by the application.

9

Applications

9.1 Name Service (DNS)

Up to this point, we have been using addresses to identify hosts. While perfectly suited for processing by routers, addresses are not exactly user friendly. It is for this reason that a unique *name* is also typically assigned to each host in a network. This section describes how a naming service can be developed to map user-friendly names into router-friendly addresses. Such a service is often the first application program implemented in a network since it frees other applications to identify hosts by name rather than by address. Name services are sometimes called *middleware* because they fill a gap between applications and the underlying network.

Host names differ from host addresses in two important ways. First, they are usually of variable length and mnemonic, thereby making them easier for humans to remember. (In contrast, fixed-length numeric addresses are easier for routers to process.) Second, names typically contain no information that helps the network locate (route packets toward) the host. Addresses, in contrast, sometimes have routing information embedded in them; *flat* addresses (those not divisible into component parts) are the exception.

Before getting into the details of how hosts are named in a network, we first introduce some basic terminology. First, a *name space* defines the set of possible names. A name space can be either *flat* (names are not divisible into components), or it can be *hierarchical* (Unix file names are the obvious example). Second, the naming system maintains a collection of *bindings* of names to values. The value can be anything we want the naming system to return when presented with a name; in many cases it is an address. Finally, a *resolution mechanism* is a procedure that, when invoked with a name, returns the corresponding value. A *name server* is a specific implementation of a resolution mechanism that is available on a network and that can be queried by sending it a message.

Because of its large size, the Internet has a particularly well-developed naming system in place—the *domain name system* (DNS). We therefore use DNS as a framework for discussing the problem of naming hosts. Note that the Internet did not always use DNS. Early in its history, when there were only a few hundred hosts on the Internet, a central authority called the Network Information Center (NIC) maintained a flat table of name-to-address bindings; this table was called *hosts.txt*. Whenever a site wanted to add a new host to the Internet, the site administrator sent email to the NIC giving the new host's name/address pair. This information was manually entered into the table, the modified table was mailed out to the various sites every few days, and the system administrator at each site installed the table on every host at the site. Name resolution was then simply implemented by a procedure that looked up a host's name in the local copy of the table and returned the corresponding address.

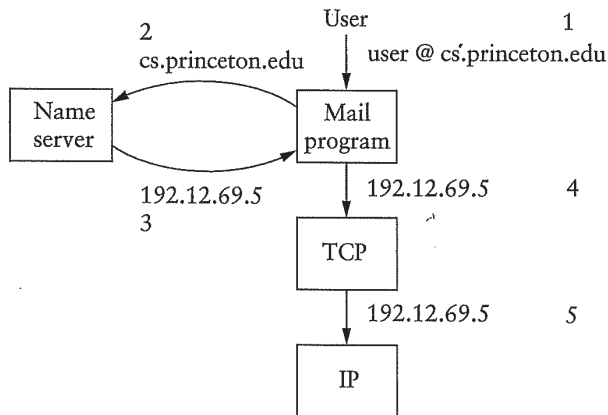


Figure 9.1 Names translated into addresses, where the numbers 1–5 show the sequence of steps in the process.

It should come as no surprise that the `hosts.txt` approach to naming did not work well as the number of hosts in the Internet started to grow. Therefore, in the mid-1980s, the domain naming system was put into place. DNS employs a hierarchical name space rather than a flat name space, and the “table” of bindings that implements this name space is partitioned into disjoint pieces and distributed throughout the Internet. These subtables are made available in name servers that can be queried over the network.

What happens in the Internet is that a user presents a host name to an application program (possibly embedded in a compound name such as an email address or URL), and this program engages the naming system to translate this name into a host address. The application then opens a connection to this host by presenting some transport protocol (e.g., TCP) with the host’s IP address. This situation is illustrated (in the case of sending email) in Figure 9.1.

9.1.1 Domain Hierarchy

DNS implements a hierarchical name space for Internet objects. Unlike Unix file names, which are processed from left to right with the naming components separated with slashes, DNS names are processed from right to left and use periods as the separator. (Although they are “processed” from right to left, humans still “read” domain names from left to right.) An example domain name for a host is `cicada.cs.princeton.edu`. Notice that we said domain names are used to name Internet “objects.” What we mean by this is that DNS is not strictly used to map host names

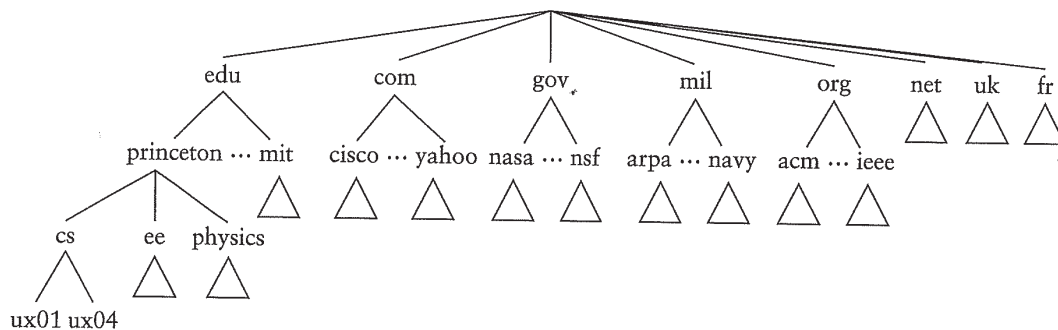


Figure 9.2 Example of a domain hierarchy.

into host addresses. It is more accurate to say that DNS maps domain names into values. For the time being, we assume that these values are IP addresses; we will come back to this issue later in this section.

Like the Unix file hierarchy, the DNS hierarchy can be visualized as a tree, where each node in the tree corresponds to a domain, and the leaves in the tree correspond to the hosts being named. Figure 9.2 gives an example of a domain hierarchy. Note that we should not assign any semantics to the term “domain” other than that it is simply a context in which additional names can be defined.

There was actually a substantial amount of discussion that took place when the domain name hierarchy was first being developed as to what conventions would govern the names that were to be handed out near the top of the hierarchy. Without going into that discussion in any detail, notice that the hierarchy is not very wide at the first level. There are domains for each country, plus the “big six” domains: edu, com, gov, mil, org, and net. These six domains are all based in the United States; the only domain names that don’t explicitly specify a country are those in the United States. Aside from this U.S. bias, you might notice a military bias in the hierarchy. This is easy to explain, since the development of DNS was originally funded by ARPA, the major research arm of the U.S. Department of Defense.

9.1.2 Name Servers

The complete domain name hierarchy exists only in the abstract. We now turn our attention to the question of how this hierarchy is actually implemented. The first step is to partition the hierarchy into subtrees called *zones*. For example, Figure 9.3 shows how the hierarchy given in Figure 9.2 might be divided into zones. Each zone can be thought of as corresponding to some administrative authority that is responsible for that portion of the hierarchy. For example, the top level of the hierarchy forms a

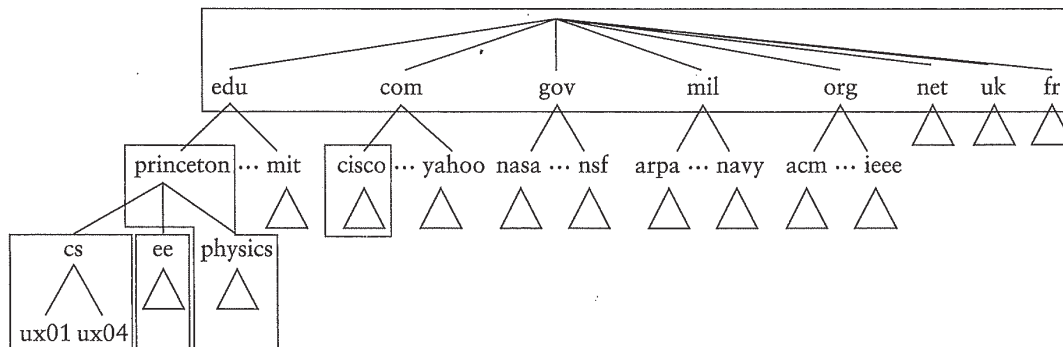


Figure 9.3 Domain hierarchy partitioned into zones.

zone that is managed by the NIC. Below this is a zone that corresponds to Princeton University. Within this zone, some departments do not want the responsibility of managing the hierarchy (and so they remain in the university-level zone), while others, like the Department of Computer Science, manage their own department-level zone.

The relevance of a zone is that it corresponds to the fundamental unit of implementation in DNS—the name server. Specifically, the information contained in each zone is implemented in two or more name servers. Each name server, in turn, is a program that can be accessed over the Internet. Clients send queries to name servers, and name servers respond with the requested information. Sometimes the response contains the final answer that the client wants, and sometimes the response contains a pointer to another server that the client should query next. Thus, from an implementation perspective, it is more accurate to think of DNS as being represented by a hierarchy of name servers rather than by a hierarchy of domains, as illustrated in Figure 9.4.

Note that each zone is implemented in two or more name servers for the sake of redundancy; that is, the information is still available even if one name server fails. On the flip side, a given name server is free to implement more than one zone.

Each name server implements the zone information as a collection of *resource records*. In essence, a resource record is a name-to-value binding, or more specifically, a 5-tuple that contains the following fields:

(Name, Value, Type, Class, TTL)

The Name and Value fields are exactly what you would expect, while the Type field specifies how the Value should be interpreted. For example, Type = A indicates that

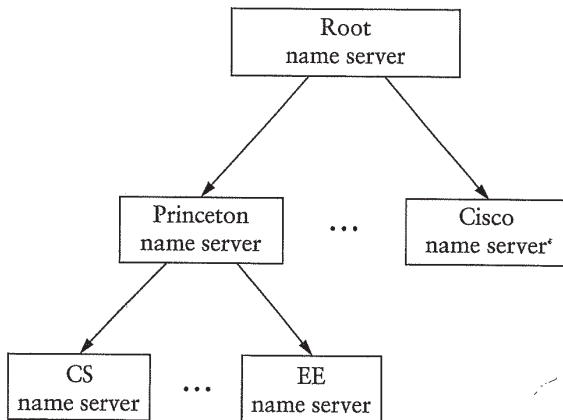


Figure 9.4 Hierarchy of name servers.

the Value is an IP address. Thus, A records implement the name-to-address mapping we have been assuming. Other record types include

- **NS:** The Value field gives the domain name for a host that is running a name server that knows how to resolve names within the specified domain.
- **CNAME:** The Value field gives the canonical name for a particular host; it is used to define aliases.
- **MX:** The Value field gives the domain name for a host that is running a mail server that accepts messages for the specified domain.

The Class field was included to allow entities other than the NIC to define useful record types. To date, the only widely used Class is the one used by the Internet; it is denoted IN. Finally, the TTL field shows how long this resource record is valid. It is used by servers that cache resource records from other servers; when the TTL expires, the server must evict the record from its cache.

To better understand how resource records represent the information in the domain hierarchy, consider the following examples drawn from the domain hierarchy given in Figure 9.2. To simplify the examples, we ignore the TTL field and we give the relevant information for only one of the name servers that implement each zone.

First, the root name server contains an NS record for each second-level server. It also has an A record that translates this name into the corresponding IP address. Taken together, these two records effectively implement a pointer from the root name server to each of the second-level servers.

```
( princeton.edu, cit.princeton.edu, NS, IN )  
( cit.princeton.edu, 128.196.128.233, A, IN )  
  
( cisco.com, ns.cisco.com, NS, IN )  
( ns.cisco.com, 128.96.32.20, A, IN )  
  
:
```

Next, the domain `princeton.edu` has a name server available on host `cit.princeton.edu` that contains the following records. Note that some of these records give the final answer (e.g., the address for host `saturn.physics.princeton.edu`), while others point to third-level name servers.

```
( cs.princeton.edu, gnat.cs.princeton.edu, NS, IN )  
( gnat.cs.princeton.edu, 192.12.69.5, A, IN )  
  
( ee.princeton.edu, helios.ee.princeton.edu, NS, IN )  
( helios.ee.princeton.edu, 128.196.28.166, A, IN )  
  
( jupiter.physics.princeton.edu, 128.196.4.1, A, IN )  
( saturn.physics.princeton.edu, 128.196.4.2, A, IN )  
  
( mars.physics.princeton.edu, 128.196.4.3, A, IN )  
( venus.physics.princeton.edu, 128.196.4.4, A, IN )  
  
:
```

Finally, a third-level name server, such as the one managed by domain `cs.princeton.edu`, contains `A` records for all of its hosts. It might also define a set of aliases (`CNAME` records) for each of those hosts. Aliases are sometimes just convenient (e.g., shorter) names for machines, but they can also be used to provide a level of indirection. For example, `www.cs.princeton.edu` is an alias for the host named `cicada.cs.princeton.edu`. This allows the site's Web server to move to another machine without affecting remote users; they simply continue to use the alias without regard for what machine currently runs the domain's Web server. The mail exchange (`MX`) records serve the same purpose for the email application—it allows an administrator to change which host receives mail on behalf of the domain without having to change everyone's email address.

```

< cs.princeton.edu, gnat.cs.princeton.edu, MX, IN >
< cicada.cs.princeton.edu, 192.12.69.60, A, IN >
< cic.cs.princeton.edu, cicada.cs.princeton.edu, CNAME, IN >
< gnat.cs.princeton.edu, 192.12.69.5, A, IN >
< gna.cs.princeton.edu, gnat.cs.princeton.edu, CNAME, IN >
< www.cs.princeton.edu, cicada.cs.princeton.edu, CNAME, IN >
< cicada.cs.princeton.edu,
    roach.cs.princeton.edu, CNAME, IN >
:

```

Note that although resource records can be defined for virtually any type of object, DNS is typically used to name hosts (including servers) and sites. It is not used to name individual people, or other objects like files or directories; other naming systems are typically used to identify such objects. For example, X.500 is an ISO naming system designed to make it easier to identify people. It allows you to name a person by giving a set of attributes: name, title, phone number, postal address, and so on. X.500 proved too cumbersome—and in some sense, was usurped by powerful search engines now available on the Web—but it did eventually evolve into LDAP (Lightweight Directory Access Protocol). LDAP is a subset of X.500 originally designed as a PC front end to X.500. Today it is gaining in popularity, mostly at the enterprise level, as a system for learning information about users.

9.1.3 Name Resolution

Given a hierarchy of name servers, we now consider the issue of how a client engages

Naming Conventions

Our description of DNS focuses on the underlying *mechanisms*, that is, how the hierarchy is partitioned over multiple servers and how the resolution process works. There is an equally interesting, but much less technical, issue of the *conventions* that are used to decide the names to use in the mechanism. For example, it is by convention that all U.S. universities are under the edu domain, while English universities are under the ac (academic) subdomain of the uk (United Kingdom) domain. In fact, the very existence of the uk domain, rather than a gb (Great Britain) domain, was a source of great controversy in the early days of DNS, since the latter does not include Northern Ireland.

The thing to understand about conventions is that they are sometimes defined without anyone making an explicit decision. For example, by convention a site hides the exact host that serves as its mail

these servers to resolve a domain name. To illustrate the basic idea, suppose the client wants to resolve the name `cicada.cs.princeton.edu` relative to the set of servers given in the previous subsection. The client first sends a query containing this name to the root server. The root server, unable to match the entire name, returns the best match it has—the NS record for `princeton.edu`. The server also returns all records that are related to this record, in this case, the A record for `cit.princeton.edu`. The client, having not received the answer it was after, next sends the same query to the name server at IP host

128.196.128.233. This server also cannot match the whole name, and so returns the NS and corresponding A records for the `cs.princeton.edu` domain. Finally, the client sends the same query as before to the server at IP host 192.12.69.5, and this time gets back the A record for `cicada.cs.princeton.edu`.

This example still leaves a couple of questions about the resolution process unanswered. The first question is, How did the client locate the root server in the first place, or said another way, How do you resolve the name of the server that knows how to resolve names? This is a fundamental problem in any naming system, and the answer is that the system has to be bootstrapped in some way. In this case, the name-to-address mapping for one or more root servers is well known, that is, published through some means outside the naming system itself.

In practice, however, not all clients know about the root servers. Instead, the client program running on each Internet host is initialized with the address of a *local* name server. For example, all the hosts in the Department of Computer Science at Princeton know about the server on `gnat.cs.princeton.edu`. This local name server, in turn, has resource records for one or more of the root servers, for example:

exchange behind the MX record. An alternative would have been to adopt the convention of sending mail to `user@mail.cs.princeton.edu`, much as we expect to find a site's public FTP directory at `ftp.cs.princeton.edu` and its WWW server at `www.cs.princeton.edu`.

Conventions also exist at the local level, where an organization names its machines according to some consistent set of rules. Given that the host names `venus`, `saturn`, and `mars` are among the most popular in the Internet, it's not too hard to figure out one common naming convention. Some host naming conventions are more imaginative, however. For example, one site named its machines `up`, `down`, `crashed`, `rebooting`, and so on, resulting in confusing statements like "rebooting has crashed" and "up is down." Of course, there are also less imaginative names, such as those who name their machines after the integers.

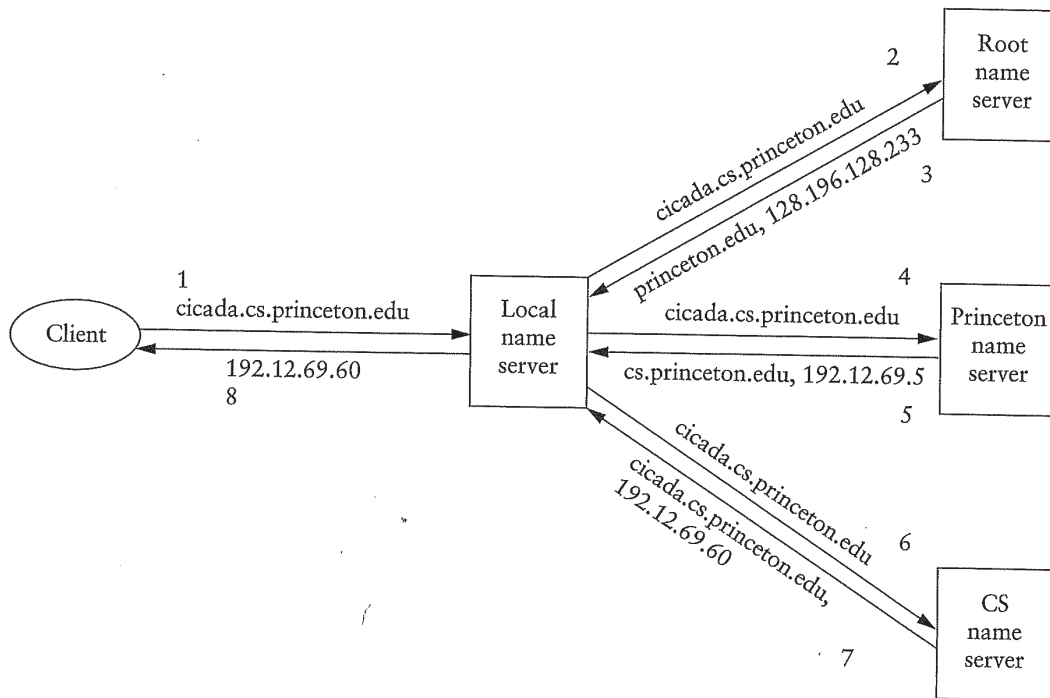


Figure 9.5 Name resolution in practice, where the numbers 1–8 show the sequence of steps in the process.

('root', venera.isi.edu, NS, IN)

(venera.isi.edu, 128.9.0.32, A, IN)

Thus, resolving a name actually involves a client querying the local server, which in turn acts as a client that queries the remote servers on the original client's behalf. This results in the client/server interactions illustrated in Figure 9.5. One advantage of this model is that all the hosts in the Internet do not have to be kept up-to-date on where the current root servers are located; only the servers have to know about the root. A second advantage is that the local server gets to see the answers that come back from queries that are posted by all the local clients. The local server caches these responses and is sometimes able to resolve future queries without having to go out over the network. The TTL field in the resource records returned by remote servers indicates how long each record can be safely cached.

The second question is how the system works when a user submits a partial name (e.g., `cicada`) rather than a complete domain name (e.g., `cicada.cs.princeton.edu`). The

answer is that the client program is configured with the local domain in which the host resides (e.g., cs.princeton.edu) and it appends this string to any simple names before sending out a query.

Just to make sure we are clear, we have now seen three different levels of identifiers—domain names, IP addresses, and physical network addresses—and the mapping of identifiers at one level into identifiers at another level happens at different points in the network architecture. First, users specify domain names when interacting with the application. Second, the application engages DNS to translate this name into an IP address; it is the IP address that is placed in each datagram, not the domain name. (As an aside, this translation process involves IP datagrams being sent over the Internet, but these datagrams are addressed to a host that runs a name server, not to the ultimate destination.) Third, IP does forwarding at each router, which often means that it maps one IP address into another; that is, it maps the ultimate destination's address into the address for the next hop router. Finally, IP engages ARP to translate the next hop IP address into the physical address for that machine; the next hop might be the ultimate destination or it might be an intermediate router. Frames sent over the physical network have these physical addresses in their headers.

9.2 Traditional Applications

The domain name system may be an essential Internet application, but it's one that users only indirectly interact with. We now turn our attention to those applications that are directly invoked by users, focusing on two of the most popular—the World Wide Web and email. We also look at network management, which although not so familiar to the average user, is the application of choice for system administrators. Like DNS, all three applications employ the request/reply paradigm—users send requests to servers, which then respond accordingly. We refer to these as “traditional” applications because they typify the sort of applications that have existed since the early days of computer networks. By contrast, the next section looks at a class of applications that have become feasible only relatively recently: streaming applications, that is, multimedia applications like video and audio.

Before taking a close look at each of these applications, there are three general points that we need to make. The first is that it is important to distinguish between application *programs* and application *protocols*. For example, the HyperText Transport Protocol (HTTP) is an application protocol that is used to retrieve Web pages from remote servers. There can be many different application programs—that is, Web clients like Internet Explorer, Mosaic, and Netscape—that provide users with a different look and feel, but all of them use the same HTTP protocol to communicate with Web servers over the Internet. This section focuses on three application protocols:

- SMTP: Simple Mail Transfer Protocol is used to exchange electronic mail.
- HTTP: HyperText Transport Protocol is used to communicate between Web browsers and Web servers.
- SNMP: Simple Network Management Protocol is used to query (and sometimes modify) the state of remote network nodes.

The second point is that since all of the application protocols described in this section follow the same request/reply communication pattern, we would expect that they are all built on top of an RPC transport protocol. This is not the case, however, as they are all implemented on top of either TCP or UDP. In effect, each protocol reinvents a simple RPC-like mechanism on top of one of the existing transport protocols. We say “simple” because each protocol is not designed to support arbitrary remote procedure calls, but is instead designed to send and respond to a specific set of request messages. In fact, it is no coincidence that two of the protocols have the word “Simple” in their name.

All three protocols have a companion protocol that specifies the format of the data that can be exchanged. This is one reason these protocols are relatively simple: Much of the complexity is managed in this companion document. For example, SMTP is a protocol for exchanging electronic mail messages, but RFC 822 (this specification has no other name) and MIME (Multipurpose Internet Mail Extensions) define the format of email messages. Similarly, HTTP is a protocol for fetching Web pages, but HTML (HyperText Markup Language) is a companion specification that defines the form of those pages. Finally, SNMP is a protocol for querying a network node, but MIB (management information base) defines the variables that can be queried.

9.2.1 Electronic Mail (SMTP, MIME)

Email is one of the oldest network applications. After all, what could be more natural than wanting to send a message to the user at the other end of a crosscountry link you just managed to get running? In fact, the pioneers of the ARPANET had not really envisioned email as a key application when the network was created—remote access to computing resources was the main design goal—but it turned out to be a surprisingly successful application. Out of this work evolved the Internet’s email system, which is now used by millions of people every day.

As with all the applications described in this section, the place to start in understanding how email works is to (1) distinguish the user interface (i.e., your mail reader) from the underlying message transfer protocol (in this case, SMTP), and (2) to distinguish between this transfer protocol and a companion protocol (RFC 822 and MIME)

that defines the format of the messages being exchanged. We start by looking at the message format.

Message Format

RFC 822 defines messages to have two parts: a *header* and a *body*. Both parts are represented in ASCII text. Originally, the body was assumed to be simple text. This is still the case, although RFC 822 has been augmented by MIME to allow the message body to carry all sorts of data. This data is still represented as ASCII text, but because it may be an encoded version of, say, a JPEG image, it's not necessarily readable by human users. More on MIME in a moment.

The message header is a series of <CRLF>-terminated lines. (<CRLF> stands for carriage-return + line-feed, which are a pair of ASCII control characters often used to indicate the end of a line of text.) The header is separated from the message body by a blank line. Each header line contains a type and value separated by a colon. Many of these header lines are familiar to users since they are asked to fill them out when they compose an email message. For example, the To: header identifies the message recipient, and the Subject: header says something about the purpose of the message. Other headers are filled in by the underlying mail delivery system. Examples include Date: (when the message was transmitted), From: (what user sent the message), and Received: (each mail server that handled this message). There are, of course, many other header lines; the interested reader is referred to RFC 822.

RFC 822 was extended in 1993 (and updated again in 1996) to allow email messages to carry many different types of data: audio, video, images, Word documents, and so on. MIME consists of three basic pieces. The first piece is a collection of header lines that augment the original set defined by RFC 822. These header lines describe, in various ways, the data being carried in the message body. They include MIME-Version: (the version of MIME being used), Content-Description: (a human-readable description of what's in the message, analogous to the Subject: line), Content-Type: (the type of data contained in the message), and Content-Transfer-Encoding (how the data in the message body is encoded).

The second piece is definitions for a set of content types (and subtypes). For example, MIME defines two different still image types, denoted image/gif and image/jpeg, each with the obvious meaning. As another example, text/plain refers to simple text you might find in a vanilla 822-style message, while text/richtext denotes a message that contains "marked up" text (e.g., text using special fonts, italics, etc.). As a third example, MIME defines an application type, where the subtypes correspond to the output of different application programs (e.g., application/postscript and application/msword).

MIME also defines a multipart type that says how a message carrying more than one data type is structured. This is like a programming language that defines both base types (e.g., integers and floats) and compound types (e.g., structures and arrays). One possible multipart subtype is mixed, which says that the message contains a set of independent data pieces in a specified order. Each piece then has its own header line that describes the type of that piece.

The third piece is a way to encode the various data types so they can be shipped in an ASCII email message. The problem is that for some data types (a JPEG image, for example), any given 8-bit byte in the image might contain one of 256 different values. Only a subset of these values are valid ASCII characters. It is important that email messages contain only ASCII, because they might pass through a number of intermediate systems (gateways, as described below) that assume all email is ASCII and would corrupt the message if it contained non-ASCII characters. To address this issue, MIME uses a straightforward encoding of binary data into the ASCII character set. The encoding is called *base64*. The idea is to map every three bytes of the original binary data into four ASCII characters. This is done by grouping the binary data into 24-bit units, and breaking each such unit into four 6-bit pieces. Each 6-bit piece maps onto one of 64 valid ASCII characters; for example, 0 maps onto *A*, 1 maps onto *B*, and so on. If you look at a message that has been encoded using the base64 encoding scheme, you'll notice only the 52 upper- and lowercase letters, the 10 digits 0 through 9, and the special characters *+* and */*. These are the first 64 values in the ASCII character set.

As one aside, so as to make reading mail as painless as possible for those of us that insist on using text-only mail readers, a MIME message that consists of regular text only can be encoded using 7-bit ASCII. There's also a readable encoding for mostly ASCII data.

Putting this all together, a message that contains some plain text, a JPEG image, and a PostScript file would look something like this:

```
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary="-----417CA6E2DE4ABCAFBC5"
From: Alice Smith <Alice@cisco.com>
To: Bob@cs.Princeton.edu
Subject: promised material
Date: Mon, 07 Sep 1998 19:45:19 -0400

-----417CA6E2DE4ABCAFBC5
Content-Type: text/plain; charset=us-ascii
```

```
Content-Transfer-Encoding: 7bit

Bob,

Here's the jpeg image and draft report I promised.

--Alice

-----417CA6E2DE4ABCAFB5
Content-Type: image/jpeg
Content-Transfer-Encoding: base64
:  unreadable encoding of a jpeg figure

-----417CA6E2DE4ABCAFB5
Content-Type: application/postscript; name="draft.ps"
Content-Transfer-Encoding: 7bit
:  readable encoding of a PostScript document
```

In this example, the Content-Type line in the message header says that this message contains various pieces, each denoted by a character string that does not appear in the data itself. Each piece then has its own Content-Type and Content-Transfer-Encoding lines.

Message Transfer

Finally, we get to SMTP—the protocol used to transfer messages from one host to another. To place SMTP in the right context, we need to identify the key players. First, users interact with a *mail reader* when they compose, file, search, and read their email. There are countless mail readers available, just like there are many Web browsers to choose from. In fact, most Web browsers now include a mail reader. Second, there is a *mail daemon* (or process) running on each host. You can think of this process as playing the role of a post office: Mail readers give the daemon messages they want to send to other users, the daemon uses SMTP running over TCP to transmit the message to a daemon running on another machine, and the daemon puts incoming messages into the user's mailbox (where that user's mail reader can later find it). Since SMTP is a protocol that anyone could implement, in theory there could be many different

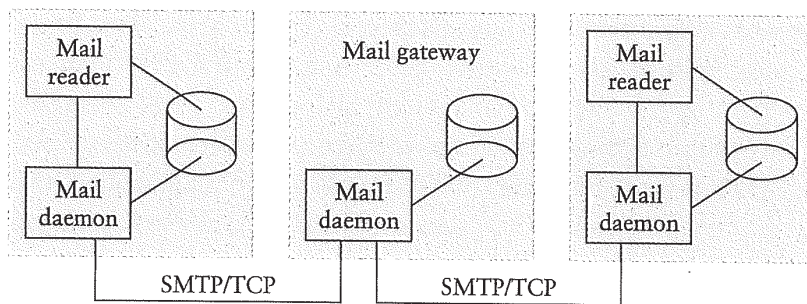


Figure 9.6 Sequence of mail gateways store and forward email messages.

implementations of the mail daemon. It turns out, though, that the mail daemon running on most hosts is derived from the `sendmail` program originally implemented on Berkeley Unix.

While it is certainly possible that the `sendmail` program on a sender's machine establishes an SMTP/TCP connection to the `sendmail` program on the recipient's machine, in many cases the mail traverses one or more *mail gateways* on its route from the sender's host to the receiver's host. Like the end hosts, these gateways also run a `sendmail` process. It's not an accident that these intermediate nodes are called "gateways" since their job is to store and forward email messages, much like an "IP gateway" (which we have referred to as a router) stores and forwards IP datagrams. The only difference is that a mail gateway typically buffers messages on disk and is willing to try retransmitting them to the next machine for several days, while an IP router buffers datagrams in memory and is only willing to retry transmitting them for a fraction of a second. Figure 9.6 illustrates a two-hop path from the sender to the receiver.

Why, you might ask, are mail gateways necessary? Why can't the sender's host send the message to the receiver's host? One reason is that the recipient does not want to include the specific host on which he or she reads email in his or her address. For example, mail delivered to `Bob@cs.princeton.edu` is first sent to a mail gateway in the CS Department at Princeton (that is, to the host named `cs.princeton.edu`), and then forwarded—involving a second SMTP/TCP connection—to the specific machine on which Bob happens to be reading his email today. The forwarding gateway maintains a database that maps users into the machine on which they currently want to receive their mail; the sender need not be aware of this specific name. (The list of Received: header lines in the message will help you trace the mail gateways that a given message traversed.) Another reason is that the recipient's machine may not always be up, in which case the mail gateway holds the message until it can be delivered. In this case,

it is possible that the recipient would use a mail fetching protocol, such as the Post Office Protocol (POP3), instead of SMTP to retrieve any messages that have arrived, but SMTP would still be used to get the message from the sender's to the recipient's gateway.

Independent of how many mail gateways are in the path, an independent SMTP connection is used between each host to move the message closer to the recipient. Each SMTP session involves a dialog between the two mail daemons, with one acting as the client and the other acting as the server. Multiple messages might be transferred between the two hosts during a single session. Since RFC 822 defines messages using ASCII as the base representation, it should come as no surprise to learn that SMTP is also ASCII based. This means it is possible for a human at a keyboard to pretend to be an SMTP client program.

SMTP is best understood by a simple example. The following is an exchange between sending host cs.princeton.edu and receiving host cisco.com. In this case, user Bob at Princeton is trying to send mail to users Alice and Tom at Cisco. The lines sent by cs.princeton.edu are shown in black and the lines sent by cisco.com are shown in green. Extra blank lines have been added to make the dialog more readable.

```
HELO cs.princeton.edu
250 Hello daemon@mail.cs.princeton.edu [128.12.169.24]

MAIL FROM:<Bob@cs.princeton.edu>
250 OK

RCPT TO:<Alice@cisco.com>
250 OK

RCPT TO:<Tom@cisco.com>
550 No such user here

DATA
354 Start mail input; end with <CRLF>.<CRLF>
Blah blah blah...
...etc. etc. etc.
<CRLF>.<CRLF>
250 OK

QUIT
221 Closing connection
```

As you can see, SMTP involves a sequence of exchanges between the client and the server. In each exchange, the client posts a command (e.g., HELO, MAIL, RCPT, DATA, QUIT) and the server responds with a code (e.g., 250, 550, 354, 221). The server also returns a human-readable explanation for the code (e.g., No such user here). In this particular example, the client first identifies itself to the server with the HELO command. It gives its domain name as an argument. The server verifies that this name corresponds to the IP address being used by the TCP connection; you'll notice the server states this IP address back to the client. The client then asks the server if it is willing to accept mail for two different users; the server responds by saying "yes" to one and "no" to the other. Then the client sends the message, which is terminated by a line with a single period (".") on it. Finally, the client terminates the connection.

There are, of course, many other commands and return codes. For example, the server can respond to a client's RCPT command with a 251 code, which indicates that the user does not have a mailbox on this host, but that the server promises to forward the message onto another mail daemon. In other words, the host is functioning as a mail gateway. As another example, the client can issue a VRFY operation to verify a user's email address, but without actually sending a message to the user.

The only other point of interest is the arguments to the MAIL and RCPT operations; for example, FROM:<Bob@cs.princeton.edu> and TO:<Alice@cisco.com>, respectively. These look a lot like 822 header fields, and in some sense, they are. What actually happens is that the mail daemon parses the message to extract the information it needs to run SMTP. The information it extracts is said to form an *envelope* for the message. The SMTP client uses this envelope to parameterize its exchange with the SMTP server. One historical note: The reason sendmail became so popular is that no one wanted to reimplement this message parsing function. While today's email addresses look pretty tame (e.g., Bob@cs.princeton.edu), this was not always the case. In the days before everyone was connected to the Internet, it was not uncommon to see email addresses of the form user%host@site!neighbor.

9.2.2 World Wide Web (HTTP)

The World Wide Web has been so successful and has made the Internet accessible to so many people that sometimes it seems to be synonymous with the Internet. One helpful way to think of the Web is as a set of cooperating clients and servers, all of whom speak the same language: HTTP. Most people are exposed to the Web through a graphical client program, or Web browser, like Netscape or Explorer. Figure 9.7 shows the Netscape browser in use, displaying a page of information from Princeton University.

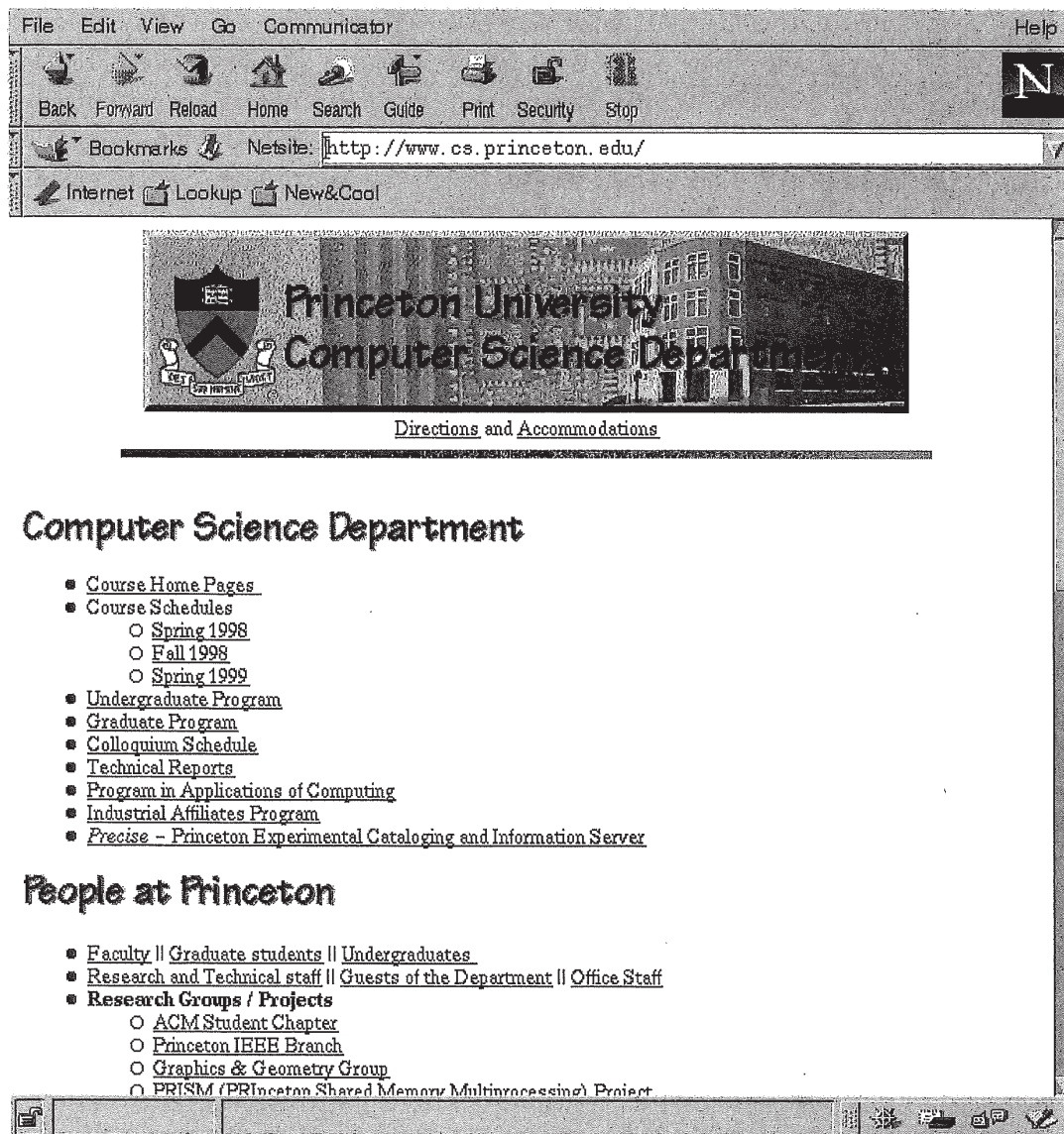


Figure 9.7 The Netscape Web browser.

Any Web browser has a function that allows the user to “open a URL.” URLs (uniform resource locators) provide information about the location of objects on the Web; they look like the following:

`http://www.cs.princeton.edu/index.html`

If you opened that particular URL, your Web browser would open a TCP connection to the Web server at a machine called `www.cs.princeton.edu` and immediately retrieve and display the file called `index.html`. Most files on the Web contain images and text, and some have audio and video clips. They also include URLs that point to other files, and your Web browser will have some way in which you can recognize URLs and ask the browser to open them. These embedded URLs are called hypertext links. When you ask your Web browser to open one of these embedded URLs (e.g., by pointing and clicking on it with a mouse), it will open a new connection and retrieve and display a new file. This is called “following a link.” It thus becomes very easy to hop from one machine to another around the network, following links to all sorts of information.

When you select to view a page, your browser (the client) fetches the page from the server using HTTP running over TCP. Like SMTP, HTTP is a text-oriented protocol. At its core, each HTTP message has the general form

```
START_LINE <CRLF>
MESSAGE_HEADER <CRLF>
<CRLF>
MESSAGE_BODY <CRLF>
```

where as before, `<CRLF>` stands for carriage-return-line-feed. The first line (`START_LINE`) indicates whether this is a request message or a response message. In effect, it identifies the “remote procedure” to be executed (in the case of a request message), or the “status” of the request (in the case of a response message). The next set of lines specify a collection of options and parameters that qualify the request or response. There are zero or more of these `MESSAGE_HEADER` lines—the set is terminated by a blank line—each of which looks like a header line in an email message. HTTP defines many possible header types, some of which pertain to request messages, some to response messages, and some to the data carried in the message body. Instead of giving the full set of possible header types, though, we just give a handful of representative examples. Finally, after the blank line comes the contents of the requested message (`MESSAGE_BODY`); this part of the message is typically empty for request messages.

Request Messages

The first line of an HTTP request message specifies three things: the operation to be performed, the Web page the operation should be performed on, and the version of HTTP being used. Although HTTP defines a wide assortment of possible request operations—including “write” operations that allow a Web page to be posted on a server—the two most common operations are `GET` (fetch the specified Web page) and `HEAD` (fetch status information about the specified Web page). The former is obviously

Operation	Description
OPTIONS	request information about available options
GET	retrieve document identified in URL
HEAD	retrieve metainformation about document identified in URL
POST	give information (e.g., annotation) to server
PUT	store document under specified URL
DELETE	delete specified URL
TRACE	loopback request message
CONNECT	for use by proxies

Table 9.1 HTTP request operations.

used when your browser wants to retrieve and display a Web page. The latter is used to test the validity of a hypertext link or to see if a particular page has been modified since the browser last fetched it. The full set of operations is summarized in Table 9.1.

For example, the `START_LINE`

```
GET http://www.cs.princeton.edu/ HTTP/1.1
```

says that the client wants the server on host `www.cs.princeton.edu` to return the page named `index.html`. This particular example uses an *absolute* URL. It is also possible to use a *relative* identifier and specify the host name in one of the `MESSAGE_HEADER` lines; for example,

```
GET / HTTP/1.1
Host: www.cs.princeton.edu
```

Here, `Host` is one of the possible `MESSAGE_HEADER` fields. One of the more interesting of these is `If-Modified-Since`, which gives the client a way to conditionally request a Web page—the server returns the page only if it has been modified since the time specified in that header line.

Response Messages

Like request messages, response messages begin with a single `START_LINE`. In this case, the line specifies the version of HTTP being used, a three-digit code indicating whether

Code	Type	Example Reasons
1xx	Informational	request received, continuing process
2xx	Success	action successfully received, understood, and accepted
3xx	Redirection	further action must be taken to complete the request
4xx	Client Error	request contains bad syntax or cannot be fulfilled
5xx	Server Error	server failed to fulfill an apparently valid request

Table 9.2 Five types of HTTP result codes.

or not the request was successful, and a text string giving the reason for the response. For example, the `START_LINE`

```
HTTP/1.1 202 Accepted
```

indicates that the server was able to satisfy the request, while

```
HTTP/1.1 404 Not Found
```

indicates that it was not able to satisfy the request because the page was not found. There are five general types of response codes, with the first digit of the code indicating its type. Table 9.2 summarizes the five types of codes.

Also similar to request messages, response messages can contain one or more `MESSAGE_HEADER` lines. These lines relay additional information back to the client. For example, the `Location` header line specifies that the requested URL is available at another location. Thus, if the Princeton CS Department Web page had moved from `http://www.cs.princeton.edu/index.html` to `http://www.princeton.edu/cs/index.html`, for example, then the server at the original address might respond with

```
HTTP/1.1 301 Moved Permanently
Location: http://www.princeton.edu/cs/index.html
```

In the common case, the response message will also carry the requested page. This page is an HTML document, but since it may carry nontextual data (e.g., a GIF image), it is encoded using MIME (see Section 9.2.1). Certain of the `MESSAGE_HEADER` lines give attributes of the page contents, including `Content-Length` (number of bytes in the contents), `Expires` (time at which the contents are considered stale), and `Last-Modified` (time at which the contents were last modified at the server).

TCP Connections

The original version of HTTP (1.0) established a separate TCP connection for each data item retrieved from the server. It's not too hard to see how this was a very inefficient mechanism: connection setup and teardown messages had to be exchanged between the client and server even if all the client wanted to do was verify that it had the most recent copy of a page. Thus, retrieving a page that included some text and a dozen icons or other small graphics would result in 13 separate TCP connections being established and closed.

The most important improvement in the latest version of HTTP (1.1) is to allow *persistent connections*—the client and server can exchange multiple request/response messages over the same TCP connection. Persistent connections have two advantages. First, they obviously eliminate the connection setup overhead, thereby reducing the load on the server, the load on the network caused by the additional TCP packets, and the delay perceived by the user. Second, because a client can send multiple request messages down a single TCP connection, TCP's congestion window mechanism is able to operate more efficiently. This is because it's not necessary to go through the slow start phase for each page.

Persistent connections do not come without a price, however. The problem is that neither the client nor server necessarily knows how long to keep a particular TCP connection open. This is especially critical on the server, which might be asked to keep connections opened on behalf of thousands of clients. The solution is that the server must time out and close a connection if it has received no requests on the connection for a period of time. Also, both the client and server must watch to see if the other side has elected to close the connection, and they must use that information as a signal that they should close their side of the connection as well. (Recall that both sides must close a TCP connection before it is fully terminated.)

Caching

One of the most active areas of research (and entrepreneurship) in the Internet today is how to effectively cache Web pages. Caching has many benefits. From the client's perspective, a page that can be retrieved from a nearby cache can be displayed much more quickly than if it has to be fetched from across the world. From the server's perspective, having a cache intercept and satisfy a request reduces the load on the server.

Caching can be implemented in many different places. For example, a user's browser can cache recently accessed pages, and simply display the cached copy if the user visits the same page again. As another example, a site can support a single sitewide cache. This allows users to take advantage of pages previously downloaded by other users. Closer to the middle of the Internet, ISPs can cache pages. Note that

in the second case, the users within the site most likely know what machine is caching pages on behalf of the site, and they configure their browsers to connect directly to the caching host. This node is sometimes called a *proxy*. In contrast, the sites that connect to the ISP are probably not aware that the ISP is caching pages. It simply happens to be the case that HTTP requests coming out of the various sites pass through a common ISP router. This router can peek inside the request message and look at the URL for the requested page. If it has the page in its cache, it returns it. If not, it forwards the request to the server, and watches for the response to fly by in the other direction. When it does, the router saves a copy in the hope that it can use it to satisfy a future request.

No matter where pages are cached, the ability to cache Web pages is important enough that HTTP has been designed to make the job easier. The trick is that the cache needs to make sure it is not responding with an out-of-date version of the page. For example, the server assigns an expiration date (the Expires header field) to each page it sends back to the client (or to a cache between the server and client). The cache remembers this date and knows that it need not reverify the page each time it is requested until after that expiration date has passed. After that time (or if that header field is not set) the cache can use the HEAD or conditional GET operation (GET with If-Modified-Since header line) to verify that it has the most recent copy of the page. More generally, there are a set of “cache directives” that must be obeyed by all caching mechanisms along the request/response chain. These directives specify whether or not a document can be cached, how long it can be cached, how fresh a document must be, and so on.

9.2.3 Network Management (SNMP)

A network is a complex system, both in terms of the number of nodes that are involved and in terms of the suite of protocols that can be running on any one node. Even if you restrict yourself to worrying about the nodes within a single administrative domain, such as a campus, there might be dozens of routers and hundreds—or even thousands—of hosts to keep track of. If you think about all the state that is maintained and manipulated on any one of those nodes—for example, address translation tables, routing tables, TCP connection state, and so on—then it is easy to become depressed about the prospect of having to manage all of this information.

It is easy to imagine wanting to know about the state of various protocols on different nodes. For example, you might want to monitor the number of IP datagram reassemblies that have been aborted, so as to determine if the timeout that garbage collects partially assembled datagrams needs to be adjusted. As another example, you might want to keep track of the load on various nodes (i.e., the number of packets

sent or received) so as to determine if new routers or links need to be added to the network. Of course, you also have to be on the watch for evidence of faulty hardware and misbehaving software.

What we have just described is the problem of network management, an issue that pervades the entire network architecture. Since the nodes we want to keep track of are distributed, our only real option is to use the network to manage the network. This means we need a protocol that allows us to read, and possibly write, various pieces of state information on different network nodes. The most widely used protocol for this purpose is the Simple Network Management Protocol (SNMP).

SNMP is essentially a specialized request/reply protocol that supports two kinds of request messages: GET and SET. The former is used to retrieve a piece of state from some node, and the latter is used to store a new piece of state in some node. (SNMP also supports a third operation—GET-NEXT—which we explain below.) The following discussion focuses on the GET operation, since it is the one most frequently used.

SNMP is used in the obvious way. A system administrator interacts with a client program that displays information about the network. This client program usually has a graphical interface. You can think of this interface as playing the same role as a Web browser. Whenever the administrator selects a certain piece of information that he or she wants to see, the client program uses SNMP to request that information from the node in question. (SNMP runs on top of UDP.) An SNMP server running on that node receives the request, locates the appropriate piece of information, and returns it to the client program, which then displays it to the user.

There is only one complication to this otherwise simple scenario: Exactly how does the client indicate which piece of information it wants to retrieve, and likewise, how does the server know which variable in memory to read to satisfy the request? The answer is that SNMP depends on a companion specification called the management information base (MIB). The MIB defines the specific pieces of information—the MIB *variables*—that you can retrieve from a network node.

The current version of MIB, called MIB-II, organizes variables into 10 different *groups*. You will recognize that most of the groups correspond to one of the protocols described in this book, and nearly all of the variables defined for each group should look familiar. For example:

- **System:** general parameters of the system (node) as a whole, including where the node is located, how long it has been up, and the system's name.
- **Interfaces:** information about all the network interfaces (adaptors) attached to this node, such as the physical address of each interface, how many packets have been sent and received on each interface.

- Address translation: information about the Address Resolution Protocol (ARP), and in particular, the contents of its address translation table.
- IP: variables related to IP, including its routing table, how many datagrams it has successfully forwarded, and statistics about datagram reassembly. Includes counts of how many times IP drops a datagram for one reason or another.
- TCP: information about TCP connections, such as the number of passive and active opens, the number of resets, the number of timeouts, default timeout settings, and so on. Per-connection information persists only as long as the connection exists.
- UDP: information about UDP traffic, including the total number of UDP datagrams that have been sent and received.

There are also groups for ICMP, EGP, and SNMP itself. The 10th group is used by different media.

Returning to the issue of the client stating exactly what information it wants to retrieve from a node, having a list of MIB variables is only half the battle. Two problems remain. First, we need a precise syntax for the client to use to state which of the MIB variables it wants to fetch. Second, we need a precise representation for the values returned by the server. Both problems are addressed using ASN.1.

Consider the second problem first. As we already saw in Chapter 7, ASN.1/BER defines a representation for different data types, such as integers. The MIB defines the type of each variable, and then it uses ASN.1/BER to encode the value contained in this variable as it is transmitted over the network. As far as the first problem is concerned, ASN.1 also defines an object identification scheme; this identification system is not described in Chapter 7. The MIB uses this identification system to assign a globally unique identifier to each MIB variable. These identifiers are given in a “dot” notation, not unlike domain names. For example, 1.3.6.1.2.1.4.3 is the unique ASN.1 identifier for the IP-related MIB variable `ipInReceives`; this variable counts the number of IP datagrams that have been received by this node. In this example, the 1.3.6.1.2.1 prefix identifies the MIB database (remember, ASN.1 object IDs are for all possible objects in the world), the 4 corresponds to the IP group, and the final 3 denotes the third variable in this group.

Thus, network management works as follows. The SNMP client puts the ASN.1 identifier for the MIB variable it wants to get into the request message, and it sends this message to the server. The server then maps this identifier into a local variable (i.e., into a memory location where the value for this variable is stored), retrieves the

current value held in this variable, and uses ASN.1/BER to encode the value it sends back to the client.

There is one final detail. Many of the MIB variables are either tables or structures. Such compound variables explain the reason for the SNMP GET-NEXT operation. This operation, when applied to a particular variable ID, returns the value of that variable plus the ID of the next variable, for example, the next item in the table or the next field in the structure. This aids the client in “walking through” the elements of a table or structure.

9.3 Multimedia Applications

Just like the traditional applications of the previous section, multimedia applications such as audio- and videoconferencing applications need application-layer protocols. Much of the initial experience in designing protocols for multimedia applications came from the “MBone tools”—applications such as *vat* and *vic* that were developed for use on the MBone, using IP multicast to enable multiparty conferencing. Initially, each application had its own protocol, but it became apparent that many multimedia applications have common requirements. This ultimately led to the development of a general-purpose protocol for use by multimedia applications called the Real-time Transport Protocol (RTP).

You might wonder why a protocol whose name identifies it as a “transport protocol” appears in a chapter on application-layer issues. The reason for this is that RTP contains a considerable amount of functionality that is specific to multimedia applications. Furthermore, it typically runs on top of one of the transport-layer protocols described in Chapter 5—UDP—which provides some of the application-independent functions you usually associate with a transport protocol. RTP is nevertheless called a transport protocol because it provides common end-to-end functions to a number of applications. (Most application-layer protocols, like HTTP and SMTP, for example, are specific to a single application.) A point to note here is the difficulty of fitting real-world protocols into a strict layerist model.

Before we look at RTP in detail, it will help to consider some of the applications that might use it. Multimedia applications are sometimes divided into two classes—*conferencing* applications and *streaming* applications. A popular example of the former class is *vat*, the audioconferencing tool that is often used over networks supporting IP multicast. The control panel for a typical *vat* conference is shown in Figure 9.8. Another conferencing application is *vic*, the videoconferencing tool discussed in Chapter 1 and illustrated in Figure 1.7.

Streaming applications typically deliver audio or video streams from a server to a client, and are typified by such commercial products as Real Audio. Because of the lack

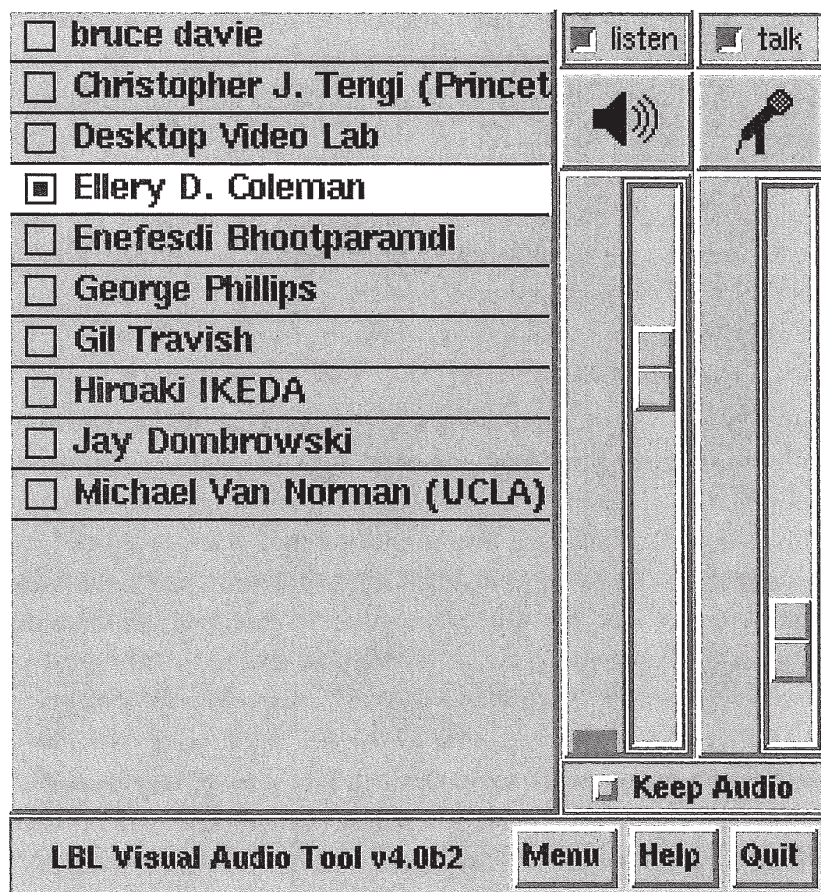


Figure 9.8 User interface of a vat audioconference.

of human interaction, such applications place somewhat different requirements on the underlying protocols. It should by now be apparent that designers of a transport protocol for multimedia applications face a real challenge in defining the requirements broadly enough to meet the needs of very different applications. They must also pay attention to the interactions among different applications, for example, the synchronization of audio and video streams. We will see how these concerns affected the design of RTP below.

Much of RTP actually derives from the application protocol that was originally embedded in vat. Newer versions of vat (and many other applications) run over RTP. RTP can run over many lower-layer protocols, but commonly runs over UDP. That leads to the protocol stack shown in Figure 9.9.

Application
RTP
UDP
IP
Subnet

Figure 9.9 Protocol stack for multimedia applications using RTP.

9.3.1 Requirements

The most basic requirement for a general-purpose multimedia protocol is that it allow similar applications to interoperate with each other. For example, it should be possible for two independently implemented audioconferencing applications to talk to each other. This immediately suggests that the applications had better use the same method of encoding and compressing voice; otherwise, the data sent by one party will be incomprehensible to the receiving party. Since there are quite a few different coding schemes for voice, each with its own trade-offs between quality, bandwidth requirements, and computational cost, it would probably be a bad idea to decree that only one such scheme can be used. Instead, our protocol should provide a way that a sender can tell a receiver which coding scheme it wants to use, and possibly negotiate until a scheme that is available to both parties is identified.

Just as with audio, there are many different video coding schemes. Thus, we see that the first common function that RTP can provide is the ability to communicate that choice of coding scheme. Note that this also serves to identify the type of application (e.g., audio or video); once we know what coding algorithm is being used, we know what type of data is being encoded as well.

Another important requirement for RTP is to enable the recipient of a data stream to determine the timing relationship among the received data. Recall from Section 6.5 that real-time applications need to place received data into a *playback buffer* to smooth out the jitter that may have been introduced into the data stream during transmission across the network. Thus, some sort of timestamping of the data will be necessary to enable the receiver to play it back at the appropriate time.

Related to the timing of a single media stream is the issue of synchronization of multiple media in a conference. The obvious example of this would be to synchronize an audio and video stream that are originating from the same sender. As we will see below, this is a slightly more complex problem than playback time determination for a single stream.

Another important function to be provided is an indication of packet loss. Note that an application with tight latency bounds generally cannot use a reliable transport like TCP because retransmission of data to correct for loss would probably cause the

packet to arrive too late to be useful. Thus, the application must be able to deal with missing packets, and the first step in dealing with them is noticing that they are in fact missing. As an example, a video application using MPEG encoding will need to take different actions when a packet is lost, depending on whether the packet came from an I frame, a B frame, or a P frame.

Since multimedia applications generally do not run over TCP, they also miss out on the congestion avoidance features of TCP (as described in Section 6.3). Yet many multimedia applications are capable of responding to congestion, for example, by changing the parameters of the coding algorithm to reduce the bandwidth consumed. Clearly, to make this work, the receiver needs to notify the sender that losses are occurring so that the sender can adjust its coding parameters.

Another common function across multimedia applications is the concept of frame boundary indication. A frame in this context is application-specific. For example, it may be helpful to notify a video application that a certain set of packets correspond to a single frame. In an audio application it is helpful to mark the beginning of a “talkspurt,” which is a collection of sounds or words followed by silence. The receiver can then identify the silences between talkspurts and use them as opportunities to move the playback point. This follows the observation that slight shortening or lengthening of the spaces between words are not perceptible to users, whereas shortening or lengthening the words themselves is both perceptible and annoying.

A final function that we might want to put into the protocol is some way of identifying senders that is more user-friendly than an IP address. Tools such as `vat` and `vic` can display strings such as `Joe User (user@domain.com)` on their control panels, and thus the application protocol should support the association of such a string with a data stream.

In addition to the functionality that is required from our protocol, we note an additional requirement: It should make reasonably efficient use of bandwidth. Put another way, we don’t want to introduce a lot of extra bits that need to be sent with every packet in the form of a long header. The reason for this is that audio packets, which are one of the most common types of multimedia data, tend to be small, so as to reduce the time it takes to fill them with samples. Long audio packets would mean high latency due to packetization, which has a negative effect on the perceived quality of conversations. (Recall that this was one of the factors in choosing the length of ATM cells.) Since the data packets themselves are short, a large header would mean that a relatively large amount of link bandwidth would be used by headers, thus reducing the available capacity for “useful” data. We will see several aspects of the design of RTP that have been influenced by the necessity of keeping the header short.

9.3.2 Real-time Transport Protocol (RTP)

Now that we have seen the rather long list of requirements for our application-layer protocol for multimedia, we turn to the details of the protocol that has been specified to meet those requirements. This protocol, RTP, was developed in the IETF and is in widespread use. The RTP standard actually defines a pair of protocols, RTP and the Real-time Transport Control Protocol (RTCP). The former is used for the exchange of multimedia data, while the latter is used to periodically send control information associated with a certain data flow. When running over UDP, the RTP data stream and the associated RTCP control stream use consecutive transport-layer ports. The RTP data uses an even port number and the RTCP control information uses the next higher (odd) port number.

Because RTP is designed to support a wide variety of applications, it provides a flexible mechanism by which new applications can be developed without repeatedly revising the RTP protocol itself. For each class of application (e.g., audio), RTP defines a *profile* and one or more *formats*. The profile provides a range of information that ensures a common understanding of the fields in the RTP header for that application class, as will be apparent when we examine the header in detail. The format specification explains how the data that follows the RTP header is to be interpreted. For example, the RTP header might just be followed by a sequence of bytes, each of which represents a single audio sample taken a defined interval after the previous one. Alternatively, the format of the data might be much more complex; an MPEG-encoded video stream, for example, would need to have a good deal of structure to represent all the different types of information.

▶ The design of RTP embodies an architectural principle known as *Application Level Framing* (ALF). This principle was put forward by Clark and Tennenhouse in 1990 as a new way to design protocols for emerging multimedia applications. They recognized that these new applications were unlikely to be well served by existing protocols such as TCP, and that furthermore they might not be well served by any sort of “one-size-fits-all” protocol. At the heart of this principle is the belief that an application understands its own needs best. For example, an MPEG video application knows how best to recover from lost frames, and how to react differently if an I frame or a B frame is lost. The same application also understands best how to segment the data for transmission—for example, it’s better to send the data from different frames in different datagrams, so that a lost packet only corrupts a single frame, not two. It is for this reason that RTP leaves so many of the protocol details to the profile and format documents that are specific to an application.

V = 2	P	X	CC	M	PT	Sequence number
Timestamp						
Synchronization source (SSRC) identifier						
Contributing source (CSRC) identifiers						
⋮						
Extension header						
RTP payload						

Figure 9.10 RTP header format.

Header Format

Figure 9.10 shows the header format used by RTP. The first 12 bytes are always present, whereas the contributing source identifiers are only used in certain circumstances. After this header there may be optional header extensions, as described below. Finally, the header is followed by the RTP payload, the format of which is determined by the application. The intention of this header is that it contain only the fields that are likely to be used by many different applications, since anything that is very specific to a single application would be more efficiently carried in the RTP payload for that application only.

The first two bits are a version identifier, which contains the value 2 in the RTP version deployed at the time of writing. You might think that the designers of the protocol were rather bold to think that 2 bits would be enough to contain all future versions of RTP, but recall that bits are at a premium in the RTP header. Furthermore, the use of profiles for different applications makes it less likely that many revisions to the base RTP protocol would be needed. In any case, if it turns out that another version of RTP is needed beyond version 2, it would be possible to consider a change to the header format so that more than one future version would be possible. For example, a new RTP header with the value 3 in the version field could have a “subversion” field somewhere else in the header.

The next bit is the “padding” (P) bit, which is set in circumstances in which the RTP payload has been padded for some reason. RTP data might be padded to fill up a block of a certain size as required by an encryption algorithm, for example. In such a case, the complete length of the RTP header, data, and padding would be conveyed by the lower-layer protocol header (e.g., the UDP header) and the last byte of the padding would contain a count of how many bytes should be ignored. This is illustrated in Figure 9.11. Note that this approach to padding removes any need for a length field

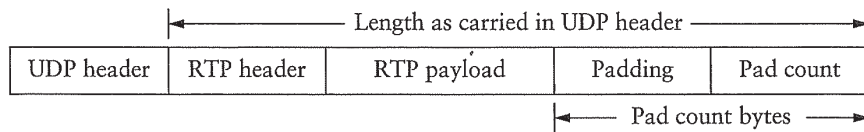


Figure 9.11 Padding of an RTP packet.

in the RTP header (thus serving the goal of keeping the header short); in the common case of no padding, the length is deduced from the lower-layer protocol.

The extension (X) bit is used to indicate the presence of an extension header, which would be defined for a specific application and follow the main header. Such headers are rarely used, since it is generally possible to define a payload-specific header as part of the payload format definition for a particular application.

The X bit is followed by a 4-bit field that counts the number of “contributing sources,” if any are included in the header. Contributing sources are discussed below.

We noted above the frequent need for some sort of frame indication; this is provided by the marker bit, which could be set at the beginning of a talkspurt, for example. The 7-bit payload type field follows; it indicates what type of multimedia data is carried in this packet. One possible use of this field would be to enable an application to switch from one coding scheme to another based on information about resource availability in the network or feedback on application quality. The exact usage of the marker bit and the payload type is determined by the application profile.

Note that the payload type is generally not used as a demultiplexing key to direct data to different applications (or to different streams within a single application, for example, the audio and video stream for a videoconference). This is because such demultiplexing is typically provided at a lower layer (e.g., by UDP, as described in Section 5.1). Thus, two media streams using RTP would typically use different UDP port numbers.

The sequence number is used to enable the receiver of an RTP stream to detect missing and misordered packets. The sender simply increments the value by one for each transmitted packet. Note that RTP does not do anything when it detects a lost packet, in contrast to TCP, which both corrects for the loss (by retransmission) and interprets the loss as a congestion indication (which may cause it to reduce its window size). Rather, it is left to the application to decide what to do when a packet is lost because this decision is likely to be highly application-dependent. For example, a video application might decide that the best thing to do when a packet is lost is to replay the last frame that was correctly received. Some applications might also decide to modify their coding algorithms to reduce bandwidth needs in response to loss, but this is not

a function of RTP. It would not be sensible for RTP to decide that the sending rate should be reduced, as this might make the application useless.

The function of the timestamp field is to enable the receiver to play back samples at the appropriate intervals and to enable different media streams to be synchronized. Because different applications may require different granularities of timing, RTP itself does not specify the units in which time is measured. Instead, the timestamp is just a counter of “ticks,” where the time between ticks is dependent on the encoding in use. For example, an audio application that samples data once every 125 μ s could use that value as its clock resolution. The clock granularity is one of the details that is specified in the RTP profile or payload format for an application.

The timestamp value in the packet is a number representing the time at which the *first* sample in the packet was generated. The timestamp is not a reflection of the time of day; only the differences between timestamps are relevant. For example, if the sampling interval is 125 μ s and the first sample in packet $n + 1$ was generated 10 ms after the first sample in packet n , then the number of sampling instants between these two samples is

$$\begin{aligned}\text{TimeBetweenPackets} \div \text{TimePerSample} &= (10 \times 10^{-3}) \div (125 \times 10^{-6}) \\ &= 80\end{aligned}$$

Assuming the clock granularity is the same as the sampling interval, then the timestamp in packet $n + 1$ would be greater than that in packet n by 80. Note that fewer than 80 samples might have been sent due to compression techniques such as silence detection, and yet the timestamp allows the receiver to play back the samples with the correct temporal relationship.

The synchronization source (SSRC) is a 32-bit number that uniquely identifies a single source of an RTP stream. In a given multimedia conference, each sender picks a random SSRC and is expected to resolve conflicts in the unlikely event that two sources pick the same value. By making the source identifier something other than the network or transport address of the source, RTP ensures independence from the lower-layer protocol. It also enables a single node with multiple sources (e.g., several cameras) to distinguish those sources. When a single node generates different media streams (e.g., audio and video), it is not required to use the same SSRC in each stream, as there are mechanisms in RTCP (described below) to allow intermedia synchronization.

The contributing source (CSRC) is used only when a number of RTP streams pass through a “mixer.” A mixer can be used to reduce the bandwidth requirements for a conference by receiving data from many sources and sending it as a single stream. For example, the audio streams from several concurrent speakers could be decoded and recoded as a single audio stream. In this case, the mixer lists itself as the

synchronization source but also lists the contributing sources—the SSRC values of the speakers who contributed to the packet in question.

Control Protocol

RTCP provides a control stream that is associated with a data stream for a multimedia application. This control stream provides three main functions:

- 1 feedback on the performance of the application and the network
- 2 a way to correlate and synchronize different media streams that have come from the same sender
- 3 a way to convey the identity of a sender for display on a user interface (e.g., the vat interface shown in Figure 9.8)

The first function may be useful for rate-adaptive applications, which may use performance data to decide to use a more aggressive compression scheme to reduce congestion, or to send a higher-quality stream when there is little congestion. It can also be useful in diagnosing network problems.

You might think that the second function is already provided by the synchronization source ID of RTP, but in fact it is not. As already noted, multiple cameras from a single node might have different SSRC values. Furthermore, there is no requirement that an audio and video stream from the same node use the same SSRC. Because collisions of SSRC values may occur, it may be necessary to change the SSRC value of a stream. To deal with this problem, RTCP uses the concept of a “canonical name” (CNAME) that is assigned to a sender, which is then associated with the various SSRC values that might be used by that sender using RTCP mechanisms.

Simply correlating two streams is only part of the problem of intermedia synchronization. Because different streams may have completely different clocks (with different granularities and even different amounts of inaccuracy, or drift), there needs to be a way to accurately synchronize streams with each other. RTCP addresses this problem.

RTCP defines a number of different packet types, including

- sender reports, which enable active senders to a session to report transmission and reception statistics
- receiver reports, which receivers who are not senders use to report reception statistics
- source descriptions, which carry CNAMEs and other sender description information
- application-specific control packets

These different RTCP packet types are sent over the lower-layer protocol, which, as we have noted, is typically UDP. Several RTCP packets can be packed into a single PDU of the lower-level protocol. It is required that at least two RTCP packets are sent in every lower-level PDU: One of these is a report packet; the other is a source description packet. Other packets may be included up to the size limits imposed by the lower-layer protocols.

Before looking closely at the contents of an RTCP packet, we note that there is a potential problem with every member of a multicast group sending periodic control traffic. Unless we take some steps to limit it, this control traffic has the potential to be a significant consumer of bandwidth. For example, in an audioconference, no more than two or three senders are likely to send audio data at any instant, since there is no point in everyone talking at once. But there is no such social limit on everyone sending control traffic, and this could be a severe problem in a conference with thousands of participants. To deal with this problem, RTCP has a set of mechanisms by which the participants scale back their reporting frequency as the number of participants increases. These rules are somewhat complex, but the basic goal is this: Limit the total amount of RTCP traffic to a small percentage (typically 5%) of the RTP data traffic. To accomplish this goal, the participants should know how much data bandwidth is likely to be in use (e.g., the amount to send three audio streams) and the number of participants. They learn the former from means outside RTP (known as session management, discussed at the end of this section), and they learn the latter from the RTCP reports of other participants. Because RTCP reports might be sent at a very low rate, it might only be possible to get an approximate count of the current number of recipients, but that is typically sufficient. Also, it is recommended to allocate more RTCP bandwidth to active senders, on the assumption that most participants would like to see reports from them, for example, to find out who is speaking.

Once a participant has determined how much bandwidth it can consume with RTCP traffic, it sets about sending periodic reports at the appropriate rate. Sender reports and receiver reports differ only in that the former include some extra information about the sender. Both types of reports contain information about the data that was received from all sources in the most recent reporting period.

The extra information in a sender report consists of

- a timestamp containing the actual time of day when this report was generated
- the RTP timestamp corresponding to the time when the report was generated
- cumulative counts of the packets and bytes sent by this sender since it began transmission

Note that the first two quantities can be used to enable synchronization of different media streams from the same source, even if those streams use different clock granularities in their RTP data streams, since it gives the key to convert time of day to the RTP timestamps.

Both sender and receiver reports contain one block of data per source that has been heard from since the last report. Each block contains the following statistics for the source in question:

- its SSRC
- the fraction of data packets from this source that were lost since the last report was sent (calculated by comparing the number of packets received with the number of packets expected; this last value can be determined from the RTP sequence numbers)
- total number of packets lost from this source since the first time it was heard from
- highest sequence number received from this source (extended to 32 bits to account for wrapping of the sequence number)
- estimated interarrival jitter for the source (calculated by comparing the interarrival spacing of received packets with the expected spacing at transmission time)
- last actual timestamp received via RTCP for this source
- delay since last sender report received via RTCP for this source

As you might imagine, the recipients of this information can learn all sorts of things about the state of the session. In particular, they can see if other recipients are getting much better quality from some sender than they are, which might be an indication that a resource reservation needs to be made, or that there is a problem in the network that needs to be attended to. In addition, if a sender notices that many receivers are experiencing high loss of its packets, it might decide that it should reduce its sending rate or use a coding scheme that is more resilient to loss.

The final aspect of RTCP that we will consider is the source description packet. Such a packet contains, at a minimum, the SSRC of the sender and the sender's CNAME. The canonical name is derived in such a way that all applications that generate media streams that might need to be synchronized (e.g., separately generated audio and video streams from the same user) will choose the same CNAME even though they might choose different SSRC values. This enables a receiver to identify the media stream that came from the same sender. The most common format of the CNAME

is `user@host`, where `host` is the fully qualified domain name of the sending machine. Thus, an application launched by the user whose user name is `jdoe` running on the machine `cicada.cs.princeton.edu` would use the string `jdoe@cicada.cs.princeton.edu` as its CNAME. The large and variable number of bytes used in this representation would make it a bad choice for the format of an SSRC, since the SSRC is sent with every data packet and must be processed in real time. Allowing CNAMEs to be bound to SSRC values in periodic RTCP messages enables a compact and efficient format for the SSRC.

Other items may be included in the source description packet, such as the real name and email address of the user. These are used in user interface displays and to contact participants, but are less essential to the operation of RTP than the CNAME.

9.3.3 Session Control and Call Control (H.323)

While RTP and RTCP provide a wide range of functionality to multimedia applications, there is an aspect of multimedia conferencing that they do not address, usually referred to as *session control*. To understand this issue, consider the following problem. Suppose you want to hold a videoconference at a certain time and make it available to a wide number of participants. Perhaps you have decided to encode the video stream using the MPEG-2 standard, to use the multicast IP address 224.1.1.1 for transmission of the data, and to send it using RTP over UDP port number 4000. How would you make all that information available to the intended participants? One way would be to put all that information in an email and send it out, but ideally there should be a standard format and protocol for disseminating this sort of information. The IETF has a working group (the Multiparty Multimedia Session Control group) that has defined protocols for just this purpose. The protocols that have been defined include

- SDP (Session Description Protocol)
- SAP (Session Announcement Protocol)
- SIP (Session Initiation Protocol)
- SCCP (Simple Conference Control Protocol)

You might think that this is a lot of protocols for a seemingly simple task, but there are many aspects of the problem and several different situations in which it must be addressed. For example, there is a difference between announcing the fact that a certain conference session is going to be made available on the MBone (which would be done using SDP and SAP) and trying to make an internet phone call to a certain user at a particular time (which could be done using SDP and SIP). In the former case, you could consider your job done once you have sent all the session information in

a standard format to a well-known multicast address. In the latter, you would need to locate one or more users, get a message to them announcing your desire to talk (analogous to ringing their phone), and perhaps negotiate a suitable audio encoding among all parties.

The ITU has also been very active in this area, which is not surprising given its relevance to telephony, the traditional realm of that body. Fortunately, there has been considerable coordination between the IETF and the ITU in this instance, so that the various protocols are somewhat interoperable. The major ITU recommendation for multimedia communication over packet networks is known as H.323, which ties together many other recommendations, including H.245 for call control. The full set of recommendations covered by H.323 runs to many hundreds of pages, and the protocol is known for its complexity, so it is only possible to give a brief overview of it here.

H.323 is popular as a protocol for Internet telephony, and we consider that application here. A device that originates or terminates calls is known as an H.323 terminal; this might be a workstation running an Internet telephony application, or it might be a specially designed “appliance”—a telephonelike device with networking software and an Ethernet port, for example. H.323 terminals can talk to each other directly, but the calls are frequently mediated by a device known as a *gatekeeper*. Gatekeepers perform a number of functions such as translating among the various address formats used for phone calls, and controlling how many calls can be placed at a given time to limit the bandwidth used by the H.323 applications. H.323 also includes the concept of a *gateway*, which connects the H.323 network to other types of networks. The most common use of a gateway is to connect an H.323 network to the public switched telephone network (PSTN) as illustrated in Figure 9.12. This enables a user running an H.323 application on a computer to talk to a person using a conventional phone on the public telephone network. One useful function performed by the gatekeeper is to help a terminal find a gateway, perhaps choosing among several options to find one that is relatively close to the ultimate destination of the call. This is clearly useful in a world where conventional phones greatly outnumber PC-based phones. When an H.323 terminal makes a call to an endpoint that is a conventional phone, the gateway becomes the effective endpoint for the H.323 call and is responsible for performing the appropriate translation of both signalling information and the media stream that need to be carried over the telephone network.

An important part of H.323 is the H.245 call control protocol. When one H.323 terminal wants to call another, it uses H.245 to negotiate the properties of the call. It might list a number of different audio codec standards that it can support, and the far endpoint of the call would reply with a list of its own supported codecs, and the two ends could pick a coding standard that they can both live with. H.245 can also be used to signal the UDP port numbers that will be used by RTP and RTCP for the

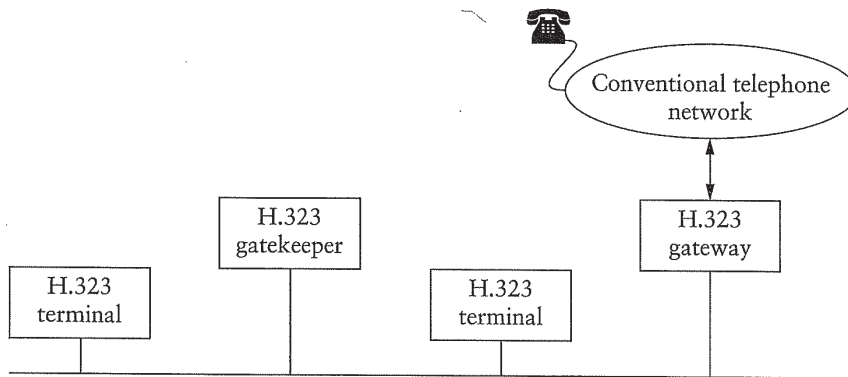


Figure 9.12 Devices in an H.323 network.

media stream (or streams—a call might include both audio and video, for example) in this call. Once this is accomplished, the call can proceed, with RTP being used to transport the media streams and RTCP carrying the relevant control information.

9.4 Summary

We have seen five different application-level protocols: the DNS protocol used by the domain naming system, SMTP used to exchange electronic mail, HTTP used to walk the World Wide Web, SNMP used to query remote nodes for the sake of network management, and RTP used by multimedia applications like *vic* and *vat*. While not by any means exhaustive, this discussion covers a number of different classes of applications. DNS, being rarely invoked directly by users, is more often thought of as a piece of infrastructure or middleware. SMTP and HTTP fall into the “traditional” application category, in contrast to real-time, multimedia applications that use RTP. And SNMP is the protocol most often relied upon by network managers, being rarely used by “average” users.

Application protocols are a curious lot. In many ways, they are like another layer of transport protocol. In other ways, they have application-specific knowledge that we would never want to pollute an application-independent transport protocol with. In effect, you could argue that all the protocols described in this chapter are just specialized transport protocols, and that transport protocols get layered on top of each other until producing the precise service needed by the application.

It's difficult to put a finger on a specific open issue in the realm of application protocols—the entire field is open as new applications are invented every day, and the networking needs of these applications are, well, application-dependent.

The real challenge to network designers is to recognize that what applications need from the network changes over time, and these changes drive the transport protocols we develop and the functionality we put into network routers.

Developing new transport protocols is a reasonably tractable problem. You may not be able to get the IETF to bless your transport protocol as an equal of TCP or UDP, but there's certainly nothing stopping you from designing the world's greatest multimedia application that comes bundled with a new end-to-end protocol that runs on top of UDP, much like happens with RTP.

On the other hand, pushing application-specific knowledge into the middle of the network—into the routers—is a much more difficult problem. This is because in order to effect a particular application, any new network service or functionality may need to be loaded into many, if not all, of the routers in the Internet. Web caching is an example of an advanced service you might want to load into network routers—cache-enhanced routers not only forward IP datagrams, but they also peek inside them and look at URLs—but Web caching works because you don't have to change all routers to gain a benefit. Just changing the router(s) you control is good enough.

In general, though, there are countless services you might want to load into the network—services that cannot be effectively provided only at the end hosts. For example, imagine a data dissemination service in which multiple sources are publishing a diverse collection of data items, and multiple clients are subscribing to some subset of those items. The items might be stock quotes, event notifications, or news articles, for example. Each source could use a traditional IP multicast group to disseminate the items to the clients, except this would be wasteful since a given client might be interested in only a small subset of items published by a given source. An alternative would be for the source to send the data items “downstream” and the clients to send their subscriptions (requests for specific item types) “upstream.” The intermediate router could be programmed to *prune* the items it sends downstream based on the subscriptions it had received on a given link.

How can applications hope to place functionality like this into the network itself? One possibility is that an emerging idea called *active networking* will eventually replace the relatively static model provided by today's routers. The idea of active networks is that packets will not only carry data, but they will also carry code (or

OPEN ISSUE

Programming the Network

possibly pointers to code) that tell the router how it should process the packet. Active networks raise many challenging problems—not the least of which is how to enforce security in a world where arbitrary applications can effectively program routers—but researchers are now starting to bring solutions together to build an *active backbone*, or ABone. Will today's vision of active networking succeed? It is hard to say, and there are voices on both sides of the argument in the networking community. What does seem certain is that there will continue to be demands for rapidly modifiable functionality in network devices.

FURTHER READING

The seminal paper on application-layer protocols is that by Clark and Tennenhouse, which is cited by the designers of RTP as their guiding vision. Rose's book on SNMP is an authoritative reference on the subject and explains many of the design decisions behind SNMP. The development of DNS is well described by Mockapetris and Dunlap.

- Clark, D., and D. Tennenhouse. Architectural considerations for a new generation of protocols. *Proceedings of the SIGCOMM '90 Symposium*, pages 200–208, September 1990.
- Rose, M. *The Simple Book. An Introduction to Internet Management*. Prentice Hall, Englewood Cliffs, NJ, 1994.
- Mockapetris, P., and K. Dunlap. Development of the domain name system. *Proceedings of the SIGCOMM '88 Symposium*, pages 123–133, August 1988.

There are a wealth of papers on naming, as well as on the related issue of resource discovery (finding out what resources exist in the first place). General studies of naming can be found in Terry [Ter86], Comer and Peterson [CP89], Birrell et al. [BLNS82], Saltzer [Sal78], Shoch [Sho78], and Watson [Wat81]; attribute-based (descriptive) naming systems are described in Peterson [Pet88] and Bowman et al. [BPY90]; and resource discovery is the subject of Bowman et al. [BDMS94].

SMTP is originally defined in RFC 821 [Pos82], and of course, RFC 822 is RFC 822 [Cro82]. MIME is defined in a series of RFCs; the original specification was in RFC 1521 [BF93] and the most recent version is defined in RFC 2045 [FB96].

Version 1.0 of HTTP is specified in RFC 1945 [BLFF96], and the latest version (1.1) is defined in RFC 2068 [FGMBL97]. There are a wealth of papers written about Web performance, especially Web caching. A good example is a paper by Danzig on Web traffic and its implications on the effectiveness of caching [Dan98].

Network management is a sufficiently large and important field that the IETF devotes an entire area to it. There are well over 100 RFCs describing various aspects of SNMP and MIBs. The two key references, however, are Case et al. [CMRW93], which defines version 2 of SNMP (SNMPv2), and McCloghrie and Rose [MR91], which defines the second version of the mandatory MIB variables (MIB-II). Many of the other SNMP/MIB-related RFCs define extensions to the core set of MIB variables—for example, variables that are specific to a particular network technology or to a particular vendor's product.

RTP is described in RFC 1889 [SCFJ96], although much of the interesting detail is in Internet drafts that are yet to be published. McCanne and Jacobson [MJ95] describe *vic*, one of the applications to use RTP.

Several research papers report results in the emerging area of active networking. Wetherall et al. [WGT99], Alexander et al. [ASNS97], and Hartman et al. [HPB⁺99] describe early active network prototypes. Smith et al. [SCM⁺99] reports on a consensus architecture coming out of the active networks research community.

Finally, we recommend the following live reference to help keep tabs on the rapid evolution of the Web:

■ <http://www.w3.org>: World Wide Web Consortium

EXERCISES

- 1 ARP and DNS both depend on caches; ARP cache entry lifetimes are typically 10 minutes, while DNS cache is on the order of days. Justify this difference. What undesirable consequences might there be in having too long a DNS cache entry lifetime?
- 2 IPv6 simplifies ARP out of existence by allowing hardware addresses to be part of the IPv6 address. How does this complicate the job of DNS? How does this affect the problem of finding your local DNS server?
- 3 DNS servers also allow reverse lookup; given an IP address 128.112.169.4, it is reversed into a text string 4.169.112.128.in-addr.arpa and looked up using DNS PTR records (which form a hierarchy of domains analogous to that for the address domain hierarchy). Suppose you want to authenticate the sender of a packet based on its host name and are confident that the source IP *address* is genuine. Explain the insecurity in converting the source address to a name as above and then comparing this name to a given list of trusted hosts. Hint: Whose DNS servers would you be trusting?

- 4 What is the relationship between a domain name (e.g., cs.princeton.edu) and an IP subnet number (e.g., 192.12.69.0)? Do all hosts on the subnet have to be identified by the same name server? What about reverse lookup, as in the previous exercise?
- 5 Suppose a host elects to use a name server not within its organization for address resolution. When would this result in no more total traffic, for queries not found in any DNS cache, than with a local name server? When might this result in a better DNS cache hit rate and possibly less total traffic?
- 6 Figure 9.4 shows the hierarchy of name servers. How would you represent this hierarchy if one name server served multiple zones? In that setting, how does the name server hierarchy relate to the zone hierarchy? How do you deal with the fact that each zone may have multiple name servers?
- 7 Use the whois utility/service to find out who is in charge of your site, at least as far as the InterNIC is concerned. Look up your site both by DNS name and by IP network number; for the latter you may have to try an alternative whois server (e.g., `whois -h whois.arin.net . . .`). Try princeton.edu and cisco.com as well.
- 8 Many smaller organizations have their Web sites maintained by a third party. How could you use whois to find if this is the case, and, if so, the identity of the third party?
- 9 One feature of the existing DNS .com hierarchy is that it is extremely “wide.”
 - (a) Propose a more hierarchical reorganization of the .com hierarchy. What objections might you foresee to your proposal’s adoption?
 - (b) What might be some of the consequences of having most DNS domain names contain four or more levels, versus the two of many existing names?
- 10 Suppose, in the other direction, we abandon any pretense at all of DNS hierarchy, and simply move all the .com entries to the root name server: `www.cisco.com` would become `www.cisco`, or perhaps just `cisco`. How would this affect root name server traffic in general? How would this affect such traffic for the specific case of resolving a name like `cisco` into a Web server address?
- 11 What DNS cache issues are involved in changing the IP address of, say, a Web server host name? How might these be minimized?

- 12 Take a suitable DNS-lookup utility (e.g., `nslookup`) and disable the recursive lookup feature (e.g., with `set norecurse`), so that when your utility sends a query to a DNS server, and that server is unable to fully answer the request from its own records, the server sends back the next DNS server in the lookup sequence rather than automatically forwarding the query to that next server. Then carry out manually a name lookup such as that in Figure 9.5; try the host name `www.cs.princeton.edu`. List each intermediate name server contacted. You may also need to specify that queries are for NS records rather than the usual A records.
- 13 Discuss how you might rewrite SMTP or HTTP to make use of a hypothetical general-purpose request/reply protocol (perhaps something like CHAN RPC). Could an appropriate analog of persistent connections be moved from the application layer into such a transport protocol? What other application tasks might be moved into this protocol?
- 14 Most Telnet clients can be used to connect to port 25, the SMTP port, instead of to the Telnet port. Using such a tool, connect to an SMTP server and send yourself (or someone else, with permission) some forged email. Then examine the headers for evidence the message isn't genuine.
- 15 What features might be used by (or added to) SMTP and/or a mail daemon such as `sendmail` to provide some resistance to email forgeries as in the previous exercise?
- 16 Find out how SMTP hosts deal with unknown commands from the other side, and how in particular this mechanism allows for the evolution of the protocol (e.g., to "extended SMTP"). You can either read the RFC, or contact an SMTP server as in Exercise 14 and test its responses to nonexistent commands.
- 17 As presented in the text, SMTP involves the exchange of several small messages. In most cases, the server responses do not affect what the client sends subsequently. The client might thus implement *command pipelining*: sending multiple commands in a single message.
 - (a) For what SMTP commands *does* the client need to pay attention to the server's responses?
 - (b) Assume the server reads each client message with `gets()` or the equivalent, which reads in a string up to a `<LF>`. What would it have to do even to detect that a client had used command pipelining?

- (c) Pipelining is nonetheless known to break with some servers; find out how a client can negotiate its use.
- 18 Find out what other features DNS MX records provide in addition to supplying an alias for a mail server; the latter could, after all, be provided by a DNS CNAME record. MX records are provided to support email; would an analogous WEB record be of use in supporting HTTP?
- 19 One of the central problems faced by a protocol such as MIME is the vast number of data formats available. Consult the MIME RFC to find out how MIME deals with new or system-specific image and text formats.
- 20 MIME supports multiple representations of the same content using the multipart/alternative syntax; for example, text could be sent as text/plain, text/richtext, and application/postscript. Why do you think plaintext is supposed to be the *first* format, even though implementations might find it easier to place plaintext after their native format?
- 21 Consult the MIME RFC to find out how base64 encoding handles binary data of a length not evenly divisible by three bytes.
- 22 In HTTP version 1.0, a server marked the end of a transfer by closing the connection. Explain why, in terms of the TCP layer, this was a problem for servers. Find out how HTTP version 1.1 avoids this. How might a general-purpose request/reply protocol address this?
- 23 Find out how to configure an HTTP server so as to eliminate the 404 not found message, and have a default (and hopefully friendlier) message returned instead. Decide if such a feature is part of the protocol or part of an implementation, or is technically even permitted by the protocol. (Documentation for the apache HTTP server can be found at www.apache.org.)
- 24 Why does the HTTP GET command on page 643,

GET <http://www.cs.princeton.edu/> HTTP/1.1

contain the name of the server being contacted? Wouldn't the server already know its name? Use Telnet, as in Exercise 14, to connect to port 80 of an HTTP server and find out what happens if you leave the host name out.

- 25** When an HTTP server initiates a `close()` at its end of a connection, it must then wait in TCP state `FIN_WAIT_2` for the client to close the other end. What mechanism within the TCP protocol could help an HTTP server deal with noncooperative or poorly implemented clients that don't close from their end? If possible, find out about the programming interface for this mechanism, and indicate how an HTTP server might apply it.
- 26** The POP3 Post Office Protocol only allows a client to retrieve email, using a password for authentication. Traditionally, to *send* email a client would simply send it to its server and expect that it be relayed.
- (a) Explain why email servers often no longer permit such relaying from arbitrary clients.
 - (b) Propose an SMTP option for remote client authentication.
 - (c) Find out what existing methods are available for addressing this issue.
- 27** Suppose a very large Web site wants a mechanism by which clients access whichever of multiple HTTP servers is "closest" by some suitable measure.
- (a) Discuss developing a mechanism within HTTP for doing this.
 - (b) Discuss developing a mechanism within DNS for doing this.
- Compare the two. Can either approach be made to work without upgrading the browser?
- 28** Find out if there is available to you an SNMP node that will answer queries you send it. If so, locate some SNMP utilities (e.g., the `ucd-snmp` suite) and try the following:
- (a) Fetch the entire system group, using something like

```
snmpwalk nodename public system
```

Also try the above with `1` in place of `system`.
 - (b) Manually walk through the system group, using multiple SNMP GET-NEXT operations (e.g., using `snmpgetnext` or equivalent), retrieving one entry at a time.
- 29** Using the SNMP device and utilities of the previous exercise, fetch the `tcp` group (numerically group 6), or some other group. Then do something to make some of the group's counters change, and fetch the group again to show the change. Try to do this in such a way that you can be sure your actions were the cause of the change recorded.

- 30 What information provided by SNMP might be useful to someone planning the IP spoofing attack of Exercise 17 in Chapter 5? What other SNMP information might be considered sensitive?
- 31 Try to find situations where an RTP application might reasonably do the following:
- Send multiple packets at essentially the same time that need different timestamps.
 - Send packets at different times that need the same timestamp.
- Argue, in consequence, that RTP timestamps must, in at least some cases, be provided (at least indirectly) by the application. Hint: Think of cases where the sending rate and playback rate might not match.
- 32 Having the timestamp clock count time in units of one frame time or one voice sample time would be the minimum resolution to ensure accurate playback. But the time unit is usually considerably smaller; what is the purpose of this?
- 33 Suppose we want returning RTCP reports from receivers to amount to no more than 5% of the outgoing primary RTP stream. If each report is 84 bytes, and the RTP traffic is 20KBps, and there are 1000 recipients, how often do individual receivers get to report? What if there are 10,000 recipients?
- 34 RFC 1889 specifies that the time interval between receiver RTCP reports include a randomization factor to avoid having all the receivers send at the same time. If all the receivers sent in the same 5% subinterval of their reply time interval, the arriving upstream RTCP traffic would rival the downstream RTP traffic.
- (a) Video receivers might reasonably wait to send their reports until the higher-priority task of processing and displaying one frame is completed; this might mean their RTCP transmissions were synchronized on frame boundaries. Is this likely to be a serious concern?
 - (b) With 10 receivers, what is the probability of their all sending in one particular 5% subinterval?
 - (c) With 10 receivers, what is the probability half will send in one particular 5% subinterval? Multiply this by 20 for an estimate of the probability half will all send in the same arbitrary 5% subinterval. Hint: How many ways can we choose 5 receivers out of 10?
- 35 What might a server actually do with the packet-loss-rate data and jitter data in receiver reports?

- 36** Suppose some receivers in a large conference can receive data at a significantly higher bandwidth than others. What sorts of things might be implemented to address this? Hint: Consider both the session announcement protocol (SAP) and the possibility of utilizing third-party “mixers.”
- 37** Propose a mechanism for deciding when to report an RTP packet as lost. How does your mechanism compare with the TCP adaptive retransmission mechanisms of Section 5.2.5?
- 38** How might you encode audio (or video) data in two packets so that if one packet is lost, then the resolution is simply reduced to what would be expected with half the bandwidth? Explain why this is much more difficult if a JPEG-type encoding is used.