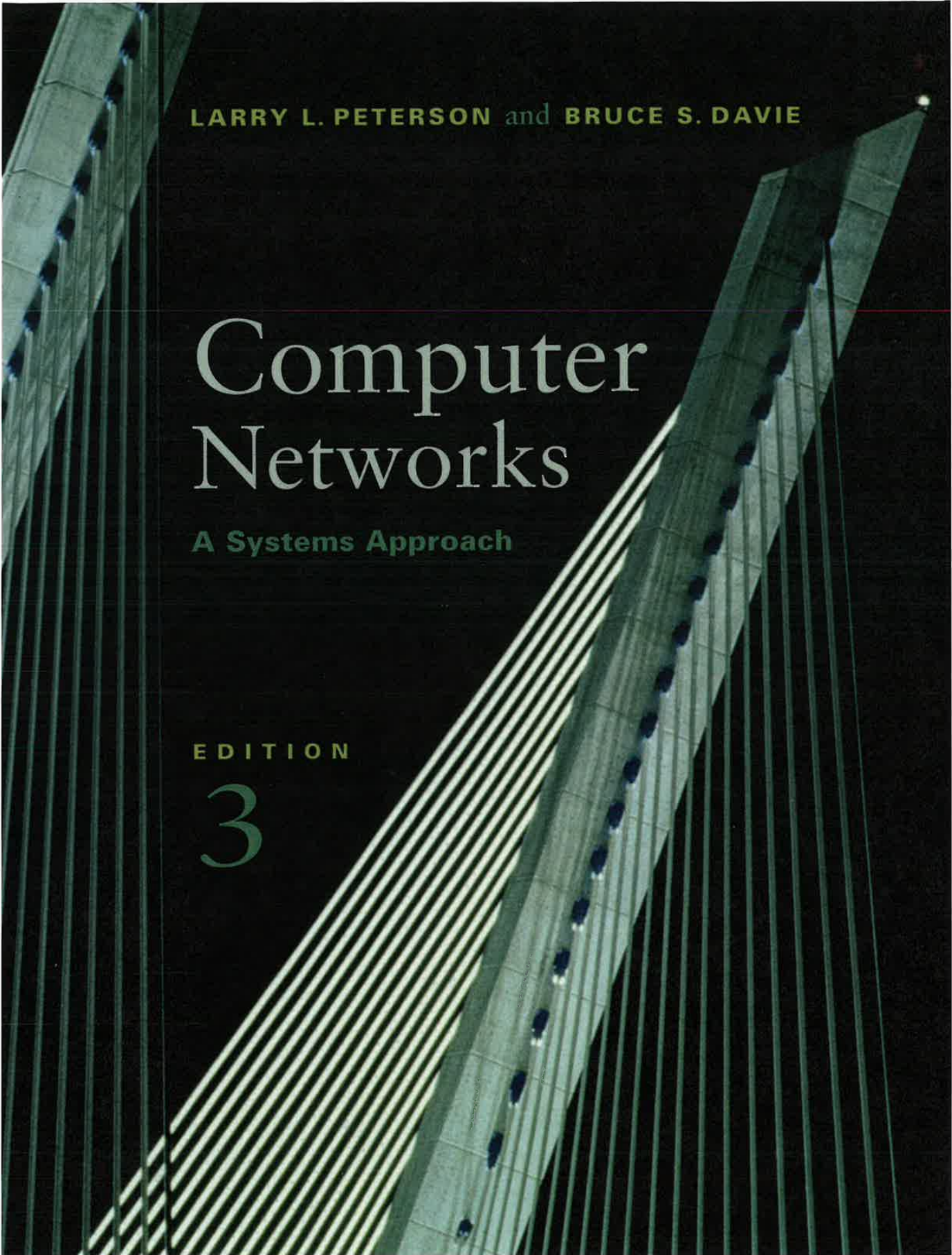




LARRY L. PETERSON and BRUCE S. DAVIE

Computer Networks

A Systems Approach

EDITION

3

THIRD EDITION

Larry L. Peterson & Bruce S. Davie

COMPUTER NETWORKS

A Systems Approach



MORGAN KAUFMANN PUBLISHERS

AN IMPRINT OF ELSEVIER SCIENCE

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

Senior Editor Rick Adams
Publishing Services Manager Simon Crump
Developmental Editor Karyn Johnson
Cover Design Ross Carron Design
Cover Image Vasco de Gama Bridge, Lisbon, Portugal
Composition/Illustration International Typesetting and Composition
Copyeditor Ken DellaPenta
Proofreader Jennifer McClain
Indexer Steve Rath
Printer Courier Corporation

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

Morgan Kaufmann Publishers
An Imprint of Elsevier Science
340 Pine Street, Sixth Floor
San Francisco, CA 94104-3205
www.mkp.com

© 2003 by Elsevier Science (USA)
All rights reserved
Printed in the United States of America

07 06 05 04 03 5 4 3 2 1

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, or otherwise—without the prior written permission of the publisher.

Library of Congress Control Number
ISBN: 1-55860-832-X (Casebound)
ISBN: 1-55860-833-8 (Paperback)

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 Applications	4
1.2 Requirements	6
1.2.1 Connectivity	7
1.2.2 Cost-Effective Resource Sharing	10
1.2.3 Support for Common Services	15
1.3 Network Architecture	19
1.3.1 Layering and Protocols	20
1.3.2 OSI Architecture	26
1.3.3 Internet Architecture	27
1.4 Implementing Network Software	30
1.4.1 Application Programming Interface (Sockets)	31
1.4.2 Example Application	33
1.4.3 Protocol Implementation Issues	36
1.5 Performance	40
1.5.1 Bandwidth and Latency	40
1.5.2 Delay $\times$ Bandwidth Product	44
1.5.3 High-Speed Networks	46
1.5.4 Application Performance Needs	48
1.6 Summary	50
Open Issue: Ubiquitous Networking	51
Further Reading	52
Exercises	55

2 Direct Link Networks

Problem: Physically Connecting Hosts	64
2.1 Hardware Building Blocks	66
2.1.1 Nodes	66
2.1.2 Links	67
2.2 Encoding (NRZ, NRZI, Manchester, 4B/5B)	75
2.3 Framing	79
2.3.1 Byte-Oriented Protocols (BISYNC, PPP, DDCMP)	80
2.3.2 Bit-Oriented Protocols (HDLC)	83
2.3.3 Clock-Based Framing (SONET)	84
2.4 Error Detection	88
2.4.1 Two-Dimensional Parity	89
2.4.2 Internet Checksum Algorithm	90
2.4.3 Cyclic Redundancy Check	92
2.5 Reliable Transmission	97
2.5.1 Stop-and-Wait	98
2.5.2 Sliding Window	101
2.5.3 Concurrent Logical Channels	110
2.6 Ethernet (802.3)	111
2.6.1 Physical Properties	112
2.6.2 Access Protocol	114
2.6.3 Experience with Ethernet	119
2.7 Token Rings (802.5, FDDI)	120
2.7.1 Physical Properties	121
2.7.2 Token Ring Media Access Control	122
2.7.3 Token Ring Maintenance	125
2.7.4 Frame Format	126
2.7.5 FDDI	127
2.8 Wireless (802.11)	131
2.8.1 Physical Properties	132
2.8.2 Collision Avoidance	133
2.8.3 Distribution System	134
2.8.4 Frame Format	136
2.9 Network Adaptors	137
2.9.1 Components	138
2.9.2 View from the Host	139
2.9.3 Memory Bottleneck	144
2.10 Summary	146

Open Issue: Does It Belong in Hardware?	147
Further Reading	148
Exercises	150
3 Packet Switching	
Problem: Not All Networks Are Directly Connected	164
3.1 Switching and Forwarding	166
3.1.1 Datagrams	168
3.1.2 Virtual Circuit Switching	170
3.1.3 Source Routing	177
3.2 Bridges and LAN Switches	180
3.2.1 Learning Bridges	181
3.2.2 Spanning Tree Algorithm	185
3.2.3 Broadcast and Multicast	189
3.2.4 Limitations of Bridges	190
3.3 Cell Switching (ATM)	192
3.3.1 Cells	193
3.3.2 Segmentation and Reassembly	198
3.3.3 Virtual Paths	203
3.3.4 Physical Layers for ATM	204
3.3.5 ATM in the LAN	205
3.4 Implementation and Performance	210
3.4.1 Ports	212
3.4.2 Fabrics	216
3.5 Summary	220
Open Issue: The Future of ATM	221
Further Reading	221
Exercises	223
4 Internetworking	
Problem: There Is More Than One Network	234
4.1 Simple Internetworking (IP)	236
4.1.1 What Is an Internetwork?	236
4.1.2 Service Model	238
4.1.3 Global Addresses	250
4.1.4 Datagram Forwarding in IP	252
4.1.5 Address Translation (ARP)	258
4.1.6 Host Configuration (DHCP)	263

4.1.7 Error Reporting (ICMP)	266
4.1.8 Virtual Networks and Tunnels	267
4.2 Routing	271
4.2.1 Network as a Graph	272
4.2.2 Distance Vector (RIP)	274
4.2.3 Link State (OSPF)	282
4.2.4 Metrics	291
4.2.5 Routing for Mobile Hosts	295
4.3 Global Internet	299
4.3.1 Subnetting	301
4.3.2 Classless Routing (CIDR)	306
4.3.3 Interdomain Routing (BGP)	308
4.3.4 Routing Areas	316
4.3.5 IP Version 6 (IPv6)	318
4.4 Multicast	331
4.4.1 Link-State Multicast	332
4.4.2 Distance-Vector Multicast	332
4.4.3 Protocol Independent Multicast (PIM)	336
4.5 Multiprotocol Label Switching (MPLS)	340
4.5.1 Destination-Based Forwarding	340
4.5.2 Explicit Routing	346
4.5.3 Virtual Private Networks and Tunnels	348
4.6 Summary	352
Open Issue: Deployment of IPV6	353
Further Reading	354
Exercises	355
 5 End-to-End Protocols	
Problem: Getting Processes to Communicate	374
5.1 Simple Demultiplexer (UDP)	376
5.2 Reliable Byte Stream (TCP)	378
5.2.1 End-to-End Issues	379
5.2.2 Segment Format	382
5.2.3 Connection Establishment and Termination	384
5.2.4 Sliding Window Revisited	389
5.2.5 Triggering Transmission	395
5.2.6 Adaptive Retransmission	397
5.2.7 Record Boundaries	401

5.2.8 TCP Extensions	402
5.2.9 Alternative Design Choices	403
5.3 Remote Procedure Call	405
5.3.1 Bulk Transfer (BLAST)	408
5.3.2 Request/Reply (CHAN)	414
5.3.3 Dispatcher (SELECT)	423
5.3.4 Putting It All Together (SunRPC, DCE)	424
5.4 Performance	431
5.5 Summary	434
Open Issue: Application-Specific Protocols	435
Further Reading	436
Exercises	437

6 Congestion Control and Resource Allocation

Problem: Allocating Resources	450
6.1 Issues in Resource Allocation	452
6.1.1 Network Model	452
6.1.2 Taxonomy	456
6.1.3 Evaluation Criteria	458
6.2 Queuing Disciplines	461
6.2.1 FIFO	462
6.2.2 Fair Queuing	463
6.3 TCP Congestion Control	468
6.3.1 Additive Increase/Multiplicative Decrease	468
6.3.2 Slow Start	471
6.3.3 Fast Retransmit and Fast Recovery	476
6.4 Congestion-Avoidance Mechanisms	478
6.4.1 DECbit	478
6.4.2 Random Early Detection (RED)	480
6.4.3 Source-Based Congestion Avoidance	486
6.5 Quality of Service	492
6.5.1 Application Requirements	493
6.5.2 Integrated Services (RSVP)	499
6.5.3 Differentiated Services (EF, AF)	509
6.5.4 ATM Quality of Service	513
6.5.5 Equation-Based Congestion Control	517
6.6 Summary	518

Open Issue: Inside versus Outside the Network	519
Further Reading	520
Exercises	521
 7 End-to-End Data	
Problem: What Do We Do with the Data?	534
7.1 Presentation Formatting	536
7.1.1 Taxonomy	537
7.1.2 Examples (XDR, ASN.1, NDR)	541
7.1.3 Markup Languages (XML)	545
7.2 Data Compression	548
7.2.1 Lossless Compression Algorithms	550
7.2.2 Image Compression (JPEG)	552
7.2.3 Video Compression (MPEG)	557
7.2.4 Transmitting MPEG over a Network	562
7.2.5 Audio Compression (MP3)	566
7.3 Summary	568
Open Issue: Computer Networks Meet Consumer Electronics	568
Further Reading	569
Exercises	570
 8 Network Security	
Problem: Securing the Data	578
8.1 Cryptographic Algorithms	580
8.1.1 Requirements	582
8.1.2 Secret Key Encryption (DES)	583
8.1.3 Public Key Encryption (RSA)	588
8.1.4 Message Digest Algorithms (MD5)	591
8.1.5 Implementation and Performance	593
8.2 Security Mechanisms	594
8.2.1 Authentication Protocols	594
8.2.2 Message Integrity Protocols	598
8.2.3 Public Key Distribution (X.509)	601
8.3 Example Systems	604
8.3.1 Pretty Good Privacy (PGP)	605
8.3.2 Secure Shell (SSH)	607
8.3.3 Transport Layer Security (TLS, SSL, HTTPS)	610
8.3.4 IP Security (IPSEC)	614

8.4	Firewalls	617
8.4.1	Filter-Based Firewalls	618
8.4.2	Proxy-Based Firewalls	620
8.4.3	Limitations	622
8.5	Summary	622
	Open Issue: Denial-of-Service Attacks	623
	Further Reading	624
	Exercises	625
9	Applications	
	Problem: Applications Need Their Own Protocols	632
9.1	Name Service (DNS)	634
9.1.1	Domain Hierarchy	635
9.1.2	Name Servers	636
9.1.3	Name Resolution	640
9.2	Traditional Applications	643
9.2.1	Electronic Mail (SMTP, MIME, IMAP)	644
9.2.2	World Wide Web (HTTP)	650
9.2.3	Network Management (SNMP)	657
9.3	Multimedia Applications	660
9.3.1	Real-time Transport Protocol (RTP)	660
9.3.2	Session Control and Call Control (SDP, SIP, H.323)	671
9.4	Overlay Networks	680
9.4.1	Routing Overlays	682
9.4.2	Peer-to-Peer Networks	690
9.4.3	Content Distribution Networks	698
9.5	Summary	704
	Open Issue: New Network Architecture	704
	Further Reading	705
	Exercises	707
	Glossary	715
	Bibliography	743
	Solutions to Selected Exercises	767
	Index	781
	About the Authors	815

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

(Dave Clark)



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

1.4 Implementing Network Software

Network architectures and protocol specifications are essential things, but a good blueprint is not enough to explain the phenomenal success of the Internet: The number of computers connected to the Internet has been doubling every year since 1981 and is now approaching 200 million; the number of people who use the Internet is estimated at well over 600 million; and it is believed that the number of bits transmitted over the Internet surpassed the corresponding figure for the voice phone system sometime in 2001.

What explains the success of the Internet? There are certainly many contributing factors (including a good architecture), but one thing that has made the Internet such a runaway success is the fact that so much of its functionality is provided by software running in general-purpose computers. The significance of this is that new functionality can be added readily with “just a small matter of programming.” As a result, new applications and services—electronic commerce, videoconferencing, and packet telephony, to name a few—have been showing up at a phenomenal pace.

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

The point to take away from this is that knowing how to implement network software is an essential part of understanding computer networks. With this in mind, this section first introduces some of the issues involved in implementing an application program on top of a network, and then goes on to identify the issues involved in

implementing the protocols running within the network. In many respects, network applications and network protocols are very similar—the way an application engages the services of the network is pretty much the same as the way a high-level protocol invokes the services of a low-level protocol. As we will see later in the section, however, there are a couple of important differences.

1.4.1 Application Programming Interface (Sockets)

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

Although each operating system is free to define its own network API (and most have), over time certain of these APIs have become widely supported; that is, they have been ported to operating systems other than their native system. This is what has happened with the *socket interface* originally provided by the Berkeley distribution of Unix, which is now supported in virtually all popular operating systems. The advantage of industrywide support for a single API is that applications can be easily ported from one OS to another. It is important to keep in mind, however, that application programs typically interact with many parts of the OS other than the network; for example, they read and write files, fork concurrent processes, and output to the graphical display. Just because two systems support the same network API does not mean that their file system, process, or graphic interfaces are the same. Still, understanding a widely adopted API like Unix sockets gives us a good place to start.

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

The main abstraction of the socket interface, not surprisingly, is the *socket*. A good way to think of a socket is as the point where a local application process attaches to the network. The interface defines operations for creating a socket, attaching the socket to the network, sending/receiving messages through the socket, and closing the

socket. To simplify the discussion, we will limit ourselves to showing how sockets are used with TCP.

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

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

The reason that this operation takes three arguments is that the socket interface was designed to be general enough to support any underlying protocol suite. Specifically, the domain argument specifies the protocol *family* that is going to be used: PF_INET denotes the Internet family, PF_UNIX denotes the Unix pipe facility, and PF_PACKET denotes direct access to the network interface (i.e., it bypasses the TCP/IP protocol stack). The type argument indicates the semantics of the communication. SOCK_STREAM is used to denote a byte stream. SOCK_DGRAM is an alternative that denotes a message-oriented service, such as that provided by UDP. The protocol argument identifies the specific protocol that is going to be used. In our case, this argument is UNSPEC because the combination of PF_INET and SOCK_STREAM implies TCP. Finally, the return value from socket is a *handle* for the newly created socket, that is, an identifier by which we can refer to the socket in the future. It is given as an argument to subsequent operations on this socket.

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

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

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

The listen operation then defines how many connections can be pending on the specified socket. Finally, the accept operation carries out the passive open. It is a blocking operation that does not return until a remote participant has established a connection, and when it does complete, it returns a *new* socket that corresponds to this just-established connection, and the address argument contains the *remote*

participant's address. Note that when `accept` returns, the original socket that was given as an argument still exists and still corresponds to the passive open; it is used in future invocations of `accept`.

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

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

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

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

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

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

1.4.2 Example Application

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

Client

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

client program enters its main loop, which reads text from standard input and sends it over the socket.

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

#define SERVER_PORT 5432
#define MAX_LINE 256

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

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

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

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

    /* active open */
    if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("simplex-talk: socket");
        exit(1);
    }
}
```

```

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

```

Server

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

```

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

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

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

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

```

```

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

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

```

1.4.3 Protocol Implementation Issues

As mentioned at the beginning of this section, the way application programs interact with the underlying network is similar to the way a high-level protocol interacts with a low-level protocol. For example, TCP needs an interface to send outgoing messages to IP, and IP needs to be able to deliver incoming messages to TCP. This is exactly the service interface introduced in Section 1.3.1.

Since we already have a network API (e.g., sockets), we might be tempted to use this same interface between every pair of protocols in the protocol stack. Although certainly an option, in practice the socket interface is not used in this way. The reason is that there are inefficiencies built into the socket interface that protocol implementers are not willing to tolerate. Application programmers tolerate them because they simplify their programming task and because the inefficiency only has to be tolerated once, but protocol implementers are often obsessed with performance and must worry about getting a message through several layers of protocols. The rest of this section discusses the two primary differences between the network API and the protocol-to-protocol interface found lower in the protocol graph.

Process Model

Most operating systems provide an abstraction called a *process*, or alternatively, a *thread*. Each process runs largely independently of other processes, and the OS is

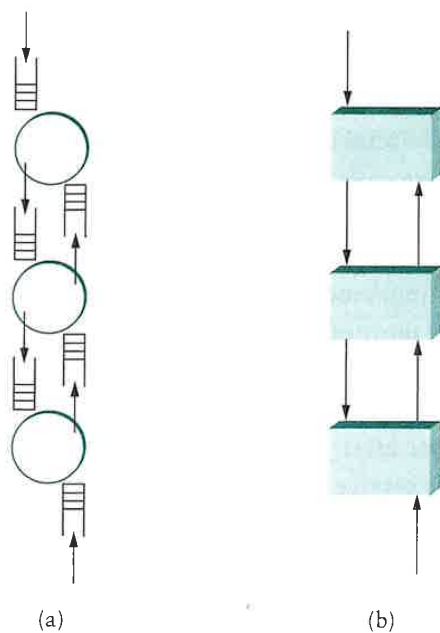


Figure 1.16 Alternative process models: (a) *process-per-protocol*; (b) *process-per-message*.

responsible for making sure that resources, such as address space and CPU cycles, are allocated to all the current processes. The process abstraction makes it fairly straightforward to have a lot of things executing concurrently on one machine; for example, each user application might execute in its own process, and various things inside the OS might execute as other processes. When the OS stops one process from executing on the CPU and starts up another one, we call the change a *context switch*.

When designing the network subsystem, one of the first questions to answer is, “Where are the processes?” There are essentially two choices, as illustrated in Figure 1.16. In the first, which we call the *process-per-protocol* model, each protocol is implemented by a separate process. This means that as a message moves up or down the protocol stack, it is passed from one process/protocol to another—the process that implements protocol i processes the message, then passes it to protocol $i - 1$, and so on. How one process/protocol passes a message to the next process/protocol depends on the support the host OS provides for interprocess communication. Typically, there is a simple mechanism for enqueueing a message with a process. The important point, however, is that a context switch is required at each level of the protocol graph—typically a time-consuming operation.

The alternative, which we call the *process-per-message* model, treats each protocol as a static piece of code and associates the processes with the messages.

That is, when a message arrives from the network, the OS dispatches a process that it makes responsible for the message as it moves up the protocol graph. At each level, the procedure that implements that protocol is invoked, which eventually results in the procedure for the next protocol being invoked, and so on. For outbound messages, the application's process invokes the necessary procedure calls until the message is delivered. In both directions, the protocol graph is traversed in a sequence of procedure calls.

Although the process-per-protocol model is sometimes easier to think about—I implement my protocol in my process, and you implement your protocol in your process—the process-per-message model is generally more efficient for a simple reason: A procedure call is an order of magnitude more efficient than a context switch on most computers. The former model requires the expense of a context switch at each level, while the latter model costs only a procedure call per level.

Now think about the relationship between the service interface as defined above and the process model. For an outgoing message, the high-level protocol invokes a `send` operation on the low-level protocol. Because the high-level protocol has the message in hand when it calls `send`, this operation can be easily implemented as a procedure call; no context switch is required. For incoming messages, however, the high-level protocol invokes the `receive` operation on the low-level protocol, and then must wait for a message to arrive at some unknown future time; this basically forces a context switch. In other words, the process running in the high-level protocol receives a message from the process running in the low-level protocol. This isn't a big deal if only the application process receives messages from the network subsystem—in fact, it's the right interface for the network API since application programs already have a process-centric view of the world—but it does have a significant impact on performance if such a context switch occurs at each layer of the protocol stack.

It is for this reason that most protocol implementations replace the `receive` operation with a `deliver` operation. That is, the low-level protocol does an *upcall*—a procedure call up the protocol stack—to deliver the message to the high-level protocol. Figure 1.17 shows the resulting interface between two adjacent protocols, TCP and IP in this case. In general, messages move down the protocol graph through a sequence of `send` operations, and up the protocol graph through a sequence of `deliver` operations.

Message Buffers

A second inefficiency of the socket interface is that the application process provides the buffer that contains the outbound message when calling `send`, and similarly it provides the buffer into which an incoming message is copied when invoking the `receive` operation. This forces the topmost protocol to copy the message from the application's buffer into a network buffer, and vice versa, as shown in Figure 1.18. It turns out that copying data from one buffer to another is one of the most expensive things a

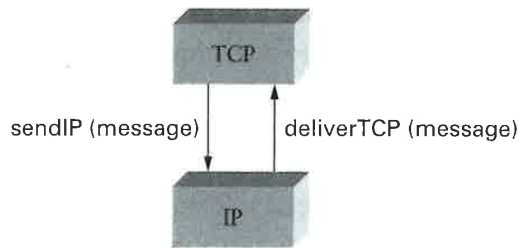


Figure 1.17 Protocol-to-protocol interface.

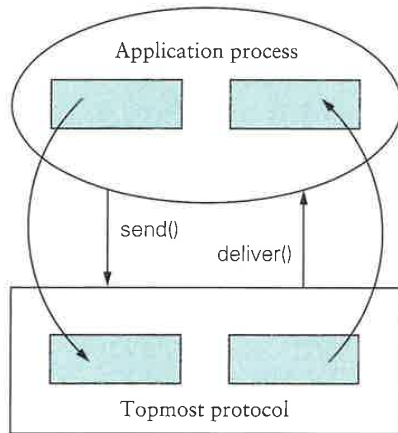


Figure 1.18 Copying incoming/outgoing messages between application buffer and network buffer.

protocol implementation can do. This is because while processors are becoming faster at an incredible pace, memory is not getting faster as quickly as processors are.

Instead of copying message data from one buffer to another at each layer in the protocol stack, most network subsystems define an abstract data type for messages that is shared by all protocols in the protocol graph. Not only does this abstraction permit messages to be passed up and down the protocol graph without copying, but it usually provides copy-free ways of manipulating messages in other ways, such as adding and stripping headers, fragmenting large messages into a set of small messages, and reassembling a collection of small messages into a single large message. The exact form of this message abstraction differs from OS to OS, but it generally involves a linked list of pointers to message buffers, similar to the one shown in Figure 1.19. We leave it as an exercise for you to define a general copy-free message abstraction.