UNIX Network Programming Volume 1

Second Edition

Networking APIs: Sockets and XTI

by W. Richard Stevens

To join a Prentice Hall PTR Internet mailing list, point to
http://www.prenhall.com/mail_lists/

ISBN 0-13-490012-X

Prentice Hall PTR
Upper Saddle River, NJ 07458

Contents

Preface

Introduction

Network programming involves writing programs that communicate with other programs across a computer network. One program is normally called the *client* and the other the *server*. Most operating systems provide precompiled programs that communicate across a network—common examples in the TCP/IP world are Web clients (browsers) and Web servers, and the FTP and Telnet clients and servers—but this book describes how to write our own network programs.

We write network programs using an *application program interface* or *API*. We describe two APIs for network programming:

1. sockets, sometimes called "Berkeley sockets" acknowledging their heritage from Berkeley Unix, and
2. XTI (X/Open Transport Interface), a slight modification of the Transport Layer Interface (TLI) developed by AT&T.

All the examples in the text are from the Unix operating system, although the foundation and concepts required for network programming are, to a large degree, operating system independent. The examples are also based on the TCP/IP protocol suite, both IP versions 4 and 6.

To write network programs one must understand the underlying operating system and the underlying networking protocols. This book builds on the foundation of my other four books in these two areas, and these books are abbreviated throughout this text as follows:

- APUE: *Advanced Programming in the UNIX Environment* [Stevens 1992],
- TCPv1: *TCP/IP Illustrated, Volume 1* [Stevens 1994],
- TCPv2: *TCP/IP Illustrated, Volume 2* [Wright and Stevens 1995], and
- TCPv3: *TCP/IP Illustrated, Volume 3* [Stevens 1996].

This second edition of *UNIX Network Programming* still contains information on both Unix and the TCP/IP protocols, but many references are made to these other four texts to allow interested readers to obtain more detailed information on various topics. This is especially the case for TCPv2, which describes and presents the actual 4.4BSD implementation of the network programming functions for the sockets API (`socket`, `bind`, `connect`, and so on). If one understands the implementation of a feature, the use of that feature in an application makes more sense.

Changes from the First Edition

This second edition is a complete rewrite of the first edition. These changes have been driven by the feedback I have received teaching this material about once a month during 1990–1996, and by following certain Usenet newsgroups during this same time, which lets one see the topics that are continually misunderstood. The following are the major changes with this new edition:

- This new edition uses ANSI C for all examples.
- The old Chapters 6 ("Berkeley Sockets") and 8 ("Library Routines") have been expanded into 25 chapters. Indeed this sevenfold expansion (based on a word count) of this material is probably the most significant change from the first to the second edition. Most of the individual sections in the old Chapter 6 have been expanded into an entire chapter with more examples added.
- The TCP and UDP portions from the old Chapter 6 have been separated and we now cover the TCP functions and a complete TCP client–server, followed by the UDP functions and a complete UDP client–server. This is easier for newcomers to understand than describing all the details of the `connect` function, for example, with its different semantics for TCP versus UDP.
- The old Chapter 7 ("System V Transport Layer Interface") has been expanded into seven chapters. We also cover the newer XTI instead of the TLI that it replaces.
- The old Chapter 2 ("The Unix Model") is gone. This chapter provided an overview of the Unix system in about 75 pages. In 1990 this chapter was needed because few books existed that adequately described the basic Unix programming interface, especially the differences between the Berkeley and System V implementations that existed in 1990. Today, however, more readers have a fundamental understanding of Unix, so concepts such as a process ID, password files, directories, and group IDs, need not be repeated. (My APUE book is a 700-page expansion of this material for readers desiring additional Unix programming details.)

Some of the advanced topics from the old Chapter 2 are covered in this new edition, but their coverage is moved to where the feature is used. For example, when showing our first concurrent server (Section 4.8) we cover the `fork` function. When we describe how to handle the `SIGCHLD` signal with our concurrent server (Section 5.9), we describe many additional features of Posix signal handling (zombies, interrupted system calls, etc.).

- Whenever possible this text describes the Posix interface. (We say more about the Posix family of standards in Section 1.10.) This includes not only the Posix.1 standard for the basic Unix functions (process control, signals, etc.), but also the forthcoming Posix.1g standard for the sockets and XTI networking APIs, and the 1996 Posix.1 standard for threads.

  The term "system call" has been changed to "function" when describing functions such as `socket` and `connect`. This follows the Posix convention that the distinction between a system call and a library function is an implementation detail that is often irrelevant for a programmer.

- The old Chapters 4 ("A Network Primer") and 5 ("Communication Protocols") have been replaced with Appendix A covering IP versions 4 (IPv4) and 6 (IPv6), and Chapter 2 covering TCP and UDP. This new material focuses on the protocol issues that network programmers are certain to encounter. The coverage of IPv6 was included, even though IPv6 implementations are just starting to appear, since during the lifetime of this text IPv6 will probably become the predominant networking protocol.

  I have found when teaching network programming that about 80% of all network programming problems have nothing to do with network programming, per se. That is, the problems are not with the API functions such as `accept` and `select`, but the problems arise from a lack of understanding of the underlying network protocols. For example, I have found that once a student understands TCP's three-way handshake and four-packet connection termination, many network programming problems are immediately understood.

  The old sections on XNS, SNA, NetBIOS, the OSI protocols, and UUCP have been removed, since it has become obvious during the early 1990s that these proprietary protocols have been eclipsed by the TCP/IP protocols. (UUCP is still popular and is not proprietary, but there is little we can show from a network programming perspective using UUCP.)

- The following new topics are covered in this second edition:
  - IPv4/IPv6 interoperability (Chapter 10),
  - protocol-independent name translation (Chapter 11),
  - routing sockets (Chapter 17),
  - multicasting (Chapter 19),
  - threads (Chapter 23),
  - IP options (Chapter 24),
  - datalink access (Chapter 26),

- client–server design alternatives (Chapter 27),
- virtual networks and tunneling (Appendix B), and
- network program debugging techniques (Appendix C).

Unfortunately, the coverage of the material from the first edition has been expanded so much that it no longer fits into a single book. Therefore at least two additional volumes are planned in the *UNIX Network Programming* series.

- Volume 2 will probably be subtitled *IPC: Interprocess Communication* and will be an expansion of the old Chapter 3, along with coverage of the 1996 Posix.1 real-time IPC mechanisms.
- Volume 3 will probably be subtitled *Applications* and will be an expansion of Chapters 9–18 of the first edition.

Even though most of the networking applications will be covered in Volume 3, a few special applications are covered in this volume: Ping, Traceroute, and `inetd`.

Readers

This text can be used as either a tutorial on network programming, or as a reference for experienced programmers. When used as a tutorial or for an introductory class on network programming, the emphasis should be on Part 2 ("Elementary Sockets," Chapters 3 through 9) followed by whatever additional topics are of interest. Part 2 covers the basic socket functions, for both TCP and UDP, along with I/O multiplexing, socket options, and basic name and address conversions. Chapter 1 should be read by all readers, especially Section 1.4, which describes some wrapper functions used throughout the text. Chapter 2 and perhaps Appendix A should be referred to as necessary, depending on the reader's background. Most of the chapters in Part 3 ("Advanced Sockets") can be read independently of the others in that part.

To aid in the use as a reference, a thorough index is provided, along with summaries on the end papers of where to find detailed descriptions of all the functions and structures. To help those reading topics in a random order, numerous references to related topics are provided throughout the text.

Although the sockets API has become the de facto standard for network programming, XTI is still used, sometimes with protocol suites other than TCP/IP. While the coverage of XTI in Part 4 is smaller than the coverage of sockets in Parts 2 and 3, much of the sockets coverage describes *concepts* that apply to XTI as well as sockets. For example, all of the concepts regarding the use of nonblocking I/O, broadcasting, multicasting, signal-driven I/O, out-of-band data, and threads, are the same, regardless of which API (sockets or XTI) is used. Indeed, many network programming problems are fundamentally similar, independent of whether the program is written using sockets or XTI, and there is hardly anything that can be done with one API that cannot be done with the other. The concepts are the same—just the function names and arguments change.

Source Code and Errata Availability

The source code for all the examples that appear in the book is available from my home page, listed at the end of the Preface. The best way to learn network programming is to take these programs, modify them, and enhance them. Actually writing code of this form is the *only* way to reinforce the concepts and techniques. Numerous exercises are also provided at the end of each chapter, and most answers are provided in Appendix E.

A current errata for the book is also available from my home page, listed at the end of the Preface.

Acknowledgments

Supporting every author is an understanding family, or nothing would ever get written! I am grateful to my family, Sally, Bill, Ellen, and David, first for their support and understanding when I wrote my first book (the first edition of this book), and for enduring this "small" revision. Their love, support, and encouragement helped make this book possible.

Numerous reviewers provided invaluable feedback (totaling 190 printed pages or 70,000 words), catching lots of errors, pointing out areas that needed more explanation, and suggesting alternative presentations, wording, and coding: Ragnvald Blindheim, Jim Bound, Gavin Bowe, Allen Briggs, Joe Doupnik, Wu-chang Feng, Bill Fenner, Bob Friesenhahn, Andrew Gierth, Wayne Hathaway, Kent Hofer, Sugih Jamin, Scott Johnson, Rick Jones, Mukesh Kacker, Marc Lampo, Marty Leisner, Jack McCann, Craig Metz, Bob Nelson, Evi Nemeth, John C. Noble, Steve Rago, Jim Reid, Chung-Shang Shao, Ian Lance Taylor, Ron Taylor, Andreas Terzis, and Dave Thaler. A special thanks to Sugih Jamin and his students in EECS 489 ("Computer Networks") at the University of Michigan who beta tested an early draft of the manuscript during the spring of 1997.

The following people answered email questions of mine, sometimes lots of questions, which improved the accuracy and presentation of the text: Dave Butenhof, Dave Hanson, Jim Hogue, Mukesh Kacker, Brian Kernighan, Vern Paxson, Steve Rago, Dennis Ritchie, Steve Summit, Paul Vixie, John Wait, Steve Wise, and Gary Wright.

A special thanks to Larry Rafsky and the wonderful team at Gari Software for handling lots of details and for many interesting technical discussions. Thank you, Larry, for everything.

Numerous individuals and their organizations went beyond the normal call of duty to provide either a loaner system, software, or access to a system, all of which were used to test some of the examples in the text.

- Meg McRoberts of SCO provided the latest releases of UnixWare, and Dion Johnson, Yasmin Kureshi, Michael Townsend, and Brian Ziel, provided support and answered questions.
- Mukesh Kacker of SunSoft provided access to a beta version of Solaris 2.6 and answered many questions about the Solaris TCP/IP implementation.

- Jim Bound, Matt Thomas, Mary Clouter, and Barb Glover of Digital Equipment Corp. provided an Alpha system and access to the latest IPv6 kits for Digital Unix.
- Michael Johnson of Red Hat Software provided the latest releases of Red Hat Linux.
- Steve Wise and Jessie Haug of IBM Austin provided an RS/6000 system and access to the latest IPv6 for AIX.
- Rick Jones of Hewlett-Packard provided access to a beta version of HP-UX 10.30 and he and William Gilliam answered many questions about it.

Many people helped with the Internet connectivity used throughout the text. My thanks once again to the National Optical Astronomy Observatories (NOAO), Sidney Wolff, Richard Wolff, and Steve Grandi, for providing access to their networks and hosts. Dave Siegel, Justus Addiss, and Paul Lucchina answered many questions, Phil Kaslo and Jim Davis provided an MBone connection, Ran Atkinson and Pedro Marques provided a 6bone connection, and Craig Metz provided lots of DNS help.

The staff at Prentice Hall, especially my editor Mary Franz, along with Noreen Regina, Sophie Papanikolaou, and Eileen Clark, have been a wonderful asset to a writer. Many thanks for letting me do so many things "my way."

As usual, but contrary to popular fads, I produced camera-ready copy of the book using the wonderful Groff package written by James Clark. I typed in all 291,972 words using the `vi` editor, created the 201 illustrations using the `gpic` program (using many of Gary Wright's macros), produced the 81 tables using the `gtbl` program, performed all the indexing, and did the final page layout. Dave Hanson's `loom` program and some scripts by Gary Wright were used to include the source code in the book. A set of `awk` scripts written by Jon Bentley and Brian Kernighan helped in producing the final index.

I welcome electronic mail from any readers with comments, suggestions, or bug fixes.

Tucson, Arizona
September 1997

W. Richard Stevens
rstevens@kohala.com
http://www.kohala.com/~rstevens

Part 1

Introduction and TCP/IP

1

Introduction

1.1 Introduction

Most network applications can be divided into two pieces: a *client* and a *server*. We can draw the communication link between the two as shown in Figure 1.1.

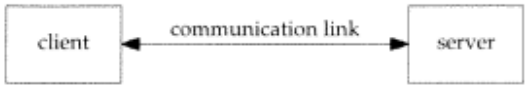

Figure 1.1 Network application: client and server.

There are numerous examples of clients and servers that most readers are probably familiar with: a Web browser (a client) communicating with a Web server; an FTP client fetching a file from an FTP server; a Telnet client that we use to log in to a remote host through a Telnet server on that remote host.

Clients normally communicate with one server at a time, although using the Web browser as an example, we might communicate with many different Web servers over, say, a 10-minute time period. But from the server's perspective at any given point in time it is not unusual for a server to be communicating with multiple clients. We show this in Figure 1.2. Later in this text we will cover several different ways for a server to handle multiple clients at the same time.

Although we think of the client application communicating with the server application, networking protocols are involved. In this text we focus on the TCP/IP protocol suite, also called the Internet protocol suite. For example, Web clients and servers communicate using the TCP protocol. TCP, in turn, uses the IP protocol, and IP communicates with a datalink layer of some form. For example, if the client and server are on the same Ethernet, we would have the arrangement shown in Figure 1.3.

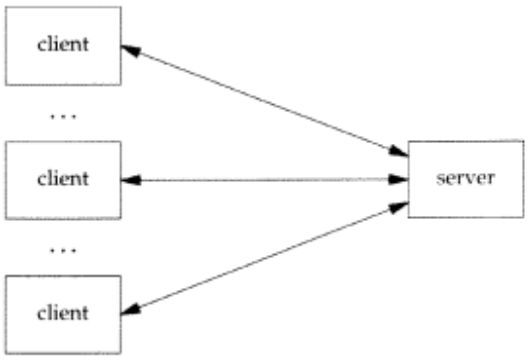

Figure 1.2 Server handling multiple clients at the same time.

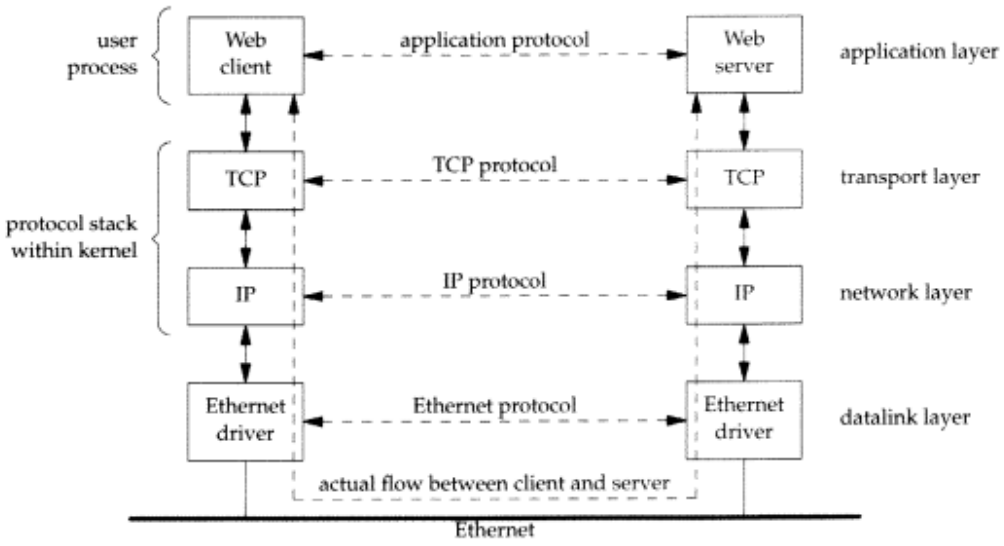

Figure 1.3 Client and server on the same Ethernet communicating using TCP.

Even though the client and server communicate using an application protocol, the transport layers communicate using TCP, and so on, we note that the actual flow of information between the client and server goes down the protocol stack on one side, across the network, and up the protocol stack on the other side.

We also note that the client and server are typically user processes, while the TCP and IP protocols are normally part of the protocol stack within the kernel. We have labeled the four layers on the right side of Figure 1.3.

TCP and IP are not the only protocols that we discuss. Some clients and servers use the UDP protocol instead of TCP and we discuss both protocols in more detail in Chapter 2. Furthermore, we have used the term "IP" but the protocol, which has been in use since the early 1980s, is officially called *IP version 4* (IPv4). A new version, *IP version 6* (IPv6) was developed during the mid-1990s and will probably replace IPv4 in the years

to come. Initial implementations of IPv6 were available at the time of this writing, and this text covers the development of network applications using both IPv4 and IPv6. Appendix A provides a comparison of IPv4 and IPv6, along with other protocols that we will encounter.

The client and server need not be attached to the same *local area network* (LAN) as we show in Figure 1.3. Instead, in Figure 1.4 we show the client and server on different LANs, with both LANs connected to a *wide area network* (WAN) using *routers*.

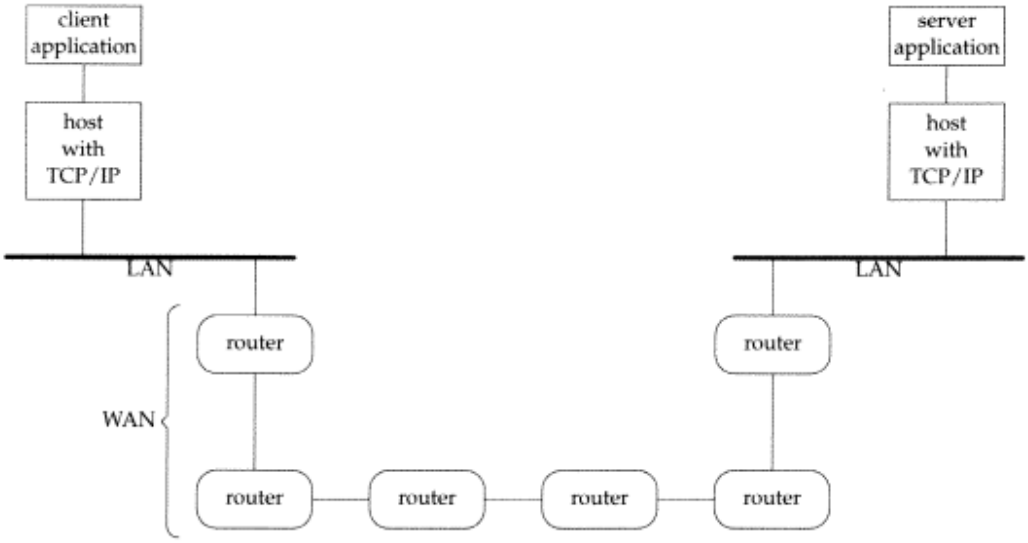

Figure 1.4 Client and server on different LANs connected through a WAN.

Routers are the building blocks of WANs. The largest WAN today is the *Internet*, although many companies build their own WANs and these private WANs may or may not be connected to the Internet.

The remainder of this chapter provides an introduction and overview to the various topics that are covered in detail later in the text. We start with a complete example of a TCP client, albeit a simple one, that demonstrates many of the function calls and concepts that we encounter throughout the text. This client works with IP version 4 only, and we show the changes required to work with IP version 6. A better solution is to write protocol-independent clients and servers, and we discuss this in Chapter 11. This chapter also shows a complete TCP server that works with our client.

To simplify all the code that we write, we define our own wrapper functions for most of the system functions that we call throughout the text. We can use these most of the time to check for an error, print an appropriate message, and terminate when an error occurs. We also show the test network, hosts, and routers used for most examples in the text, along with their hostnames, IP addresses, and operating systems.

Most discussions of Unix these days include the term *Posix*, which is the standard that most vendors have adopted. We describe the history of Posix and how it affects the APIs that we describe in this text, along with the other players in the standards area.

1.2 A Simple Daytime Client

Let us consider a specific example to introduce many of the concepts and terms that we will encounter throughout the book. Figure 1.5 is an implementation of a TCP time-of-day client. This client establishes a TCP connection with a server and the server simply sends back the current time and date in a human-readable format.

intro/daytimetcpcli.c

```
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd, n;
 6     char    recvline[MAXLINE + 1];
 7     struct sockaddr_in servaddr;

 8     if (argc != 2)
 9         err_quit("usage: a.out <IPaddress>");

10     if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
11         err_sys("socket error");

12     bzero(&servaddr, sizeof(servaddr));
13     servaddr.sin_family = AF_INET;
14     servaddr.sin_port = htons(13);  /* daytime server */
15     if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
16         err_quit("inet_pton error for %s", argv[1]);

17     if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
18         err_sys("connect error");

19     while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
20         recvline[n] = 0;        /* null terminate */
21         if (fputs(recvline, stdout) == EOF)
22             err_sys("fputs error");
23     }
24     if (n < 0)
25         err_sys("read error");

26     exit(0);
27 }
```

intro/daytimetcpcli.c

Figure 1.5 TCP daytime client.

This is the format that we use for all the source code in the text. Each nonblank line is numbered. The text describing portions of the code begins with the starting and ending line numbers in the left margin, as shown shortly. Sometimes the paragraph is preceded by a short descriptive bold heading, providing a summary statement of the code being described.

The horizontal rules at the beginning and end of the code fragment specify the source code filename: the file `daytimetcpcli.c` in the directory `intro` for this example. Since the source code for all the examples in the text is freely available (see the Preface), this lets you locate the appropriate source file. Compiling, running, and especially modifying these programs while reading this text is an excellent way to learn the concepts of network programming.

Throughout the text we will use indented, parenthetical notes such as this to describe implementation details and historical points.

If we compile the program into the default `a.out` file and execute it, we have the following output.

```
solaris % a.out 206.62.226.35          our input
Fri Jan 12 14:27:52 1996               the program's output
```

Whenever we display interactive input and output we show our typed input in a **`bold font`**, and the computer output `like this`. *Comments are added on the right side in italics.* We always include the name of the system as part of the shell prompt (`solaris` in this example) to show on which host the command was run. Figure 1.16 shows the systems used to run most of the examples in this book. The hostnames usually describe the operating system.

There are many details to now consider in this 27-line program. We mention them briefly here, in case this is your first encounter with a network program, and provide more information on these topics later in the text.

Include our own header

1 We include our own header, `unp.h`, which we show in Section D.1. This header includes numerous system headers that are needed by most network programs and defines various constants that we use (e.g., `MAXLINE`).

Command-line arguments

2-3 This is the definition of the `main` function along with the command-line arguments. We have written the code in this text assuming an ANSI (American National Standards Institute) C compiler.

Create a TCP socket

10-11 The `socket` function creates an Internet (`AF_INET`) stream (`SOCK_STREAM`) socket, which is a fancy name for a TCP socket. The function returns a small integer descriptor that we use to identify the socket in all future function calls (e.g., the calls to `connect` and `read` that follow).

The `if` statement contains a call to the `socket` function, an assignment of the return value to the variable named `sockfd`, and then a test of whether this assigned value is less than 0. While we could break this into two C statements,

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
```

it is a common C idiom to combine the two lines. The set of parentheses around the function call and assignment are required, given the precedence rules of C (the less-than operator has a higher precedence than assignment). As a personal style issue, the author always places a space between the two opening parentheses, as a visual indicator that the left-hand side of the comparison is also an assignment. (The author first saw this style in the Minix source code [Tanenbaum 1987] and has copied it ever since.) We use this same style in the `while` statement later in the program.

We will encounter many different uses of the term *socket*. First, the *application programming interface*, or API, that we are using is called the *sockets API*. In the preceding

paragraph we refer to a function named socket that is part of the sockets API. In the preceding paragraph we also refer to a "TCP socket," which is synonymous with a "TCP endpoint."

If the call to socket fails, we abort the program by calling our own err_sys function. It prints our error message along with a description of the system error that occurred (e.g., "Protocol not supported" is one possible error from socket) and terminates the process. This function, and a few others of our own that begin with err_, are called throughout the text. We describe them in Section D.4.

Specify server's IP address and port

12-16 We fill in an Internet socket address structure (a sockaddr_in structure named servaddr) with the server's IP address and port number. We set the entire structure to 0 using bzero, set the address family to AF_INET, set the port number to 13 (which is the well-known port of the daytime server on any TCP/IP host that supports this service, as shown in Figure 2.13), and set the IP address to the value specified as the first command-line argument (argv[1]). The IP address and port number fields in this structure must be in specific formats: we call the library function htons ("host to network short") to convert the binary port number, and we call the library function inet_pton ("presentation to numeric") to convert the ASCII command-line argument (such as 206.62.226.35 when we ran this example) into the proper format.

> bzero is not an ANSI C function. It is derived from early Berkeley networking code. Nevertheless, we use it throughout the text, instead of the ANSI C memset function, because bzero is easier to remember (with only two arguments) than memset (with three arguments). Almost every vendor that supports the sockets API also provides bzero, and if not, we provide a macro definition of it in our unp.h header.

> Indeed, the author made the mistake of swapping the second and third arguments to memset in 10 occurrences in the first printing of TCPv3. A C compiler cannot catch this error because both arguments are of the same type. (Actually, the second argument is an int and the third argument is size_t, which is typically an unsigned int, but the values specified, 0 and 16, respectively, are still OK for the other type of argument.) The call to memset still worked but did nothing: the number of bytes to initialize was specified as 0. The programs still worked, because only a few of the socket functions actually require that the final 8 bytes of an Internet socket address structure be set to 0. Nevertheless, it was an error, and one that can be avoided by using bzero, because swapping the two arguments to bzero will always be caught by the C compiler if function prototypes are used.

> This may be your first encounter with the inet_pton function. It is new with IP version 6 (which we talk more about in Appendix A). Older code uses the inet_addr function to convert an ASCII dotted-decimal string into the correct format, but this function has numerous limitations that inet_pton corrects. Do not worry if your system does not (yet) support this function; we provide an implementation of it in Section 3.7.

Establish connection with server

17-18 The connect function, when applied to a TCP socket, establishes a TCP connection with the server specified by the socket address structure pointed to by the second argument. We must also specify the length of the socket address structure as the third argument to connect, and for Internet socket address structures we always let the compiler calculate the length using C's sizeof operator.

In the `unp.h` header we `#define SA` to be `struct sockaddr`, that is, a generic socket address structure. Every time one of the socket functions requires a pointer to a socket address structure, that pointer must be cast to a pointer to a generic socket address structure. This is because the socket functions predate the ANSI C standard, so the `void *` pointer type was not available in the early 1980s when these functions were developed. The problem is that "`struct sockaddr`" is 15 characters and often causes the source code line to extend past the right edge of the screen (or page in the case of a book), so we shorten it to `SA`. We talk more about generic socket address structures with Figure 3.3.

Read and display server's reply

14–25 We `read` the server's reply and display the result using the standard I/O `fputs` function. We must be careful when using TCP because it is a *byte-stream* protocol with no record boundaries. The server's reply is normally a 26-byte string of the form

```
Fri Jan 12 14:27:52 1996\r\n
```

where `\r` is the ASCII carriage return and `\n` is the ASCII linefeed. With a byte-stream protocol these 26 bytes can be returned in numerous ways: a single TCP segment containing all 26 bytes of data, in 26 TCP segments each containing 1 byte of data, or any other combination that totals to 26 bytes. Normally a single segment containing all 26 bytes of data is returned, but with larger data sizes we cannot assume that the server's reply is returned by a single `read`. Therefore, when reading from a TCP socket we *always* need to code the `read` in a loop and terminate the loop when either `read` returns 0 (i.e., the other end closed the connection) or less than 0 (an error).

In this example the end of record is being denoted by the server closing the connection. This technique is also used by HTTP, the Hypertext Transfer Protocol. Other techniques are available. For example, FTP (File Transfer Protocol) and SMTP (Simple Mail Transfer Protocol) both mark the end of a record with the 2-byte sequence of an ASCII carriage return followed by an ASCII linefeed. Sun RPC (Remote Procedure Call) and the DNS (Domain Name System) place a binary count containing the record length in front of each record that is sent when using TCP. The important concept here is that TCP itself provides no record markers: if the application wants to delineate the end of records, it must do so itself and there are a few common ways to accomplish this.

Terminate program

26 `exit` terminates the program. Unix always closes all open descriptors when a process terminates, so our TCP socket is now closed.

As we mentioned, the text goes into much more detail on all the points that we just described.

1.3 Protocol Independence

Our program in Figure 1.5 is *protocol dependent* on IP version 4 (IPv4). We allocate and initialize a `sockaddr_in` structure, we set the family of this structure to `AF_INET`, and we specify the first argument to `socket` as `AF_INET`.

If we want to modify the program to work under IP version 6 (IPv6) we must change the code. Figure 1.6 shows a version that works under IPv6, with the changes highlighted in a bolder font.

intro/daytimetcpcliv6.c

```
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd, n;
 6     char    recvline[MAXLINE + 1];
 7     struct sockaddr_in6 servaddr;

 8     if (argc != 2)
 9         err_quit("usage: a.out <IPaddress>");

10     if ( (sockfd = socket(AF_INET6, SOCK_STREAM, 0)) < 0)
11         err_sys("socket error");

12     bzero(&servaddr, sizeof(servaddr));
13     servaddr.sin6_family = AF_INET6;
14     servaddr.sin6_port = htons(13);    /* daytime server */
15     if (inet_pton(AF_INET6, argv[1], &servaddr.sin6_addr) <= 0)
16         err_quit("inet_pton error for %s", argv[1]);

17     if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
18         err_sys("connect error");

19     while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
20         recvline[n] = 0;    /* null terminate */
21         if (fputs(recvline, stdout) == EOF)
22             err_sys("fputs error");
23     }
24     if (n < 0)
25         err_sys("read error");

26     exit(0);
27 }
```

intro/daytimetcpcliv6.c

Figure 1.6 Version of Figure 1.5 for IP version 6.

Only five lines are changed, but what we now have is another protocol-dependent program, this time dependent on IP version 6. It is better to make the program *protocol independent*. Figure 11.7 shows a version of this client that is protocol independent by using the `getaddrinfo` function (which is called by `tcp_connect`).

Another deficiency in our programs is that the user must enter the server's IP address as a dotted-decimal number (e.g., 206.62.226.35 for the IPv4 version). Humans work better with names instead of numbers (e.g., `laptop.kohala.com` or just `laptop`). In Chapters 9 and 11 we discuss the functions that convert between hostnames and IP addresses, and between service names and ports. We purposely put off the discussion of these functions and continue using IP addresses and port numbers so we know exactly what goes into the socket address structures that we must fill in and examine. This also avoids complicating our discussion of network programming with the details of yet another set of functions.

1.4 Error Handling: Wrapper Functions

In any real-world program it is essential to check *every* function call for an error return. In Figure 1.5 we check for errors from `socket`, `inet_pton`, `connect`, `read`, and `fputs`, and when one occurs we call our own functions `err_quit` and `err_sys` to print an error message and terminate the program. We find that most of the time this is what we want to do. Occasionally we want to do something other than terminate when one of these functions returns an error, as in Figure 5.12, when we must check for an interrupted system call.

Since terminating on an error is the common case, we can shorten our programs by defining a *wrapper function* that performs the actual function call, tests the return value, and terminates on an error. The convention we use is to capitalize the name of the function, as in

```
sockfd = Socket(AF_INET, SOCK_STREAM, 0);
```

Our wrapper function is shown in Figure 1.7.

lib/wrapsock.c

```
172 int
173 Socket(int family, int type, int protocol)
174 {
175     int     n;

176     if ( (n = socket(family, type, protocol)) < 0)
177         err_sys("socket error");
178     return (n);
179 }
```

lib/wrapsock.c

Figure 1.7 Our wrapper function for the `socket` function.

> *Whenever you encounter a function name in the text that begins with an uppercase letter, that is a wrapper function of our own. It calls a function whose name is the same but beginning with the lowercase letter.*
>
> *When describing the source code that is presented in the text, we always refer to the lowest level function being called (e.g.,* `socket`*) and not the wrapper function (e.g.,* `Socket`*).*

While these wrapper functions might not seem like a big savings, when we discuss threads in Chapter 23 we will find that the thread functions do not set the standard Unix `errno` variable when an error occurs; instead the `errno` value is the return value of the function. This means that every time we call one of the pthread functions we must allocate a variable, save the return value in that variable, and then set `errno` to this value before calling `err_sys`. To avoid cluttering the code with braces, we can use C's comma operator to combine the assignment into `errno` and the call of `err_sys` into a single statement, as in the following:

```
int    n;

if ( (n = pthread_mutex_lock(&ndone_mutex)) != 0)
    errno = n, err_sys("pthread_mutex_lock error");
```

Alternately we could define a new error function that takes the system's error number as an argument. But we can make this piece of code much easier to read as just

```
Pthread_mutex_lock(&ndone_mutex);
```

by defining our own wrapper function, shown in Figure 1.8.

lib/wrappthread.c

```
72 void
73 Pthread_mutex_lock(pthread_mutex_t *mptr)
74 {
75     int     n;

76     if ( (n = pthread_mutex_lock(mptr)) == 0)
77         return;
78     errno = n;
79     err_sys("pthread_mutex_lock error");
80 }
```

lib/wrappthread.c

Figure 1.8 Our wrapper function for `pthread_mutex_lock`.

With careful C coding we could use macros instead of functions, providing a little run-time efficiency, but these wrapper functions are rarely, if ever, the performance bottleneck of a program.

Our choice of capitalizing the first character of the function name is a compromise. Many other styles were considered: prefixing the function name with an `e` (as done on p. 182 of [Kernighan and Pike 1984]), appending `_e` to the function name, and so on. Our style seems the least distracting while still providing a visual indication that some other function is really being called.

This technique has the side benefit of checking for errors from functions whose error returns are often ignored: `close` and `listen`, for example.

Throughout the rest of this book we will use these wrapper functions unless we need to check for an explicit error and handle it in some form other than terminating the process. We do not show the source code for all our wrapper functions, but the code is freely available (see the Preface).

Unix `errno` Value

When an error occurs in a Unix function (such as one of the socket functions), the global variable `errno` is set to a positive value indicating the type of error and the function normally returns –1. Our `err_sys` function looks at the value of `errno` and prints the corresponding error message string (e.g., "Connection timed out" if `errno` equals `ETIMEDOUT`).

The value of `errno` is set by a function only if an error occurs. Its value is undefined if the function does not return an error. All of the positive error values are constants with an all uppercase name beginning with `E` and are normally defined in the `<sys/errno.h>` header. No error has the value of 0.

Storing `errno` in a global variable does not work with multiple threads that share all global variables. We talk about solutions to this problem in Chapter 23.

Throughout the text we use phrases of the form "the `connect` function returns `ECONNREFUSED`" as shorthand to mean that the function returns an error (typically a return value of –1) with `errno` set to the specified constant.

1.5 A Simple Daytime Server

We can also write a simple version of a TCP daytime server, which will work with the client from Section 1.2. We use the wrapper functions that we described in the previous section and show this server in Figure 1.9.

intro/daytimetcpsrv.c

```
 1 #include    "unp.h"
 2 #include    <time.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     listenfd, connfd;
 7     struct sockaddr_in servaddr;
 8     char    buff[MAXLINE];
 9     time_t  ticks;

10     listenfd = Socket(AF_INET, SOCK_STREAM, 0);

11     bzero(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
14     servaddr.sin_port = htons(13);  /* daytime server */

15     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

16     Listen(listenfd, LISTENQ);

17     for ( ; ; ) {
18         connfd = Accept(listenfd, (SA *) NULL, NULL);

19         ticks = time(NULL);
20         snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
21         Write(connfd, buff, strlen(buff));

22         Close(connfd);
23     }
24 }
```

intro/daytimetcpsrv.c

Figure 1.9 TCP daytime server.

Create a TCP socket

10 The creation of the TCP socket is identical to the client code.

Bind server's well-known port to socket

11-15 The server's well-known port (13 for the daytime service) is bound to the socket by filling in an Internet socket address structure and calling `bind`. We specify the IP address as `INADDR_ANY`, which allows the server to accept a client connection on any interface, in case the server host has multiple interfaces. Later we will see how we can restrict the server to accepting a client connection on just a single interface, if we so desire.

Convert socket to listening socket

16 By calling `listen` the socket is converted into a listening socket, on which incoming connections from clients will be accepted by the kernel. These three steps, `socket`, `bind`, and `listen`, are the normal steps for any TCP server to prepare what we call the *listening descriptor* (`listenfd` in this example).

The constant `LISTENQ` is from our `unp.h` header. It specifies the maximum number of client connections that the kernel will queue for this listening descriptor. We say much more about this queueing in Section 4.5.

Accept client connection, send reply

17-21 Normally the server process is put to sleep in the call to `accept`, waiting for a client connection to arrive and be accepted. A TCP connection uses what is called a *three-way handshake* to establish a connection and when this handshake completes, `accept` returns, and the return value from the function is a new descriptor (`connfd`) that is called the *connected descriptor*. This new descriptor is used for communication with the new client. A new descriptor is returned by `accept` for each client that connects to our server.

> The style used throughout the book for an infinite loop is
>
> ```
> for ( ; ; ) {
>     . . .
> }
> ```

The current time and date is returned by the library function `time`, which returns the number of seconds since the Unix Epoch: 00:00:00 January 1, 1970, UTC (Coordinated Universal Time). The next library function, `ctime`, converts this integer value into a human-readable string such as

```
Fri Jan 12 14:27:52 1996
```

A carriage return and linefeed are appended to the string by `snprintf` and the result is written to the client by `write`.

> This may be your first encounter with `snprintf`. Lots of existing code calls `sprintf` instead, but `sprintf` cannot check for overflow of the destination buffer. `snprintf`, on the other hand, requires that the second argument be the size of the destination buffer, and this buffer will not be overflowed.

`snprintf` is not yet part of the ANSI C standard but is being considered for a revision of the standard, currently called *C9X*. Nevertheless, many vendors are providing it as part of the standard C library. We use `snprintf` throughout the text, providing our own version that just calls `sprintf` when it is not provided.

It is remarkable how many network break-ins have occurred by a hacker sending data to cause a server's call to `sprintf` to overflow its buffer. Other functions that we should be careful with are `gets`, `strcat`, and `strcpy`, normally calling `fgets`, `strncat`, and `strncpy` instead. Additional tips on writing secure network programs are in Chapter 23 of [Garfinkel and Spafford 1996].

Terminate connection

The server closes its connection with the client by calling `close`. This initiates the normal TCP connection termination sequence: a FIN is sent in each direction and each FIN is acknowledged by the other end. We say much more about TCP's three-way handshake and the four TCP packets used to terminate a TCP connection in Section 2.5.

As with the client in the previous section, we have only examined this server briefly, saving all the details for later in the book. Note the following points:

- As with the client, the server is protocol dependent on IPv4. We will show a protocol-independent version in Figure 11.9 that uses the `getaddrinfo` function.
- Our server handles only one client at a time. If multiple client connections arrive at about the same time, the kernel queues them, up to some limit, and returns them to `accept` one at a time. This daytime server, which requires calling two library functions, `time` and `ctime`, is quite fast. But if the server took more time to service each client (say a few seconds or a minute), we would need some way to overlap the service of one client with another client. The server that we show in Figure 1.9 is called an *iterative server*, because it iterates through each client, one at a time. There are numerous techniques for writing a *concurrent server*, one that handles multiple clients at the same time. The simplest technique for a concurrent server is to call the Unix `fork` function (Section 4.7), creating one child process for each client. Other techniques are to use threads instead of `fork` (Section 23.4) or to pre-`fork` a fixed number of children when the server starts (Section 27.6).
- If we start a server like this from a shell command line, we might want the server to run for a long time, since servers often run for as long as the system is up. This requires that we add code to the server to run correctly as a Unix *daemon*: a process that can run in the background, unattached to a terminal. We cover this in Section 12.4.

1.6 Road Map to Client–Server Examples in the Text

Two client–server examples are used predominantly throughout the text to illustrate the various techniques used in network programming:

- a daytime client–server (which we started in Figures 1.5, 1.6, and 1.9), and
- an echo client–server (which starts in Chapter 5).

To provide a road map for the different topics that are covered in this text, we summarize the programs that we develop, and the starting figure number and page number in which the source code appears. Figure 1.10 lists the versions of the daytime client, two versions of which we have already seen. Figure 1.11 lists the versions of the daytime server. Figure 1.12 lists the versions of the echo client and Figure 1.13 lists the versions of the echo server.

Figure	Page	Description
1.5	6	TCP/IPv4, protocol dependent
1.6	10	TCP/IPv6, protocol dependent
9.8	253	TCP/IPv4, protocol dependent, calls `gethostbyname` and `getservbyname`
11.7	287	TCP, protocol independent, calls `getaddrinfo` and `tcp_connect`
11.12	295	UDP, protocol independent, calls `getaddrinfo` and `udp_client`
15.11	411	TCP, uses nonblocking `connect`
28.13	779	TCP/IPv4, XTI, protocol dependent
29.7	795	TCP, XTI, protocol independent, calls `netdir_getbyname` and `tcp_connect`
31.3	823	UDP, XTI, protocol independent, calls `netdir_getbyname` and `udp_client`
31.4	826	UDP, XTI, protocol independent, receives asynchronous errors
31.7	830	UDP, XTI, protocol independent, reads datagrams in pieces
33.8	857	TCP, protocol dependent, uses TPI instead of sockets or XTI
E.1	929	TCP, protocol dependent, generates `SIGPIPE`
E.5	932	TCP, protocol dependent, prints socket receive buffer sizes and MSS
E.13	942	TCP, protocol dependent, allows hostname (`gethostbyname`) or IP address
E.14	943	TCP, protocol independent, allows hostname (`gethostbyname`)

Figure 1.10 Different versions of the daytime client developed in the text.

Figure	Page	Description
1.9	13	TCP/IPv4, protocol dependent
11.9	290	TCP, protocol independent, calls `getaddrinfo` and `tcp_listen`
11.10	292	TCP, protocol independent, calls `getaddrinfo` and `tcp_listen`
11.15	298	UDP, protocol independent, calls `getaddrinfo` and `udp_server`
12.5	338	TCP, protocol independent, runs as stand-alone daemon
12.12	345	TCP, protocol independent, spawned from `inetd` daemon
30.5	805	TCP, XTI, protocol independent, calls `netdir_getbyname` and `tcp_listen`
31.6	828	UDP, XTI, protocol independent, calls `netdir_getbyname` and `udp_server`

Figure 1.11 Different versions of the daytime server developed in the text.

Figure	Page	Description
5.4	114	TCP/IPv4, protocol dependent
6.9	157	TCP, uses `select`
6.13	162	TCP, uses `select` and works in a batch mode
8.7	216	UDP/IPv4, protocol dependent
8.9	219	UDP, verifies server's address
8.17	227	UDP, calls `connect` to obtain asynchronous errors
13.2	352	UDP, timeout when reading server's reply using `SIGALRM`
13.4	354	UDP, timeout when reading server's reply using `select`
13.5	355	UDP, timeout when reading server's reply using `SO_RCVTIMEO`
14.4	380	Unix domain stream, protocol dependent
14.6	381	Unix domain datagram, protocol dependent
15.3	400	TCP, uses nonblocking I/O
15.9	408	TCP, uses two processes (`fork`)
15.21	423	TCP, establishes connection then sends RST
18.5	476	UDP, broadcasts with race condition
18.6	479	UDP, broadcasts with race condition
18.7	481	UDP, broadcasts, race condition fixed by using `pselect`
18.9	483	UDP, broadcasts, race condition fixed by using `sigsetjmp` and `siglongjmp`
18.10	485	UDP, broadcasts, race condition fixed by using IPC from signal handler
20.6	545	UDP, reliable using timeout, retransmit, and sequence number
21.14	583	TCP, heartbeat test to server using out-of-band data
23.2	606	TCP, uses two threads
24.6	642	TCP/IPv4, specifies a source route

Figure 1.12 Different versions of the echo client developed in the text.

Figure	Page	Description
5.2	113	TCP/IPv4, protocol dependent
5.12	128	TCP/IPv4, protocol dependent, reaps terminated children
6.21	165	TCP/IPv4, protocol dependent, uses `select`, one process handles all clients
6.25	172	TCP/IPv4, protocol dependent, uses `poll`, one process handles all clients
8.3	214	UDP/IPv4, protocol dependent
8.24	234	TCP and UDP/IPv4, protocol dependent, uses `select`
13.14	367	TCP, uses standard I/O library
14.3	379	Unix domain stream, protocol dependent
14.5	380	Unix domain datagram, protocol dependent
14.15	393	Unix domain stream, with credential passing from client
20.4	537	UDP, receive destination address and received interface; truncated datagrams
20.15	554	UDP, bind all interface addresses
21.15	585	TCP, heartbeat test to client using out-of-band data
22.4	594	UDP, uses signal-driven I/O
23.3	607	TCP, one thread per client
23.4	610	TCP, one thread per client, portable argument passing
24.6	642	TCP/IPv4, prints received source route
25.30	689	UDP, uses `icmpd` to receive asynchronous errors
E.17	955	UDP, bind all interface addresses

Figure 1.13 Different versions of the echo server developed in the text.

1.7 OSI Model

A common way to describe the layers in a network is the International Organization for Standardization (ISO) *open systems interconnection* model (OSI) for computer communications. This is a seven-layer model, which we show in Figure 1.14, along with the approximate mapping to the Internet protocol suite.

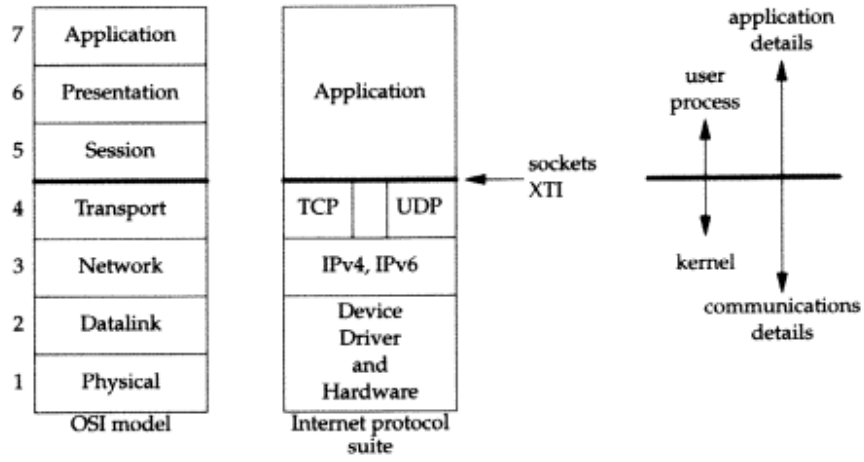

Figure 1.14 Layers in OSI model and Internet protocol suite.

We consider the bottom two layers of the OSI model as the device driver and networking hardware that are supplied with the system. Normally we need not concern ourselves with these layers other than being aware of some properties of the datalink, such as the 1500-byte Ethernet MTU (which we describe in Section 2.9).

The network layer is handled by the IPv4 and IPv6 protocols, both of which we describe in Appendix A. The transport layers that we can choose from are TCP and UDP, and we describe these in Chapter 2. We show a gap between TCP and UDP in Figure 1.14 to indicate that it is possible for an application to bypass the transport layer and use IPv4 or IPv6 directly. This is called a *raw socket* and we talk about this in Chapter 25.

The upper three layers of the OSI model are combined into a single layer called the application. This is the Web client (browser), Telnet client, the Web server, the FTP server, or whatever application we are using. With the Internet protocols there is rarely any distinction between the upper three layers of the OSI model.

The two programming interfaces that we describe in this book, sockets and XTI, are interfaces from the upper three layers (the "application") into the transport layer. This is the focus of this book: how to write applications using either sockets or XTI that use either TCP or UDP. We already mentioned raw sockets and in Chapter 26 we will see that we can even bypass the IP layer completely to read and write our own datalink layer frames.

Why do both sockets and XTI provide the interface from the upper three layers of the OSI model into the transport layer? There are two reasons for this design, which we note on the right side of Figure 1.14. First, the upper three layers handle all the details of the application (FTP, Telnet, or HTTP, for example) and know little about the communication details. The lower four layers know little about the application but handle all the communication details: sending data, waiting for an acknowledgment, sequencing data that arrives out of order, calculating and verifying checksums, and so on. The second reason is that the upper three layers often form what is called a *user process* while the lower four layers are normally provided as part of the operating system kernel. Unix provides this separation between the user process and the kernel, as do many other contemporary operating systems. Therefore the interface between layers 4 and 5 is the natural place to build the application programming interface (API).

1.8 BSD Networking History

The sockets API originated with the 4.2BSD system, released in 1983. Figure 1.15 shows the development of the various BSD releases, noting the major TCP/IP developments. A few changes to the sockets API also took place in 1990 with the 4.3BSD Reno release, when the OSI protocols went into the BSD kernel.

The path down the page from 4.2BSD through 4.4BSD are the releases from the Computer Systems Research Group (CSRG) at Berkeley that required the recipient to already have a source code license for Unix. But all of the networking code, both the kernel support (such as the TCP/IP and Unix domain protocol stacks and the socket interface), along with the applications (such as the Telnet and FTP clients and servers), were developed independently from the AT&T-derived Unix code. Therefore starting in 1989 Berkeley provided the first of the BSD networking releases, which contained all of the networking code and various other pieces of the BSD system that were not constrained by the Unix source code license. These releases were "publicly available" and eventually available by anonymous FTP to anyone on the Internet.

The final releases from Berkeley were 4.4BSD-Lite in 1994 and 4.4BSD-Lite2 in 1995. We note that these two releases were then used as the base for other systems: BSD/OS, FreeBSD, NetBSD, and OpenBSD, all four of which are still being actively developed and enhanced. More information on the various BSD releases, and on the history of the various Unix systems in general, can be found in Chapter 1 of [McKusick et al. 1996].

Many Unix systems started with some version of the BSD networking code, including the sockets API, and we refer to these implementations as *Berkeley-derived implementations*. Many commercial versions of Unix are based on System V Release 4 (SVR4) and some of these have Berkeley-derived networking code (e.g., UnixWare 2.x), while the networking code in other SVR4 systems has been independently derived (e.g., Solaris 2.x). We also note that the Linux system, a popular, freely available implementation of Unix, does *not* fit into the Berkeley-derived classification: its networking code and sockets API were developed from scratch.

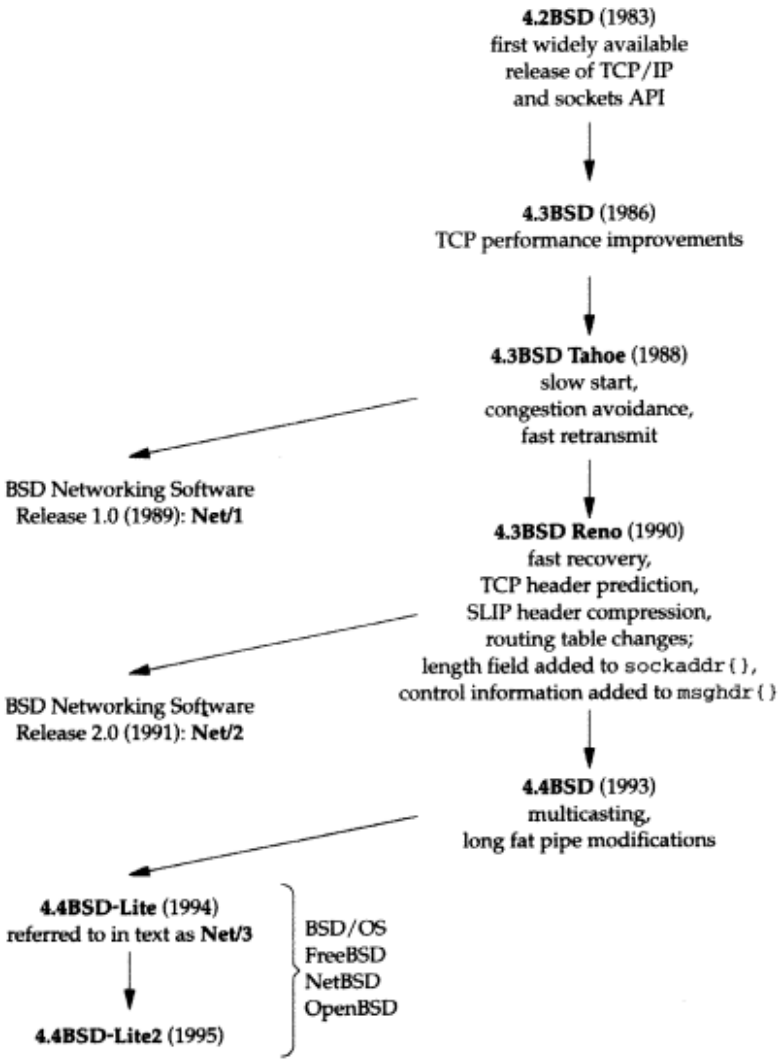

Figure 1.15 History of various BSD releases.

1.9 Test Networks and Hosts

Figure 1.16 shows the various networks and hosts used in the examples throughout the text. For each host we show the operating system and the type of hardware (since some of the operating systems run on more than one type of hardware). The name within each box is the hostname that appears in the text.

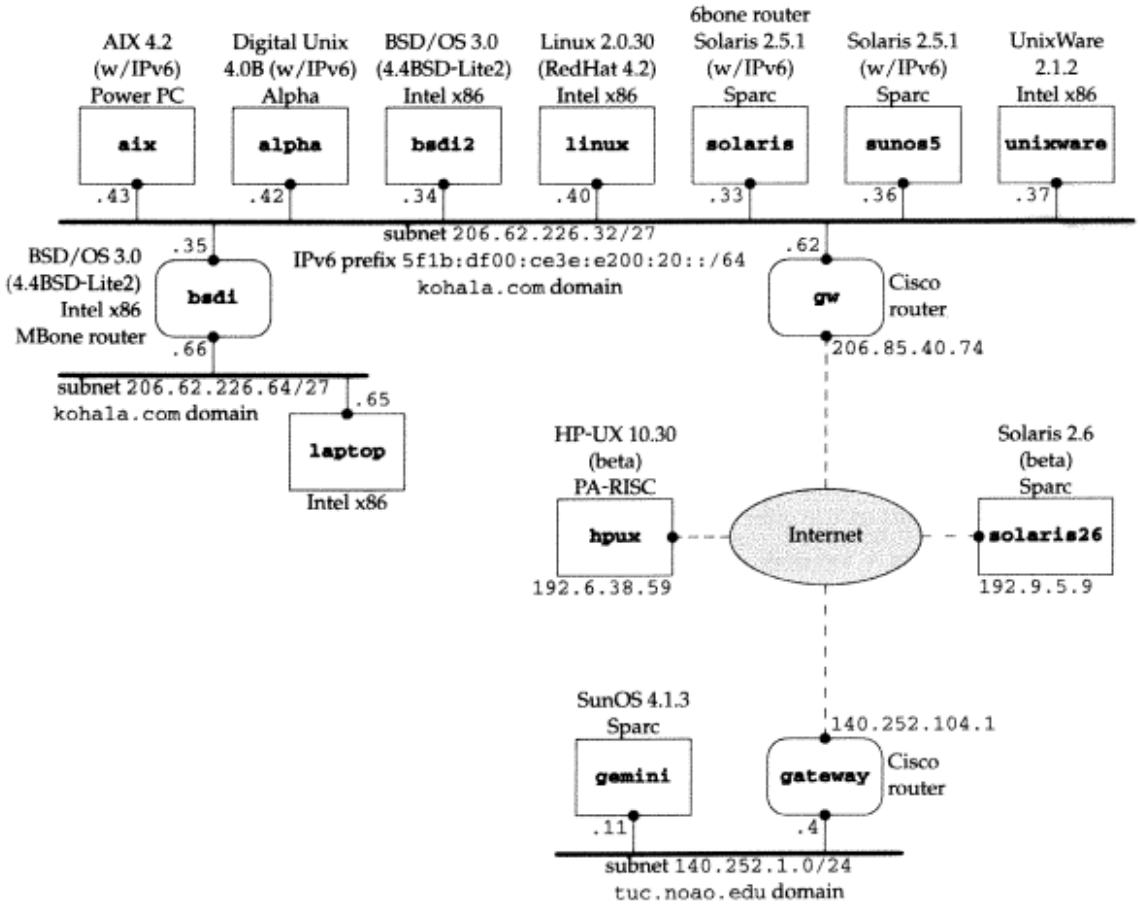

Figure 1.16 Networks and hosts used for most examples in the text.

The hosts on the top two Ethernets with the subnet addresses 206.62.226.32/27 and 206.62.226.64/27 are all in the `kohala.com` domain. The hosts on the bottom Ethernet with the subnet address 140.252.1.0/24 are all in the `tuc.noao.edu` domain, which is run by the National Optical Astronomy Observatories. The notation /27 and /24 indicates the number of consecutive bits starting from the leftmost bit of the address used to identify the network and subnet. Figures A.6 and A.5 show these two IPv4 addresses in more detail and Section A.4 talks about the /n notation used today to designate subnet boundaries.

Also in Figure 1.16 we draw nodes that function as routers with rounded corners, and nodes that are only hosts with square corners. We follow this convention throughout the book, as sometimes the distinction between a host and a router is important.

We note that the real name of the Sun operating system is SunOS 5.x and not Solaris 2.x, but everyone calls it Solaris.

Discovering Network Topology

We show the network topology in Figure 1.16 for the hosts used for the examples throughout this text, but you need to know your own network topology to run the examples and exercises on your own network. Although there are no current Unix standards with regard to network configuration and administration, two basic commands are provided by most Unix systems and can be used to discover details of a network: `netstat` and `ifconfig`. We show examples on some different systems from Figure 1.16. Check the manual pages for these commands on your system to see the details on the information that is output. Also be aware that some vendors place these commands in an administrative directory, such as `/sbin` or `/usr/sbin`, instead of the normal `/usr/bin`, and these directories might not be in your normal shell search path (`PATH`).

1. `netstat -i` provides information on the interfaces. We also specify the `-n` flag to print numeric addresses, instead of trying to find names for the networks. This shows us the interfaces and their names.

```
linux % netstat -ni
Kernel Interface table
Iface   MTU Met  RX-OK RX-ERR RX-DRP RX-OVR  TX-OK TX-ERR TX-DRP TX-OVR Flags
lo     3584   0     32      0      0      0     32      0      0      0 BLRU
eth0   1500   0 483929      0      0      0 449881      0      0      0 BRU
```

   The loopback interface is called `lo` and the Ethernet is called `eth0`. The next example shows a host with IPv6 support.

```
alpha % netstat -ni
Name  Mtu   Network     Address                 Ipkts Ierrs    Opkts Oerrs Coll
ln0   1500  <Link>      08:00:2b:37:64:26       11220     0     4893     0    4
ln0   1500  DLI         none                    11220     0     4893     0    4
ln0   1500  206.62.226. 206.62.226.42            11220     0     4893     0    4
ln0   1500  IPv6 FE80::800:2B37:6426            11220     0     4893     0    4
ln0   1500  IPv6 5F1B:DF00:CE3E:E200:20:800:2B37:6426 11220 0 4893     0    4
lo0   1536  <Link>      Link#3                  12432     0    12432     0    0
lo0   1536  127         127.0.0.1               12432     0    12432     0    0
lo0   1536  IPv6        ::1                     12432     0    12432     0    0
tun0  576   <Link>      Link#4                      0     0        0     0    0
tun0  576   IPv6        ::206.62.226.42             0     0        0     0    0
```

2. `netstat -r` shows the routing table, which is another way to determine the interfaces. We normally specify the `-n` flag to print numeric addresses. This also shows the IP address of the default router.

```
aix % netstat -rn
Routing tables
Destination      Gateway           Flags   Refs   Use    MTU   Netif Expire

Route tree for Protocol Family 2 (Internet):
default          206.62.226.62     UG        0     0      -    en0
127/8            127.0.0.1         U         0     0      -    lo0
206.62.226.32/27 206.62.226.43     U         4   475      -    en0
```

```
Route tree for Protocol Family 24 (Internet v6):
::/96              0.0.0.0              UC       0      0  1480    sit0   - =>
default       fe80::2:0:800:2078:e3e3 UG       0      0     -     en0
::1                ::1                  UH       0      0 16896    lo0
5f1b:df00:ce3e:e200:20::/80
                   link#2               UC       0      0  1500    en0    -
fe80::/16          link#2               UC       0      0  1500    en0    -
fe80::2:0:800:2078:e3e3
                   link#2               UHDL     1      0  1500    en0    -
ff01::/16          ::1                  U        0      0     -     lo0
ff02::/16     fe80::800:5afc:2b36       U        1      3  1500    en0
ff11::/16          ::1                  U        0      0     -     lo0
ff12::/16     fe80::800:5afc:2b36       U        0      0  1500    en0
```

(We have wrapped some of the longer lines to align the output fields.)

3. Given the interface names, we execute `ifconfig` to obtain the details for each interface.

```
linux % ifconfig eth0
eth0 Link encap:10Mbps Ethernet  HWaddr 00:A0:24:9C:43:34
     inet addr:206.62.226.40  Bcast:206.62.226.63  Mask:255.255.255.224
     UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
     RX packets:484461 errors:0 dropped:0 overruns:0
     TX packets:450113 errors:0 dropped:0 overruns:0
     Interrupt:10 Base address:0x300
```

This shows the IP address, subnet mask, and broadcast address. The `MULTICAST` flag is often an indication that the host supports multicasting.

```
alpha % ifconfig ln0
ln0: flags=c63<UP,BROADCAST,NOTRAILERS,RUNNING,MULTICAST,SIMPLEX>
     inet 206.62.226.42 netmask ffffffe0 broadcast 206.62.226.63 ipmtu 1500
```

Some implementations provide a `-a` flag that prints the information on all configured interfaces.

4 One way to find the IP address of many hosts on the local network is to `ping` the broadcast address (which we found in the previous step).

```
bsdi % ping 206.62.226.63
PING 206.62.226.63 (206.62.226.63): 56 data bytes
64 bytes from 206.62.226.35: icmp_seq=0 ttl=255 time=0.316 ms
64 bytes from 206.62.226.40: icmp_seq=0 ttl=64 time=1.369 ms (DUP!)
64 bytes from 206.62.226.34: icmp_seq=0 ttl=255 time=1.822 ms (DUP!)
64 bytes from 206.62.226.42: icmp_seq=0 ttl=64 time=2.27 ms (DUP!)
64 bytes from 206.62.226.37: icmp_seq=0 ttl=64 time=2.717 ms (DUP!)
64 bytes from 206.62.226.33: icmp_seq=0 ttl=255 time=3.281 ms (DUP!)
64 bytes from 206.62.226.62: icmp_seq=0 ttl=255 time=3.731 ms (DUP!)
^?                                        type our interrupt key (DEL)
--- 206.62.226.63 ping statistics ---
1 packets transmitted, 1 packets received, +6 duplicates, 0% packet loss
round-trip min/avg/max = 0.316/2.215/3.731 ms
```

1.10 Unix Standards

Most activity these days with regard to Unix standardization is being done by Posix and The Open Group.

POSIX

Posix is an acronym for "Portable Operating System Interface." Posix is not a single standard, but a family of standards being developed by the Institute for Electrical and Electronics Engineers, Inc., normally called the *IEEE*. The Posix standards are also being adopted as international standards by ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission), called ISO/IEC.

The first of the Posix standards was IEEE Std 1003.1–1988 (317 pages) and it specified the C language interface into a Unix-like kernel covering the following areas: process primitives (`fork`, `exec`, signals, timers), the environment of a process (user IDs, process groups), files and directories (all the I/O functions), terminal I/O, the system databases (password file and group file), and the `tar` and `cpio` archive formats.

> The first Posix standard was a trial use version in 1986 known as "IEEEIX." The name Posix was suggested by Richard Stallman.

This standard was updated in 1990 by IEEE Std 1003.1–1990 (356 pages), which was also International Standard ISO/IEC 9945-1: 1990. Minimal changes were made from the 1988 to the 1990 version. Appended to the title was "Part 1: System Application Program Interface (API) [C Language]" indicating that this standard was the C language API.

The next of the Posix standards was IEEE Std 1003.2–1992, and its title contained "Part 2: Shell and Utilities." It was published in two volumes, totaling about 1300 pages. This part defines the shell (based on the System V Bourne shell) and about 100 utilities (programs normally executed from a shell, from `awk` and `basename` to `vi` and `yacc`). Throughout this text we refer to this standard as *Posix.2*.

Next came IEEE Std 1003.1b–1993, formerly known as IEEE P1003.4. This was an update to the 1003.1–1990 standard to include the realtime extensions developed by the P1003.4 working group. The 1003.1b–1993 standard added the following items to the 1990 standard: file synchronization, asynchronous I/O, semaphores, memory management (`mmap` and shared memory), execution scheduling, clocks and timers, and message queues. The 1003.1b–1993 standard totaled 590 pages.

The next Posix standard was IEEE Std 1003.1, 1996 Edition [IEEE 1996], which includes 1003.1–1990 (the base API), 1003.1b–1993 (realtime extensions), 1003.1c–1995 (pthreads), and 1003.1i–1995 (technical corrections to 1003.1b). This standard is also called ISO/IEC 9945-1: 1996. Three chapters on threads were added for a total size of 743 pages. Throughout this text we refer to this standard as *Posix.1*.

> Over one-quarter of the 743 pages are an appendix titled "Rationale and Notes." This rationale contains historical information and reasons why certain features were included or omitted. Often the rationale is as informative as the official standard.

This standard also contains a Foreword stating that ISO/IEC 9945 consists of the following parts:

- Part 1: System application program interface (API) [C language],
- Part 2: Shell and utilities, and
- Part 3: System administration (under development).

The Posix work that affects most of this book is IEEE Std 1003.1g: Protocol Independent Interfaces (PII), a product of the P1003.1g working group. This is the networking API standard and it defines two APIs, which it calls DNIs (Detailed Network Interfaces):

1. DNI/Socket, based on the 4.4BSD sockets API.
2. DNI/XTI, based on the X/Open XPG4 specification.

Work on this standard started in the late 1980s as the P1003.12 working group (later renamed P1003.1g), but as of this writing, the standard is not complete (but getting close!). Draft 6.4 (May 1996) was the first draft to obtain more than 75% approval from the balloting group. Draft 6.6 (March 1997) appears to be the final draft [IEEE 1997a]. Sometime in 1998 or 1999 a new version of IEEE Std 1003.1 should be printed to include the P1003.1g standard.

Even though the P1003.1g standard is not officially complete, this book uses the features from Draft 6.6 of this standard whenever possible. Throughout this text we refer to this draft as *Posix.1g*. For example, the third argument to the `connect` function (Section 4.3) is shown as a `socklen_t` datatype, even though this is new with Posix.1g. Similarly we describe the new Posix.1g `sockatmark` function (Section 21.3) and provide an implementation of it using the `ioctl` function. We also use the Posix.1g protocol value of `AF_LOCAL` instead of `AF_UNIX` for Unix domain sockets. Differences between current practice and Posix.1g are noted throughout the book. Although no vendors today support Posix.1g (since it is not final), once the standard is complete vendor support should be forthcoming.

Work on all of the Posix standards continues and it is a moving target for any book that attempts to cover it. The current status of the various Posix standards is available from `http://www.pasc.org/standing/sd11.html`.

The Open Group

The Open Group was formed in 1996 by the consolidation of the X/Open Company (founded in 1984) and the Open Software Foundation (OSF, founded in 1988). It is an international consortium of vendors and end-user customers from industry, government, and academia.

X/Open published the *X/Open Portability Guide*, Issue 3 (XPG3) in 1989. Issue 4 was published in 1992 followed by Issue 4, Version 2 in 1994. This latest version was also known as "Spec 1170," with the magic number 1170 being the sum of the number of system interfaces (926), the number of headers (70), and the number of commands (174). The latest name for this set of specifications is the "X/Open Single Unix Specification" although it is also called "Unix 95."

In March 1997 Version 2 of the Single Unix Specification was announced. Products conforming to this specification can be called "Unix 98." We refer to this specification as just "Unix 98" throughout this text. The number of interfaces required by Unix 98 increases from 1170 to 1434, although for a workstation this jumps to 3030, because it includes the CDE (Common Desktop Environment), which in turn requires the X Window System and the Motif user interface. Details are available in [Josey 1997] and `http://www.opengroup.org/public/tech/unix/version2`.

We are interested in the networking services that are part of Unix 98. These are defined in [Open Group 1997] for both the sockets and XTI APIs. This specification is nearly identical to Draft 6.6 of Posix.1g.

> Unfortunately X/Open refers to their networking standards as *XNS*: X/Open Networking Services. The version of this document that defines sockets and XTI for Unix 98 ([Open Group 1997]) is called "XNS Issue 5." In the networking world XNS has always been the acronym for the Xerox Network Systems architecture. We avoid this use of XNS and refer to this X/Open document as just the Unix 98 network API standard.

Internet Engineering Task Force

The *IETF*, the Internet Engineering Task Force, is a large open international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet. It is open to any interested individual.

The Internet standards process is documented in RFC 2026 [Bradner 1996]. Internet standards normally deal with protocol issues and not with programming APIs. Nevertheless, two RFCs ([Gilligan et al. 1997] and [Stevens and Thomas 1997]) specify the sockets API for IP version 6. These are informational RFCs, not standards, and were produced to speed the deployment of portable applications by the numerous vendors working on early releases of IPv6. Standards bodies tend to take a long time. Nevertheless, at some time the IPv6 APIs will probably be standardized more formally.

Unix Versions and Portability

Most Unix systems today conform to some version of Posix.1 and Posix.2. We must use the qualifier "some" because as updates to Posix occur (e.g., the realtime extensions in 1993 and the pthreads addition in 1996) it takes vendors a year or two (sometimes more) to incorporate these latest changes.

Historically most Unix systems show either a Berkeley heritage or a System V heritage, but these differences are slowly disappearing as most vendors adopt the Posix standards. The main differences still existing deal with system administration, one area that no Posix standard currently addresses.

The focus of this book is on the forthcoming Posix.1g standard, with our main focus on the sockets API. Whenever possible we use the Posix functions.

1.11 64-bit Architectures

During the mid to late 1990s the trend is toward 64-bit architectures and 64-bit software. One reason is for larger addressing within a process (i.e., 64-bit pointers) that can address large amounts of memory (more than 2^{32} bytes). The common programming model for existing 32-bit Unix systems is called the *ILP32* model, denoting that integers (I), long integers (L), and pointers (P) occupy 32 bits. The model that is becoming most prevalent for 64-bit Unix systems is called the *LP64* model, meaning only long integers (L) and pointers (P) require 64 bits. Figure 1.17 compares these two models.

Datatype	ILP32 model	LP64 model
`char`	8	8
`short`	16	16
`int`	32	32
`long`	32	64
pointer	32	64

Figure 1.17 Comparison of number of bits to hold various datatypes for ILP32 and LP64 models.

From a programming perspective the LP64 model means we cannot assume that a pointer can be stored in an integer. We must also consider the effect of the LP64 model on the existing APIs.

ANSI C invented the `size_t` datatype, and this is used, for example, as the argument to `malloc` (the number of bytes to allocate), and the third argument to `read` and `write` (the number of bytes to read or write). On a 32-bit system `size_t` is a 32-bit value, but on a 64-bit system it must be a 64-bit value, to take advantage of the larger addressing model. This means a 64-bit system will probably contain a `typedef` of `size_t` to be an `unsigned long`. The networking API problem is that some drafts of Posix.1g specified that function arguments containing the size of a socket address structures have the `size_t` datatype (e.g., the third argument to `bind` and `connect`). Some XTI structures also had members with a datatype of `long` (e.g., the `t_info` and `t_opthdr` structures). If these had been left as is, both would change from 32-bit values to 64-bit values when a Unix system changes from the ILP32 to the LP64 model. In both instances there is no need for a 64-bit datatype: the length of a socket address structure is a few hundred bytes at the most, and the use of `long` for the XTI structure members was a mistake.

What we will see are new datatypes invented to handle these scenarios. The sockets API uses the `socklen_t` datatype for lengths of socket address structures and XTI uses the `t_scalar_t` and `t_uscalar_t` datatypes. The reason for not changing these values from 32 bits to 64 bits is to make it easier to provide binary compatibility on the new 64-bit systems for application compiled under the 32-bit systems.

1.12 Summary

Figure 1.5 shows a complete, albeit simple, TCP client that fetches the current time and date from a specified server and Figure 1.9 shows a complete version of the server. These two examples introduce many of the terms and concepts that are expanded on throughout the rest of the book.

Our client was protocol dependent on IPv4 and we modified it to use IPv6 instead. But this just gave us another protocol-dependent program. In Chapter 11 we develop some functions that let us write protocol-independent code, which will be important as the Internet starts using IPv6.

Throughout the text we will use the wrapper functions developed in Section 1.4 to reduce the size of our code, yet still check every function call for an error return. Our wrapper functions all begin with a capital letter.

The IEEE Posix standards—Posix.1 defining the basic C interface to Unix, Posix.2 defining the standard commands, and Posix.1g defining the networking APIs—have been the standards that most vendors are moving toward. The Posix standards, however, are rapidly being absorbed and expanded by the commercial standards, notably The Open Group's Unix standards, such as Unix 98.

Readers interested in the history of Unix networking should consult [Salus 1994] for a description of the Unix history, and [Salus 1995] for the history of TCP/IP and the Internet.

Exercises

1.1 Go through the steps at the end of Section 1.9 to discover information about your network topology.

1.2 Obtain the source code for the examples in this text (see the Preface). Compile and test the TCP daytime client in Figure 1.5. Run the program a few times, specifying a different IP address as the command-line argument each time.

1.3 Modify the first argument to `socket` in Figure 1.5 to be 9999. Compile and run the program. What happens? Find the `errno` value corresponding to the error that is printed. How can you find more information on this error?

1.4 Modify Figure 1.5 by placing a counter in the `while` loop, counting the number of times `read` returns a value greater than 0. Print the value of the counter before terminating. Compile and run your new client.

1.5 Modify Figure 1.9 as follows. First change the port number assigned to the `sin_port` member from 13 to 9999. Next, change the single call to `write` into a loop that calls `write` for each byte of the result string. Compile this modified server and start it running in the background. Next modify the client from the previous exercise (which prints the counter before terminating), changing the port number assigned to the `sin_port` member from 13 to 9999. Start this client, specifying the IP address of the host on which the modified server is running as the command-line argument. What value is printed as the client's counter? If possible, also try to run the client and server on different hosts.

2

The Transport Layer: TCP and UDP

2.1 Introduction

This chapter provides an overview of the TCP/IP protocols that are used in the examples throughout the book. Our goal is to provide enough detail to understand how to use the protocols from a network programming perspective and provide references to more detailed descriptions of the actual design, implementation, and history of the protocols.

This chapter focuses on the transport layer, the TCP and UDP protocols, because most client–server applications use either TCP or UDP. These two protocols in turn use the network-layer protocol IP, either IP version 4 (IPv4) or IP version 6 (IPv6). While it is possible to use IPv4 or IPv6 directly, bypassing the transport layer, this technique (called *raw sockets*) is used less frequently. Therefore, we place a more detailed description of IPv4 and IPv6, along with ICMPv4 and ICMPv6, in Appendix A.

UDP is a simple, unreliable, datagram protocol, while TCP is a sophisticated, reliable, byte-stream protocol. We need to understand the services provided by these two transport layers to the application, so that we know what is handled by the protocol and what we must handle in the application.

There are features of TCP that, when understood, make it easier for us to write robust clients and servers. Also, when we understand these features it becomes easier to debug our clients and servers using commonly provided tools such as `netstat`. We cover various topics in this chapter that fall into this category: TCP's three-way handshake, TCP's connection termination sequence, TCP's TIME_WAIT state, TCP and UDP buffering by the socket layer, and so on.

2.2 The Big Picture

Although the protocol suite is called "TCP/IP," there are more members of this family than just TCP and IP. Figure 2.1 shows an overview of these protocols.

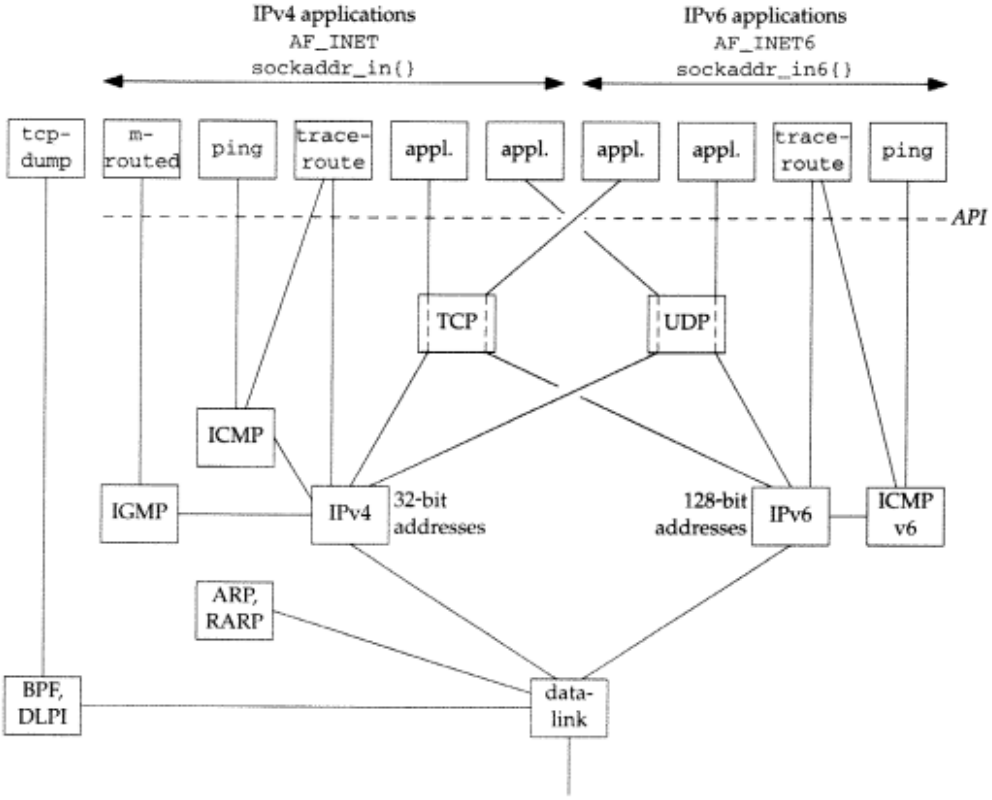

Figure 2.1 Overview of TCP/IP protocols.

We show both IPv4 and IPv6 in this figure. Moving from right to left in this figure, the rightmost four applications are using IPv6 and we talk about the `AF_INET6` constant in Chapter 3 along with the `sockaddr_in6` structure. The next five applications use IPv4.

The leftmost application, `tcpdump`, communicates directly with the datalink using either BPF (BSD Packet Filter) or DLPI (Data Link Provider Interface). We mark the dashed line beneath the nine applications on the right as the *API*, which is normally sockets or XTI. The interface to either BPF or DLPI does not use sockets or XTI.

> There is an exception to this, which we describe in more detail in Chapter 25: Linux provides access to the datalink using a special type of socket called `SOCK_PACKET`.

We also note in Figure 2.1 that the `traceroute` program uses two sockets: one for IP and another for ICMP. In Chapter 25 we develop IPv4 and IPv6 versions of both `ping` and `traceroute`.

We now describe each of the protocol boxes in this figure.

IPv4 *Internet Protocol, version 4.* IPv4, which we often denote as just IP, has been the workhorse protocol of the Internet protocol suite since the early 1980s. It uses 32-bit addresses (Section A.4). IPv4 provides the packet delivery service for TCP, UDP, ICMP, and IGMP.

IPv6 *Internet Protocol, version 6.* IPv6 was designed in the mid-1990s as a replacement for IPv4. The major change is a larger address comprising 128 bits (Section A.5), to deal with the explosive growth of the Internet in the 1990s. IPv6 provides the packet delivery service for TCP, UDP, and ICMPv6.

We often use the adjective *IP*, as in *IP layer* and *IP address*, when the distinction between IPv4 and IPv6 is not needed.

TCP *Transmission Control Protocol.* TCP is a connection-oriented protocol that provides a reliable, full-duplex, byte stream for a user process. TCP sockets are an example of *stream sockets*. TCP takes care of details such as acknowledgments, timeouts, retransmissions, and the like. Most Internet application programs use TCP. Notice that TCP can use either IPv4 or IPv6.

UDP *User Datagram Protocol.* UDP is a connectionless protocol and UDP sockets are an example of *datagram sockets*. Unlike TCP, which is a reliable protocol, there is no guarantee that UDP datagrams ever reach their intended destination. As with TCP, UDP can use either IPv4 or IPv6.

ICMP *Internet Control Message Protocol.* ICMP handles error and control information between routers and hosts. These messages are normally generated by and processed by the TCP/IP networking software itself, not user processes, although we show the Ping program, which uses ICMP. We sometimes refer to this protocol as ICMPv4 to distinguish it from ICMPv6.

IGMP *Internet Group Management Protocol.* IGMP is used with multicasting (Chapter 19), which is optional with IPv4.

ARP *Address Resolution Protocol.* ARP maps an IPv4 address into a hardware address (such as an Ethernet address). ARP is normally used on broadcast networks such as Ethernet, token ring, and FDDI but is not needed on point-to-point networks.

RARP *Reverse Address Resolution Protocol.* RARP maps a hardware address into an IPv4 address. It is sometimes used when a diskless node such as an X terminal is booting.

ICMPv6 *Internet Control Message Protocol, version 6.* ICMPv6 combines the functionality of ICMPv4, IGMP, and ARP.

BPF *BSD Packet Filter*. This interface provides access to the datalink for a process. It is normally found on Berkeley-derived kernels.

DLPI *Data Link Provider Interface*. This interface provides access to the datalink and is normally provided with SVR4.

All the Internet protocols are defined by *Request for Comments (RFCs)*, which are their formal specifications. The solution to Exercise 2.1 shows how to obtain RFCs.

We use the terms *IPv4/IPv6 host* and *dual-stack host* to denote a host that supports both IPv4 and IPv6.

Additional details on the TCP/IP protocols themselves are in TCPv1. The 4.4BSD implementation of TCP/IP is described in TCPv2.

2.3 UDP: User Datagram Protocol

UDP is a simple transport-layer protocol. It is described in RFC 768 [Postel 1980]. The application writes a *datagram* to a UDP socket, which is *encapsulated* as either an IPv4 datagram or an IPv6 datagram, which is then sent to its destination. But there is no guarantee that a UDP datagram ever reaches its final destination.

The problem that we encounter with network programming using UDP is its lack of reliability. If we want to be certain that a datagram reaches its destination, we must build lots of features into our application: acknowledgments from the other end, timeouts, retransmissions, and the like.

Each UDP datagram has a length and we can consider a datagram as a *record*. If the datagram reaches its final destination correctly (that is, the packet arrives without a checksum error), then the length of the datagram is passed to the receiving application. We have already mentioned that TCP is a *byte-stream* protocol, without any record boundaries at all (Section 1.2), which differs from UDP.

We also say that UDP provides a *connectionless* service as there need not be any long-term relationship between a UDP client and server. For example, a UDP client can create a socket and send a datagram to a given server and then immediately send another datagram on the same socket to a different server. Similarly a UDP server can receive five datagrams in a row on a single UDP socket, each from five different clients.

2.4 TCP: Transmission Control Protocol

The service provided by TCP to an application is different from the service provided by UDP. (TCP is described in RFC 793 [Postel 1981c]). First, TCP provides *connections* between clients and servers. A TCP client establishes a connection with a given server, exchanges data with that server across the connection, and then terminates the connection.

TCP also provides *reliability*. When TCP sends data to the other end, it requires an acknowledgment in return. If an acknowledgment is not received, TCP automatically retransmits the data and waits a longer amount of time. After some number of retransmissions, TCP will give up, with the total amount of time spent trying to send data typically between 4 and 10 minutes (depending on the implementation). TCP contains algorithms to estimate the *round-trip time* (RTT) between a client and server dynamically so that it knows how long to wait for an acknowledgment. For example, the RTT on a LAN can be milliseconds while across a WAN it can be seconds. Furthermore, TCP can measure an RTT of 1 second between a client and server and then 30 seconds later measure an RTT of 2 seconds on the same connection, caused by variations in the network traffic.

TCP also *sequences* the data by associating a sequence number with every byte that it sends. For example, assume an application writes 2048 bytes to a TCP socket, causing TCP to send two segments, the first containing the data with sequence numbers 1–1024 and the second containing the data with sequence numbers 1025–2048. (A *segment* is the unit of data that TCP passes to IP.) If the segments arrive out of order, the receiving TCP will reorder the two segments based on their sequence numbers before passing the data to the receiving application. If TCP receives duplicate data from its peer (say the peer thought a segment was lost and retransmitted it, when it wasn't really lost, the network was just overloaded), it can detect that the data has been duplicated (from the sequence numbers), and the duplicate data is discarded.

> There is no reliability provided by UDP. UDP itself does not provide anything like acknowledgments, sequence numbers, RTT estimation, timeouts, or retransmissions. If a UDP datagram is duplicated in the network, two copies can be delivered to the receiving host. Also, if a UDP client sends two datagrams to the same destination, they can be reordered by the network and arrive out of order. UDP applications must handle all these cases, as we show in Section 20.5.

TCP provides *flow control*. TCP always tells its peer exactly how many bytes of data it is willing to accept from the peer. This is called the advertised *window*. At any time, the window is the amount of room currently available in the receive buffer, guaranteeing that the sender cannot overflow the receiver's buffer. The window changes dynamically over time: as data is received from the sender, the window size decreases, but as the receiving application reads data from the buffer, the window increases. It is possible for the window to reach 0: TCP's receive buffer for this socket is full and it must wait for the application to read data from the buffer before it can take any more data from the peer.

> UDP provides no flow control. It is easy for a fast UDP sender to transmit datagrams at a rate that the UDP receiver cannot keep up with, as we show in Section 8.13.

Finally, a TCP connection is also *full-duplex*. This means that an application can send and receive data in both directions on a given connection at any time. This means that TCP must keep track of state information such as sequence numbers and window sizes for each direction of data flow: sending and receiving.

> UDP can be full-duplex.

2.5 TCP Connection Establishment and Termination

To aid our understanding of the `connect`, `accept`, and `close` functions and to help us debug TCP applications using the `netstat` program, we must understand how TCP connections are established and terminated, and TCP's state transition diagram. This is an example of increased knowledge of the underlying protocols helping us with network programming.

Three-Way Handshake

The following scenario occurs when a TCP connection is established:

1. The server must be prepared to accept an incoming connection. This is normally done by calling `socket`, `bind`, and `listen` and is called a *passive open*.
2. The client issues an *active open* by calling `connect`. This causes the client TCP to send a SYN segment (which stands for "synchronize") to tell the server the client's initial sequence number for the data that the client will send on the connection. Normally there is no data sent with the SYN: it just contains an IP header, a TCP header, and possible TCP options (which we talk about shortly).
3. The server must acknowledge the client's SYN and the server must also send its own SYN containing the initial sequence number for the data that the server will send on the connection. The server sends its SYN and the ACK of the client's SYN in a single segment.
4. The client must acknowledge the server's SYN.

The minimum number of packets required for this exchange is three; hence this is called TCP's *three-way handshake*. We show a picture of the three segments in Figure 2.2.

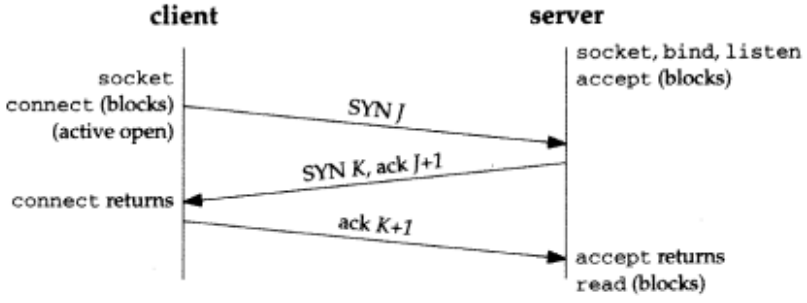

Figure 2.2 TCP three-way handshake.

We show the client's initial sequence number as J and the server's initial sequence number as K. The acknowledgment number in an ACK is the next expected sequence number for the end sending the ACK. Since a SYN occupies 1 byte of the sequence number space, the acknowledgment number in the ACK of each SYN is the initial

sequence number plus one. Similarly the ACK of each FIN is the sequence number of the FIN plus one.

> An everyday analogy for establishing a TCP connection is the telephone system [Nemeth 1997]. The `socket` function is the equivalent of having a telephone to use. `bind` is telling other people your telephone number so that they can call you. `listen` is turning on the ringer so that you will hear when an incoming call arrives. `connect` requires that we know the other person's phone number and dial it. `accept` is when the person being called answers the phone. Having the client's identity returned by `accept` (where the identify is the client's IP address and port number) is similar to having the caller ID feature show the caller's phone number. One difference, however, is that `accept` returns the client's identity only after the connection has been established, whereas the caller ID feature shows the caller's phone number before we choose whether to answer the phone or not. If the Domain Name System is used (Chapter 9), it provides a service analogous to a telephone book. `gethostbyname` is similar to looking up a person's phone number in the phone book. `gethostbyaddr` would be the equivalent of having a phone book sorted by telephone numbers that we could search, instead of sorted by name.

TCP Options

Each SYN can contain TCP options. Commonly used options are the following:

- MSS option. With this option the TCP sending the SYN announces its *maximum segment size*, the maximum amount of data that it is willing to accept in each TCP segment, on this connection. We will see how to fetch and set this TCP option with the `TCP_MAXSEG` socket option (Section 7.9).
- Window scale option. The maximum window that either TCP can advertise to the other TCP is 65535, because the corresponding field in the TCP header occupies 16 bits. But high-speed connections (45 Mbits/sec and faster, as described in RFC 1323 [Jacobson, Braden, and Borman 1992]) or long-delay paths (satellite links) require a larger window to obtain the maximum throughput possible. This newer option specifies that the advertised window in the TCP header must be scaled (left-shifted) by 0–14 bits, providing a maximum window of almost one gigabyte (65535×2^{14}). Both end systems must support this option for the window scale to be used on a connection. We will see how to effect this option with the `SO_RCVBUF` socket option (Section 7.5).

  > To provide interoperability with older implementations that do not support this option, the following rules apply. TCP can send the option with its SYN as part of an active open. But it can scale its windows only if the other end also sends the option with its SYN. Similarly the server's TCP can send this option only if it receives the option with the client's SYN. This logic assumes that implementations ignore options that they do not understand, which is required and common, but unfortunately, not guaranteed with all implementations.

- Timestamp option. This option is needed for high-speed connections to prevent possible data corruption caused by lost packets that then reappear. Since it is a newer option, it is negotiated similarly to the window scale option. As a network programmer there is nothing we need worry about with this option.

The MSS option is supported by most implementations, while the window scale and timestamp options are newer. These latter two are sometimes called the "RFC 1323 options" as that RFC [Jacobson, Braden, and Borman 1992] specifies the options. They are also called the "long fat pipe" options, since a network with either a high bandwidth or a long delay is called a *long fat pipe*. Chapter 24 of TCPv1 contains more details on these newer options.

TCP Connection Termination

While it takes three segments to establish a connection, it takes four to terminate a connection.

1. One application calls `close` first, and we say that this end performs the *active close*. This end's TCP sends a FIN segment, which means it is finished sending data.
2. The other end that receives the FIN performs the *passive close*. The received FIN is acknowledged by TCP. The receipt of the FIN is also passed to the application as an end-of-file (after any data that may already be queued for the application to receive), since the receipt of the FIN means the application will never receive any additional data on the connection.
3. Sometime later the application that received the end-of-file will `close` its socket. This causes its TCP to send a FIN.
4. The TCP on the system that receives this final FIN (the end that did the active close) acknowledges the FIN.

Since a FIN and an ACK are required in each direction, four segments are normally required. We use the qualifier "normally" because in some scenarios the FIN in step 1 is sent with data. Also, the segments in steps 2 and 3 are both from the end performing the passive close and could be combined into one segment. We show these packets in Figure 2.3.

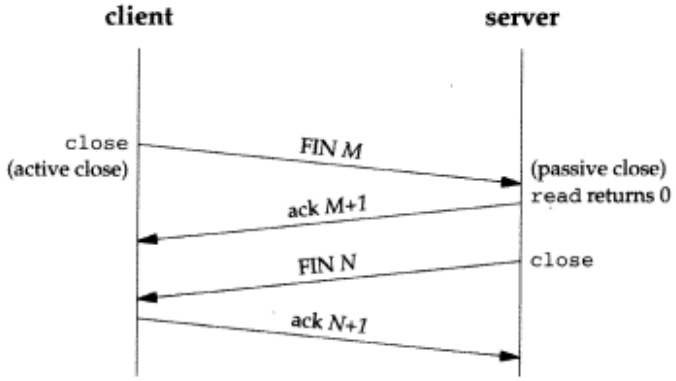

Figure 2.3 Packets exchanged when a TCP connection is closed.

A FIN occupies 1 byte of sequence number space just like a SYN. Therefore the ACK of each FIN is the sequence number of the FIN plus one.

Between steps 2 and 3 it is possible for data to flow from the end doing the passive close to the end doing the active close. This is called a *half-close* and we talk about this in detail with the `shutdown` function in Section 6.6.

The sending of each FIN occurs when the socket is closed. We indicated that the application calls `close` for this to happen but realize that when a Unix process terminates, either voluntarily (calling `exit` or having the `main` function return) or involuntarily (receiving a signal that terminates the process), all open descriptors are closed, which will also cause a FIN to be sent on any TCP connection that is still open.

Although we show the client in Figure 2.3 performing the active close, either end—the client or the server—can perform the active close. Often the client performs the active close, but with some protocols (notably HTTP), the server performs the active close.

TCP State Transition Diagram

The operation of TCP with regard to connection establishment and connection termination can be specified with a *state transition diagram*. We show this in Figure 2.4.

There are 11 different states defined for a connection and the rules of TCP dictate the transitions from one state to another, based on the current state and the segment received in that state. For example, if the application performs an active open in the CLOSED state, TCP sends a SYN and the new state is SYN_SENT. If TCP next receives a SYN with an ACK, it sends an ACK and the new state is ESTABLISHED. This final state is where most data transfer occurs.

The two arrows leading from the ESTABLISHED state deal with the termination of a connection. If the application calls `close` before receiving an end-of-file (an active close), the transition is to the FIN_WAIT_1 state. But if the application receives a FIN while in the ESTABLISHED state (a passive close), the transition is to the CLOSE_WAIT state.

We denote the normal client transitions with a darker solid line and the normal server transitions with a darker dashed line. We also note that there are two transitions that we have not talked about: a simultaneous open (when both ends send SYNs at about the same time and the SYNs cross in the network) and a simultaneous close (when both ends send FINs at the same time). Chapter 18 of TCPv1 contains examples and discussion of both scenarios, which are possible but rare.

One reason for showing the state transition diagram is to show the 11 TCP states with their names. These states are displayed by `netstat`, which is a useful tool when debugging client–server applications. We will use `netstat` to monitor the state changes in Chapter 5.

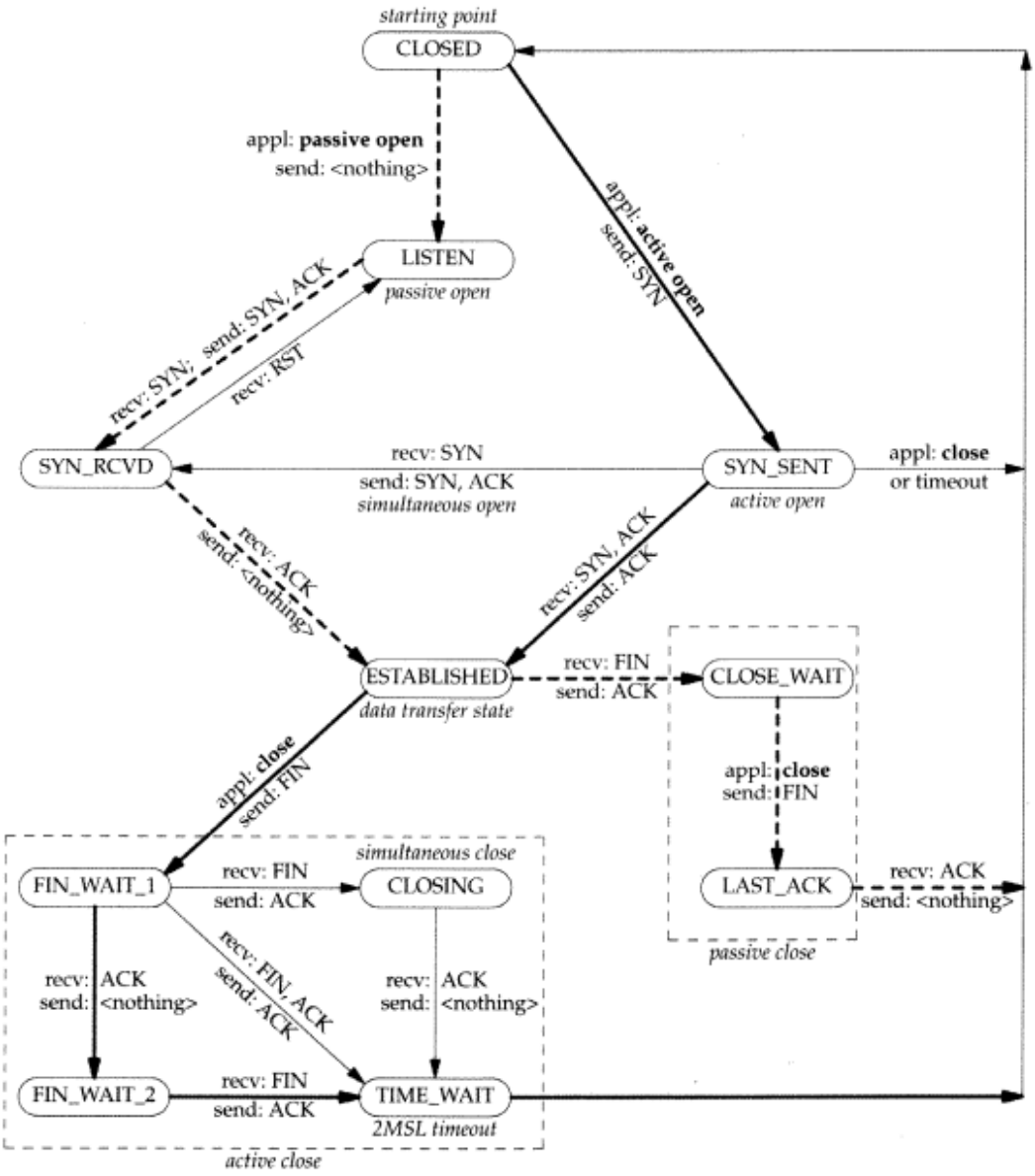

Figure 2.4 TCP state transition diagram.

Watching the Packets

Figure 2.5 shows the actual packet exchange that takes place for a complete TCP connection: the connection establishment, data transfer, and connection termination. We also show the TCP states through which each endpoint passes.

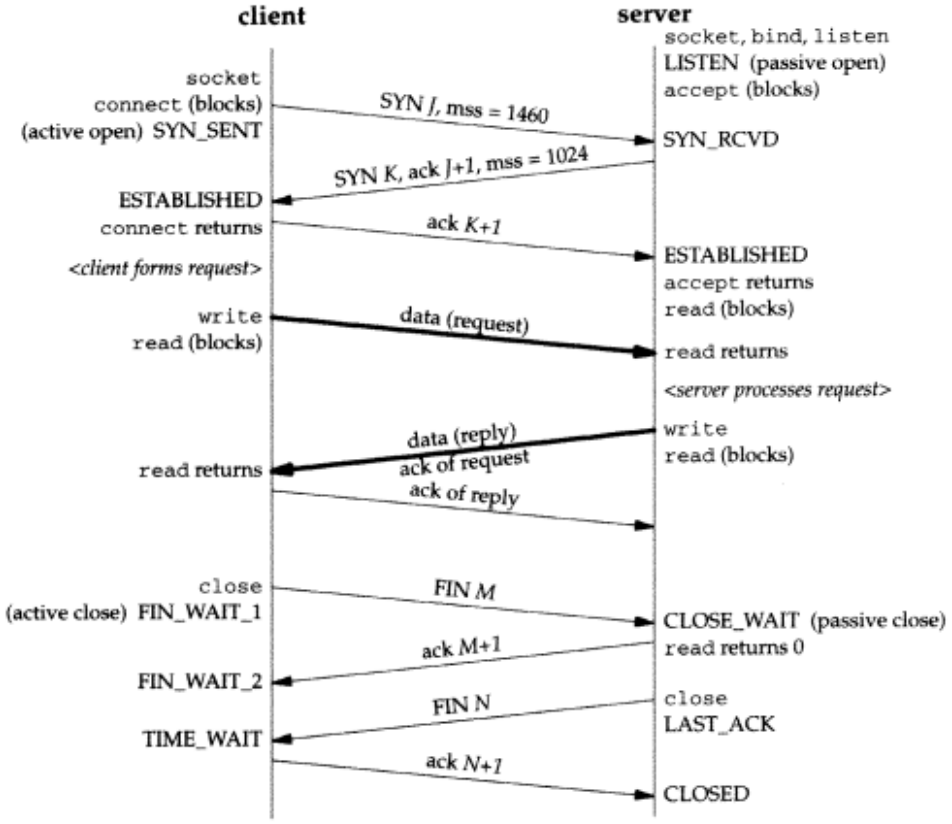

Figure 2.5 Packet exchange for TCP connection.

The client in this example announces an MSS of 1460 (typical for IPv4 on an Ethernet) and the server announces an MSS of 1024 (typical for older Berkeley-derived implementations on an Ethernet). It is OK for the MSS to be different in each direction. (See also Exercise 2.5.)

Once the connection is established, the client forms a request and sends it to the server. We assume this request fits into a single TCP segment (i.e., less than 1024 bytes given the server's announced MSS). The server processes the request and sends a reply, and we assume that the reply fits in a single segment (less than 1460 in this example).

We show both data segments as bolder arrows. Notice that the acknowledgment of the client's request is sent with the server's reply. This is called *piggybacking* and will normally happen when the time it takes the server to process the request and generate the reply is less than around 200 ms. If the server takes longer, say 1 second, we would see the acknowledgment followed later by the reply. (The dynamics of TCP data flow are covered in detail in Chapters 19 and 20 of TCPv1.)

We then show the four segments that terminate the connection. Notice that the end that performs the active close (the client in this scenario) enters the TIME_WAIT state. We discuss this in the next section.

It is important to notice in Figure 2.5 that if the entire purpose of this connection was to send a one-segment request and receive a one-segment reply, there are eight segments of overhead involved when using TCP. If UDP were used instead, only two packets would be exchanged: the request and the reply. But switching from TCP to UDP removes all the reliability that TCP provides to the application, pushing lots of these details from the transport layer (TCP) into the UDP application. Another important feature provided by TCP is congestion control, which must then be handled by the UDP application. Nevertheless, it is important to understand that many applications that are built using UDP do so because the application exchanges small amounts of data and UDP avoids the overhead of the TCP connection establishment and connection termination.

> An alternative to UDP in this scenario is T/TCP: TCP for Transactions. We describe this in Section 13.9.

2.6 TIME_WAIT State

Undoubtedly, the most misunderstood aspect of TCP with regard to network programming is its TIME_WAIT state. We can see in Figure 2.4 that the end that performs the active close goes through this state. The duration that this endpoint remains in this state is twice the MSL (*maximum segment lifetime*), sometimes called 2MSL.

Every implementation of TCP must choose a value for the MSL. The recommended value in RFC 1122 [Braden 1989] is 2 minutes, although Berkeley-derived implementations have traditionally used a value of 30 seconds instead. This means the duration of the TIME_WAIT state is between 1 and 4 minutes. The MSL is the maximum amount of time that any given IP datagram can live in an internet. We know this time is bounded because every datagram contains an 8-bit hop limit (the IPv4 TTL field in Figure A.1 and the IPv6 hop limit field in Figure A.2) with a maximum value of 255. Although this is a hop limit and not a true time limit, the assumption is made that a packet with the maximum hop limit of 255 cannot exist in an internet for more than MSL seconds.

The way in which a packet gets "lost" in an internet is usually the result of routing anomalies. A router crashes or a link between two routers goes down and it takes the routing protocols seconds or minutes to stabilize and find an alternate path. During that time period routing loops can occur (router A sends packets to router B, and B sends them back to A) and packets can get caught in these loops. In the meantime, assuming the lost packet is a TCP segment, the sending TCP times out and retransmits

the packet, and the retransmitted packet gets to the final destination by some alternate path. But sometime later (up to MSL seconds after the lost packet started on its journey) the routing loop is corrected and the packet that was lost in the loop is sent to the final destination. This original packet is called a *lost duplicate* or a *wandering duplicate*. TCP must handle these duplicates.

There are two reasons for the TIME_WAIT state:

1. to implement TCP's full-duplex connection termination reliably, and
2. to allow old duplicate segments to expire in the network.

The first reason can be explained by looking at Figure 2.5 and assuming that the final ACK is lost. The server will resend its final FIN so the client must maintain state information allowing it to resend the final ACK. If it did not maintain this information, it would respond with an RST (a different type of TCP segment), which would be interpreted by the server as an error. If TCP is performing all the work necessary to terminate both directions of data flow cleanly for a connection (its full-duplex close), then it must correctly handle the loss of any of these four segments. This example also shows why the end that performs the active close is the end that remains in the TIME_WAIT state: because that end is the one that might have to retransmit the final ACK.

To understand the second reason for the TIME_WAIT state, assume we have a TCP connection between 206.62.226.33 port 1500 and 198.69.10.2 port 21. This connection is closed and then sometime later we establish another connection between the same IP addresses and ports: 206.62.226.33 port 1500 and 198.69.10.2 port 21. This latter connection is called an *incarnation* of the previous connection since the IP addresses and ports are the same. TCP must prevent old duplicates from a connection from reappearing at some time later and being misinterpreted as belonging to a new incarnation of the same connection. To do this TCP will not initiate a new incarnation of a connection that is currently in the TIME_WAIT state. Since the duration of the TIME_WAIT state is twice the MSL, this allows MSL seconds for a packet in one direction to be lost, and another MSL seconds for the reply to be lost. By enforcing this rule we are guaranteed that when we successfully establish a TCP connection, all old duplicates from previous incarnations of this connection have expired in the network.

> There is an exception to this rule. Berkeley-derived implementations will initiate a new incarnation of a connection that is currently in the TIME_WAIT state if the arriving SYN has a sequence number that is "greater than" the ending sequence number from the previous incarnation. Pages 958–959 of TCPv2 talk about this in more detail. This requires the server to perform the active close, since the TIME_WAIT state must exist on the end that receives the next SYN. This capability is used by the `rsh` command. RFC 1185 [Jacobson, Braden, and Zhang 1990] talks about some pitfalls in doing this.

2.7 Port Numbers

At any given time, multiple processes can use either UDP or TCP. Both TCP and UDP use 16-bit integer *port numbers* to differentiate between these processes.

When a client wants to contact a server, the client must identify the server with which it wants to communicate. Both TCP and UDP define a group of *well-known ports* to identify well-known services. For example, every TCP/IP implementation that supports FTP assigns the well-known port of 21 (decimal) to the FTP server. TFTP servers, for the Trivial File Transfer Protocol, are assigned the UDP port of 69.

Clients, on the other hand, use *ephemeral ports*, that is, short-lived ports. These port numbers are normally assigned automatically by TCP or UDP to the client. Clients normally do not care about the value of the ephemeral port; the client just needs to be certain that the ephemeral port is unique on the client host. The TCP and UDP codes guarantee this uniqueness.

RFC 1700 [Reynolds and Postel 1994] contains the list of port number assignments from the *Internet Assigned Numbers Authority* (IANA). But usually the file `ftp://ftp.isi.edu/in-notes/iana/assignments/port-numbers` is more up to date than the RFC. The port numbers are divided into three ranges:

1. The *well-known ports*: 0 through 1023. These port numbers are controlled and assigned by the IANA. When possible, the same port is assigned to a given service for both TCP and UDP. For example, port 80 is assigned for a Web server, for both protocols, even though all implementations currently use only TCP.
2. The *registered ports*: 1024 through 49151 . These are not controlled by the IANA, but the IANA registers and lists the uses of these ports as a convenience to the community. When possible, the same port is assigned to a given service for both TCP and UDP. For example, ports 6000 through 6063 are assigned for an X Window server, for both protocols, even though all implementations currently use only TCP. The upper limit of 49151 for these ports is new, as RFC 1700 [Reynolds and Postel 1994] lists the upper range as 65535.
3. The *dynamic* or *private* ports, 49152 through 65535. The IANA says nothing about these ports. These are what we call *ephemeral* ports.

(The magic number 49152 is three-fourths of 65536.) Figure 2.6 shows this division along with the common allocation of the port numbers.

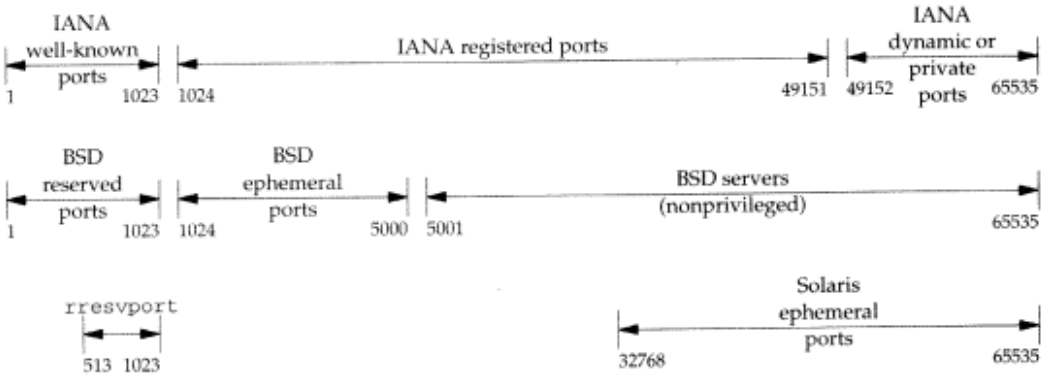

Figure 2.6 Allocation of port numbers.

We note the following points from this figure.

- Unix systems have the concept of a *reserved port*, which is any port less than 1024. These ports can only be assigned to a socket by a superuser process. All the IANA well-known ports are reserved ports; hence the server allocating this port (such as the FTP server) must have superuser privileges when it starts.
- Historically, Berkeley-derived implementations (starting with 4.3BSD) have allocated ephemeral ports in the range 1024–5000. This was fine in the early 1980s, when server hosts were not capable of handling too many clients at once, but it is easy today to find a host that can support more than 3977 clients at any given time. Therefore some systems allocate ephemeral ports differently (e.g., Solaris as we show in Figure 2.6) to provide more ephemeral ports.

  As it turns out, the upper limit of 5000 for the ephemeral ports, which many systems currently implement, was a typo [Borman 1997a]. The limit should have been 50,000.

- There are a few clients (not servers) that require a reserved port as part of the client–server authentication: the `rlogin` and `rsh` clients are the common examples. These clients call the library function `rresvport` to create a TCP socket and assign an unused port in the range 513–1023 to the socket. This function normally tries to bind port 1023 and if that fails, tries to bind 1022, and so on, until it either succeeds, or fails on port 513.

  Notice that the BSD reserved ports and the `rresvport` function both overlap with the upper half of the IANA well-known ports. This is because the IANA well-known ports used to stop at 255. RFC 1340 (a previous "Assigned Numbers" RFC) in 1992 started assigning well-known ports between 256 and 1023. The previous "Assigned Numbers" document, RFC 1060 in 1990, called ports 256–1023 the *Unix Standard Services*. There are numerous Berkeley-derived servers that picked their well-known ports in the 1980s starting at 512 (leaving 256–511 untouched). The `rresvport` function chose to start at the top of the 512–1023 range and work down.

Socket Pair

The *socket pair* for a TCP connection is the 4-tuple that defines the two endpoints of the connection: the local IP address, local TCP port, foreign IP address, and foreign TCP port. A socket pair uniquely identifies every TCP connection on an internet.

The two values that identify each endpoint, an IP address and a port number, are often called a *socket*.

We can extend the concept of a socket pair to UDP, even though UDP is connectionless. When we describe the socket functions (`bind`, `connect`, `getpeername`, etc.), we will note which functions specify which values in the socket pair. For example, `bind` lets the application specify the local IP address and local port, for both TCP and UDP sockets.

2.8 TCP Port Numbers and Concurrent Servers

With a concurrent server, where the main server loop spawns a child to handle each new connection, what happens if the child continues to use the well-known port number while servicing a long request? Let's examine a typical sequence. First, the server is started on the host `bsdi` (Figure 1.16), which is multihomed with IP addresses 206.62.226.35 and 206.62.226.66 and the server does a passive open using its well-known port number (21, for this example). It is now waiting for a client request which we show in Figure 2.7.

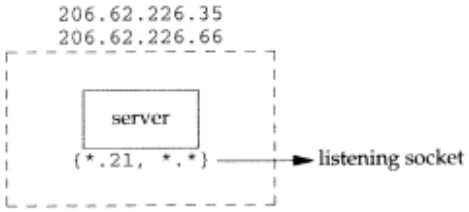

Figure 2.7 TCP server with a passive open on port 21.

We use the notation `{*.21, *.*}` to indicate the server's socket pair. The server is waiting for a connection request on any local interface (the first asterisk), on port 21. The foreign IP address and foreign port are not specified and we denote them as `*.*`. We also call this a *listening socket*.

> We use a period to separate the IP address from the port number because that is what `netstat` uses. This is sometimes confusing because decimal points are used in both domain names (`solaris.kohala.com.21`) and in IPv4 dotted-decimal notation (`206.62.226.33.21`).

When we specify the local IP address as an asterisk, it is called the *wildcard* character. If the host on which the server is running is multihomed (as in this example), the server can specify that it wants only to accept incoming connections that arrive destined to one specific local interface. This is a one-or-any choice for the server. The server cannot specify a list of multiple addresses. The wildcard local address is the "any" choice. In Figure 1.9 the wildcard address was specified by setting the IP address in the socket address structure to `INADDR_ANY` before calling `bind`.

At some later time a client starts on the host with IP address 198.69.10.2 and executes an active open to the server's IP address of 206.62.226.35. We assume the ephemeral port chosen by the client TCP is 1500 for this example. This is shown in Figure 2.8. Beneath the client we show its socket pair.

When the server receives and accepts the client's connection, it `forks` a copy of itself, letting the child handle the client, as we show in Figure 2.9. (We describe the `fork` function in Section 4.7.)

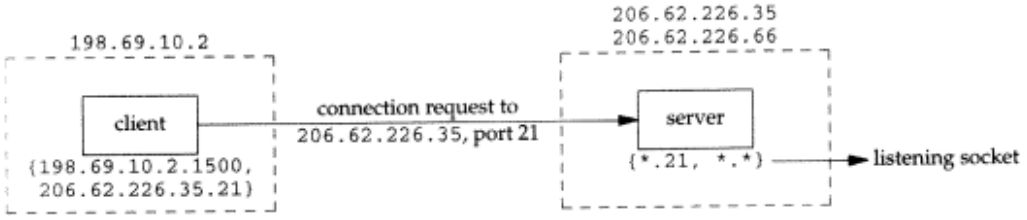

Figure 2.8 Connection request from client to server.

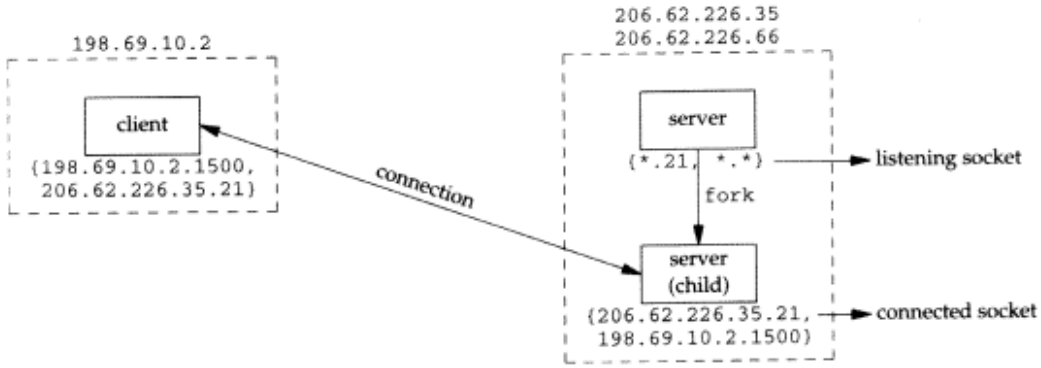

Figure 2.9 Concurrent server has child handle client.

At this point we must distinguish on the server host between the listening socket and the connected socket. Notice that the connected socket uses the same local port (21) as used for the listening socket. Also notice that on the multihomed server the local address is filled in for the connected socket (206.62.226.35) once the connection is established.

The next step assumes that another client process on the client host requests a connection with the same server. The TCP code on the client host assigns the new client socket an unused ephemeral port number, say 1501. This gives us the scenario shown in Figure 2.10. On the server the two connections are distinct: the socket pair for the first connection differs from the socket pair for the second connection because the client's TCP chooses an unused port for the second connection (1501).

Notice from this example that TCP cannot demultiplex incoming segments by looking at just the destination port number. TCP must look at all four elements in the socket pair to determine which endpoint receives an arriving segment. In Figure 2.10 we have three sockets with the same local port (21). If a segment arrives from 198.69.10.2 port 1500 destined for 206.62.226.35 port 21, it is delivered to the first child. If a segment arrives from 198.69.10.2 port 1501 destined for 206.62.226.35 port 21, it is delivered to the second child. All other TCP segments destined for port 21 are delivered to the original server with the listening socket.

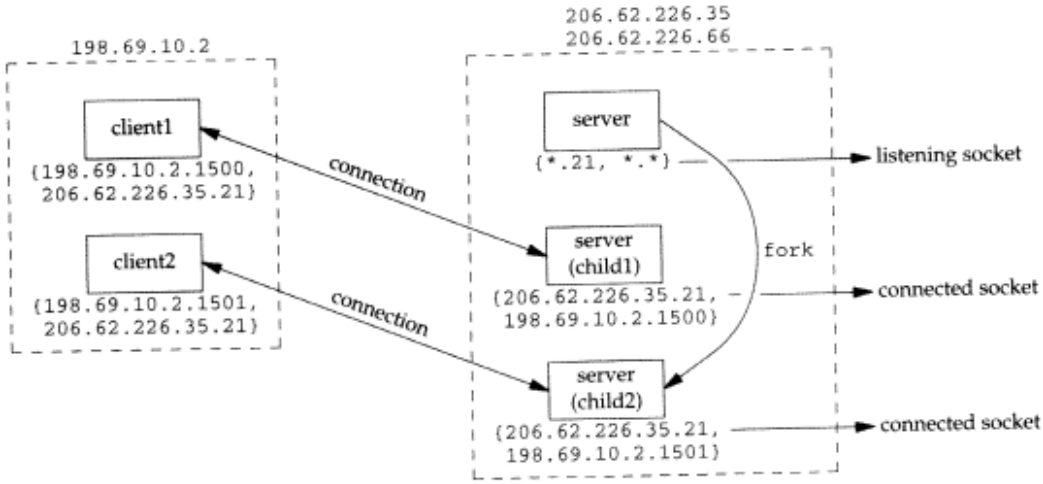

Figure 2.10 Second client connection with same server.

2.9 Buffer Sizes and Limitations

There are certain limits that affect the size of IP datagrams. We first describe these limits and then tie them all together with regard to how they affect the data an application can transmit.

- The maximum size of an IPv4 datagram is 65535 bytes, including the IPv4 header. This is because of the 16-bit total length field in Figure A.1.
- The maximum size of an IPv6 datagram is 65575 bytes, including the 40-byte IPv6 header. This is because of the 16-bit payload length field in Figure A.2. Notice that the IPv6 payload length field does not include the size of the IPv6 header, while the IPv4 total length field does include the header size.

  IPv6 has a jumbo payload option, which extends the payload length field to 32 bits, but this option is supported only on those datalinks with an MTU that exceeds 65535. (This is intended for host-to-host interconnects, such as HIPPI, which often have no inherent MTU.)
- Many networks have an *MTU*, *maximum transmission unit*, which can be dictated by the hardware. For example, the Ethernet MTU is 1500 bytes. Other datalinks, such as point-to-point links using the PPP protocol, have a configurable MTU. Older SLIP links often used an MTU of 296 bytes.

  The minimum link MTU for IPv4 is 68 bytes. The minimum link MTU for IPv6 is 576 bytes.
- The smallest MTU in the path between two hosts is called the *path MTU*. Today, the Ethernet MTU of 1500 bytes is often the path MTU. The path MTU need not

be the same in both directions between any two hosts, because routing in the Internet is often asymmetric [Paxson 1996]. That is, the route from A to B can differ from the route from B to A.

- When an IP datagram is to be sent out an interface, if the size of the datagram exceeds the link MTU, *fragmentation* is performed by both IPv4 and IPv6. The fragments are never *reassembled* until they reach the final destination. IPv4 hosts perform fragmentation on datagrams that they generate and IPv4 routers perform fragmentation on datagrams that they forward. But with IPv6 only hosts perform fragmentation on datagrams that they generate; IPv6 routers do not fragment datagrams that they are forwarding.

  We must be careful with our terminology. A box labeled an IPv6 router may indeed perform fragmentation, but only on datagrams that the router itself generates, never on datagrams that it is forwarding. When this box generates IPv6 datagrams, it is really acting as a host. For example, most routers support the Telnet protocol and this is used for configuration of the router by administrators. The IP datagrams generated by the router's Telnet server are generated by the router, not forwarded by the router.

  You may notice that fields exist in the IPv4 header (Figure A.1) to handle IPv4 fragmentation, but there are no fields in the IPv6 header (Figure A.2) for fragmentation. Since fragmentation is the exception, rather than the rule, IPv6 contains an option header with the fragmentation information.

- If the DF bit ("don't fragment") is set in the IPv4 header (Figure A.1) it specifies that this datagram must not be fragmented, either by the sending host or by any router. A router that receives an IPv4 datagram with the DF bit set whose size exceeds the outgoing link's MTU generates an ICMPv4 "destination unreachable, fragmentation needed but DF bit set" error message (Figure A.15).

  Since IPv6 routers do not perform fragmentation, there is an implied DF bit with every IPv6 datagram. When an IPv6 router receives a datagram whose size exceeds the outgoing link's MTU, it generates an ICMPv6 "packet too big" error message (Figure A.16).

  The IPv4 DF bit and its implied IPv6 counterpart can be used for *path MTU discovery* (RFC 1191 [Mogul and Deering 1990] for IPv4 and RFC 1981 [McCann, Deering, and Mogul 1996] for IPv6). For example, if TCP uses this technique with IPv4, then it sends all of its datagrams with the DF bit set. If some intermediate router returns an ICMP "destination unreachable, fragmentation needed but DF bit set" error, TCP decreases the amount of data it sends per datagram and retransmits. Path MTU discovery is optional with IPv4 but should be supported by all IPv6 implementations.

- IPv4 and IPv6 define a *minimum reassembly buffer size*: the minimum datagram size that we are guaranteed any implementation must support. For IPv4 this is 576 bytes. IPv6 raises this to 1500 bytes. With IPv4, for example, we have no idea whether a given destination can accept a 577-byte datagram or not. Therefore many IPv4 applications that use UDP (DNS, RIP, TFTP, BOOTP, SNMP) prevent the application from generating IP datagrams that exceed this size.

- TCP has an *MSS, maximum segment size,* that announces to the peer TCP the maximum amount of TCP data that the peer can send per segment. We saw the MSS option on the SYN segments in Figure 2.5. The goal of the MSS is to tell the peer the actual value of the reassembly buffer size and to try to avoid fragmentation. The MSS is often set to the interface MTU minus the fixed sizes of the IP and TCP headers. On an Ethernet using IPv4 this would be 1460, and on an Ethernet using IPv6 this would be 1440. (The TCP header is 20 bytes for both, but the IPv4 header is 20 bytes and the IPv6 header is 40 bytes.)

  The MSS value in the TCP MSS option is a 16-bit field, limiting the value to 65535. This is fine for IPv4, since the maximum amount of TCP data in an IPv4 datagram is 65495 (65535 minus the 20-byte IPv4 header and minus the 20-byte TCP header). But with the IPv6 jumbo payload option, a different technique is used (RFC 2147 [Borman 1997b]). First, the maximum amount of TCP data in an IPv6 datagram without the jumbo payload option is 65515 (65535 minus the 20-byte TCP header). Therefore the MSS value of 65535 is considered a special case that designates "infinity." This value is used only if the jumbo payload option is being used, which requires an MTU that exceeds 65535. If TCP is using the jumbo payload option and receives an MSS announcement of 65535 from the peer, the limit on the datagram sizes that it sends is just the interface MTU. If this turns out to be too large (i.e., there is a link in the path with a smaller MTU) then path MTU discovery will determine the smaller value.

TCP Output

Given all these terms and definitions, Figure 2.11 shows what happens when an application writes data to a TCP socket.

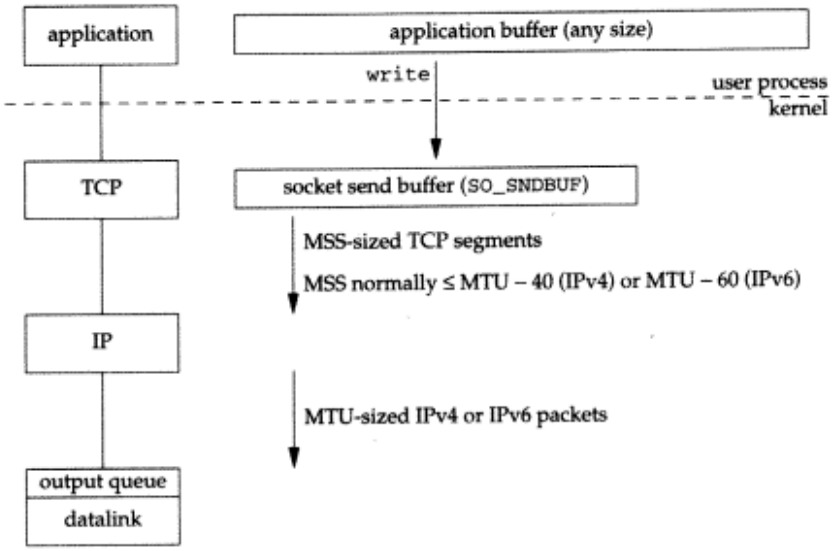

Figure 2.11 Steps and buffers involved when application writes to a TCP socket.

Every TCP socket has a send buffer and we can change the size of this buffer with the `SO_SNDBUF` socket option (Section 7.5). When the application calls `write`, the kernel copies all the data from the application buffer into the socket send buffer. If there is insufficient room in the socket buffer for all of the application's data (either the application buffer is larger than the socket send buffer, or there is already data in the socket send buffer), the process is put to sleep. This assumes the normal default of a blocking socket. (We talk about nonblocking sockets in Chapter 15.) The kernel will not return from the `write` until the final byte in the application buffer has been copied into the socket send buffer. Therefore the successful return from a `write` to a TCP socket only tells us that we can reuse our application buffer. It does *not* tell us that either the peer TCP has received the data, or that the peer application has received the data. (We talk about this more with the `SO_LINGER` socket option in Section 7.5.)

TCP takes the data in the socket send buffer and sends it to the peer TCP, based on all the rules of TCP data transmission (Chapters 19 and 20 of TCPv1). The peer TCP must acknowledge the data, and as the ACKs arrive from the peer, only then can our TCP discard the acknowledged data from the socket send buffer. TCP must keep a copy of our data until it is acknowledged by the peer.

TCP sends the data to IP in MSS-sized chunks or smaller, prepending its TCP header to each segment, where the MSS is the value announced by the peer, or 536 if the peer did not send an MSS option. IP prepends its header, searches the routing table for the destination IP address (the matching routing table entry specifies the outgoing interface), and passes the datagram to the appropriate datalink. IP might perform fragmentation before passing the datagram to the datalink, but as we said earlier, one goal of the MSS option is to try to avoid fragmentation and newer implementations also use path MTU discovery. Each datalink has an output queue and if this queue is full, the packet is discarded and an error is returned up the protocol stack: from the datalink to IP and then from IP to TCP. TCP will note this error and try sending the segment later. The application is not told of this transient condition.

UDP Output

Figure 2.12 shows what happens when an application writes data to a UDP socket. This time we show the socket send buffer as a dashed box because it doesn't really exist. A UDP socket has a send buffer size (which we can change with the `SO_SNDBUF` socket option, Section 7.5), but this is simply an upper limit on the maximum sized UDP datagram that can be written to the socket. If an application writes a datagram larger than the socket send buffer size, `EMSGSIZE` is returned. Since UDP is unreliable, it does not need to keep a copy of the application's data and does not need an actual send buffer. (The application data is normally copied into a kernel buffer of some form as it passes down the protocol stack, but this copy is discarded by the datalink layer after the data is transmitted.)

UDP simply prepends its 8-byte header and passes the datagram to IP. IPv4 or IPv6 prepends its header, determines the outgoing interface by performing the routing function, and then either adds the datagram to the datalink output queue (if it fits within the MTU) or fragments the datagram and adds each fragment to the datalink output queue.

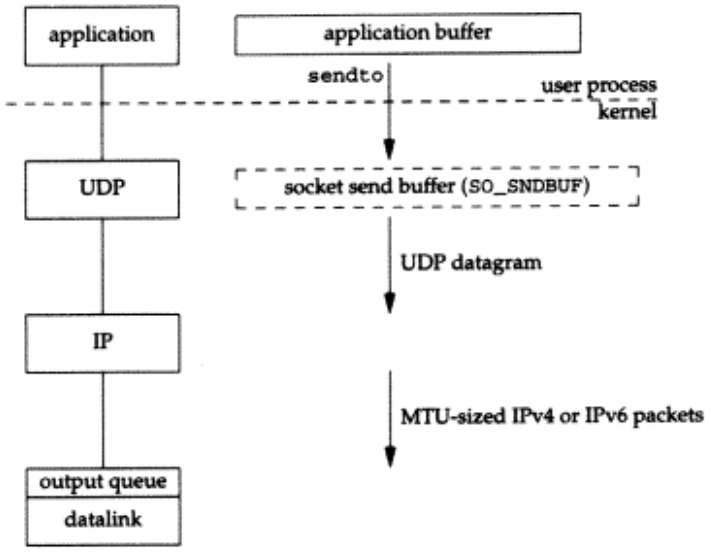

Figure 2.12 Steps and buffers involved when application writes to a UDP socket.

If a UDP application sends large datagrams (say 2000-byte datagrams), there is a much higher probability of fragmentation than with TCP, because TCP breaks the application data into MSS-sized chunks, something that has no counterpart in UDP.

The successful return from a write to a UDP socket tells us that either the datagram or all fragments of the datagram have been added to the datalink output queue. If there is no room on the queue for the datagram or one of its fragments, `ENOBUFS` is often returned to the application.

> Unfortunately some implementations do not return this error, giving the application no indication that the datagram was discarded without even being transmitted.

2.10 Standard Internet Services

Figure 2.13 lists several standard services that are provided by most implementations of TCP/IP. Notice that all are provided using both TCP and UDP and the port number is the same for both protocols.

Name	TCP port	UDP port	RFC	Description
echo	7	7	862	Server returns whatever the client sends.
discard	9	9	863	Server discards whatever the client sends.
daytime	13	13	867	Server returns the time and date in a human-readable format.
chargen	19	19	864	TCP server sends a continual stream of characters, until the connection is terminated by the client. UDP server sends a datagram containing a random number of characters each time the client sends a datagram.
time	37	37	868	Server returns the time as a 32-bit binary number. This number represents the number of seconds since midnight January 1, 1900, UTC.

Figure 2.13 Standard TCP/IP services provided by most implementations.

Often these services are provided by the `inetd` daemon on Unix hosts (Section 12.5). These standard services provide an easy testing facility, using the standard Telnet client. For example, the following tests both the daytime and echo servers.

```
solaris % telnet bsdi daytime
Trying 206.62.226.35...                 output by Telnet client
Connected to bsdi.kohala.com.           output by Telnet client
Escape character is '^]'.               output by Telnet client
Tue Mar 19 11:06:49 1996                output by daytime server
Connection closed by foreign host.      output by Telnet client (server closes connection)

solaris % telnet bsdi echo
Trying 206.62.226.35...                 output by Telnet client
Connected to bsdi.kohala.com.           output by Telnet client
Escape character is '^]'.               output by Telnet client
hello, world                            we type this
hello, world                            and it is echoed back by the server
^]                                      we type control and right bracket to talk to Telnet client
telnet> quit                            and tell client we are done
Connection closed.                      client closes the connection this time
```

In these two examples we type the name of the host and the name of the service (`daytime` and `echo`). These service names are mapped into the port numbers shown in Figure 2.13 by the `/etc/services` file, as we describe in Section 9.9.

Notice that when we connect to the `daytime` server, the server performs the active close, while with the `echo` server, the client performs the active close. Recall from Figure 2.4 that the end performing the active close is the end that goes through the TIME_WAIT state.

2.11 Protocol Usage by Common Internet Applications

Figure 2.14 summarizes the protocol usage of various common Internet applications.

Application	IP	ICMP	UDP	TCP
Ping		•		
Traceroute		•	•	
OSPF (routing protocol)	•			
RIP (routing protocol)			•	
BGP (routing protocol)				•
BOOTP (bootstrap protocol)			•	
DHCP (bootstrap protocol)			•	
NTP (time protocol)			•	
TFTP (trivial FTP)			•	
SNMP (network management)			•	
SMTP (electronic mail)				•
Telnet (remote login)				•
FTP (file transfer)				•
HTTP (the Web)				•
NNTP (network news)				•
DNS (domain name system)			•	•
NFS (network file system)			•	•
Sun RPC (remote procedure call)			•	•
DCE RPC (remote procedure call)			•	•

Figure 2.14 Protocol usage of various common Internet applications.

The first two applications, Ping and Traceroute, are diagnostic applications that use ICMP. Traceroute builds its own UDP packets to send and reads ICMP replies. The three popular routing protocols demonstrate the variety of transport protocols used by routing protocols. OSPF uses IP directly, using a raw socket, while RIP uses UDP and BGP uses TCP.

The next five are UDP-based applications, followed by five TCP applications. The final four are applications that use both UDP and TCP.

2.12 Summary

UDP is a simple, unreliable, connectionless protocol, while TCP is a complex, reliable, connection-oriented protocol. While most applications on the Internet use TCP (the Web, Telnet, FTP, and email), there is a need for both transport layers. In Section 20.4 we discuss the reasons to choose UDP instead of TCP.

TCP establishes connections using a three-way handshake and terminates connections using a four-packet exchange. When a TCP connection is established, it goes from the CLOSED state to the ESTABLISHED state, and when it is terminated, it goes back to the CLOSED state. There are 11 states in which a TCP connection can be, and a state transition diagram gives the rules on how to go between the states. Understanding this

diagram is essential to diagnosing problems using the `netstat` command and understanding what happens when we call functions such as `connect`, `accept`, and `close`.

TCP's TIME_WAIT state is a continual source of confusion with network programmers. This state exists to implement TCP's full-duplex connection termination (i.e., to handle the case of the final ACK being lost), and to allow old duplicate segments to expire in the network.

Exercises

2.1 We have mentioned IP versions 4 and 6. What happened to version 5 and what were versions 0, 1, 2, and 3? (*Hint*: Find the latest "Assigned Numbers" RFC. Feel free to skip ahead to the solution if you do not know how to obtain RFCs electronically.)

2.2 Where would you look to find more information about the protocol that is assigned IP version 5?

2.3 With Figure 2.11 we said that TCP assumes an MSS of 536 if it does not receive an MSS option from the peer. Why is this value used?

2.4 Draw a figure like Figure 2.5 for the daytime client–server in Chapter 1, assuming the server returns the 26 bytes of data in a single TCP segment.

2.5 A connection is established between a host on an Ethernet, whose TCP advertises an MSS of 1460, and a host on a token ring, whose TCP advertises an MSS of 4096. Neither host implements path MTU discovery. Watching the packets we never see more than 1460 bytes of data in either direction. Why?

2.6 In Figure 2.14 we said that OSPF uses IP directly. What is the value of the protocol field in the IPv4 header (Figure A.1) for these OSPF datagrams?

Part 2

Elementary Sockets

3

Sockets Introduction

3.1 Introduction

This chapter begins the description of the sockets API (application program interface). We begin with socket address structures, which will be found in almost every example in the text. These structures can be passed in two directions: from the process to the kernel, and from the kernel to the process. The latter case is an example of a value–result argument, and we will encounter other examples of these arguments throughout the text.

The address conversion functions convert between a text representation of an address and the binary value that goes into a socket address structure. Most existing IPv4 code uses `inet_addr` and `inet_ntoa`, but two new functions, `inet_pton` and `inet_ntop`, handle both IPv4 and IPv6.

One problem with these address conversion functions is that they are protocol dependent on the type of address being converted: IPv4 or IPv6. We develop a set of functions whose names begin with `sock_` that work with socket address structures in a protocol-independent fashion. We will use these throughout the text to make our code protocol independent.

3.2 Socket Address Structures

Most of the socket functions require a pointer to a socket address structure as an argument. Each supported protocol suite defines its own socket address structure. The names of these structures begin with `sockaddr_` and end with a unique suffix for each protocol suite.

IPv4 Socket Address Structure

An IPv4 socket address structure, commonly called an "Internet socket address structure," is named `sockaddr_in` and defined by including the `<netinet/in.h>` header. Figure 3.1 shows the Posix.1g definition.

```
struct in_addr {
  in_addr_t   s_addr;          /* 32-bit IPv4 address */
                               /* network byte ordered */
};

struct sockaddr_in {
  uint8_t         sin_len;     /* length of structure (16) */
  sa_family_t     sin_family;  /* AF_INET */
  in_port_t       sin_port;    /* 16-bit TCP or UDP port number */
                               /* network byte ordered */
  struct in_addr  sin_addr;    /* 32-bit IPv4 address */
                               /* network byte ordered */
  char            sin_zero[8]; /* unused */
};
```

Figure 3.1 The Internet (IPv4) socket address structure: `sockaddr_in`.

There are several points we need to make about socket address structures in general, using this example.

- The length member, `sin_len`, was added with 4.3BSD-Reno, when support for the OSI protocols was added (Figure 1.15). Before this release, the first member was `sin_family`, which was historically an `unsigned short`. Not all vendors support a length field for socket address structures and Posix.1g does not require this member. The datatype that we show, `uint8_t`, is typical, and datatypes of this form are new with Posix.1g (Figure 3.2).

  Having a length field simplifies the handling of variable-length socket address structures.

- Even if the length field is present, we need never set it and need never examine it, unless we are dealing with routing sockets (Chapter 17). It is used within the kernel by the routines that deal with socket address structures from various protocol families (e.g., the routing table code).

  The four socket functions that pass a socket address structure from the process to the kernel, `bind`, `connect`, `sendto`, and `sendmsg`, all go through the `sockargs` function in a Berkeley-derived implementation (p. 452 of TCPv2). This function copies the socket address structure from the process and explicitly sets its `sin_len` member to the size of the structure that was passed as an argument to these four functions. The five socket functions that pass a socket address structure from the kernel to the process, `accept`, `recvfrom`, `recvmsg`, `getpeername`, and `getsockname`, all set the `sin_len` member before returning to the process.

  Unfortunately there is normally no simple compile-time test to determine whether an implementation defines a length field for its socket address structures. In our code we test our own

HAVE_SOCKADDR_SA_LEN constant (Figure D.2), but whether to define this constant or not requires trying to compile a simple test program that uses this optional structure member and seeing if the compilation succeeds or not. We will see in Figure 3.4 that IPv6 implementations are required to define SIN6_LEN if socket address structures have a length field. Some IPv4 implementations (e.g., Digital Unix) provide the length field of the socket address structure to the application based on a compile-time option (e.g., _SOCKADDR_LEN). This feature provides compatibility for older programs.

- Posix.1g requires only three members in the structure: sin_family, sin_addr, and sin_port. It is acceptable for a Posix-compliant implementation to define additional structure members, and this is normal for an Internet socket address structure. Almost all implementations add the sin_zero member so that all socket address structures are at least 16 bytes in size.

- We show the Posix.1g datatypes for the s_addr, sin_family, and sin_port members. The in_addr_t datatype must be an unsigned integer type of at least 32 bits, in_port_t must be an unsigned integer type of at least 16 bits, and sa_family_t can be any unsigned integer type. The latter is normally an 8-bit unsigned integer if the implementation supports the length field, or an unsigned 16-bit integer if the length field is not supported. Figure 3.2 lists these three Posix-defined datatypes, along with some other Posix.1g datatypes that we will encounter.

Datatype	Description	Header
int8_t	signed 8-bit integer	<sys/types.h>
uint8_t	unsigned 8-bit integer	<sys/types.h>
int16_t	signed 16-bit integer	<sys/types.h>
uint16_t	unsigned 16-bit integer	<sys/types.h>
int32_t	signed 32-bit integer	<sys/types.h>
uint32_t	unsigned 32-bit integer	<sys/types.h>
sa_family_t	address family of socket address structure	<sys/socket.h>
socklen_t	length of socket address structure, normally uint32_t	<sys/socket.h>
in_addr_t	IPv4 address, normally uint32_t	<netinet/in.h>
in_port_t	TCP or UDP port, normally uint16_t	<netinet/in.h>

Figure 3.2 Datatypes required by Posix.1g.

- You will also encounter the datatypes u_char, u_short, u_int, and u_long, which are all unsigned. Posix.1g defines these with the note that they are obsolete. They are provided for backward compatibility.

- Both the IPv4 address and the TCP or UDP port number are always stored in the structure in network byte order. We must be cognizant of this when using these members. (We say more about the difference between host byte order and network byte order in Section 3.4.)
- The 32-bit IPv4 address can be accessed in two different ways. For example, if serv is defined as an Internet socket address structure, then serv.sin_addr references

the 32-bit IPv4 address as an `in_addr` structure, while `serv.sin_addr.s_addr` references the same 32-bit IPv4 address as an `in_addr_t` (typically an unsigned 32-bit integer). We must be certain that we are referencing the IPv4 address correctly, especially when it is used as an argument to a function, because compilers often pass structures differently from integers.

> The reason the `sin_addr` member is a structure, and not just an `unsigned long`, is historical. Earlier releases (4.2BSD) defined the `in_addr` structure as a `union` of various structures, to allow access to each of the 4 bytes and to both of the 16-bit values contained within the 32-bit IPv4 address. This was used with class A, B, and C addresses to fetch the appropriate bytes of the address. But with the advent of subnetting and then the disappearance of the various address classes with classless addressing (Section A.4), the need for the `union` disappeared. Most systems today have done away with the `union` and just define `in_addr` as a structure with a single `unsigned long` member.

- The `sin_zero` member is unused, but we *always* set it to 0 when filling in one of these structures. By convention, we always set the entire structure to 0 before filling it in, not just the `sin_zero` member.

  > Although most uses of the structure do not require that this member be 0, when binding a nonwildcard IPv4 address, this member must be 0 (pp. 731–732 of TCPv2).

- Socket address structures are used only on a given host: the structure itself is not communicated between different hosts although certain fields (e.g., the IP address and port) are used for communication.

Generic Socket Address Structure

Socket address structures are *always* passed by reference when passed as an argument to any of the socket functions. But the socket functions that take one of these pointers as an argument must deal with socket address structures from *any* of the supported protocol families.

A problem is how to declare the type of pointer that is passed. With ANSI C the solution is simple: `void *` is the generic pointer type. But the socket functions predate ANSI C and the solution chosen in 1982 was to define a *generic* socket address structure in the `<sys/socket.h>` header, which we show in Figure 3.3.

```
struct sockaddr {
  uint8_t       sa_len;
  sa_family_t   sa_family;    /* address family: AF_xxx value */
  char          sa_data[14];  /* protocol-specific address */
};
```

Figure 3.3 The generic socket address structure: `sockaddr`.

The socket functions are then defined as taking a pointer to the generic socket address structure, as shown here in the ANSI C function prototype for the `bind` function:

```
int bind(int, struct sockaddr *, socklen_t);
```

This requires that any calls to these functions must cast the pointer to the protocol-specific socket address structure to be a pointer to a generic socket address structure. For example,

```
struct sockaddr_in   serv;      /* IPv4 socket address structure */

/* fill in serv{} */

bind(sockfd, (struct sockaddr *) &serv, sizeof(serv));
```

If we omit the cast "`(struct sockaddr *)`" the C compiler generates a warning of the form "warning: passing arg 2 of 'bind' from incompatible pointer type" assuming the system's headers have an ANSI C prototype for the `bind` function.

From an application programmer's point of view, the *only* use of these generic socket address structures is to cast pointers to protocol-specific structures.

> Recall in Section 1.2 that in our `unp.h` header we define `SA` to be the string "`struct sockaddr`" just to shorten the code that we must write to cast these pointers.
>
> From the kernel's perspective another reason for using pointers to generic socket address structures as arguments is that the kernel must take the caller's pointer, cast it to a `struct sockaddr *` and then look at the value of `sa_family` to determine the type of structure. But from an application programmer's perspective, it would be simpler if the pointer type were `void *`, omitting the need for the explicit cast.

IPv6 Socket Address Structure

The IPv6 socket address is defined by including the `<netinet/in.h>` header, and we show it in Figure 3.4.

```
struct in6_addr {
  uint8_t  s6_addr[16];          /* 128-bit IPv6 address */
                                 /* network byte ordered */
};

#define SIN6_LEN      /* required for compile-time tests */

struct sockaddr_in6 {
  uint8_t         sin6_len;      /* length of this struct (24) */
  sa_family_t     sin6_family;   /* AF_INET6 */
  in_port_t       sin6_port;     /* transport layer port# */
                                 /* network byte ordered */
  uint32_t        sin6_flowinfo; /* priority & flow label */
                                 /* network byte ordered */
  struct in6_addr sin6_addr;     /* IPv6 address */
                                 /* network byte ordered */
};
```

Figure 3.4 IPv6 socket address structure: `sockaddr_in6`.

The extensions to the sockets API for IPv6 are defined in RFC 2133 [Gilligan et al. 1997]. Posix.1g says nothing about IPv6. But some of the datatypes in Figure 3.4 differ from RFC 2133 because we use what would be the Posix.1g datatypes in the figure, but these were finalized in a Posix.1g draft that came after RFC 2133.

Note the following points about Figure 3.4:

- The `SIN6_LEN` constant must be defined if the system supports the length member for socket address structures.
- The IPv6 family is `AF_INET6`, whereas the IPv4 family is `AF_INET`.
- The members in this structure are ordered so that if the `sockaddr_in6` structure is 64-bit aligned, so is the 128-bit `sin6_addr` member. On some 64-bit processors, data accesses of 64-bit values are optimized if stored on a 64-bit boundary.
- The `sin6_flowinfo` member is divided into three fields:
  - the low-order 24 bits are the flow label,
  - the next 4 bits are the priority,
  - the next 4 bits are reserved.

  The flow label and priority fields are described with Figure A.2. We note that the use of the priority field is still a research topic.

Comparison of Socket Address Structures

Figure 3.5 shows a comparison of the four socket address structures that we encounter in this text: IPv4, IPv6, Unix domain (Figure 14.1), and datalink (Figure 17.1). In this figure we assume that the socket address structures all contain a 1-byte length field, that the family field also occupies 1 byte, and that any field that must be at least some number of bits is exactly that number of bits. Two of the socket address structures are fixed length, while the Unix domain structure and the datalink structure are variable length. To handle variable-length structures whenever we pass a pointer to a socket address structure as an argument to one of the socket functions, we pass its length as another argument. We show the size in bytes (for the 4.4BSD implementation) of the fixed-length structures beneath each structure.

The `sockaddr_un` structure itself is not variable length (Figure 14.1), but the amount of information—the pathname within the structure—is variable length. When passing pointers to these structures, we must be careful how we handle the length field, both the length field in the socket address structure itself (if supported by the implementation), and the length to and from the kernel.

This figure shows a style that we follow throughout the text: structure names are always shown in a bolder font followed by braces, as in **`sockaddr_in{}`**.

We noted earlier that the length field was added to all the socket address structures with the 4.3BSD Reno release. Had the length field been present with the original release of sockets,

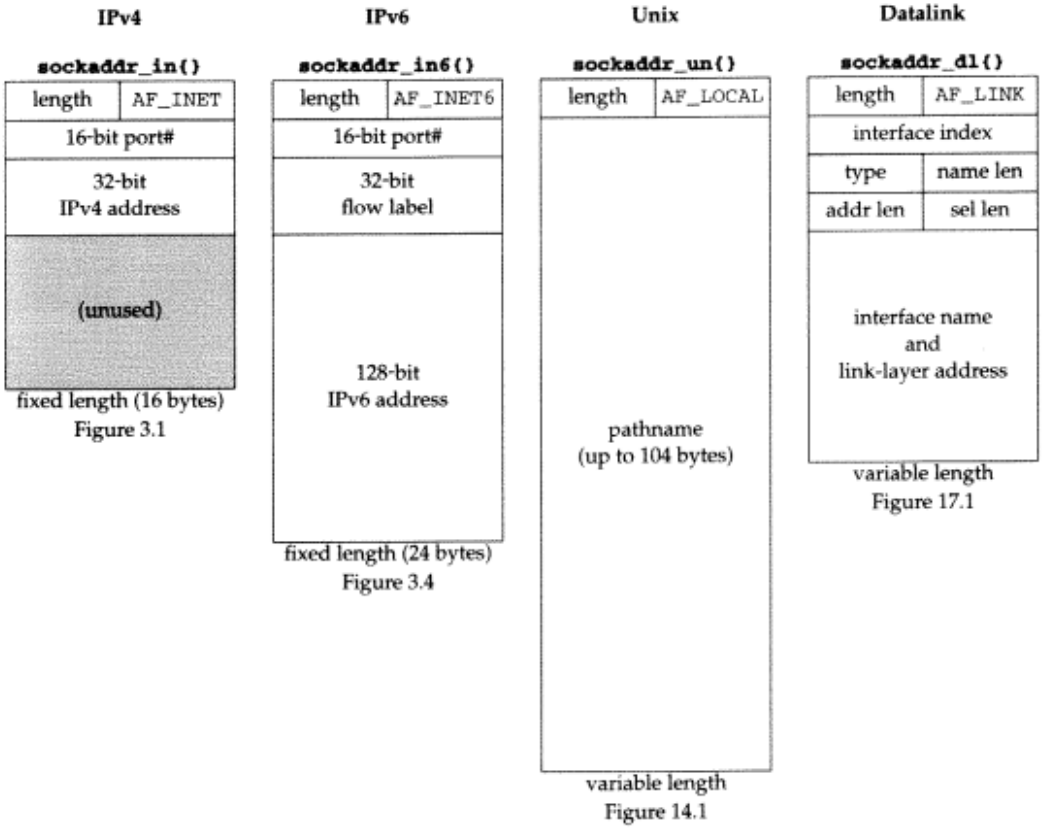

Figure 3.5 Comparison of various socket address structures.

there would be no need for the length argument to all the socket functions: the third argument to `bind` and `connect`, for example. Instead, the size of the structure could be contained in the length field of the structure.

3.3 Value–Result Arguments

We mentioned that when a socket address structure is passed to any of the socket functions, it is always passed by reference. That is, a pointer to the structure is passed. The length of the structure is also passed as an argument. But the way in which the length is passed depends on which direction the structure is being passed: from the process to the kernel, or vice versa.

1. The three functions `bind`, `connect`, and `sendto` pass a socket address structure from the process to the kernel. One argument to these three functions is the pointer to the socket address structure and another argument is the integer size of the structure, as in

```
struct sockaddr_in  serv;

/* fill in serv{} */
connect(sockfd, (SA *) &serv, sizeof(serv));
```

Since the kernel is passed both the pointer and the size of what the pointer points to, it knows exactly how much data to copy from the process into the kernel. Figure 3.6 shows this scenario.

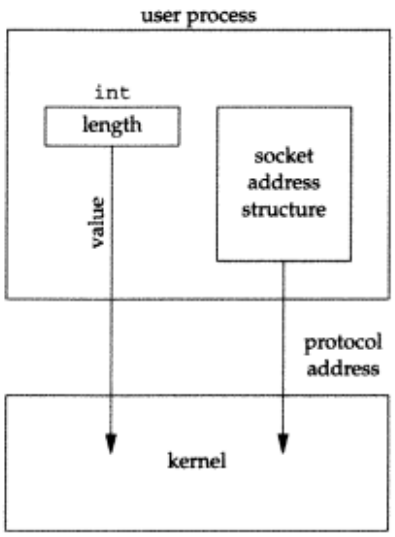

Figure 3.6 Socket address structure passed from process to kernel.

We will see in the next chapter that the datatype for the size of a socket address structure is actually `socklen_t` and not `int`, but Posix.1g recommends that `socklen_t` be defined as `uint32_t`.

2. The four functions `accept`, `recvfrom`, `getsockname`, and `getpeername` pass a socket address structure from the kernel to the process, the reverse direction from the previous scenario. Two of the arguments to these four functions are the pointer to the socket address structure along with a pointer to an integer containing the size of the structure, as in

```
struct sockaddr_un  cli;    /* Unix domain */
socklen_t  len;

len = sizeof(cli);          /* len is a value */
getpeername(unixfd, (SA *) &cli, &len);
/* len may have changed */
```

The reason that the size changes from an integer to be a pointer to an integer is because the size is both a *value* when the function is called (it tells the kernel the size of the structure, so that the kernel does not write past the end of the structure when filling it in) and a *result* when the function returns (it tells the process

how much information the kernel actually stored in the structure). This type of argument is called a *value–result* argument. Figure 3.7 shows this scenario.

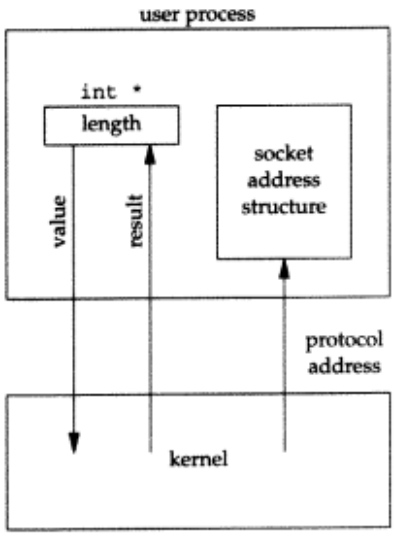

Figure 3.7 Socket address structure passed from kernel to process.

We will see an example of value–result arguments in Figure 4.11.

> We have been talking about socket address structures being passed between the process and the kernel. For an implementation such as 4.4BSD, where all the socket functions are system calls within the kernel, this is correct. But in some implementations, notably System V, the socket functions are just library functions that execute as part of a normal user process. How these functions interface with the protocol stack in the kernel is an implementation detail that normally does not affect us. Nevertheless, for simplicity we continue to talk about these structures as being passed between the process and the kernel by functions such as `bind` and `connect`. (We will see in Section C.1 that System V implementations do indeed pass user's socket address structures between the process and the kernel, but as part of streams messages.)
>
> Two other functions pass socket address structures: `recvmsg` and `sendmsg` (Section 13.5). But we will see that the length field is not a function argument but a structure member.

When using value–result arguments for the length of socket address structures, if the socket address structure is fixed length (Figure 3.5), the value returned by the kernel will always be that fixed size: 16 for an IPv4 `sockaddr_in` and 24 for an IPv6 `sockaddr_in6`, for example. But with a variable-length socket address structure (e.g., a Unix domain `sockaddr_un`) the value returned can be less than the maximum size of the structure (as we will see with Figure 14.2).

With network programming the most common example of a value–result argument is the length of a returned socket address structure. But we will encounter other value–result arguments in this text:

- The middle three arguments for the `select` function (Section 6.3).
- The length argument for the `getsockopt` function (Section 7.2).
- The `msg_namelen` and `msg_controllen` members of the `msghdr` structure, when used with `recvmsg` (Section 13.5).
- The `ifc_len` member of the `ifconf` structure (Figure 16.2).
- The first of the two length arguments for the `sysctl` function (Section 17.4).

3.4 Byte Ordering Functions

Consider a 16-bit integer that is made up of 2 bytes. There are two ways to store the 2 bytes in memory: with the low-order byte at the starting address, known as *little-endian* byte order, or with the high-order byte at the starting address, known as *big-endian* byte order. We show these two formats in Figure 3.8.

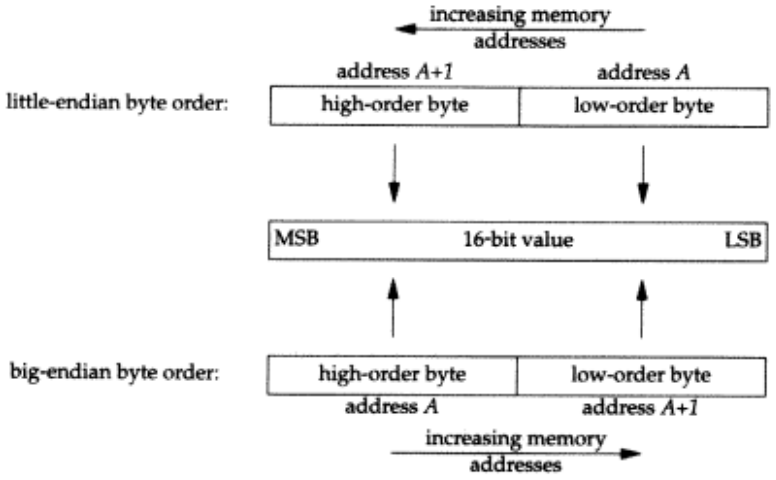

Figure 3.8 Little-endian byte order and big-endian byte order for a 16-bit integer.

In this figure we show increasing memory addresses going from right to left in the top, and from left to right in the bottom. We also show the *MSB* (most significant bit) as the leftmost bit of the 16-bit value and the *LSB* (least significant bit) as the rightmost bit.

> The terms "little endian" and "big endian" indicate which end of the multibyte value, the little end or the big end, is stored at the starting address of the value.

Unfortunately there is no standard between these two byte orderings and we encounter systems that use both formats. We refer to the byte ordering used by a given system as the *host byte order*. The program shown in Figure 3.9 prints the host byte order.

intro/byteorder.c

```
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     union {
 6         short   s;
 7         char    c[sizeof(short)];
 8     } un;

 9     un.s = 0x0102;
10     printf("%s: ", CPU_VENDOR_OS);
11     if (sizeof(short) == 2) {
12         if (un.c[0] == 1 && un.c[1] == 2)
13             printf("big-endian\n");
14         else if (un.c[0] == 2 && un.c[1] == 1)
15             printf("little-endian\n");
16         else
17             printf("unknown\n");
18     } else
19         printf("sizeof(short) = %d\n", sizeof(short));

20     exit(0);
21 }
```

intro/byteorder.c

Figure 3.9 Program to determine host byte order.

We store the 2-byte value `0x0102` into the short integer and then look at the two consecutive bytes `c[0]` (the address *A* in Figure 3.8) and `c[1]` (the address *A+1* in Figure 3.8) to determine the byte order.

The string `CPU_VENDOR_OS` is determined by the GNU `autoconf` program when the software in this book is configured, and it identifies the CPU type, vendor, and operating system release. We show some examples here in the output from this program when run on the various systems in Figure 1.16.

```
aix % byteorder
powerpc-ibm-aix4.2.0.0: big-endian

alpha % byteorder
alpha-dec-osf4.0: little-endian

bsdi % byteorder
i386-pc-bsdi3.0: little-endian

gemini % byteorder
sparc-sun-sunos4.1.4: big-endian

hpux % byteorder
hppa1.1-hp-hpux10.30: big-endian

linux % byteorder
i586-pc-linux-gnu: little-endian

solaris % byteorder
sparc-sun-solaris2.5.1: big-endian
```

```
unixware % byteorder
i386-univel-sysv4.2MP: little-endian
```

We have talked about the byte ordering of a 16-bit integer, and obviously the same discussion applies to a 32-bit integer.

> There are systems that can change between little-endian and big-endian byte ordering ([Dewar and Smosna 1990]), either when the system is reset (MIPS 2000), or at any point while a program is running (Intel i860).

We must deal with the byte ordering differences as network programmers because networking protocols must specify a *network byte order*. For example, in a TCP segment there is a 16-bit port number and a 32-bit IPv4 address. The sending protocol stack and the receiving protocol stack must agree on the order in which the bytes of these multibyte fields are transmitted. The Internet protocols use big-endian byte ordering for these multibyte integers.

In theory an implementation could store the fields in a socket address structure in host byte order and then convert to and from the network byte order when moving the fields to and from the protocol headers, saving us from having to worry about this detail. But both history and Posix.1g specify that certain fields in the socket address structures be maintained in network byte order. Our concern is therefore converting between the host byte order and the network byte order. We use the following four functions to convert between these two byte orders.

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

uint16_t htons(uint16_t host16bitvalue);

uint32_t htonl(uint32_t host32bitvalue);
```

Both return: value in network byte order

```
uint16_t ntohs(uint16_t net16bitvalue);

uint32_t ntohl(uint32_t net32bitvalue);
```

Both return: value in host byte order

In the names of these functions `h` stands for *host*, `n` stands for *network*, `s` stands for *short*, and `l` stands for *long*. The terms *short* and *long* are historical artifacts from the Digital VAX implementation of 4.2BSD. We should instead think of `s` as a 16-bit value (such as a TCP or UDP port number) and `l` as a 32-bit value (such as an IPv4 address). Indeed, on the 64-bit Digital Alpha, a long integer occupies 64 bits, yet the `htonl` and `ntohl` functions operate on 32-bit values.

When using these functions we do not care about the actual values (big endian or little endian) for the host byte order and the network byte order. What we must do is be certain to call the appropriate function to convert a given value between the host and network byte order. On those systems that have the same byte ordering as the Internet protocols (big endian), these four functions are usually defined as null macros.

We talk more about the byte ordering problem, with respect to the data contained in a network packet, as opposed to the fields in the protocol headers, in Section 5.18 and Exercise 5.8.

We have not defined the term *byte*. We use the term to mean an 8-bit quantity since almost all current computer systems use 8-bit bytes. Most Internet standards use the term *octet* instead of byte to mean an 8-bit quantity. This started in the early days of TCP/IP because much of the early work was done on systems such as the DEC-10, which did not use 8-bit bytes.

> A common network programming error in the 1980s was to develop code on Sun workstations (big-endian Motorola 68000s) and forget to call any of these four functions. The code worked fine on these workstations but would not work when ported to little-endian machines (such as VAXes).

3.5 Byte Manipulation Functions

There are two groups of functions that operate on multibyte fields, without interpreting the data, and without assuming that the data is a null-terminated C string. We need these types of functions when dealing with socket address structures, because we need to manipulate fields such as IP addresses, which can contain bytes of 0, but these fields are not C character strings. The functions beginning with `str` (for string), defined by including the `<string.h>` header, deal with null-terminated C character strings.

The first group of functions, whose names begin with `b` (for byte), are from 4.2BSD and are still provided by almost any system that supports the socket functions. The second group of functions, whose names begin with `mem` (for memory), are from the ANSI C standard and are provided with any system that supports an ANSI C library.

We first show the Berkeley-derived functions, although the only one of these that we use in this text is `bzero`. (We use it because it has only two arguments and is easier to remember than the three-argument `memset` function, as explained on p. 8.) The other two functions, `bcopy` and `bcmp`, you may encounter in existing applications.

```
#include <strings.h>

void bzero(void *dest, size_t nbytes);

void bcopy(const void *src, void *dest, size_t nbytes);

int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);
```

Returns: 0 if equal, nonzero if unequal

> This is our first encounter with the ANSI C `const` qualifier. In the three uses here it indicates that what is pointed to by the pointer with this qualification, *src*, *ptr1*, and *ptr2*, is not modified by the function. Worded another way, the memory pointed to by the `const` pointer is read but not modified by the function.

`bzero` sets the specified number of bytes to 0 in the destination. We often use this function to initialize a socket address structure to 0. `bcopy` moves the specified number of bytes from the source to the destination. `bcmp` compares two arbitrary byte strings. The return value is zero if the two byte strings are identical; otherwise it is nonzero.

The following functions are the ANSI C functions:

```
#include <string.h>

void *memset(void *dest, int c, size_t len);

void *memcpy(void *dest, const void *src, size_t nbytes);

int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);
```

Returns: 0 if equal, <0 or >0 if unequal (see text)

`memset` sets the specified number of bytes to the value *c* in the destination. `memcpy` is similar to `bcopy` but the order of the two pointer arguments is swapped. `bcopy` correctly handles overlapping fields, while the behavior of `memcpy` is undefined if the source and destination overlap. The ANSI C `memmove` function must be used when the fields overlap (Exercise 30.3).

> One way to remember the order of the two pointers for `memcpy` is to remember that they are written in the same left-to-right order as an assignment statement in C:
>
> ```
> dest = src;
> ```
>
> One way to remember the order of the final two arguments to `memset` is to realize that all of the ANSI C `memXXX` functions require a length argument, and it is always the final argument.

`memcmp` compares two arbitrary byte strings and returns 0 if they are identical. If not identical, the return value is either greater than 0 or less than 0, depending whether the first unequal byte pointed to by *ptr1* is greater than or less than the corresponding byte pointed to by *ptr2*. The comparison is done assuming the two unequal bytes are `unsigned chars`.

3.6 `inet_aton`, `inet_addr`, and `inet_ntoa` Functions

There are two groups of address conversion functions that we describe in this section and the next. They convert Internet addresses between ASCII strings (what humans prefer to use) and network byte ordered binary values (values that are stored in socket address structures).

1. `inet_aton`, `inet_ntoa`, and `inet_addr` convert an IPv4 address between a dotted-decimal string (e.g., `"206.62.226.33"`) and its 32-bit network byte ordered binary value. You will probably encounter these functions in lots of existing code.

2. The newer functions `inet_pton` and `inet_ntop` handle both IPv4 and IPv6 addresses. We describe these two functions in the next section and use them throughout the text.

```
#include <arpa/inet.h>

int inet_aton(const char *strptr, struct in_addr *addrptr);
```

Returns: 1 if string was valid, 0 on error

```
in_addr_t inet_addr(const char *strptr);
```

Returns: 32-bit binary network byte ordered IPv4 address; INADDR_NONE if error

```
char *inet_ntoa(struct in_addr inaddr);
```

Returns: pointer to dotted-decimal string

The first of these, `inet_aton`, converts the C character string pointed to by *strptr* into its 32-bit binary network byte ordered value, which is stored through the pointer *addrptr*. If successful, 1 is returned; otherwise 0 is returned.

> An undocumented feature of `inet_aton` is that if *addrptr* is a null pointer, the function still performs its validation of the input string but does not store any result.

`inet_addr` does the same conversion, returning the 32-bit binary network byte ordered value as the return value. The problem with this function is that all 2^{32} possible binary values are valid IP addresses (0.0.0.0 through 255.255.255.255), but the function returns the constant `INADDR_NONE` (typically 32 one-bits) on an error. This means the dotted-decimal string 255.255.255.255 (the IPv4 limited broadcast address, Section 18.2) cannot be handled by this function, since its binary value appears to indicate failure of the function.

> Many older versions of Ping output the error "unknown host" if we try execute `ping 255.255.255.255`. The reason for this incorrect error is that `inet_addr` appears to fail, so it tries to look up the dotted-decimal string as a hostname, which fails.
>
> Another potential problem with `inet_addr` is that some manual pages state that it returns –1 on an error, instead of `INADDR_NONE`. This can lead to problems, depending on the C compiler, when comparing the return value of the function (an unsigned value) to a negative constant.

Today `inet_addr` is deprecated and any new code should use `inet_aton` instead. Better still is to use the newer functions described in the next section, which handle both IPv4 and IPv6.

The `inet_ntoa` function converts a 32-bit binary network byte ordered IPv4 address into its corresponding dotted-decimal string. The string pointed to by the return value of the function resides in static memory. This means the function is not reentrant, which we discuss in Section 11.14. Finally, notice that this function takes a structure as its argument, not a pointer to a structure.

Functions that take actual structures as arguments are rare. It is more common to pass a pointer to the structure.

3.7 `inet_pton` and `inet_ntop` Functions

These two functions are new with IPv6 and work with both IPv4 and IPv6 addresses. We use these two functions throughout the text. The letters `p` and `n` stand for *presentation* and *numeric*. The presentation format for an address is often an ASCII string and the numeric format is the binary value that goes into a socket address structure.

```
#include  <arpa/inet.h>

int inet_pton(int family, const char *strptr, void *addrptr);
```

Returns: 1 if OK, 0 if input not a valid presentation format, –1 on error

```
const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t len);
```

Returns: pointer to result if OK, NULL on error

The *family* argument for both functions is either `AF_INET` or `AF_INET6`. If *family* is not supported, both functions return an error with `errno` set to `EAFNOSUPPORT`.

The first function tries to convert the string pointed to by *strptr*, storing the binary result through the pointer *addrptr*. If successful, the return value is 1. If the input string is not a valid presentation format for the specified *family*, 0 is returned.

`inet_ntop` does the reverse conversion, from numeric (*addrptr*) to presentation (*strptr*). The *len* argument is the size of the destination, to prevent the function from overflowing the caller's buffer. To help specify this size, the following two definitions are defined by including the `<netinet/in.h>` header:

```
#define  INET_ADDRSTRLEN    16    /* for IPv4 dotted-decimal */
#define  INET6_ADDRSTRLEN   46    /* for IPv6 hex string */
```

If *len* is too small to hold the resulting presentation format, including the terminating null, a null pointer is returned and `errno` is set to `ENOSPC`.

The *strptr* argument to `inet_ntop` cannot be a null pointer. The caller must allocate memory for the destination and specify its size. On success this pointer is the return value of the function.

Figure 3.10 summarizes the five functions that we have described in this section and the previous section.

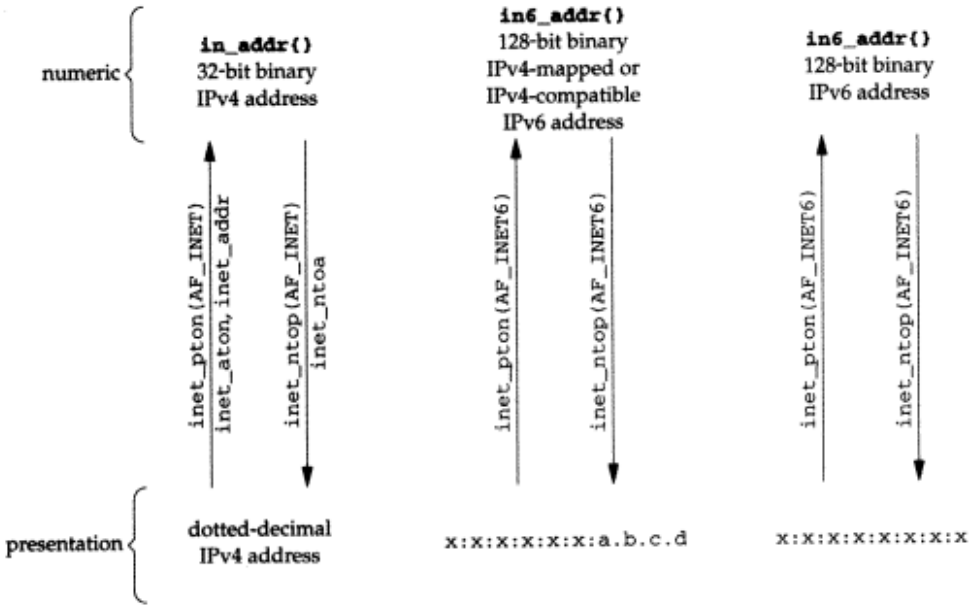

Figure 3.10 Summary of address conversion functions.

Example

Even if your system does not yet include support for IPv6, you can start using these newer functions by replacing calls of the form

```
foo.sin_addr.s_addr = inet_addr(cp);
```

with

```
inet_pton(AF_INET, cp, &foo.sin_addr);
```

and replacing calls of the form

```
ptr = inet_ntoa(foo.sin_addr);
```

with

```
char  str[INET_ADDRSTRLEN];
ptr = inet_ntop(AF_INET, &foo.sin_addr, str, sizeof(str));
```

Figure 3.11 shows a simple definition of `inet_pton` that supports only IPv4. Similarly, Figure 3.12 shows a simple version of `inet_ntop` that supports only IPv4.

libfree/inet_pton_ipv4.c

```
10 int
11 inet_pton(int family, const char *strptr, void *addrptr)
12 {
13     if (family == AF_INET) {
14         struct in_addr in_val;

15         if (inet_aton(strptr, &in_val)) {
16             memcpy(addrptr, &in_val, sizeof(struct in_addr));
17             return (1);
18         }
19         return (0);
20     }
21     errno = EAFNOSUPPORT;
22     return (-1);
23 }
```

libfree/inet_pton_ipv4.c

Figure 3.11 Simple version of `inet_pton` that supports only IPv4.

libfree/inet_ntop_ipv4.c

```
 8 const char *
 9 inet_ntop(int family, const void *addrptr, char *strptr, size_t len)
10 {
11     const u_char *p = (const u_char *) addrptr;

12     if (family == AF_INET) {
13         char    temp[INET_ADDRSTRLEN];

14         snprintf(temp, sizeof(temp), "%d.%d.%d.%d",
15                  p[0], p[1], p[2], p[3]);
16         if (strlen(temp) >= len) {
17             errno = ENOSPC;
18             return (NULL);
19         }
20         strcpy(strptr, temp);
21         return (strptr);
22     }
23     errno = EAFNOSUPPORT;
24     return (NULL);
25 }
```

libfree/inet_ntop_ipv4.c

Figure 3.12 Simple version of `inet_ntop` that supports only IPv4.

3.8 sock_ntop and Related Functions

A basic problem with inet_ntop is that it requires the caller to pass a pointer to a binary address. This address is normally contained in a socket address structure, requiring the caller to know the format of the structure and the address family. That is, to use it we must write code of the form

```
struct sockaddr_in   addr;

inet_ntop(AF_INET, &addr.sin_addr, str, sizeof(str));
```

for IPv4, or

```
struct sockaddr_in6   addr6;

inet_ntop(AF_INET6, &addr6.sin6_addr, str, sizeof(str));
```

for IPv6. This makes our code protocol dependent.

To solve this we will write our own function named sock_ntop that takes a pointer to a socket address structure, looks inside the structure, and calls the appropriate function to return the presentation format of the address.

```
#include "unp.h"

char *sock_ntop(const struct sockaddr *sockaddr, socklen_t addrlen);
```

Returns: nonnull pointer if OK, NULL on error

This is the notation we use for functions of our own that we use throughout the book that are not standard system functions: the box around the function prototype and return value is dashed. The header that is included at the beginning is usually our unp.h header.

sockaddr points to a socket address structure whose length is *addrlen*. The function uses its own static buffer to hold the result and a pointer to this buffer is the return value.

Notice that using static storage for the result prevents the function from being *reentrant* or *thread-safe*. We talk more about this in Section 11.14. We make this design decision for this function to allow us to easily call it from the simple examples in the book.

The presentation format is the dotted-decimal form of an IPv4 address or the hex string form of an IPv6 address, followed by a terminator (we use a period, similar to netstat), followed by the decimal port number, followed by a null character. Hence the buffer size must be at least INET_ADDRSTRLEN plus 6 bytes for IPv4 (16 + 6 = 22), or INET6_ADDRSTRLEN plus 6 bytes for IPv6 (46 + 6 = 52).

We show the source code for only the AF_INET case in Figure 3.13.

lib/sock_ntop.c

```
 5 char *
 6 sock_ntop(const struct sockaddr *sa, socklen_t salen)
 7 {
 8     char    portstr[7];
 9     static char str[128];       /* Unix domain is largest */

10     switch (sa->sa_family) {
11     case AF_INET:{
12             struct sockaddr_in *sin = (struct sockaddr_in *) sa;

13             if (inet_ntop(AF_INET, &sin->sin_addr, str, sizeof(str)) == NULL)
14                 return (NULL);
15             if (ntohs(sin->sin_port) != 0) {
16                 snprintf(portstr, sizeof(portstr), ".%d", ntohs(sin->sin_port));
17                 strcat(str, portstr);
18             }
19             return (str);
20         }
```

lib/sock_ntop.c

Figure 3.13 Our `sock_ntop` function.

There are a few other functions that we define to operate on socket address structures, and these will simplify the portability of our code between IPv4 and IPv6.

```
#include "unp.h"

int sock_bind_wild(int sockfd, int family);
```

Returns: 0 if OK, –1 on error

```
int sock_cmp_addr(const struct sockaddr *sockaddr1,
                  const struct sockaddr *sockaddr2, socklen_t addrlen);
```

Returns: 0 if the addresses are of the same family and equal, else nonzero

```
int sock_cmp_port(const struct sockaddr *sockaddr1,
                  const struct sockaddr *sockaddr2, socklen_t addrlen);
```

Returns: 0 if the addresses are of the same family and the ports are equal, else nonzero

```
int sock_get_port(const struct sockaddr *sockaddr, socklen_t addrlen);
```

Returns: nonnegative port number for IPv4 or IPv6 address, else –1

```
char *sock_ntop_host(const struct sockaddr *sockaddr, socklen_t addrlen);
```

Returns: nonnull pointer if OK, NULL on error

```
void sock_set_addr(const struct sockaddr *sockaddr, socklen_t addrlen, void *ptr);

void sock_set_port(const struct sockaddr *sockaddr, socklen_t addrlen, int port);

void sock_set_wild(struct sockaddr *sockaddr, socklen_t addrlen);
```

`sock_bind_wild` binds the wildcard address and an ephemeral port to a socket. `sock_cmp_addr` compares the address portion of two socket address structures and `sock_cmp_port` compares the port number of two socket address structures. `sock_get_port` returns just the port number and `sock_ntop_host` converts just the host portion of a socket address structure to presentation format (not the port number). `sock_set_addr` sets just the address portion of a socket address structure to the value pointed to by *ptr* and `sock_set_port` sets just the port number of a socket address structure. `sock_set_wild` sets the address portion of a socket address structure to the wildcard. As with all of the functions in the text, we provide a wrapper function whose name begins with `S` for all of these functions that return other than `void` and normally call the wrapper function from our programs. We do not show the source code for all these functions, but it is freely available (see the Preface).

3.9 readn, writen, and readline Functions

Stream sockets (e.g., TCP sockets) exhibit a behavior with the `read` and `write` functions that differs from normal file I/O. A `read` or `write` on a stream socket might input or output fewer bytes than requested, but this is not an error condition. The reason is that buffer limits might be reached for the socket in the kernel. All that is required is for the caller to invoke the `read` or `write` function again, to input or output the remaining bytes. (Some versions of Unix also exhibit this behavior when writing more than 4096 bytes to a pipe.) This scenario is always a possibility on a stream socket with `read`, but is normally seen with `write` only if the socket is nonblocking. Nevertheless, we always call our `writen` function instead of `write`, in case the implementation returns a short count.

We provide the following three functions that we use whenever we read from or write to a stream socket.

```
#include "unp.h"

ssize_t readn(int filedes, void *buff, size_t nbytes);

ssize_t writen(int filedes, const void *buff, size_t nbytes);

ssize_t readline(int filedes, void *buff, size_t maxlen);
```

All return: number of bytes read or written, –1 on error

Figure 3.14 shows the `readn` function. Figure 3.15 shows the `writen` function, and Figure 3.16 shows the `readline` function.

lib/readn.c

```
1 #include    "unp.h"

2 ssize_t                           /* Read "n" bytes from a descriptor. */
3 readn(int fd, void *vptr, size_t n)
4 {
5     size_t  nleft;
6     ssize_t nread;
7     char   *ptr;

8     ptr = vptr;
9     nleft = n;
10    while (nleft > 0) {
11        if ( (nread = read(fd, ptr, nleft)) < 0) {
12            if (errno == EINTR)
13                nread = 0;      /* and call read() again */
14            else
15                return (-1);
16        } else if (nread == 0)
17            break;              /* EOF */

18        nleft -= nread;
19        ptr += nread;
20    }
21    return (n - nleft);         /* return >= 0 */
22 }
```

lib/readn.c

Figure 3.14 `readn` function: read *n* bytes from a descriptor.

lib/writen.c

```
1 #include    "unp.h"

2 ssize_t                           /* Write "n" bytes to a descriptor. */
3 writen(int fd, const void *vptr, size_t n)
4 {
5     size_t  nleft;
6     ssize_t nwritten;
7     const char *ptr;

8     ptr = vptr;
9     nleft = n;
10    while (nleft > 0) {
11        if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
12            if (errno == EINTR)
13                nwritten = 0;   /* and call write() again */
14            else
15                return (-1);    /* error */
16        }
17        nleft -= nwritten;
18        ptr += nwritten;
19    }
20    return (n);
21 }
```

lib/writen.c

Figure 3.15 `writen` function: write *n* bytes to a descriptor.

test/readline1.c

```
 1 #include    "unp.h"

 2 ssize_t
 3 readline(int fd, void *vptr, size_t maxlen)
 4 {
 5     ssize_t n, rc;
 6     char    c, *ptr;

 7     ptr = vptr;
 8     for (n = 1; n < maxlen; n++) {
 9       again:
10         if ( (rc = read(fd, &c, 1)) == 1) {
11             *ptr++ = c;
12             if (c == '\n')
13                 break;           /* newline is stored, like fgets() */
14         } else if (rc == 0) {
15             if (n == 1)
16                 return (0);      /* EOF, no data read */
17             else
18                 break;           /* EOF, some data was read */
19         } else {
20             if (errno == EINTR)
21                 goto again;
22             return (-1);         /* error, errno set by read() */
23         }
24     }

25     *ptr = 0;                    /* null terminate like fgets() */
26     return (n);
27 }
```

test/readline1.c

Figure 3.16 `readline` function: read a text line from a descriptor, 1 byte at a time.

Our three functions look for the error `EINTR` (the system call was interrupted by a caught signal, which we discuss in more detail in Section 5.9) and continue reading or writing if the error occurs. We handle the error here, instead of forcing the caller to call `readn` or `writen` again, since the purpose of these three functions is to prevent the caller from having to handle a short count.

In Section 13.3 we mention that the `MSG_WAITALL` flag can be used with the `recv` function to replace the need for a separate `readn` function.

Note that our `readline` function calls the system's `read` function once for every byte of data. This is inefficient, as shown in Section 3.9 of APUE. What we would like to do is buffer the data by calling `read` to obtain as much data as we can and then examine the buffer 1 byte at a time. One way to do this is to use the standard I/O library, as we describe in Section 13.8.

Figure 3.17 shows a faster version of the `readline` function, which reads up to `MAXLINE` bytes at a time and then returns one character at a time.

lib/readline.c

```
1 #include    "unp.h"

2 static ssize_t
3 my_read(int fd, char *ptr)
4 {
5     static int read_cnt = 0;
6     static char *read_ptr;
7     static char read_buf[MAXLINE];

8     if (read_cnt <= 0) {
9       again:
10        if ( (read_cnt = read(fd, read_buf, sizeof(read_buf))) < 0) {
11            if (errno == EINTR)
12                goto again;
13            return (-1);
14        } else if (read_cnt == 0)
15            return (0);
16        read_ptr = read_buf;
17    }
18    read_cnt--;
19    *ptr = *read_ptr++;
20    return (1);
21 }

22 ssize_t
23 readline(int fd, void *vptr, size_t maxlen)
24 {
25    ssize_t n, rc;
26    char    c, *ptr;

27    ptr = vptr;
28    for (n = 1; n < maxlen; n++) {
29        if ( (rc = my_read(fd, &c)) == 1) {
30            *ptr++ = c;
31            if (c == '\n')
32                break;          /* newline is stored, like fgets() */
33        } else if (rc == 0) {
34            if (n == 1)
35                return (0);     /* EOF, no data read */
36            else
37                break;          /* EOF, some data was read */
38        } else
39            return (-1);        /* error, errno set by read() */
40    }

41    *ptr = 0;                   /* null terminate like fgets() */
42    return (n);
43 }
```

lib/readline.c

Figure 3.17 Better version of `readline` function.

2-21 The internal function my_read reads up to MAXLINE characters at a time and then returns them, one at a time.

29 The only change to the readline function itself is to call my_read instead of read.

Making this small change to our readline function makes a big difference. If we measure the time required by the old and new versions to read a 2781-line file (135,816 bytes) the *clock times* are 8.8 seconds for the old version and 0.3 seconds for the new version. Almost all of the difference is in the time spent within the kernel, the *system time*, as the old version performs 135,816 system calls while the new version performs only 34 system calls (135,816 divided by MAXLINE, which is 4096).

> Unfortunately by using static variables in my_read to maintain the state information across successive calls, the function is no longer *reentrant* or *thread-safe*. We discuss this in Sections 11.14 and 23.5. We develop a thread-safe version using thread-specific data in Figure 23.11.

3.10 isfdtype Function

There are times when we need to test a descriptor to see if it is of a specified type. Historically this has been done by calling the Posix.1 fstat function and then testing the returned st_mode value using one of the S_IS*xxx* macros. (This is discussed on pp. 73–76 of APUE.) Many implementations, but not all, define the S_ISSOCK macro that tests whether or not a descriptor is a socket. Since there are some implementations that cannot tell whether a descriptor is a socket based on just the information returned by the fstat function, Posix.1g provides the new isfdtype function.

```
#include <sys/stat.h>

int isfdtype(int fd, int fdtype);
```

Returns: 1 if descriptor of specified type, 0 if not, –1 on error

To test for a socket, *fdtype* is S_IFSOCK. One use for this function is in a program that is execed by another program (Section 4.7) to test whether an expected descriptor is really a socket.

Figure 3.18 shows a sample implementation of this function, assuming that the implementation supports the S_IFSOCK mode returned by fstat.

lib/isfdtype.c

```
 1 #include    "unp.h"

 2 #ifndef S_IFSOCK
 3 #error S_IFSOCK not defined
 4 #endif

 5 int
 6 isfdtype(int fd, int fdtype)
 7 {
 8     struct stat buf;

 9     if (fstat(fd, &buf) < 0)
10         return (-1);

11     if ((buf.st_mode & S_IFMT) == fdtype)
12         return (1);
13     else
14         return (0);
15 }
```

lib/isfdtype.c

Figure 3.18 Implementation of `isfdtype` using `fstat`.

There are numerous other `S_IFxxx` constants defined by including the `<sys/stat.h>` header and our implementation allows them. Posix.1g, however, only specifies that this function works when *fdtype* is `S_IFSOCK`.

3.11 Summary

Socket address structures are an integral part of every network program. We allocate them, fill them in, and pass pointers to them to the various socket functions. Sometimes we pass a pointer to one of these structures to the socket function and it fills in the contents. We always pass these structures by reference (that is we pass a pointer to the structure, not the structure itself) and always pass the size of the structure as another argument. When the socket function fills in the structure, the length is also passed by reference, so that its value can be updated by the function, and we call these value–result arguments.

Socket address structures are self-defining because they always begin with a field (the "family") that identifies the address family contained in the structure. Newer implementations that support variable-length socket address structures also contain a length field at the beginning, which contains the length of the entire structure.

The two functions that convert IP addresses between presentation format (what we write, such as ASCII characters) and numerical format (what goes into a socket address structure) are `inet_pton` and `inet_ntop`. Although we will use these two functions in the coming chapters, they are protocol dependent. A better technique is to manipulate socket address structures as opaque objects, knowing just the pointer to the structure and its size, and we developed a set of `sock_` functions that help make our programs protocol independent. We complete the development of our protocol-independent tools in Chapter 11 with the `getaddrinfo` and `getnameinfo` functions.

TCP sockets provide a byte stream to the application: there are no record markers. The return value from a `read` can be less than what we asked for, but this does not indicate an error. To help read and write a byte stream we developed three functions, `readn`, `writen`, and `readline`, which we use throughout the text.

Exercises

3.1 Why must value–result arguments such as the length of a socket address structure be passed by reference?

3.2 Why do the `readn` and `writen` functions both copy the `void*` pointer into a `char*` pointer?

3.3 The `inet_aton` and `inet_addr` functions have traditionally been liberal in what they accept as a dotted-decimal IPv4 address string: allowing from one to four numbers separated by decimal points, and allowing a leading `0x` to specify a hexadecimal number, or a leading 0 to specify an octal number. (Try `telnet 0xe` to see this behavior.) `inet_pton` is much stricter with IPv4 address and requires exactly four numbers separated by three decimal points, with each number being a decimal number between 0 and 255. `inet_pton` does not allow a dotted-decimal number to be specified when the address family is `AF_INET6`, although one could argue that these should be allowed and the return value is then the IPv4-mapped IPv6 address for the dotted-decimal string (Figure A.10). Write a new function named `inet_pton_loose` that handles these scenarios: if the address family is `AF_INET` and `inet_pton` returns 0, call `inet_aton` and see if it succeeds. Similarly, if the address family is `AF_INET6` and `inet_pton` returns 0, call `inet_aton` and if it succeeds, return the IPv4-mapped IPv6 address.

4

Elementary TCP Sockets

4.1 Introduction

This chapter describes the elementary socket functions required to write a complete TCP client and server. We first describe all of the elementary socket functions that we will be using and then develop the client and server in the next chapter. We will work with this client and server throughout the text, enhancing it many times (Figures 1.12 and 1.13).

We also describe concurrent servers, a common Unix technique for providing concurrency when numerous clients are connected to the same server at the same time. Each client connection causes the server to `fork` a new process just for that client. In this chapter we consider only the one-*process*-per-client model using `fork` but consider a different one-*thread*-per-client model when we describe threads in Chapter 23.

Figure 4.1 shows a time line of the typical scenario that takes place between a TCP client and server. First the server is started, then sometime later a client is started that connects to the server. We assume that the client sends a request to the server, the server processes the request, and the server sends back a reply to the client. This continues until the client closes its end of the connection, which sends an end-of-file notification to the server. The server then closes its end of the connection and either terminates or waits for a new client connection.

4.2 `socket` Function

To perform network I/O, the first thing a process must do is call the `socket` function, specifying the type of communication protocol desired (TCP using IPv4, UDP using IPv6, Unix domain stream protocol, etc.).

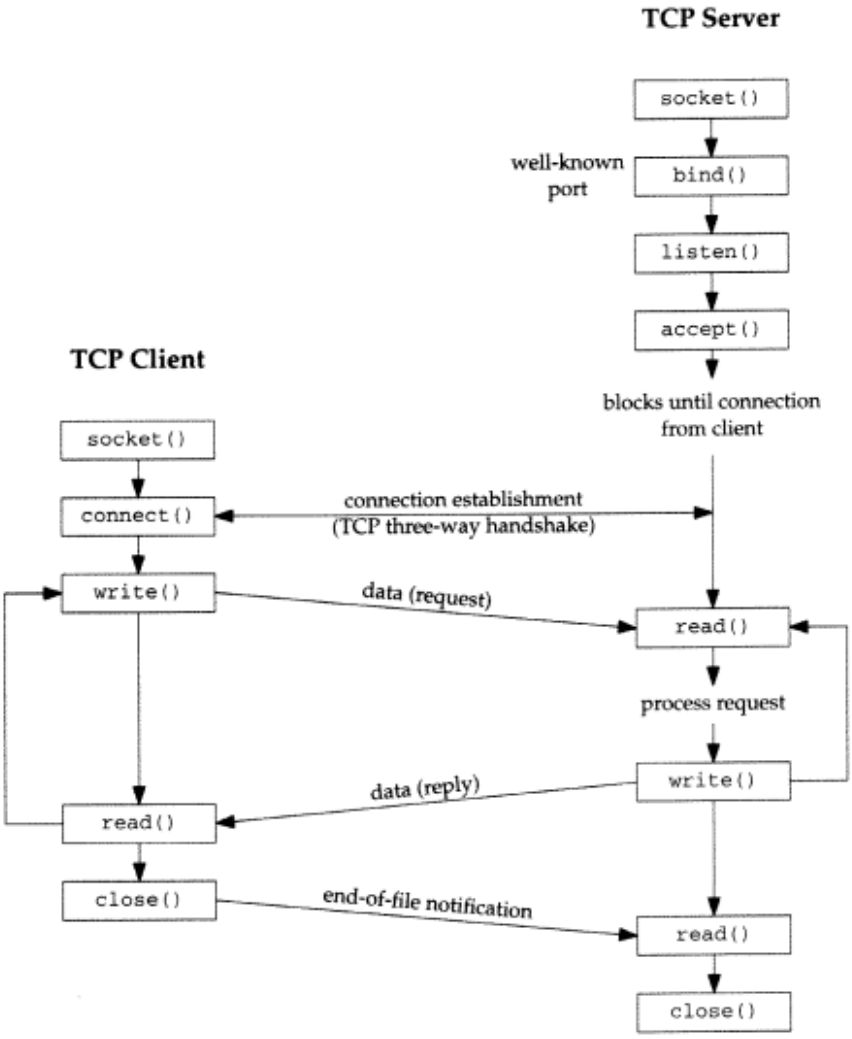

Figure 4.1 Socket functions for elementary TCP client–server.

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

int socket(int family, int type, int protocol);
```

Returns: nonnegative descriptor if OK, –1 on error

The *family* specifies the protocol family and is one of the constants shown in Figure 4.2. The socket *type* is one of the constants shown in Figure 4.3. Normally the *protocol* argument to the `socket` function is set to 0 except for raw sockets. We will discuss these in Chapter 25.

Not all combinations of socket *family* and *type* are valid. Figure 4.4 shows the valid combinations, along with the actual protocol that is selected by the pair. The boxes marked "Yes" are valid but do not have handy acronyms. The blank boxes are not supported.

family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols (Chapter 14)
AF_ROUTE	Routing sockets (Chapter 17)
AF_KEY	Key socket

Figure 4.2 Protocol *family* constants for `socket` function.

type	Description
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_RAW	raw socket

Figure 4.3 *type* of socket for `socket` function.

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP	TCP	Yes		
SOCK_DGRAM	UDP	UDP	Yes		
SOCK_RAW	IPv4	IPv6		Yes	Yes

Figure 4.4 Combinations of *family* and *type* for the `socket` function.

You may also encounter the corresponding `PF_xxx` constant as the first argument to `socket`. We say more about this at the end of this section.

We note that you may encounter `AF_UNIX` (the historical Unix name) instead of `AF_LOCAL` (the Posix.1g name), and we say more about this in Chapter 14.

There are other values for the *family* and *type* arguments. For example, 4.4BSD supports both `AF_NS` (the Xerox NS protocols, often called XNS) and `AF_ISO` (the OSI protocols). But few people use either of these protocols today. Similarly, the *type* of `SOCK_SEQPACKET`, a sequenced-packet socket, is implemented by both the Xerox NS protocols and the OSI protocols. But TCP is a byte stream and supports only `SOCK_STREAM` sockets.

Linux supports a new socket type, `SOCK_PACKET`, that provides access to the datalink, similar to BPF and DLPI in Figure 2.1. We say more about this in Chapter 26.

The key socket, AF_KEY, is new. IPv6 requires support for cryptographic security and many systems will probably support this for IPv4 too. Similar to the way that a routing socket (AF_ROUTE) is an interface to the kernel's routing table, the key socket is an interface into the kernel's key table. Preliminary documentation on this family is in [McDonald, Phan, and Atkinson 1996] and [McDonald, Metz, and Phan 1997].

On success the socket function returns a small nonnegative integer value, similar to a file descriptor. We call this a *socket descriptor*, or a *sockfd*. To obtain this socket descriptor, all we have specified is a protocol family (IPv4, IPv6, or Unix) and the socket type (stream, datagram, or raw). We have not yet specified either the local protocol address or the foreign protocol address.

AF_*xxx* versus PF_*xxx*

The AF_ prefix stands for "address family" and the PF_ prefix stands for "protocol family." Historically the intent was that a single protocol family might support multiple address families and that the PF_ value was used to create the socket and the AF_ value was used in socket address structures. But in actuality, a protocol family supporting multiple address families has never been supported and the <sys/socket.h> header defines the PF_ value for a given protocol to be equal to the AF_ value for that protocol. While there is no guarantee that this equality between the two will always be true, should anyone change this for existing protocols, lots of existing code would break. To conform to existing coding practice, we use only the AF_ constants in this text, although you may encounter the PF_ value, mainly in calls to socket.

Looking at 137 programs that call socket in the BSD/OS 2.1 release shows 143 calls that specify the AF_ value and only 8 that specify the PF_ value.

Historically, the reason for the similar sets of constants with the AF_ and PF_ prefixes goes back to 4.1cBSD [Lanciani 1996] and a version of the socket function that predates the one we are describing (which appeared with 4.2BSD). The 4.1cBSD version of socket took four arguments, one of which was a pointer to a sockproto structure. The first member of this structure was named sp_family and its value was one of the PF_ values. The second member, sp_protocol, was a protocol number, similar to the third argument to socket today. Specifying this structure was the only way to specify the protocol family. Therefore, in this early system the PF_ values were used as structure tags to specify the protocol family in the sockproto structure, and the AF_ values were used as structure tags to specify the address family in the socket address structures. The sockproto structure is still in 4.4BSD (pp. 626–627 of TCPv2) but is only used internally by the kernel. The original definition had the comment "protocol family" for the sp_family member, but this has been changed to "address family" in the 4.4BSD source code.

To confuse this difference between the AF_ and PF_ constants even more, the Berkeley kernel data structure that contains the value that is compared to the first argument to socket (the dom_family member of the domain structure, p. 187 of TCPv2) has the comment that it contains an AF_ value. But some of the domain structures within the kernel are initialized to the corresponding AF_ value (p. 192 of TCPv2) while others are initialized to the PF_ value (p. 646 of TCPv2 and p. 229 of TCPv3).

As another historical note, the 4.2BSD manual page for socket, dated July 1983, calls its first argument *af* and lists the possible values as the AF_ constants.

Finally, we note that Posix.1g specifies the first argument to `socket` be a `PF_` value, and the `AF_` value be used for socket address structure. But it then defines only one family value in the `addrinfo` structure (Section 11.2), intended for use in either a call to `socket` or in a socket address structure!

4.3 connect Function

The `connect` function is used by a TCP client to establish a connection with a TCP server.

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

int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

Returns: 0 if OK, –1 on error

sockfd is a socket descriptor that was returned by the `socket` function. The second and third arguments are a pointer to a socket address structure, and its size, as described in Section 3.3. The socket address structure must contain the IP address and port number of the server. We saw an example of this function in Figure 1.5.

The client does not have to call `bind` (which we describe in the next section) before calling `connect`: the kernel will choose both an ephemeral port and the source IP address if necessary.

In the case of a TCP socket, the `connect` function initiates TCP's three-way handshake (Section 2.5). The function returns only when the connection is established or an error occurs. There are several different error returns possible.

1. If the client TCP receives no response to its SYN segment, `ETIMEDOUT` is returned. 4.4BSD, for example, sends one SYN when `connect` is called, another 6 seconds later, and another 24 seconds later (p. 828 of TCPv2). If no response is received after a total of 75 seconds, the error is returned.

   Some systems provide administrative control over this timeout; see Appendix E of TCPv1.

2. If the server's response to the client's SYN is an RST, this indicates that no process is waiting for connections on the server host at the port that we specified (i.e., the server process is probably not running). This is a *hard error* and the error `ECONNREFUSED` is returned to the client as soon as the RST is received.

   An RST, meaning "reset," is a type of TCP segment that is sent by TCP when something is wrong. Three conditions that generate an RST are when a SYN arrives for a port that has no listening server (what we just described), when TCP wants to abort an existing connection, and when TCP receives a segment for a connection that does not exist. (TCPv1 pp. 246–250 contains additional information.)

3. If the client's SYN elicits an ICMP destination unreachable from some intermediate router, this is considered a *soft error*. The client kernel saves the message

but keeps sending SYNs with the same time between each SYN as in the first scenario. But if no response is received after some fixed amount of time (75 seconds for 4.4BSD), the saved ICMP error is returned to the process as either EHOSTUNREACH or ENETUNREACH.

> Many earlier systems, such as 4.2BSD, incorrectly aborted the connection establishment attempt when the ICMP destination unreachable was received. This is wrong because this ICMP error can indicate a transient condition. For example, it could be that the condition is caused by a routing problem that will be corrected in 15 seconds.
>
> Notice that ENETUNREACH is not listed in Figure A.15, even when the error indicates that the destination network is unreachable. Network unreachables are considered obsolete, and even if 4.4BSD receives one, EHOSTUNREACH is returned to the application.

We see these different error conditions with our simple client from Figure 1.5. We first specify the local host (127.0.0.1), which is running the daytime server and see the normal output.

```
solaris % daytimetcpcli 127.0.0.1
Tue Jan 16 16:45:07 1996
```

To see a different format for the returned reply, we specify the local Cisco router (Figure 1.16).

```
solaris % daytimetcpcli 206.62.226.62
Tuesday, May 7, 1996 11:01:33-MST
```

Next we specify an IP address that is on the local subnet (206.62.226) but the host ID (55) is nonexistent. That is, there does not exist a host on the subnet with a host ID of 55, so when the client host sends out ARP requests (asking for that host to respond with its hardware address), it will never receive an ARP reply.

```
solaris % daytimetcpcli 206.62.226.55
connect error: Connection timed out
```

We only get the error after the `connect` times out (which we said was 3 minutes with Solaris 2.5). Notice that our `err_sys` function prints the human-readable string associated with the ETIMEDOUT error.

Our next example is to the host `gateway`, which is a Cisco router, that is not running a daytime server.

```
solaris % daytimetcpcli 140.252.1.4
connect error: Connection refused
```

The server responds immediately with an RST.

Our final example specifies an IP address that is not reachable on the Internet. If we watch the packets with `tcpdump`, we see that a router six hops away returns an ICMP host unreachable error.

```
solaris % daytimetcpcli 192.3.4.5
connect error: No route to host
```

As with the ETIMEDOUT error, in this example the `connect` returns the EHOSTUNREACH error only after waiting its specified amount of time.

In terms of the TCP state transition diagram (Figure 2.4), `connect` moves from the CLOSED state (the state in which a socket begins when it is created by the `socket` function) to the SYN_SENT state and then, on success, to the ESTABLISHED state. If the `connect` fails, the socket is no longer usable and must be closed. We cannot call `connect` again on the socket. In Figure 11.6 we will see that when we call `connect` in a loop, trying each IP address for a given host until one works, each time `connect` fails we must `close` the socket descriptor and call `socket` again.

4.4 bind Function

The `bind` function assigns a local protocol address to a socket. With the Internet protocols the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number.

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

int bind(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
```

Returns: 0 if OK, –1 on error

> Historically the manual page description of `bind` has said "`bind` assigns a name to an unnamed socket." The use of the term "name" is confusing and gives the connotation of domain names (Chapter 9) such as `foo.bar.com`. The `bind` function has nothing to do with names. `bind` assigns a protocol address to a socket, and what that protocol address means depends on the protocol.

The second argument is a pointer to a protocol-specific address and the third argument is the size of this address structure. With TCP, calling `bind` lets us specify a port number, an IP address, both, or neither.

- Servers bind their well-known port when they start. We saw this in Figure 1.9. If a TCP client or server does not do this, the kernel chooses an ephemeral port for the socket when either `connect` or `listen` is called. It is normal for a TCP client to let the kernel choose an ephemeral port, unless the application requires a reserved port (Figure 2.6), but it is rare for a TCP server to let the kernel choose an ephemeral port, since servers are known by their well-known port.

  > Exceptions to this rule are RPC (Remote Procedure Call) servers. They normally let the kernel choose an ephemeral port for their listening socket since this port is then registered with the RPC port mapper. Clients have to contact the port mapper to obtain the ephemeral port before they can `connect` to the server. This also applies to RPC servers using UDP.

- A process can `bind` a specific IP address to its socket. The IP address must belong to an interface on the host. For a TCP client, this assigns the source IP address that will be used for IP datagrams sent on the socket. For a TCP server, this restricts the socket to receive incoming client connections destined only to that IP address.

Normally a TCP client does not `bind` an IP address to its socket. The kernel then chooses the source IP address when the socket is connected, based on the outgoing interface that is used, which in turn is based on the route required to reach the server (p. 737 of TCPv2).

If a TCP server does not bind an IP address to its socket, the kernel uses the destination IP address of the client's SYN as the server's source IP address (p. 943 of TCPv2).

As we said, calling `bind` lets us specify the IP address, the port, both, or neither. Figure 4.5 summarizes the values to which we set the `sin_addr` and `sin_port`, or the `sin6_addr` and `sin6_port`, depending on the desired result.

Process specifies		Result
IP address	port	
wildcard	0	kernel chooses IP address and port
wildcard	nonzero	kernel chooses IP address, process specifies port
local IP address	0	process specifies IP address, kernel chooses port
local IP address	nonzero	process specifies IP address and port

Figure 4.5 Result when specifying IP address and/or port number to `bind`.

If we specify a port number of 0, the kernel chooses an ephemeral port when `bind` is called. But if we specify a wildcard IP address, the kernel does not choose the local IP address until either the socket is connected (TCP) or until a datagram is sent on the socket (UDP).

With IPv4 the *wildcard* address is specified by the constant `INADDR_ANY`, whose value is normally 0. This tells the kernel to choose the IP address. We saw the use of this in Figure 1.9 with the assignment

```
struct sockaddr_in   servaddr;

servaddr.sin_addr.s_addr = htonl(INADDR_ANY);     /* wildcard */
```

While this works with IPv4, where an IP address is a 32-bit value that can be represented as a simple numeric constant (0 in this case), we cannot use this technique with IPv6, since the 128-bit IPv6 address is stored in a structure. (In C we cannot represent a constant structure on the right-hand side of an assignment.) To solve this problem, we write

```
struct sockaddr_in6    serv;

serv.sin6_addr = in6addr_any;     /* wildcard */
```

The system allocates and initializes the `in6addr_any` variable to the constant `IN6ADDR_ANY_INIT`. The `<netinet/in.h>` header contains the `extern` declaration for `in6addr_any`.

The value of `INADDR_ANY` (0) is the same in either network or host byte order, so the use of `htonl` is not really required. But since all the `INADDR_` constants defined by the `<netinet/in.h>` header are defined in host byte order, we should use `htonl` with any of these constants.

If we tell the kernel to choose an ephemeral port number for our socket, notice that `bind` does not return the chosen value. Indeed, it cannot return this value since the second argument to `bind` has the `const` qualifier. To obtain the value of the ephemeral port assigned by the kernel we must call `getsockname` to return the protocol address.

A common example of a process binding a nonwildcard IP address to a socket is on a host that provides Web servers to multiple organizations (Section 14.2 of TCPv3). First, each organization has its own domain name, such as *www.organization*.com. Next, each organization's domain name maps into a different IP address, but typically on the same subnet. For example, if the subnet is 198.69.10, the first organization's IP address could be 198.69.10.128, the next 198.69.10.129, and so on. All of these IP addresses are then *aliased* onto a single network interface (using the `alias` option of the `ifconfig` command on 4.4BSD, for example) so that the IP layer will accept incoming datagrams destined for any of the aliased addresses. Finally, one copy of the HTTP server is started for each organization and each copy `bind`s only the IP address for that organization.

> An alternative technique is to run a single server that binds the wildcard address. When a connection arrives, the server calls `getsockname` to obtain the destination IP address from the client, which in our discussion above could be 198.69.10.128, 198.69.10.129, and so on. The server then handles the client request based on the IP address to which the connection was issued.
>
> One advantage in binding a nonwildcard IP address is that the demultiplexing of a given destination IP address to a given server process is then done by the kernel.
>
> We must be careful to distinguish between the interface on which a packet arrives versus the destination IP address of that packet. In Section 8.8 we talk about the weak end system model and the strong end system model. Most implementations employ the former, meaning it is OK for a packet to arrive with a destination IP address that identifies an interface other than the interface on which the packet arrives. (This assumes a multihomed host.) Binding a nonwildcard IP address restricts the datagrams that will be delivered to the socket based only on the destination IP address. It says nothing about the arriving interface, unless the host employs the strong end system model.

A common error from `bind` is `EADDRINUSE` ("Address already in use"). We say more about this in Section 7.5 when we talk about the `SO_REUSEADDR` and `SO_REUSEPORT` socket options.

4.5 `listen` Function

The `listen` function is called only by a TCP server and it performs two actions.

1. When a socket is created by the `socket` function, it is assumed to be an active socket, that is, a client socket that will issue a `connect`. The `listen` function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket. In terms of the TCP state transition diagram (Figure 2.4) the call to `listen` moves the socket from the CLOSED state to the LISTEN state.

2. The second argument to this function specifies the maximum number of connections that the kernel should queue for this socket.

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

int listen(int sockfd, int backlog);
```

Returns: 0 if OK, −1 on error

This function is normally called after both the `socket` and `bind` functions and must be called before calling the `accept` function.

To understand the *backlog* argument we must realize that for a given listening socket, the kernel maintains two queues:

1. An *incomplete connection queue*, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake. These sockets are in the SYN_RCVD state (Figure 2.4).
2. A *completed connection queue*, which contains an entry for each client with whom the TCP three-way handshake has completed. These sockets are in the ESTABLISHED state (Figure 2.4).

Figure 4.6 depicts these two queues for a given listening socket.

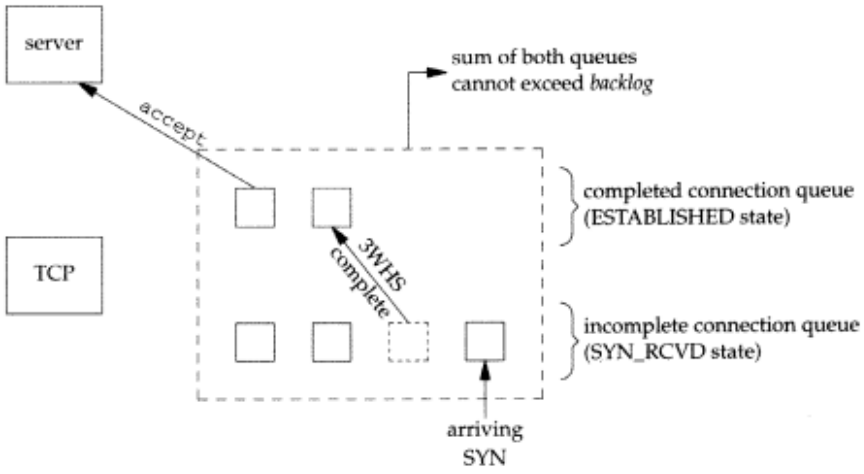

Figure 4.6 The two queues maintained by TCP for a listening socket.

Figure 4.7 depicts the packets exchanged during the connection establishment with these two queues.

When a SYN arrives from a client, TCP creates a new entry on the incomplete queue and then responds with the second segment of the three-way handshake: the server's SYN with an ACK of the client's SYN (Section 2.5). This entry will remain on the

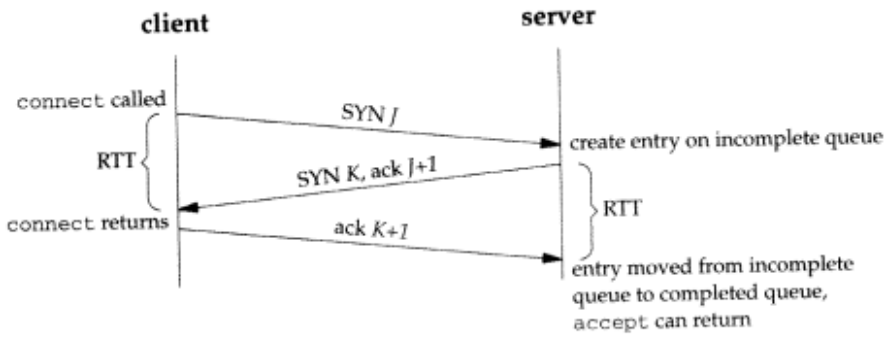

Figure 4.7 TCP three-way handshake and the two queues for a listening socket.

incomplete queue until the third segment of the three-way handshake arrives (the client's ACK of the server's SYN), or until the entry times out. (Berkeley-derived implementations have a timeout of 75 seconds for these incomplete entries.) If the three-way handshake completes normally, the entry moves from the incomplete queue to the end of the completed queue. When the process calls `accept`, which we describe in the next section, the first entry on the completed queue is returned to the process or, if the queue is empty, the process is put to sleep until an entry is placed onto the completed queue.

There are several points to consider about the handling of these two queues.

- The *backlog* argument to the `listen` function has historically specified the maximum value for the sum of both queues.

  > There has never been a formal definition of what the *backlog* means. The 4.2BSD manual page says that it "defines the maximum length the queue of pending connections may grow to." Many manual pages and even Posix.1g copy this definition verbatim, but this definition does not say whether a pending connection is one in the SYN_RCVD state, one in the ESTABLISHED state that has not yet been accepted, or either. The historical definition in this bullet is the Berkeley implementation, dating back to 4.2BSD, and copied by many others.

- Berkeley-derived implementations add a fudge factor to the *backlog* that we specify: it is multiplied by 1.5 (p. 257 of TCPv1 and p. 462 of TCPv2). For example, the commonly specified *backlog* of 5 really allows up to eight queued entries on these systems, as we show in Figure 4.10.

  > The reason for adding this fudge factor appears lost to history [Joy 1994]. But if we consider the *backlog* as specifying the maximum number of completed connections that the kernel will queue for a socket ([Borman 1997c], as discussed shortly), then the reason for the fudge factor is to take into account incomplete connections on the queue.

- Do not specify a *backlog* of 0, as different implementations interpret this differently (Figure 4.10). Some implementations allow one queued connection, while others do not allow any queued connections. If you do not want any clients connecting to your listening socket, then close the listening socket.

- Assuming the three-way handshake completes normally (i.e., no lost segments and no retransmissions), an entry remains on the incomplete connection queue for one round-trip time (RTT), whatever that value happens to be between this particular client and the server. Section 14.4 of TCPv3 shows that for one Web server the median RTT between many clients and the server was 187 ms. (The median is often used for this statistic, since a few large values can noticeably skew the mean.)
- Historically, sample code always shows a *backlog* of 5, as that was the maximum value supported by 4.2BSD. This was adequate in the 1980s when busy servers would handle only a few hundred connections per day. But with the growth of the World Wide Web (WWW), where busy servers handle millions of connections per day, this small number is completely inadequate (pp. 187–192 of TCPv3). Busy HTTP servers must specify a much larger *backlog*, and newer kernels must support larger values.

  Many current systems allow the administrator to modify the maximum value for the *backlog*.

- A problem is: what value should the application specify for the *backlog*, since 5 is often inadequate? There is no easy answer for this. HTTP servers now specify a larger value, but if the value specified is a constant in the source code to increase the constant requires recompiling the server. Another method is to assume some default but allow a command-line option or an environment variable to override the default. It is always OK to specify a value that is larger than supported by the kernel, as the kernel should silently truncate the value to the maximum value that it supports, without returning an error (p. 456 of TCPv2).

  We can provide a simple solution to this problem by modifying our wrapper function for the `listen` function. Figure 4.8 shows the actual code. We allow the environment variable `LISTENQ` to override the value specified by the caller.

lib/wrapsock.c

```
74 void
75 Listen(int fd, int backlog)
76 {
77     char    *ptr;

78         /* can override 2nd argument with environment variable */
79     if ( (ptr = getenv("LISTENQ")) != NULL)
80         backlog = atoi(ptr);

81     if (listen(fd, backlog) < 0)
82         err_sys("listen error");
83 }
```

lib/wrapsock.c

Figure 4.8 Wrapper function for `listen` that allows an environment variable to specify *backlog*.

- Manuals and books have historically said that the reason for queueing a fixed number of connections is to handle the case of the server process being busy between successive calls to `accept`. This implies that of the two queues, the completed queue should normally have more entries than the incomplete queue. Again, busy

Web servers have shown that this is false. The reason for specifying a large *backlog* is because the incomplete connection queue can grow as client SYNs arrive, waiting for completion of the three-way handshake.

Figure 4.9 shows the actual number of entries on each queue measured on a moderately busy Web server. These values were obtained by sampling these two counters for a listening HTTP socket approximately every 84 ms for 2 hours during the middle of a weekday.

#entries on queue	Incomplete queue	Complete queue
0	3,033	90,358
1	7,158	107
2	10,551	59
3	12,960	52
4	11,949	38
5	9,836	27
6	7,754	31
7	6,165	22
8	4,829	30
9	3,687	35
10	2,674	30
11	1,893	25
12	1,431	29
13	1,083	25
14	1,065	49
15	980	7
16	784	
17	696	
18	514	
19	382	
20	294	
21	248	
22	161	
23	152	
24	121	
25	77	
26	48	
27	33	
28	79	
29	78	
30	90	
31	70	
32	29	
33	16	
34	4	
	90,924	90,924

Figure 4.9 Number of entries on incomplete and completed connection queues.

The completed connection queue was empty 99.4% of the time, but there were periods when this queue was not empty. The system on which this server was running

(BSD/OS 2.0.1) had a maximum backlog of 64, although the values shown do not appear to have reached this limit.

- If the queues are full when a client SYN arrives, TCP ignores the arriving SYN (pp. 930–931 of TCPv2), it does not send an RST. This is because the condition is considered temporary, and the client TCP will retransmit its SYN, hopefully finding room on the queue in the near future. If the server TCP were to send an RST, the client's `connect` would immediately return an error, forcing the application to handle this condition, instead of letting TCP's normal retransmission take over. Also, the client could not differentiate between an RST in response to a SYN meaning "there is no server at this port" versus "there is a server at this port but its queues are full."

  Posix.1g allows either behavior: ignoring the new SYN or responding to the new SYN with an RST. Historically, all Berkeley-derived implementations have ignored the new SYN.

- Data that arrives after the three-way handshake completes, but before the server calls `accept`, should be queued by the server TCP, up to the size of the connected socket's receive buffer.

Figure 4.10 shows the actual number of queued connections provided for different values of the *backlog* argument for the various operating systems in Figure 1.16. For nine different operating systems there are six distinct columns, showing the variety of interpretations about what the backlog means!

	Maximum actual number of queued connections					
backlog	AIX 4.2, BSD/OS 3.0	DUnix 4.0, Linux 2.0.27, UWare 2.1.2	HP-UX 10.30	SunOS 4.1.4	Solaris 2.5.1	Solaris 2.6
0	1	0	1	1	1	1
1	2	1	1	2	2	3
2	4	2	3	4	3	4
3	5	3	4	5	4	6
4	7	4	6	7	5	7
5	8	5	7	8	6	9
6	10	6	9	8	7	10
7	11	7	10	8	8	12
8	13	8	12	8	9	13
9	14	9	13	8	10	15
10	16	10	15	8	11	16
11	17	11	16	8	12	18
12	19	12	18	8	13	19
13	20	13	18	8	14	21
14	22	14	19	8	15	22

Figure 4.10 Actual number of queued connections for values of *backlog*.

AIX, BSD/OS, and SunOS 4 have the traditional Berkeley algorithm, although the latter does not allow the *backlog* to go above 5. HP-UX and Solaris 2.6 add a different fudge

factor to the *backlog*. Digital Unix, Linux, and UnixWare interpret the *backlog* literally, and Solaris 2.5.1 just adds one to the *backlog*.

Linux allowed an unlimited number of connections for a *backlog* of 0, indicating a bug.

The program to measure these values is shown in the solution for Exercise 14.5.

As we said, historically the backlog has specified the maximum value for the sum of both queues. During 1996 a new type of attack was launched on the Internet, called *SYN flooding* [CERT 1996b]. The hacker writes a program to send SYNs at a high rate to the victim, filling the incomplete connection queue for one or more TCP ports. (We use the term *hacker* to mean the attacker, as described in the Preface of [Cheswick and Bellovin 1994].) Additionally the source IP address of each SYN is set to a random number (this is called *IP spoofing*) so that the server's SYN/ACK goes nowhere. This also prevents the server from knowing the real IP address of the hacker. By filling the incomplete queue with bogus SYNs, legitimate SYNs are not queued, providing a *denial of service* to legitimate clients. There are two commonly used methods of handling these attacks, summarized in [Borman 1997c]. But what is most interesting in this note is revisiting what the `listen` backlog really means. It should specify the maximum number of *completed* connections for a given socket that the kernel will queue. The purpose of having a limit on these completed connections is to stop the kernel from accepting new connection requests for a given socket when the application is not accepting them (for whatever reason). If a system implements this interpretation, as does BSD/OS 3.0, then the application need not specify huge *backlog* values just because the server handles lots of client requests (e.g., a busy Web server) or to provide protection against SYN flooding. The kernel handles lots of incomplete connections, regardless of whether they are legitimate or from a hacker. But even with this interpretation, we can see in Figure 4.9 that scenarios do occur when the completed connection queue accumulates entries (up to 15 in this figure), where the traditional value of 5 is inadequate.

4.6 accept Function

`accept` is called by a TCP server to return the next completed connection from the front of the completed connection queue (Figure 4.6). If the completed connection queue is empty, the process is put to sleep (assuming the default of a blocking socket).

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

int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

Returns: nonnegative descriptor if OK, –1 on error

The *cliaddr* and *addrlen* arguments are used to return the protocol address of the connected peer process (the client). *addrlen* is a value–result argument (Section 3.3): before the call, we set the integer value referenced by **addrlen* to the size of the socket address structure pointed to by *cliaddr*, and on return this integer value contains the actual number of bytes stored by the kernel in the socket address structure.

If `accept` is successful, its return value is a brand new descriptor that was automatically created by the kernel. This new descriptor refers to the TCP connection with the client. When discussing `accept` we call the first argument to `accept` the *listening socket* (the descriptor created by `socket` and then used as the first argument to both

`bind` and `listen`), and we call the return value from `accept` the *connected socket*. It is important to differentiate between these two sockets. A given server normally creates only one listening socket, which then exists for the lifetime of the server. The kernel then creates one connected socket for each client connection that is `accept`ed (i.e., for which the TCP three-way handshake completes). When the server is finished serving a given client, the connected socket is closed.

This function returns up to three values: an integer return code that is either a new socket descriptor or an error indication, the protocol address of the client process (through the *cliaddr* pointer), and the size of this address (through the *addrlen* pointer). If we are not interested in having the protocol address of the client returned, we set both *cliaddr* and *addrlen* to null pointers.

Figure 1.9 shows these points. The connected socket is closed each time through the loop, but the listening socket remains open for the life of the server. We also see that the second and third arguments to `accept` are null pointers, since we were not interested in the identity of the client.

Example: Value–Result Arguments

We will now show how to handle the value–result argument to `accept` by modifying the code from Figure 1.9 to print the IP address and port of the client. We show this in Figure 4.11.

New declarations

7-8 We define two new variables: `len`, which will be a value–result variable, and `cliaddr`, which will contain the client's protocol address.

Accept connection and print client's address

19-23 We initialize `len` to the size of the socket address structure and pass a pointer to the `cliaddr` structure and a pointer to `len` as the second and third arguments to `accept`. We call `inet_ntop` (Section 3.7) to convert the 32-bit IP address in the socket address structure into a dotted-decimal ASCII string and call `ntohs` (Section 3.4) to convert the 16-bit port number from network byte order to host byte order.

> Calling `sock_ntop` instead of `inet_ntop` would make our server more protocol independent, but this server is already dependent on IPv4. We show a protocol-independent version of this server in Figure 11.9.

If we run our new server and then run our client on the same host, connecting to our server twice in a row, we have the following output from the client:

```
solaris % daytimetcpcli 127.0.0.1
Wed Jan 17 15:42:35 1996
solaris % daytimetcpcli 206.62.226.33
Wed Jan 17 15:42:53 1996
```

We first specify the server's IP address as the loopback address (127.0.0.1) and then as its own IP address (206.62.226.33). Here is the corresponding server output.

intro/daytimetcpsrv1.c

```
 1 #include    "unp.h"
 2 #include    <time.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     listenfd, connfd;
 7     socklen_t len;
 8     struct sockaddr_in servaddr, cliaddr;
 9     char    buff[MAXLINE];
10     time_t  ticks;

11     listenfd = Socket(AF_INET, SOCK_STREAM, 0);

12     bzero(&servaddr, sizeof(servaddr));
13     servaddr.sin_family = AF_INET;
14     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
15     servaddr.sin_port = htons(13);  /* daytime server */

16     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

17     Listen(listenfd, LISTENQ);

18     for ( ; ; ) {
19         len = sizeof(cliaddr);
20         connfd = Accept(listenfd, (SA *) &cliaddr, &len);
21         printf("connection from %s, port %d\n",
22                Inet_ntop(AF_INET, &cliaddr.sin_addr, buff, sizeof(buff)),
23                ntohs(cliaddr.sin_port));

24         ticks = time(NULL);
25         snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
26         Write(connfd, buff, strlen(buff));

27         Close(connfd);
28     }
29 }
```

intro/daytimetcpsrv1.c

Figure 4.11 Daytime server that prints client IP address and port.

```
solaris # daytimetcpsrv1
connection from 127.0.0.1, port 33188
connection from 206.62.226.33, port 33189
```

Notice what happens with the client's IP address. Since our daytime client (Figure 1.5) does not call bind, we said in Section 4.4 that the kernel chooses the source IP address based on the outgoing interface that is used. In the first case the kernel sets the source IP address to the loopback address and in the second case it sets the address to the IP address of the Ethernet interface. We can also see in this example that the ephemeral port chosen by the Solaris kernel is 33188, and then 33189 (recall Figure 2.6).

As a final point, our shell prompt for the server script changes to the pound sign (#), the commonly used prompt for the superuser. Our server must run with superuser

privileges to bind the reserved port of 13. If we do not have superuser privileges, the call to bind fails:

```
solaris % daytimetcpsrv1
bind error: Permission denied
```

4.7 fork and exec Functions

Before describing how to write a concurrent server in the next section we must describe the Unix fork function. This function is the only way in Unix to create a new process.

```
#include <unistd.h>

pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, –1 on error

If you have never seen this function before, the hard part in understanding fork is that it is called *once* but it returns *twice*. It returns once in the calling process (called the parent) with a return value that is the process ID of the newly created process (the child). It also returns once in the child, with a return value of 0. Hence the return value tells the process whether it is the parent or the child.

The reason fork returns 0 in the child, instead of the parent's process ID, is because a child has only one parent and it can always obtain the parent's process ID by calling getppid. A parent, on the other hand, can have any number of children, and there is no way to obtain the process IDs of its children. If the parent wants to keep track of the process IDs of all its children, it must record the return values from fork.

All descriptors open in the parent before the call to fork are shared with the child after fork returns. We will see this feature used by network servers: the parent calls accept and then calls fork. The connected socket is then shared between the parent and child. Normally the child then reads and writes the connected socket and the parent closes the connected socket.

There are two typical uses of fork.

1. A process makes a copy of itself so that one copy can handle one operation while the other copy does another task. This is typical for network servers. We will see many examples of this later in the text.
2. A process wants to execute another program. Since the only way to create a new process is by calling fork, the process first calls fork to make a copy of itself, and then one of the copies (typically the child process) calls exec (described next) to replace itself with the new program. This is typical for programs such as shells.

The only way in which an executable program file on disk is executed by Unix is for an existing process to call one of the six exec functions. (We often refer generically to

"the exec function" when it does not matter which of the six is called.) exec replaces the current process image with the new program file and this new program normally starts at the main function. The process ID does not change. We refer to the process that calls exec as the *calling process* and the newly executed program as the *new program*.

> Older manuals and books incorrectly refer to the new program as the *new process*, which is wrong, because a new process is not created.

The difference in the six exec functions is (a) whether the program file to execute is specified by a *filename* or a *pathname*, (b) whether the arguments to the new program are listed one by one or referenced through an array of pointers, and (c) whether the environment of the calling process is passed to the new program or whether a new environment is specified.

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *) 0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *) 0, char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0, ... /* (char *) 0 */ );

int execvp(const char *filename, char *const argv[]);
```

All six return: −1 on error, no return on success

These functions return to the caller only if an error occurs. Otherwise control passes to the start of the new program, normally the main function.

The relationship among these six functions is shown in Figure 4.12. Normally only execve is a system call within the kernel and the other five are library functions that call execve.

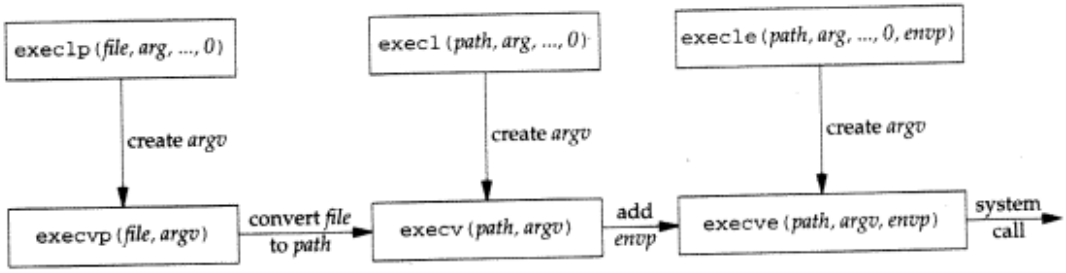

Figure 4.12 Relationship among the six exec functions.

Note the following differences among these six functions:

1. The three functions in the top row specify each argument string as a separate argument to the `exec` function, with a null pointer terminating the variable number of arguments. The three functions in the second row have an *argv* array, containing the pointers to the argument strings. This *argv* array must contain a null pointer to specify its end, since a count is not specified.
2. The two functions in the left column specify a *filename* argument. This is converted into a *pathname* using the current `PATH` environment variable. If the *filename* argument to `execlp` or `execvp` contains a slash (`/`) anywhere in the string, the `PATH` variable is not used. The four functions in the right two columns specify a fully qualified *pathname* argument.
3. The four functions in the left two columns do not specify an explicit environment pointer. Instead the current value of the external variable `environ` is used for building an environment list that is passed to the new program. The two functions in the right column specify an explicit environment list. The *envp* array of pointers must be terminated by a null pointer.

Descriptors open in the process before calling `exec` normally remain open across the `exec`. We use the qualifier "normally" because this can be disabled using `fcntl` to set the `FD_CLOEXEC` descriptor flag. The `inetd` server uses this feature, as we describe in Section 12.5.

4.8 Concurrent Servers

The server in Figure 4.11 is an *iterative server*. For something as simple as a daytime server, this is fine. But when the client request can take longer to service, we do not want to tie up a single server with one client; we want to handle multiple clients at the same time. The simplest way to write a *concurrent server* under Unix is to `fork` a child process to handle each client. Figure 4.13 shows the outline for a typical concurrent server.

When a connection is established, `accept` returns, the server calls `fork`, and then the child process services the client (on `connfd`, the connected socket) and the parent process waits for another connection (on `listenfd`, the listening socket). The parent closes the connected socket since the child handles this new client.

In Figure 4.13 we assume that the function `doit` does whatever is required to service the client. When this function returns, we explicitly `close` the connected socket in the child. This is not required since the next statement calls `exit`, and part of process termination is closing all open descriptors by the kernel. Whether to include this explicit call to `close` or not is a matter of personal programming taste.

We said in Section 2.5 that calling `close` on a TCP socket causes a FIN to be sent, followed by the normal TCP connection termination sequence. Why doesn't the `close` of `connfd` in Figure 4.13 by the parent terminate its connection with the client? To understand what's happening we must understand that every file or socket has a reference count. The reference count is maintained in the file table entry (pp. 57–60 of APUE). This is a count of the number of descriptors that are currently open that refer to

```
pid_t  pid;
int    listenfd, connfd;

listenfd = Socket( ... );

    /* fill in sockaddr_in{} with server's well-known port */
Bind(listenfd, ... );
Listen(listenfd, LISTENQ);

for ( ; ; ) {
    connfd = Accept(listenfd, ... );      /* probably blocks */

    if ( (pid = Fork()) == 0) {
        Close(listenfd);    /* child closes listening socket */
        doit(connfd);       /* process the request */
        Close(connfd);      /* done with this client */
        exit(0);            /* child terminates */
    }

    Close(connfd);          /* parent closes connected socket */
}
```

Figure 4.13 Outline for typical concurrent server.

this file or socket. In Figure 4.13, after `socket` returns, the file table entry associated with `listenfd` has a reference count of 1. After `accept` returns, the file table entry associated with `connfd` has a reference count of 1. But after `fork` returns, both descriptors are shared (i.e., duplicated) between the parent and child, so the file table entries associated with both sockets now have a reference count of 2. Therefore, when the parent closes `connfd`, it just decrements the reference count from 2 to 1 and that is all. A real close on the descriptor does not take place until the reference count reaches 0. This will occur at some time later when the child closes `connfd`.

We can also visualize the sockets and the connection that occurs in Figure 4.13 as follows. First, Figure 4.14 shows the status of the client and server while the server is blocked in the call to `accept` and the connection request arrives from the client.

Figure 4.14 Status of client–server before call to `accept` returns.

Immediately after `accept` returns we have the scenario shown in Figure 4.15. The connection is accepted by the kernel and a new socket, `connfd`, is created. This is a connected socket and data can now be read and written across the connection.

The next step in the concurrent server is to call `fork`. Figure 4.16 shows the status after `fork` returns.

Figure 4.15 Status of client–server after return from `accept`.

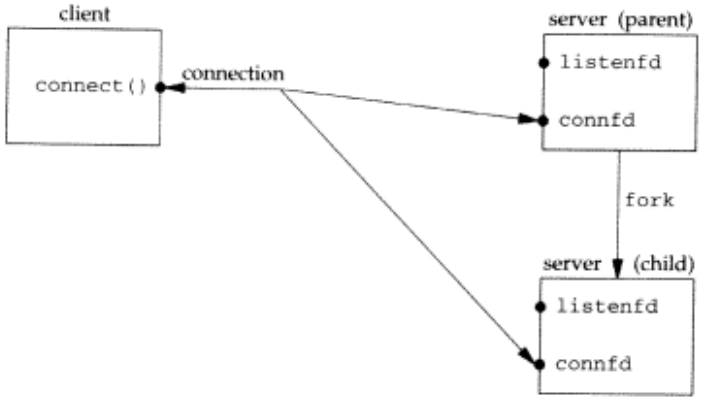

Figure 4.16 Status of client–server after `fork` returns.

Notice that both descriptors, `listenfd` and `connfd`, are shared (duplicated) between the parent and child.

The next step is for the parent to close the connected socket and the child to close the listening socket. This is shown in Figure 4.17.

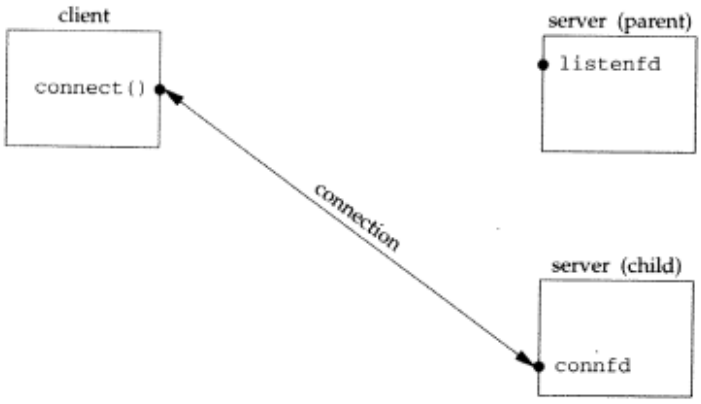

Figure 4.17 Status of client–server after parent and child close appropriate sockets.

This is the desired final state of the sockets. The child is handling the connection with the client and the parent can call `accept` again on the listening socket, to handle the next client connection.

4.9 close Function

The normal Unix `close` function is also used to close a socket and terminate a TCP connection.

```
#include <unistd.h>

int close(int sockfd);
```

Returns: 0 if OK, –1 on error

The default action of `close` with a TCP socket is to mark the socket as closed and return to the process immediately. The socket descriptor is no longer usable by the process: it cannot be used as an argument to `read` or `write`. But TCP will try to send any data that is already queued to be sent to the other end, and after this occurs the normal TCP connection termination sequence takes place (Section 2.5).

In Section 7.5 we describe the `SO_LINGER` socket option which lets us change this default action with a TCP socket. In that section we also describe what a TCP application must do to be guaranteed that the peer application has received any outstanding data.

Descriptor Reference Counts

At the end of Section 4.8 we mentioned that when the parent process in our concurrent server `closes` the connected socket, this just decrements the reference count for the descriptor. Since the reference count was still greater than 0, this call to `close` did not initiate TCP's four-packet connection termination sequence. This is the behavior we want with our concurrent server with the connected socket that is shared between the parent and child.

If we really want to send a FIN on a TCP connection, the `shutdown` function can be used (Section 6.6) instead of `close`. We describe the motivation for this in Section 6.5.

We must also be aware of what happens in our concurrent server if the parent does not call `close` for each connected socket returned by `accept`. First, the parent will eventually run out of descriptors, as there is usually a limit to the number of open descriptors that any process can have open at any time. But more importantly, none of the client connections will be terminated. When the child closes the connected socket, its reference count will go from 2 to 1 and it will remain at 1 since the parent never `closes` the connected socket. This will prevent TCP's connection termination sequence from occurring, and the connection remains open.

4.10 getsockname and getpeername Functions

These two functions return either the local protocol address associated with a socket (`getsockname`) or the foreign protocol address associated with a socket (`getpeername`).

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

int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);

int getpeername(int sockfd, struct sockaddr *peeraddr, socklen_t *addrlen);
```

Both return: 0 if OK, −1 on error

Notice that the final argument for both functions is a value–result argument. That is, both functions fill in the socket address structure pointed to by *localaddr* or *peeraddr*.

> We mentioned with our discussion of `bind` that the term "name" is misleading. These two functions return the protocol address associated with one of the two ends of a network connection, which for IPv4 and IPv6 is the combination of an IP address and port number. These functions have nothing to do with domain names (Chapter 9).

These two functions are required for the following reasons:

- After `connect` successfully returns in a TCP client that does not call `bind`, `getsockname` returns the local IP address and local port number assigned to the connection by the kernel.
- After calling `bind` with a port number of 0 (telling the kernel to choose the local port number), `getsockname` returns the local port number that was assigned.
- `getsockname` can be called to obtain the address family of a socket, as we show in Figure 4.19.
- In a TCP server that `bind`s the wildcard IP address (Figure 1.9), once a connection is established with a client (`accept` returns successfully), the server can call `getsockname` to obtain the local IP address assigned to the connection. The socket descriptor argument in this call must be that of the connected socket, and not the listening socket.
- When a server is `exec`ed by the process that calls `accept`, the only way the server can obtain the identity of the client is to call `getpeername`. This is what happens whenever `inetd` (Section 12.5) `fork`s and `exec`s a TCP server. Figure 4.18 shows this scenario. `inetd` calls `accept` (top left box) and two values are returned: the connected socket descriptor, `connfd`, is the return value of the function, and the small box we label "peer's address" (an Internet socket address structure) contains the IP address and port number of the client. `fork` is called and a child of `inetd` is created. Since the child starts with a copy of the parent's memory image, the socket address structure is available to the child, as is the connected socket descriptor (since the descriptors are shared between the parent and child). But when the child `exec`s the real server (say the Telnet server that we show), the memory image of the child is replaced with the new program file for the Telnet server (i.e., the socket address structure containing the peer's address is lost), but the connected socket descriptor remains open across the `exec`. One of the first function calls performed by the Telnet server is `getpeername` to obtain the IP address and port number of the client.

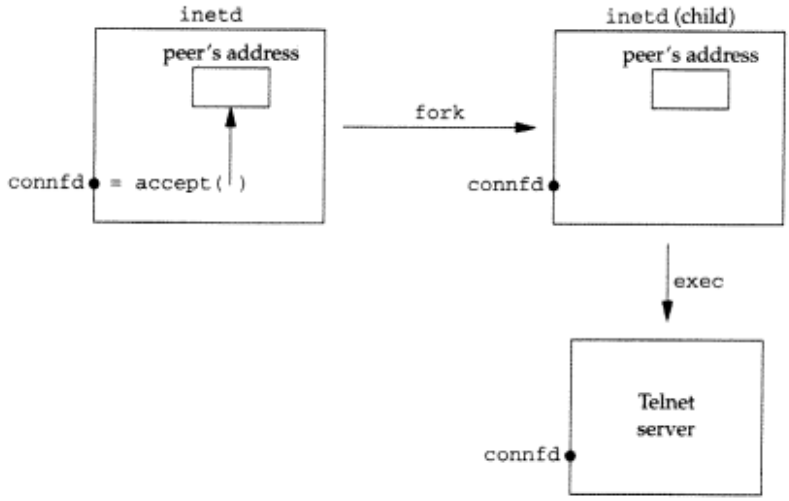

Figure 4.18 Example of `inetd` spawning a server.

Obviously the Telnet server in this final example must know the value of `connfd` when it starts. There are two common ways to do this. First, the process calling `exec` can format the descriptor number as a character string and pass it as a command-line argument to the newly execed program. Alternately a convention can be established that a certain descriptor is always set to the connected socket before calling `exec`. The latter is what `inetd` does, always setting descriptors 0, 1, and 2 to be the connected socket.

Example: Obtaining the Address Family of a Socket

The `sockfd_to_family` function shown in Figure 4.19 returns the address family of a socket.

lib/sockfd_to_family.c

```
1 #include    "unp.h"
2 int
3 sockfd_to_family(int sockfd)
4 {
5     union {
6         struct sockaddr sa;
7         char    data[MAXSOCKADDR];
8     } un;
9     socklen_t len;
10    len = MAXSOCKADDR;
11    if (getsockname(sockfd, (SA *) un.data, &len) < 0)
12        return (-1);
13    return (un.sa.sa_family);
14 }
```

lib/sockfd_to_family.c

Figure 4.19 Return the address family of a socket.

Allocate room for largest socket address structure

5-8 Since we do not know what type of socket address structure to allocate, we use the constant `MAXSOCKADDR` in our `unp.h` header, which is the size in bytes of the largest socket address structure. We define a `char` array of this size within a `union` that includes a generic socket address structure.

Call `getsockname`

10-13 We call `getsockname` and return the address family.

Since Posix.1g allows a call to `getsockname` on an unbound socket, this function should work for any open socket descriptor.

4.11 Summary

All clients and servers begin with a call to `socket`, returning a socket descriptor. Clients then call `connect`, while servers call `bind`, `listen`, and `accept`. Sockets are normally closed with the standard `close` function, although we will see another way to do this with the `shutdown` function (Section 6.6) and we will also examine the effect of the `SO_LINGER` socket option (Section 7.5).

Most TCP servers are concurrent with the server calling `fork` for every client connection that it handles. We will see that most UDP servers are iterative. While these two models have been used successfully for many years, in Chapter 27 we will look at other server design options, using threads and processes.

Exercises

4.1 In Section 4.4 we stated that the `INADDR_` constants defined by the `<netinet/in.h>` header are in host byte order. How can we tell this?

4.2 Modify Figure 1.5 to call `getsockname` after `connect` returns success. Print the local IP address and local port assigned to the TCP socket using `sock_ntop`. In what range (Figure 2.6) are your system's ephemeral ports?

4.3 In a concurrent server assume the child runs first after the call to `fork`. The child then completes the service of the client before the call to `fork` returns to the parent. What happens in the two calls to `close` in Figure 4.13?

4.4 In Figure 4.11 first change the server's port from 13 to 9999 (so that we do not need superuser privileges to start the program). Remove the call to `listen`. What happens?

4.5 Continue the previous exercise. Remove the call to `bind`, but allow the call to `listen`. What happens?

5

TCP Client–Server Example

5.1 Introduction

We now use the elementary functions from the previous chapter to write a complete TCP client–server example. Our simple example is an echo server that performs the following steps:

1. The client reads a line of text from its standard input and writes the line to the server.
2. The server reads the line from its network input and echoes the line back to the client.
3. The client reads the echoed line and prints it on its standard output.

Figure 5.1 depicts this simple client–server along with the functions used for input and output.

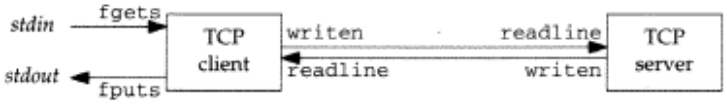

Figure 5.1 Simple echo client and server.

We show two arrows between the client and server but this is one full-duplex TCP connection. The `fgets` and `fputs` functions are from the standard I/O library and the `writen` and `readline` functions are shown in Section 3.9.

While we develop our own implementation of an echo server, most TCP/IP implementations provide such a server, using both TCP and UDP (Section 2.10). We will also use this server with our own client.

A client–server that echoes input lines is a valid, yet simple, example of a network application. All the basic steps required to implement any client–server are illustrated by this example. To expand this example into your own application all you need to do is change what the server does with the input it receives from its clients.

Besides running our client and server in its normal mode (type in a line and watch it echo) we examine lots of boundary conditions for this example: what happens when the client and server are started; what happens when the client terminates normally; what happens to the client if the server process terminates before the client is done; what happens to the client if the server host crashes; and so on. By looking at all these scenarios and understanding what happens at the network level, and how this appears to the sockets API, we will understand more about what goes on at these levels and how to code our applications to handle these scenarios.

In all these examples, we have "hard-coded" protocol-specific constants such as addresses and ports. There are two reasons for this. First, we need to understand exactly what needs to be stored in the protocol-specific address structures. Second, we have not yet covered the library functions that make this more portable. These functions are covered in Chapters 9 and 11.

We note now that we will make many changes to both the client and server in successive chapters as we learn more about network programming (Figures 1.12 and 1.13).

5.2 TCP Echo Server: `main` Function

Our TCP client and server follow the flow of functions that we diagramed in Figure 4.1. We show the concurrent server program in Figure 5.2.

Create socket, bind server's well-known port

9–15 A TCP socket is created. An Internet socket address structure is filled in with the wildcard address (`INADDR_ANY`) and the server's well-known port (`SERV_PORT`, which is defined as 9877 in our `unp.h` header). Binding the wildcard address tells the system that we will accept a connection destined for any local interface, in case the system is multihomed. Our choice of the TCP port number is based on Figure 2.6. It should be greater than 1023 (we do not need a reserved port), greater than 5000 (to avoid conflict with the ephemeral ports allocated by many Berkeley-derived implementations), less than 49152 (to avoid conflict with the "correct" range of ephemeral ports), and should not conflict with any registered port. The socket is converted into a listening socket by `listen`.

Wait for client connection to complete

17–18 The server blocks in the call to `accept`, waiting for a client connection to complete.

Concurrent server

19–24 For each client, `fork` spawns a child, and the child handles the new client. As we discussed in Section 4.8, the child closes the listening socket and the parent closes the connected socket. The child then calls `str_echo` (Figure 5.3) to handle the client.

tcpcliserv/tcpserv01.c

```
 1 #include    "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     listenfd, connfd;
 6     pid_t   childpid;
 7     socklen_t clilen;
 8     struct sockaddr_in cliaddr, servaddr;
 9     listenfd = Socket(AF_INET, SOCK_STREAM, 0);
10     bzero(&servaddr, sizeof(servaddr));
11     servaddr.sin_family = AF_INET;
12     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
13     servaddr.sin_port = htons(SERV_PORT);
14     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
15     Listen(listenfd, LISTENQ);
16     for ( ; ; ) {
17         clilen = sizeof(cliaddr);
18         connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
19         if ( (childpid = Fork()) == 0) {     /* child process */
20             Close(listenfd);    /* close listening socket */
21             str_echo(connfd);   /* process the request */
22             exit(0);
23         }
24         Close(connfd);          /* parent closes connected socket */
25     }
26 }
```

tcpcliserv/tcpserv01.c

Figure 5.2 TCP echo server (improved in Figure 5.12).

5.3 TCP Echo Server: `str_echo` Function

The function `str_echo`, shown in Figure 5.3, performs the server processing for each client: reading the lines from the client and echoing them back to the client.

Read a line and echo the line

7-11 `readline` reads the next line from the socket and the line is echoed back to the client by `writen`. If the client closes the connection (the normal scenario), the receipt of the client's FIN causes the child's `readline` to return 0. This causes the `str_echo` function to return, which terminates the child in Figure 5.2.

5.4 TCP Echo Client: `main` Function

Figure 5.4 shows the TCP client `main` function.

lib/str_echo.c

```
1 #include    "unp.h"
2 void
3 str_echo(int sockfd)
4 {
5     ssize_t n;
6     char    line[MAXLINE];
7     for ( ; ; ) {
8         if ( (n = Readline(sockfd, line, MAXLINE)) == 0)
9             return;              /* connection closed by other end */
10        Writen(sockfd, line, n);
11    }
12 }
```

lib/str_echo.c

Figure 5.3 `str_echo` function: echo lines on a socket.

tcpcliserv/tcpcli01.c

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int     sockfd;
6     struct sockaddr_in servaddr;
7     if (argc != 2)
8         err_quit("usage: tcpcli <IPaddress>");
9     sockfd = Socket(AF_INET, SOCK_STREAM, 0);
10    bzero(&servaddr, sizeof(servaddr));
11    servaddr.sin_family = AF_INET;
12    servaddr.sin_port = htons(SERV_PORT);
13    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
14    Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
15    str_cli(stdin, sockfd);     /* do it all */
16    exit(0);
17 }
```

tcpcliserv/tcpcli01.c

Figure 5.4 TCP echo client.

Create socket, fill in Internet socket address structure

9–13 A TCP socket is created and an Internet socket address structure is filled in with the server's IP address and port number. We take the server's IP address from the command-line argument and the server's well-known port (`SERV_PORT`) is from our `unp.h` header.

Connect to server

14–15 `connect` establishes the connection with the server. The function `str_cli` (Figure 5.5) then handles the rest of the client processing.

5.5 TCP Echo Client: `str_cli` Function

This function, shown in Figure 5.5, handles the client processing loop: read a line of text from standard input, write it to the server, read back the server's echo of the line, and output the echoed line to standard output.

lib/str_cli.c

```
 1 #include    "unp.h"

 2 void
 3 str_cli(FILE *fp, int sockfd)
 4 {
 5     char    sendline[MAXLINE], recvline[MAXLINE];

 6     while (Fgets(sendline, MAXLINE, fp) != NULL) {

 7         Writen(sockfd, sendline, strlen(sendline));

 8         if (Readline(sockfd, recvline, MAXLINE) == 0)
 9             err_quit("str_cli: server terminated prematurely");

10         Fputs(recvline, stdout);
11     }
12 }
```

lib/str_cli.c

Figure 5.5 `str_cli` function: client processing loop.

Read a line, write to server

6-7 `fgets` reads a line of text and `writen` sends the line to the server.

Read echoed line from server, write to standard output

8-10 `readline` reads the line echoed back from the server and `fputs` writes it to the standard output.

Return to `main`

11-12 The loop terminates when `fgets` returns a null pointer, which occurs when it encounters either an end-of-file or an error. Our `Fgets` wrapper function checks for an error and aborts if one occurs, so `Fgets` returns a null pointer only when an end-of-file is encountered.

5.6 Normal Startup

Although our TCP example is small (about 150 lines of code for the two `main` functions, `str_echo`, `str_cli`, `readline`, and `writen`), it is essential that we understand how the client and server start, how they end, and most importantly, what happens when something goes wrong: the client host crashes, the client process crashes, network connectivity is lost, and so on. Only by understanding these boundary conditions, and their interaction with the TCP/IP protocols, can we write robust clients and servers that can handle these conditions.

We first start the server in the background on the host bsdi.

```
bsdi % tcpserv01 &
[1]     21130
```

When the server starts, it calls socket, bind, listen, and accept, blocking in the call to accept. (We have not started the client yet.) Before starting the client, we run the netstat program to verify the state of the server's listening socket.

```
bsdi % netstat -a
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  *.9877                 *.*                    LISTEN
```

Here we show only the first line of output (the heading), and the line that we are interested in. This command shows the status of *all* sockets on the system, which can be lots of output. We must specify the -a flag to see listening sockets.

The output is what we expect. A socket is in the LISTEN state with a wildcard for the local IP address and a local port of 9877. netstat prints an asterisk for an IP address of 0 (INADDR_ANY, the wildcard) or for a port of 0.

We then start the client on the same host, specifying the server's IP address of 127.0.0.1. We could have also specified this address as 206.62.226.35 (Figure 1.16).

```
bsdi % tcpcli01 127.0.0.1
```

The client calls socket and connect, the latter causing TCP's three-way handshake to take place. When the three-way handshake completes, connect returns in the client and accept returns in the server. The connection is established. The following steps then take place:

1. The client calls str_cli, which will block in the call to fgets, because we have not typed a line of input yet.
2. When accept returns in the server, it calls fork and the child calls str_echo. This function calls readline, which calls read, which blocks, waiting for a line to be sent from the client.
3. The server parent, on the other hand, calls accept again, and blocks, waiting for the next client connection.

We have three processes, and all three are asleep (blocked): client, server parent, and server child.

> When the three-way handshake completes, we purposely list the client step first, and then the server steps. The reason can be seen in Figure 2.5: connect returns when the second segment of the handshake is received by the client but accept does not return until the third segment of the handshake is received by the server, one-half of a round-trip time after connect returns.

We purposely run the client and server on the same host, because this is the easiest way to experiment with client–server applications. Since we are running the client and server on the same host, netstat now shows two additional lines of output, corresponding to the TCP connection.

```
bsdi % netstat -a
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  localhost.9877         localhost.1052         ESTABLISHED
tcp        0      0  localhost.1052         localhost.9877         ESTABLISHED
tcp        0      0  *.9877                 *.*                    LISTEN
```

The first of the ESTABLISHED lines corresponds to the server child's socket, since the local port is 9877. The second of the ESTABLISHED lines is the client's socket, since the local port is 1052. If we were running the client and server on different hosts, the client host would display only the client's socket, and the server host would display only the two server sockets.

We can also use the `ps` command to check the status and relationship of these processes.

```
bsdi % ps -l
  PID   PPID    WCHAN  STAT  TT       TIME COMMAND
19130 19129    wait   Is    p1    0:04.99 -ksh (ksh)
21130 19130    netcon I     p1    0:00.06 tcpserv01
21131 19130    ttyin  I+    p1    0:00.09 tcpcli01 127.0.0.1
21132 21130    netio  I     p1    0:00.01 tcpserv01
21134 21133    wait   Ss    p2    0:03.50 -ksh (ksh)
21149 21134    -      R+    p2    0:00.05 ps -l
```

(We have removed several columns of output that do not affect this discussion.) In this output we ran the client and server from the same window (`p1`, which stands for pseudo-terminal number 1) and ran the `ps` command from another (`p2`). The PID and PPID columns show the parent and child relationships. We can tell that the first `tcpserv01` line is the parent and the second `tcpserv01` line is the child since the PPID of the child is the parent's PID. Also the PPID of the parent is the shell (`ksh`).

The STAT column for all three of our network processes is "I" meaning the process is idle (i.e., asleep). The plus sign at the end of two of the STAT columns indicates that the process is in the foreground process group of its control terminal. If the process is asleep, the WCHAN column specifies the condition. 4.4BSD prints `netcon` if the process is blocked in either `accept` or `connect`, `netio` if the process is blocked on socket input or output, or `ttyin` or `ttyout` if the process is blocked on terminal I/O. The WCHAN values for our three network processes make sense.

5.7 Normal Termination

At this point the connection is established and whatever we type to the client is echoed back.

```
bsdi % tcpcli01 127.0.0.1          we showed this line earlier
hello, world                       we now type this
hello, world                       and the line is echoed
good bye
good bye
^D                                 Control-D is our terminal EOF character
```

We type in two lines, each one is echoed, and then we type our terminal EOF character (Control-D) which terminates the client. If we immediately execute `netstat` we have

```
bsdi % netstat -a | grep 9877
tcp        0      0  localhost.1052         localhost.9877         TIME_WAIT
tcp        0      0  *.9877                 *.*                    LISTEN
```

The client's side of the connection (since the local port is 1052) enters the TIME_WAIT state (Section 2.6), and the listening server is still waiting for another client connection. (This time we pipe the output of `netstat` into `grep`, printing only the lines with our server's well-known port. But doing this also removes the heading line.)

We can follow through the steps involved in the normal termination of our client and server.

1. When we type our EOF character, `fgets` returns a null pointer and the function `str_cli` (Figure 5.5) returns.
2. When `str_cli` returns to the client `main` function (Figure 5.4), the latter terminates by calling `exit`.
3. Part of process termination is the closing of all open descriptors, so the client socket is closed by the kernel. This sends a FIN to the server, to which the server TCP responds with an ACK. This is the first half of the TCP connection termination sequence. At this point the server socket is in the CLOSE_WAIT state and the client socket is in the FIN_WAIT_2 state (Figures 2.4 and 2.5).
4. When the server TCP receives the FIN, the server child is blocked in a call to `readline` (Figure 5.3), and `readline` then returns 0. This causes the `str_echo` function to return to the server child `main`.
5. The server child terminates by calling `exit` (Figure 5.2).
6. All open descriptors in the server child are closed. Closing the connected socket by the child causes the final two segments of the TCP connection termination to take place: a FIN from the server to the client, and an ACK from the client (Figure 2.5). At this point the connection is completely terminated. The client socket enters the TIME_WAIT state.
7. Another part of process termination is for the `SIGCHLD` signal to be sent to the parent when the server child terminates. That occurs in this example, but we do not catch the signal in our code, and the default action of this signal is to be ignored. The child enters the zombie state. We can verify this with the `ps` command.

```
bsdi % ps
  PID  TT  STAT      TIME COMMAND
19130  p1  Ss     0:05.08 -ksh (ksh)
21130  p1  I      0:00.06 tcpserv01
21132  p1  Z      0:00.00 (tcpserv01)
21167  p1  R+     0:00.10 ps
```

The STAT of the child is now `Z` (for zombie).

We need to clean up our zombie processes and doing this requires dealing with Unix signals. In the next section we give an overview of signal handling and the following section continues our example.

5.8 Posix Signal Handling

A *signal* is a notification to a process that an event has occurred. Signals are sometimes called *software interrupts*. Signals usually occur *asynchronously*. By this we mean that the process doesn't know ahead of time exactly when a signal will occur.

Signals can be sent

- by one process to another process (or to itself),
- by the kernel to a process.

The `SIGCHLD` signal that we described at the end of the previous section is one that is sent by the kernel whenever a process terminates, to the parent of the terminating process.

Every signal has a *disposition*, which is also called the *action* associated with the signal. We set the disposition of a signal by calling the `sigaction` function (described shortly) and we have three choices for the disposition.

1. We can provide a function that is called whenever a specific signal occurs. This function is called a *signal handler* and this action is called *catching* the signal. The two signals `SIGKILL` and `SIGSTOP` cannot be caught. Our function is called with a single integer argument that is the signal number and the function returns nothing. Its function prototype is therefore

   ```
   void handler(int signo);
   ```

   For most signals, calling `sigaction` and specifying a function to be called when the signal occurs is all that is required to catch a signal. But we will see later that a few signals, `SIGIO`, `SIGPOLL`, and `SIGURG`, all require additional actions on the part of the process to catch the signal.
2. We can *ignore* a signal by setting its disposition to `SIG_IGN`. The two signals `SIGKILL` and `SIGSTOP` cannot be ignored.
3. We can set the *default* disposition for a signal by setting its disposition to `SIG_DFL`. The default is normally to terminate a process on the receipt of a signal, with certain signals also generating a core image of the process in its current working directory. There are a few signals whose default disposition is to be ignored: `SIGCHLD` and `SIGURG` (sent on the arrival of out-of-band data, Chapter 21) are two that we encounter in this text that are ignored by default.

`signal` Function

The Posix way to establish the disposition of a signal is to call the `sigaction` function. This gets complicated, however, as one argument to the function is a structure that we

must allocate and fill in. An easier way to set the disposition for a signal is to call the `signal` function. The first argument is the signal name and the second argument is either a pointer to a function or one of the constants `SIG_IGN` or `SIG_DFL`. But `signal` is a historical function that predates Posix.1 and different implementations provide different signal semantics when it is called, providing backward compatibility, whereas Posix explicitly spells out the semantics when `sigaction` is called. The solution is to define our own function named `signal` that just calls the Posix `sigaction` function. This provides a simple interface with the desired Posix semantics. We include this function in our own library, along with our `err_XXX` functions and our wrapper functions, for example, that we specify when building any of our programs in this text. This function is shown in Figure 5.6.

lib/signal.c

```
 1 #include    "unp.h"

 2 Sigfunc *
 3 signal(int signo, Sigfunc *func)
 4 {
 5     struct sigaction act, oact;

 6     act.sa_handler = func;
 7     sigemptyset(&act.sa_mask);
 8     act.sa_flags = 0;
 9     if (signo == SIGALRM) {
10 #ifdef  SA_INTERRUPT
11         act.sa_flags |= SA_INTERRUPT;    /* SunOS 4.x */
12 #endif
13     } else {
14 #ifdef  SA_RESTART
15         act.sa_flags |= SA_RESTART;      /* SVR4, 4.4BSD */
16 #endif
17     }
18     if (sigaction(signo, &act, &oact) < 0)
19         return (SIG_ERR);
20     return (oact.sa_handler);
21 }
```

lib/signal.c

Figure 5.6 `signal` function that calls the Posix `sigaction` function.

Simplify function prototype using `typedef`

2–3 The normal function prototype for `signal` is complicated by the level of nested parentheses:

```
void (*signal(int signo, void (*func)(int)))(int);
```

To simplify this we define the `Sigfunc` type in our `unp.h` header as

```
typedef  void  Sigfunc(int);
```

stating that signal handlers are functions with an integer argument and the function returns nothing (`void`). The function prototype is then

```
Sigfunc *signal(int signo, Sigfunc *func);
```

A pointer to a signal handling function is the second argument to the function, as well as the return value from the function.

Set handler

6 The `sa_handler` member of the `sigaction` structure is set to the *func* argument.

Set signal mask for handler

7 Posix allows us to specify a set of signals that will be *blocked* when our signal handler is called. Any signal that is blocked cannot be *delivered* to the process. We set the `sa_mask` member to the empty set, which means that no additional signals are blocked while our signal handler is running. Posix guarantees that the signal being caught is always blocked while its handler is executing.

Set `SA_RESTART` flag

8-17 An optional flag is `SA_RESTART` and if set, a system call interrupted by this signal will be automatically restarted by the kernel. (We talk more about interrupted system calls in the next section when we continue our example.) If the signal being caught is not `SIGALRM`, we specify the `SA_RESTART` flag, if defined. (The reason for making a special case for `SIGALRM` is that the purpose of generating this signal is normally to place a timeout on an I/O operation, as we show in Section 13.2, in which case we want the blocked system call to be interrupted by the signal.) Older systems, notably SunOS 4.x, automatically restart an interrupted system call by default and then define the complement of this flag as `SA_INTERRUPT`. If this flag is defined, we set it if the signal being caught is `SIGALRM`.

Call `sigaction`

18-20 We call `sigaction` and then return the old action for the signal as the return value of the `signal` function.

Throughout this text we use the `signal` function from Figure 5.6.

Posix Signal Semantics

We summarize the following points about signal handling on a Posix-compliant system.

- Once a signal handler is installed, it remains installed. (Older systems removed the signal handler each time it was executed.)
- While a signal handler is executing, the signal being delivered is blocked. Furthermore any additional signals that were specified in the `sa_mask` signal set passed to `sigaction` when the handler was installed are also blocked. In Figure 5.6 we set `sa_mask` to the empty set, meaning no additional signals are blocked other than the signal being caught.
- If a signal is generated one or more times while it is blocked, it is normally delivered only one time after the signal is unblocked. That is, by default Unix signals are not *queued*. We will see an example of this in the next section. The Posix realtime standard, 1003.1b, defines a set of reliable signals that are queued, but we do not use them in this text.

- It is possible to selectively block and unblock a set of signals using the `sigprocmask` function. This lets us protect a critical region of code by preventing certain signals from being caught while that region of code is executing.

5.9 Handling `SIGCHLD` Signals

The purpose of the zombie state is to maintain information about the child for the parent to fetch at some later time. This information includes the process ID of the child, its termination status, and information on the resource utilization of the child (CPU time, memory, etc.). If a process terminates, and that process has children in the zombie state, the parent process ID of all the zombie children is set to 1 (the `init` process), which will inherit the children and clean them up (i.e., `init` will `wait` for them, which removes the zombie). Some Unix systems show the COMMAND column for a zombie process as `<defunct>`.

Handling Zombies

Obviously we do not want to leave zombies around. They take up space in the kernel and eventually we can run out of processes. Whenever we `fork` children, we must `wait` for them to prevent them from becoming zombies. To do this we establish a signal handler to catch `SIGCHLD` and within the handler we call `wait`. (We describe the `wait` and `waitpid` functions in Section 5.10.) We establish the signal handler by adding the function call

```
Signal(SIGCHLD, sig_chld);
```

in Figure 5.2, after the call to `listen`. (It must be done sometime before we `fork` the first child and need be done only once.) We then define the signal handler, the function `sig_chld`, which we show in Figure 5.7.

tcpcliserv/sigchldwait.c

```
 1 #include    "unp.h"
 2 void
 3 sig_chld(int signo)
 4 {
 5     pid_t   pid;
 6     int     stat;
 7     pid = wait(&stat);
 8     printf("child %d terminated\n", pid);
 9     return;
10 }
```

tcpcliserv/sigchldwait.c

Figure 5.7 Version of `SIGCHLD` signal handler that calls `wait` (improved in Figure 5.11).

Warning: Calling standard I/O functions such as `printf` in a signal handler is not recommended, for reasons that we discuss in Section 11.14. We call `printf` here as a diagnostic tool to see when the child terminates.

Under System V and Unix 98 the child of a process does not become a zombie if the process sets the disposition of SIGCHLD to SIG_IGN. Unfortunately this works only under System V and Unix 98. Posix.1 explicitly states that this behavior is unspecified. The portable way to handle zombies is to catch SIGCHLD and call wait or waitpid.

If we compile this program—Figure 5.2, with the call to Signal, with our sig_chld handler—under Solaris 2.5 and use the signal function from the system library (not our version from Figure 5.6), we have the following:

```
solaris % tcpserv02 &                    start server in background
[2]     16939
solaris % tcpcli01 127.0.0.1             then client in foreground
hi there                                 we type this
hi there                                 and this is echoed
^D                                       we type our EOF character
child 16942 terminated                   output by printf in signal handler
accept error: Interrupted system call    but main function aborts
```

The sequence of steps is as follows:

1. We terminate the client by typing our EOF character. The client TCP sends a FIN to the server and the server responds with an ACK.
2. The receipt of the FIN delivers an EOF to the child's pending readline. The child terminates.
3. The parent is blocked in its call to accept when the SIGCHLD signal is delivered. The sig_chld function executes (our signal handler), wait fetches the child's PID and termination status, and printf is called from the signal handler. The signal handler returns.
4. Since the signal was caught by the parent while the parent was blocked in a slow system call (accept), the kernel causes the accept to return an error of EINTR (interrupted system call). The parent does not handle this error (Figure 5.2), so it aborts.

The purpose of this example is to show that when writing network programs that catch signals, we must be cognizant of interrupted system calls, and we must handle them. In this specific example, running under Solaris 2.5, the signal function provided in the standard C library does not cause an interrupted system call to be automatically restarted by the kernel. That is, the SA_RESTART flag that we set in Figure 5.6 is not set by the signal function in the system library. Some other systems automatically restart the interrupted system call. If we run the same example under 4.4BSD, using its library version of the signal function, the kernel restarts the interrupted system call and accept does not return an error. To handle this potential problem between different operating systems is one reason we define our own version of the signal function that we use throughout the text (Figure 5.6).

We always code an explicit return in our signal handlers (Figure 5.7), even though falling off the end of the function does the same thing for a function returning void. This provides a warning that the return may interrupt a system call.

Handling Interrupted System Calls

We used the term *slow system call* to describe `accept` and we use this term for any system call that can block forever. That is, the system call need never return. Most networking functions fall into this category. For example, there is no guarantee that a server's call to `accept` will ever return, if there are no clients that will connect to the server. Similarly our server's call to `read` (through `readline`) in Figure 5.3 will never return if the client never sends a line for the server to echo. Other examples of slow system calls are reads and writes of pipes and terminal devices. A notable exception is disk I/O, which usually returns to the caller (assuming no catastrophic hardware failure).

The basic rule that applies here is that when a process is blocked in a slow system call *and* the process catches a signal *and* the signal handler returns, the system call *can* return an error of `EINTR`. *Some* kernels automatically restart *some* interrupted system calls. For portability when we write a program that catches signals (most concurrent servers catch `SIGCHLD`), we must be prepared for slow system calls to return `EINTR`. Portability problems are caused by the qualifiers "can" and "some" used earlier and the fact that support for the Posix `SA_RESTART` flag is optional. Even if the implementation supports the `SA_RESTART` flag, not all interrupted system calls may automatically be restarted. Most Berkeley-derived implementations, for example, never automatically restart `select` and some of these implementations never restart `accept` or `recvfrom`.

To handle an interrupted `accept` we change the call to `accept` in Figure 5.2, the beginning of the `for` loop, to the following:

```
for ( ; ; ) {
    clilen = sizeof(cliaddr);
    if ( (connfd = accept(listenfd, (SA *) &cliaddr, &clilen)) < 0) {
        if (errno == EINTR)
            continue;       /* back to for() */
        else
            err_sys("accept error");
    }
```

Notice that we call `accept` and not our wrapper function `Accept`, since we must handle the failure of the function ourself.

What we are doing in this piece of code is restarting the interrupted system call ourself. This is fine for `accept` along with the functions such as `read`, `write`, `select`, and `open`. But there is one function that we cannot restart ourself: `connect`. If this function returns `EINTR`, we cannot call it again, as doing so will return an immediate error. When `connect` is interrupted by a caught signal and is not automatically restarted, we must call `select` to wait for the connection to complete, as we describe in Section 15.3.

5.10 `wait` and `waitpid` Functions

In Figure 5.7 we called the `wait` function to handle the terminated child.

```
#include <sys/wait.h>

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);
```

Both return: process ID if OK, 0, or –1 on error

`wait` and `waitpid` both return two values: the return value of the function is the process ID of the terminated child, and the termination status of the child (an integer) is returned through the *statloc* pointer. There are three macros that we can call that examine the termination status and tell us if the child terminated normally, was killed by a signal, or is just job-control stopped. Additional macros let us then fetch the exit status of the child, or the value of the signal that killed the child, or the value of the job-control signal that stopped the child. We use the `WIFEXITED` and `WEXITSTATUS` macros in Figure 14.10.

If there are no terminated children for the process calling `wait`, but the process has one or more children that are still executing, then `wait` blocks until the first of the existing children terminate.

`waitpid` gives us more control over which process to wait for and whether or not to block. First, the *pid* argument lets us specify the process ID that we want to wait for. A value of –1 says to wait for the first of our children to terminate. (There are other options, dealing with process group IDs, but we do not need them in this text.) The *options* argument lets us specify additional options. The most common option is `WNOHANG`. This option tells the kernel not to block if there are no terminated children.

Difference between `wait` and `waitpid`

We now want to illustrate the difference between the `wait` and `waitpid` functions, when used to clean up terminated children. To do this we modify our TCP client as shown in Figure 5.9. The client establishes five connections with the server and then uses only the first one (`sockfd[0]`) in the call to `str_cli`. The purpose of establishing multiple connections is to spawn multiple children from the concurrent server, as shown in Figure 5.8.

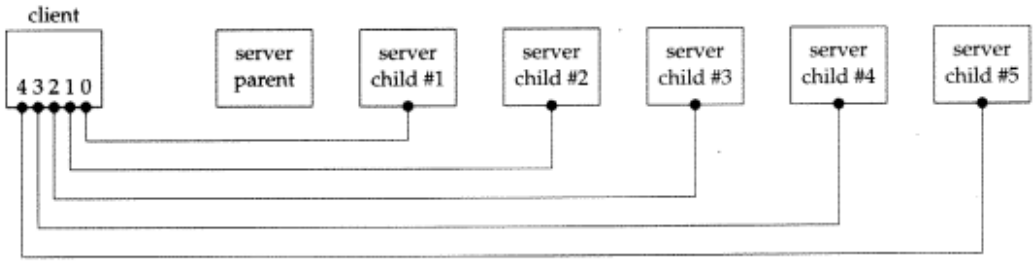

Figure 5.8 Client with five established connections to same concurrent server.

tcpcliserv/tcpcli04.c

```
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     i, sockfd[5];
 6     struct sockaddr_in servaddr;

 7     if (argc != 2)
 8         err_quit("usage: tcpcli <IPaddress>");

 9     for (i = 0; i < 5; i++) {
10         sockfd[i] = Socket(AF_INET, SOCK_STREAM, 0);

11         bzero(&servaddr, sizeof(servaddr));
12         servaddr.sin_family = AF_INET;
13         servaddr.sin_port = htons(SERV_PORT);
14         Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

15         Connect(sockfd[i], (SA *) &servaddr, sizeof(servaddr));
16     }

17     str_cli(stdin, sockfd[0]);  /* do it all */

18     exit(0);
19 }
```

tcpcliserv/tcpcli04.c

Figure 5.9 TCP client that establishes five connections with server.

When the client terminates, all open descriptors are closed automatically by the kernel (we do not call `close`, only `exit`), and all five connections are terminated at about the same time. This causes five FINs to be sent, one on each connection, which in turn causes all five server children to terminate at about the same time. This causes five `SIGCHLD` signals to be delivered to the parent at about the same time, which we show in Figure 5.10.

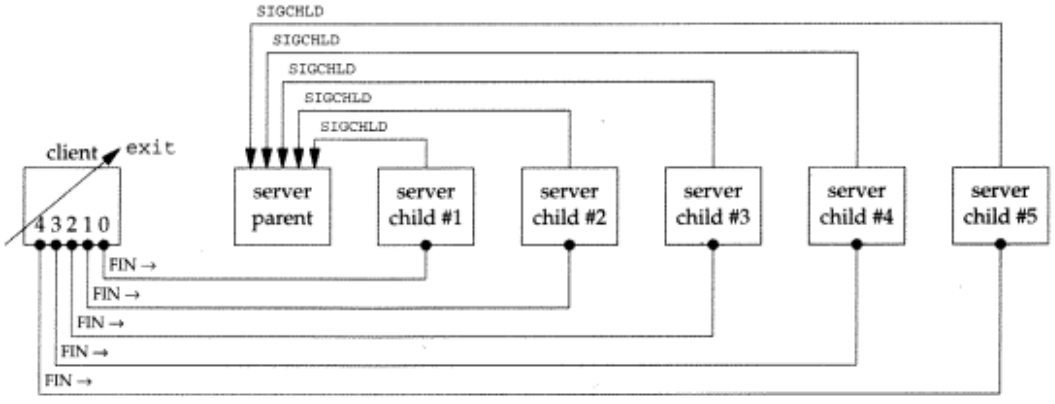

Figure 5.10 Client terminates, closing all five connections, terminating all five children.

this delivery of multiple occurrences of the same signal that causes the problem we are about to see.

We first run the server in the background and then our new client. Our server is Figure 5.2, modified to call `signal` to establish Figure 5.7 as a signal handler for `SIGCHLD`.

```
bsdi % tcpserv03 &
[1]     21282
bsdi % tcpcli04 206.62.226.35
hello                                   we type this
hello                                   and it is echoed
^D                                      we then type our EOF character
child 21288 terminated                  output by server
```

The first thing we notice is that only one `printf` is output, when we expect all five children to have terminated. If we execute `ps`, we see that the other four children still exist as zombies.

```
  PID  TT  STAT      TIME COMMAND
21282  p1  S      0:00.09 tcpserv03
21284  p1  Z      0:00.00 (tcpserv03)
21285  p1  Z      0:00.00 (tcpserv03)
21286  p1  Z      0:00.00 (tcpserv03)
21287  p1  Z      0:00.00 (tcpserv03)
```

Establishing a signal handler and calling `wait` from that handler are insufficient for preventing zombies. The problem is that all five signals are generated before the signal handler is executed, and the signal handler is executed only one time because Unix signals are normally not *queued*. Furthermore this problem is nondeterministic. In the example we just ran, with the client and server on the same host, the signal handler is executed once, leaving four zombies. But if we run the client and server on different hosts, the signal handler is normally executed two times: once as a result of the first signal being generated, and since the other four signals occur while the signal handler is executing, the handler is called only one more time. This leaves three zombies. But sometimes, probably dependent on the timing of the FINs arriving at the server host, the signal handler is executed three or even four times.

The correct solution is to call `waitpid` instead of `wait`. Figure 5.11 shows the version of our `sig_chld` function that handles `SIGCHLD` correctly. This version works because we call `waitpid` within a loop, fetching the status of any of our children that have terminated. We must specify the `WNOHANG` option: this tells `waitpid` not to block if there exist running children that have not yet terminated. In Figure 5.7 we cannot call `wait` in a loop, because there is no way to prevent `wait` from blocking if there exist running children that have not yet terminated.

Figure 5.12 shows the final version of our server. It correctly handles a return of `EINTR` from `accept` and it establishes a signal handler (Figure 5.11) that calls `waitpid` for all terminated children.

tcpcliserv/sigchldwaitpid.c

```
 1 #include    "unp.h"
 2 void
 3 sig_chld(int signo)
 4 {
 5     pid_t   pid;
 6     int     stat;
 7     while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
 8         printf("child %d terminated\n", pid);
 9     return;
10 }
```

tcpcliserv/sigchldwaitpid.c

Figure 5.11 Final (correct) version of `sig_chld` function that calls `waitpid`.

tcpcliserv/tcpserv04.c

```
 1 #include    "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     listenfd, connfd;
 6     pid_t   childpid;
 7     socklen_t clilen;
 8     struct sockaddr_in cliaddr, servaddr;
 9     void    sig_chld(int);
10     listenfd = Socket(AF_INET, SOCK_STREAM, 0);
11     bzero(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
14     servaddr.sin_port = htons(SERV_PORT);
15     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
16     Listen(listenfd, LISTENQ);
17     Signal(SIGCHLD, sig_chld);  /* must call waitpid() */
18     for ( ; ; ) {
19         clilen = sizeof(cliaddr);
20         if ( (connfd = accept(listenfd, (SA *) &cliaddr, &clilen)) < 0) {
21             if (errno == EINTR)
22                 continue;       /* back to for() */
23             else
24                 err_sys("accept error");
25         }
26         if ( (childpid = Fork()) == 0) {     /* child process */
27             Close(listenfd);    /* close listening socket */
28             str_echo(connfd);   /* process the request */
29             exit(0);
30         }
31         Close(connfd);          /* parent closes connected socket */
32     }
33 }
```

tcpcliserv/tcpserv04.c

Figure 5.12 Final (correct) version of TCP server that handles an error of `EINTR` from `accept`.

The purpose of this section has been to demonstrate three scenarios that we can encounter with network programming.

1. We must catch the SIGCHLD signal when forking child processes.
2. We must handle interrupted system calls when we catch signals.
3. A SIGCHLD handler must be coded correctly using waitpid to prevent any zombies from being left around.

The final version of our TCP server (Figure 5.12) along with the SIGCHLD handler in Figure 5.11 handles all three scenarios.

5.11 Connection Abort before accept Returns

There is another condition, similar to the interrupted system call example in the previous section, that can cause accept to return a nonfatal error, in which case we should just call accept again. The sequence of packets shown in Figure 5.13 has been seen on busy servers (typically busy Web servers).

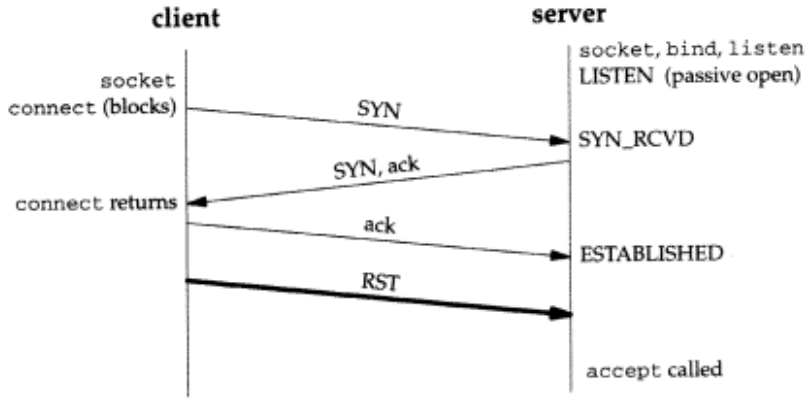

Figure 5.13 Receiving an RST for an ESTABLISHED connection before accept is called.

The three-way handshake completes, the connection is established, and then the client TCP sends an RST (reset). On the server side the connection is queued by its TCP, waiting for the server process to call accept when the RST arrives. Some time later the server process calls accept.

> An easy way to simulate this scenario is to start the server, have it call socket, bind, and listen, and then go to sleep for a short period of time before calling accept. While the server process is asleep, start the client, have it call socket and connect. As soon as connect returns, set the SO_LINGER socket option to generate the RST (which we describe in Section 7.5 and show an example of in Figure 15.21) and terminate.

Unfortunately, what happens to the aborted connection is implementation dependent. Berkeley-derived implementations handle the aborted connection completely within the kernel, and the server process never sees it. Most SVR4 implementations, however, return an error to the process as the return from `accept`, and the error depends on the implementation. These SVR4 implementations return an `errno` of `EPROTO` ("protocol error"), but Posix.1g specifies that the return must be `ECONNABORTED` ("software caused connection abort") instead. The reason for the Posix.1g change is that `EPROTO` is also returned when some fatal protocol-related events occur on the streams subsystem. Returning the same error for the nonfatal abort of an established connection by the client makes it impossible for the server to know whether to call `accept` again or not. In the case of the `ECONNABORTED` error, the server can ignore the error and just call `accept` again.

> Solaris 2.6 implements the Posix.1g change.
>
> The steps involved in Berkeley-derived kernels that never pass this error to the process can be followed in TCPv2. The RST is processed on p. 964, causing `tcp_close` to be called. This function calls `in_pcbdetach` on p. 897, which in turn calls `sofree` on p. 719. `sofree` (p. 473) finds that the socket being aborted is still on the listening socket's completed connection queue and removes the socket from the queue and frees the socket. When the server gets around to calling `accept`, it will never know that a connection that had completed has since been removed from the queue.

We return to these aborted connections in Section 15.6 and see how they can present a problem when combined with `select` and a listening socket in the normal blocking mode.

5.12 Termination of Server Process

We now start our client–server and then kill the server child process. This simulates the crashing of the server process, so we can see what happens to the client. (We must be careful to distinguish between the crashing of the server *process*, which we are about to describe, and the crashing of the server *host*, which we describe in Section 5.14.) The following steps take place:

1. We start the server and client on different hosts and type one line to the client to verify that all is OK. That line is echoed normally by the server child.
2. We find the process ID of the server child and `kill` it. As part of process termination all open descriptors in the child are closed. This causes a FIN to be sent to the client, and the client TCP responds with an ACK. This is the first half of the TCP connection termination.
3. The `SIGCHLD` signal is sent to the server parent and handled correctly (Figure 5.12).
4. Nothing happens at the client. The client TCP receives the FIN from the server TCP, and responds with an ACK, but the problem is that the client process is blocked in the call to `fgets` waiting for a line from the terminal.

5. Running `netstat` at this point from another window on the client shows the state of the client socket.

```
solaris % netstat | grep 9877
Local Address   Remote Address  Swind Send-Q Rwind Recv-Q  State
solaris.34673   bsdi.9877        8760      0  8760      0  CLOSE_WAIT
```

(This is the first time we have shown the `netstat` output from Solaris so we have added the heading line. The format is slightly different from the BSD output, but the information is similar.) We also run `netstat` from another window on the server:

```
bsdi % netstat | grep 9877
tcp        0      0  bsdi.9877        solaris.34673       FIN_WAIT_2
```

From Figure 2.4 we see that half of the TCP connection termination sequence has taken place.

6. We can still type a line of input to the client. Here is what happens at the client starting from step 1.

```
solaris % tcpcli01 206.62.226.35      start client
hello                                 the first line that we type
hello                                 it is echoed correctly
                                      here we kill the server child on the server host
another line                          we then type a second line to the client
str_cli: server terminated prematurely
```

When we type "another line", `str_cli` calls `writen` and the client TCP sends the data to the server. This is allowed by TCP because the receipt of the FIN by the client TCP only indicates that the server process has closed its end of the connection and will not be sending any more data. The receipt of the FIN does *not* tell the client TCP that the server process has terminated (which in this case it has). We cover this again in Section 6.6 when we talk about TCP's half-close.

When the server TCP receives the data from the client, it responds with an RST since the process that had that socket open has terminated. We can verify that the RST is sent by watching the packets with `tcpdump`.

7. But the client process will not see the RST because it calls `readline` immediately after the call to `writen` and `readline` returns 0 (end-of-file) immediately because of the FIN that was received in step 2. Our client is not expecting to receive an end-of-file at this point (Figure 5.5) so it quits with the error message "server terminated prematurely."

8. When the client terminates (by calling `err_quit` in Figure 5.5), all its open descriptors are closed.

What we have described also depends on the timing of the example. When we run the client and server on different hosts, as we just described, it takes a few milliseconds for the data to be sent from the client to the server (the "another line") and the server's RST to be received by the client. That is why the client's call to `readline` returns 0 because the FIN that was received earlier is ready to be read. But if we run the client and server on the same host, or if we were

to put a slight pause in the client before its call to `readline`, then the received RST takes precedence over the FIN that was received earlier. This would cause `readline` to return an error and `errno` would contain `ECONNRESET` ("Connection reset by peer").

The problem in this example is that the client is blocked in the call to `fgets` when the FIN arrives on the socket. The client is really working with two descriptors—the socket and the user input—and instead of blocking on input from only one of the two sources (as `str_cli` is currently coded), it should block on input from either source. Indeed, this is one purpose of the `select` and `poll` functions, which we describe in Chapter 6. When we recode the `str_cli` function in Section 6.4, as soon as we `kill` the server child, the client is notified of the received FIN.

5.13 SIGPIPE Signal

What happens if the client ignores the error return from `readline` and writes more data to the server? This can happen, for example, if the client needs to perform two writes to the server before reading anything back, with the first write eliciting the RST.

The rule that applies is: when a process writes to a socket that has received an RST, the `SIGPIPE` signal is sent to the process. The default action of this signal is to terminate the process so the process must catch the signal to avoid being involuntarily terminated.

If the process either catches the signal and returns from the signal handler, or ignores the signal, the write operation returns `EPIPE`.

A frequently asked question (FAQ) on Usenet is how to obtain this signal on the first write, and not the second. This is not possible. Following our discussion above, the first write elicits the RST and the second write elicits the signal. It is OK to write to a socket that has received a FIN, but it is an error to write to a socket that has received an RST.

To see what happens with `SIGPIPE` we modify our client as shown in Figure 5.14.

tcpcliserv/str_cli11.c

```
1 #include    "unp.h"
2 void
3 str_cli(FILE *fp, int sockfd)
4 {
5     char    sendline[MAXLINE], recvline[MAXLINE];
6     while (Fgets(sendline, MAXLINE, fp) != NULL) {
7         Writen(sockfd, sendline, 1);
8         sleep(1);
9         Writen(sockfd, sendline + 1, strlen(sendline) - 1);
10        if (Readline(sockfd, recvline, MAXLINE) == 0)
11            err_quit("str_cli: server terminated prematurely");
12        Fputs(recvline, stdout);
13    }
14 }
```

tcpcliserv/str_cli11.c

Figure 5.14 `str_cli` that calls `writen` twice.

All we have changed is to call `writen` two times: the first time the first byte of data is written to the socket, followed by a pause of 1 second, followed by the remainder of the line. The intent is for the first `writen` to elicit the RST and then for the second `writen` to generate `SIGPIPE`.

If we run the client on our BSD/OS host, we get:

```
bsdi % tcpcli11 206.62.226.34
hi there                                  we type this line
hi there                                  this is echoed by the server
                                          here we kill the server child
bye                                       then we type this line
bsdi % echo $?                            what is the KornShell's return value of last command?
269                                       269 = 256 + 13
bsdi % grep SIGPIPE /usr/include/sys/signal.h
#define SIGPIPE 13      /* write on a pipe with no one to read it */
```

We start the client, type in one line, see that line echoed correctly, and then terminate the server child on the server host. We then type another line ("bye") but nothing is echoed and we just get a shell prompt. Since the default action of `SIGPIPE` is to terminate the process without generating a `core` file, nothing is printed by the KornShell. This is the problem with programs terminated by `SIGPIPE`: normally nothing is output even by the shell to indicate what has happened.

We must execute `echo $?` to print the shell's return value, which is 269. We then print the numeric value of the constant `SIGPIPE` and see that the KornShell's return value is 256 plus the signal number. But if we execute this program under Digital Unix 4.0, Solaris 2.5, or UnixWare 2.1.2, the KornShell's return value is 141, or 128 plus 13.

> The 11/16/88 version of the KornShell returned 128 plus the signal number, while newer versions return 256 plus the signal number. All that Posix.2 specifies is that the return value be greater than 128. Other shells may return different values.

The recommended way to handle `SIGPIPE` depends on what the application wants to do when this occurs. If there is nothing special to do, then setting the signal disposition to `SIG_IGN` is easy, assuming that subsequent output operations will catch the error of `EPIPE` and terminate. If special actions are needed when the signal occurs (writing to a log file perhaps), then the signal should be caught and any desired actions can be performed in the signal handler. Be aware, however, that if multiple sockets are in use, the delivery of the signal does not tell us which socket encountered the error. If we need to know which `write` caused the error, then we must either ignore the signal or return from the signal handler and handle `EPIPE` from the `write`.

5.14 Crashing of Server Host

Our next scenario is to see what happens when the server host crashes. To simulate this we must run the client and server on different hosts. We then start the server, start the client, type in a line to the client to verify that the connection is up, disconnect the server host from the network, and type in another line at the client. This also covers the scenario of the server host being unreachable when the client sends data (i.e., some intermediate router is down after the connection has been established).

The following steps take place:

1. When the server host crashes, nothing is sent out on the existing network connections. That is, we are assuming the host crashes, and is not shut down by an operator (which we cover in Section 5.16).
2. We type a line of input to the client, it is written by `writen` (Figure 5.5), and is sent by the client TCP as a data segment. The client then blocks in the call to `readline`, waiting for the echoed reply.
3. If we watch the network with `tcpdump`, we will see the client TCP continually retransmits the data segment, trying to receive an ACK from the server. Section 25.11 of TCPv2 shows a typical pattern for TCP retransmissions: Berkeley-derived implementations retransmit the data segment 12 times, waiting for around 9 minutes before giving up. When the client TCP finally gives up (assuming the server host has not been rebooted during this time, or if the server host has not crashed but was unreachable on the network, assuming the host was still unreachable), an error is returned to the client process. Since the client is blocked in the call to `readline`, it returns an error. Assuming the server host had crashed and there were no responses at all to the client's data segments, the error is `ETIMEDOUT`. But if some intermediate router determined that the server host was unreachable and responded with an ICMP destination unreachable message, the error is either `EHOSTUNREACH` or `ENETUNREACH`.

Although our client discovers (eventually) that the peer is down or unreachable, there are times when we want to detect this quicker than having to wait 9 minutes. The solution is to place a timeout on the call to `readline`, which we discuss in Section 13.2.

The scenario that we just discussed detects that the server host has crashed only when we send data to that host. If we want to detect the crashing of the server host even if we are not actively sending it data, another technique is required. We discuss the `SO_KEEPALIVE` socket option in Section 7.5 and some client–server heartbeat functions in Section 21.5.

5.15 Crashing and Rebooting of Server Host

In this scenario we establish the connection between the client and server and then assume the server host crashes and reboots. In the previous section the server host was still down when we sent it data. Here we will let the server host reboot before sending it data. The easiest way to simulate this is to establish the connection, disconnect the server from the network, shut down the server host and then reboot it, and then reconnect the server host to the network. We do not want the client to see the server host shut down (which we cover in Section 5.16).

As stated in the previous section, if the client is not actively sending data to the server when the server host crashes, the client is not aware that the server host has crashed. (This assumes we are not using the `SO_KEEPALIVE` socket option.) The following steps take place:

1. We start the server and then the client. We type a line to verify that the connection is established.
2. The server host crashes and reboots.
3. We type a line of input to the client, which is sent as a TCP data segment to the server host
4. When the server host reboots after crashing, its TCP loses all information about connections that existed before the crash. Therefore the server TCP responds to the received data segment from the client with an RST.
5. Our client is blocked in the call to `readline` when the RST is received, causing `readline` to return the error `ECONNRESET`.

If it is important for our client to detect the crashing of the server host, even if the client is not actively sending data, then some other technique (such as the `SO_KEEPALIVE` socket option or some client–server heartbeat functions) is required.

5.16 Shutdown of Server Host

The previous two sections discussed the crashing of the server host, or the server host being unreachable across the network. We now consider what happens if the server host is shut down by an operator while our server process is running on that host.

When a Unix system is shut down, the `init` process normally sends the `SIGTERM` signal to all processes (we can catch this signal), waits some fixed amount of time (often between 5 and 20 seconds), and then sends the `SIGKILL` signal (which we cannot catch) to any processes still running. This gives all running processes a short amount of time to clean up and terminate. If we do not catch `SIGTERM` and terminate, our server will be terminated by the `SIGKILL` signal. When the process terminates, all open descriptors are closed, and we then follow the same sequence of steps discussed in Section 5.12. As we stated there, we must use the `select` or `poll` function in our client to have the client detect the termination of the server process as soon as it occurs.

5.17 Summary of TCP Example

Before any TCP client and server can communicate with each other, each end must specify the socket pair for the connection: the local IP address, local port, foreign IP address, and foreign port. In Figure 5.15 we show these four values as bullets. This figure is from the client's perspective. The foreign IP address and foreign port must be specified by the client in the call to `connect`. The two local values are normally chosen by the kernel as part of the `connect` function. The client has the option of specifying either or both of the local values, by calling `bind` before `connect`, but this is not common.

As we mentioned in Section 4.10, the client can obtain the two local values chosen by the kernel by calling `getsockname` after the connection is established.

Figure 5.16 shows the same four values, but from the server's perspective.

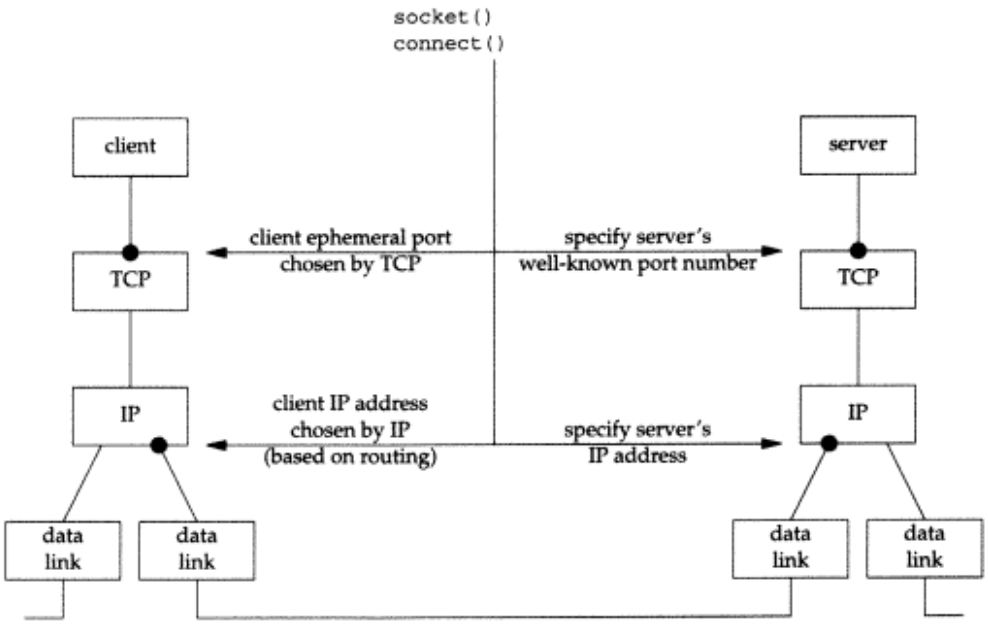

Figure 5.15 Summary of TCP client–server from client's perspective.

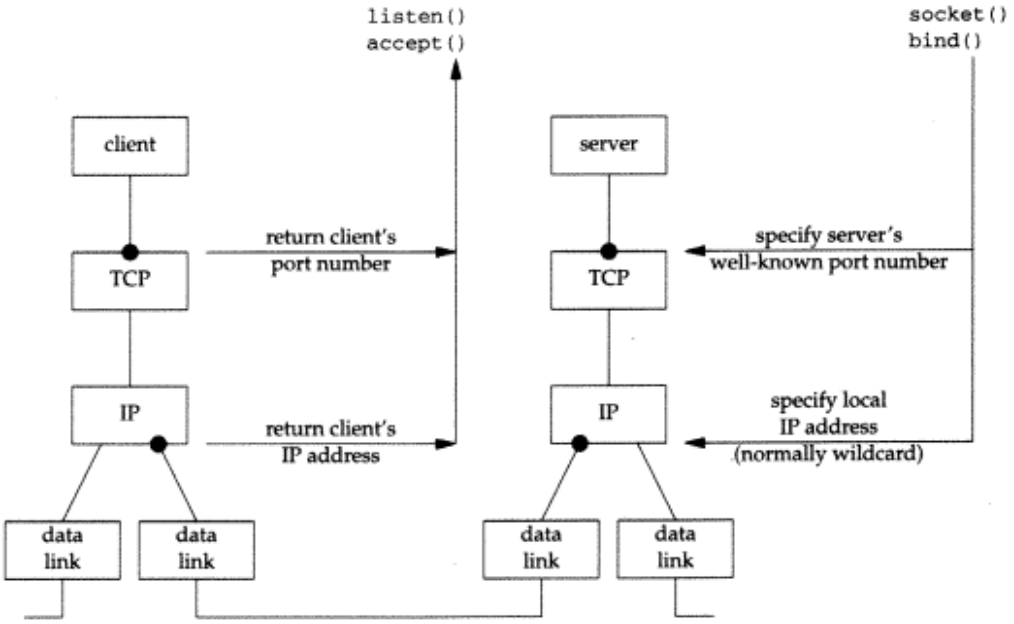

Figure 5.16 Summary of TCP client–server from server's perspective.

The local port (the server's well-known port) is specified by `bind`. Normally the server also specifies the wildcard IP address in this call, although the server can restrict itself to receiving connections destined for one particular local interface by binding a nonwildcard IP address. If the server binds the wildcard IP address on a multihomed host, it can determine the local IP address by calling `getsockname` after the connection is established (Section 4.10). The two foreign values are returned to the server by `accept`. As we mentioned in Section 4.10, if another program is `exec`ed by the server that calls `accept`, that program can call `getpeername` to determine the client's IP address and port, if necessary.

5.18 Data Format

In our example the server never examines the request that it receives from the client. The server just reads all the data up through and including the newline and sends it back to the client, looking for only the newline. This is an exception, not the rule, and normally we must worry about the format of the data exchanged between the client and server.

Example: Passing Text Strings between Client and Server

Let's modify our server so that it still reads a line of text from the client, but the server now expects that line to contain two integers separated by white space, and the server returns the sum of those two integers. Our client and server `main` functions remain the same, as does our `str_cli` function. All that changes is our `str_echo` function, which we show in Figure 5.17.

tcpcliserv/str_echo08.c

```
 1 #include    "unp.h"
 2 void
 3 str_echo(int sockfd)
 4 {
 5     long    arg1, arg2;
 6     ssize_t n;
 7     char    line[MAXLINE];
 8     for ( ; ; ) {
 9         if ( (n = Readline(sockfd, line, MAXLINE)) == 0)
10             return;             /* connection closed by other end */
11         if (sscanf(line, "%ld%ld", &arg1, &arg2) == 2)
12             snprintf(line, sizeof(line), "%ld\n", arg1 + arg2);
13         else
14             snprintf(line, sizeof(line), "input error\n");
15         n = strlen(line);
16         Writen(sockfd, line, n);
17     }
18 }
```

tcpcliserv/str_echo08.c

Figure 5.17 `str_echo` function that adds two numbers.

11-14 We call `sscanf` to convert the two arguments from text strings to long integers, and then `snprintf` to convert the result into a text string.

This new client and server work fine, regardless of the byte ordering of the client and server hosts.

Example: Passing Binary Structures between Client and Server

We now modify our client and server to pass binary values across the socket, instead of text strings. We will see that this does not work when the client and server are run on hosts with different byte orders, or on hosts that do not agree on the size of a long integer (Figure 1.17).

Our client and server `main` functions do not change. We define one structure for the two arguments, another structure for the result, and place both definitions in our `sum.h` header, shown in Figure 5.18. Figure 5.19 shows the `str_cli` function.

tcpcliserv/sum.h

```
1 struct args {
2     long    arg1;
3     long    arg2;
4 };

5 struct result {
6     long    sum;
7 };
```

tcpcliserv/sum.h

Figure 5.18 `sum.h` header.

tcpcliserv/str_cli09.c

```
1 #include    "unp.h"
2 #include    "sum.h"

3 void
4 str_cli(FILE *fp, int sockfd)
5 {
6     char    sendline[MAXLINE];
7     struct args args;
8     struct result result;

9     while (Fgets(sendline, MAXLINE, fp) != NULL) {

10        if (sscanf(sendline, "%ld%ld", &args.arg1, &args.arg2) != 2) {
11            printf("invalid input: %s", sendline);
12            continue;
13        }
14        Writen(sockfd, &args, sizeof(args));

15        if (Readn(sockfd, &result, sizeof(result)) == 0)
16            err_quit("str_cli: server terminated prematurely");

17        printf("%ld\n", result.sum);
18    }
19 }
```

tcpcliserv/str_cli09.c

Figure 5.19 `str_cli` function that sends two binary integers to server.

11-14 `sscanf` converts the two arguments from text strings to binary and we call `writen` to send the structure to the server.

15-17 We call `readn` to read the reply, and print the result using `printf`.

Figure 5.20 shows our `str_echo` function.

tcpcliserv/str_echo09.c

```
 1 #include    "unp.h"
 2 #include    "sum.h"

 3 void
 4 str_echo(int sockfd)
 5 {
 6     ssize_t n;
 7     struct args args;
 8     struct result result;

 9     for ( ; ; ) {
10         if ( (n = Readn(sockfd, &args, sizeof(args))) == 0)
11             return;              /* connection closed by other end */

12         result.sum = args.arg1 + args.arg2;
13         Writen(sockfd, &result, sizeof(result));
14     }
15 }
```

tcpcliserv/str_echo09.c

Figure 5.20 `str_echo` function that adds two binary integers.

9-14 We read the arguments by calling `readn`, calculate and store the sum, and call `writen` to send back the result structure.

If we run the client and server on two machines of the same architecture, say `solaris` and `sunos5` in Figure 1.16, everything works fine. Here is the client interaction:

```
sunos5 % tcpcli09 206.62.226.33
11 22                                   we type this
33                                      and this is the server's reply
-11 -44
-55
```

But when the client and server are on two machines of different architectures (the server on the big-endian Sparc system `solaris` and the client on the little-endian Intel system `bsdi`) it does not work.

```
bsdi % tcpcli09 206.62.226.33
1 2                                     we type this
3                                       and it works
-22 -77                                 then we type this
-16777314                               and it does not work
```

The problem is that the two binary integers are sent across the socket in little-endian format by the client, but interpreted as big-endian integers by the server. We see that it appears to work for positive integers but fails for negative integers (see Exercise 5.8). There are really three potential problems with this example.

1. Different implementations store binary numbers in different formats. The most common formats are big endian and little endian, as we described in Section 3.4.
2. Different implementations can store the same C datatype differently. For example, most 32-bit Unix systems use 32 bits for a `long` but 64-bit systems typically use 64 bits for the same datatype (Figure 1.17). There is no guarantee that a `short`, `int`, or `long` is of any certain size.
3. Different implementations pack structures differently, depending on the number of bits used for the various datatypes and the alignment restrictions of the machine. Therefore it is never wise to send binary structures across a socket.

There are two common solutions to this data format problem.

1. Pass all the numeric data as text strings. This is what we did in Figure 5.17. This assumes that both hosts have the same character set.
2. Explicitly define the binary formats of the supported datatypes (number of bits, big or little endian) and pass all data between the client and server in this format. Remote procedure call (RPC) packages normally use this technique. RFC 1832 [Srinivasan 1995] describes the *External Data Representation* (XDR) standard that is used with the Sun RPC package.

5.19 Summary

The first version of our echo client–server totaled about 150 lines (including the `readline` and `writen` functions), yet provided lots of details to examine. The first problem we encountered was zombie children and we caught the `SIGCHLD` signal to handle this. Our signal handler then called `waitpid` and we demonstrated that we must call this function instead of the older `wait` function, since Unix signals are not queued. This led us into some of the details of Posix signal handling, and additional information on this topic is provided in Chapter 10 of APUE.

The next problem we encountered was the client not being notified when the server process terminated. We saw that our client's TCP was notified, but we did not receive that notification since we were blocked waiting for user input. We will use the `select` or `poll` function in Chapter 6 to handle this scenario, by waiting for any one of multiple descriptors to be ready, instead of blocking on a single descriptor.

We also discovered that if the server host crashes, we do not detect this until the client sends data to the server. Some applications must be made aware of this fact sooner and in Section 7.5 we look at the `SO_KEEPALIVE` socket option, and in Section 21.5 we develop a set of client–server heartbeat functions.

Our simple example exchanged lines of text, which was OK since the server never looked at the lines that it echoed. Sending numerical data between the client and server can lead to a new set of problems, as shown.

Exercises

5.1 Build the TCP server from Figures 5.2 and 5.3 and the TCP client from Figures 5.4 and 5.5. Start the server and then start the client. Type in a few lines to verify that the client and server work. Terminate the client by typing your end-of-file character and note the time. Use `netstat` on the client host to verify that the client's end of the connection goes through the TIME_WAIT state. Execute `netstat` every 5 seconds or so to see when the TIME_WAIT state ends. What is the MSL for this implementation?

5.2 What happens with our echo client–server if we run the client and redirect standard input to a binary file?

5.3 What is the difference between our echo client–server and using the Telnet client to communicate with our echo server?

5.4 In our example in Section 5.12 we verified that the first two segments of the connection termination are sent (the FIN from the server that is then ACKed by the client) by looking at the socket states using `netstat`. Are the final two segments exchanged (a FIN from client that is ACKed by the server)? If so, when, and if not, why?

5.5 What happens in the example outlined in Section 5.14 if between steps 2 and 3 we restart our server application on the server host?

5.6 To verify what we claimed happens with `SIGPIPE` in Section 5.13, modify Figure 5.4 as follows. Write a signal handler for `SIGPIPE` that just prints a message and returns. Establish this signal handler before calling `connect`. Change the server's port number to 13, the daytime server. When the connection is established, `sleep` for 2 seconds, `write` a few bytes to the socket, `sleep` for another 2 seconds, and `write` a few more bytes to the socket. Run the program. What happens?

5.7 What happens in Figure 5.15 if the IP address of the server host that is specified by the client in its call to `connect` is the IP address associated with the rightmost datalink on the server, instead of the IP address associated with the leftmost datalink on the server?

5.8 In our example output from Figure 5.20 when the client and server were on different endian systems, the example worked for small positive numbers, but not for small negative numbers. Why? (*Hint*: Draw a picture of the values exchanged across the socket, similar to Figure 3.8.)

5.9 In our example in Figures 5.19 and 5.20 can we solve the byte ordering problem by having the client convert the two arguments into network byte order using `htonl`, having the server then call `ntohl` on each argument before doing the addition, and then doing a similar conversion on the result?

5.10 What happens in Figures 5.19 and 5.20 if the client is on a Sparc that stores a `long` in 32 bits, but the server is on a Digital Alpha that stores a `long` in 64 bits? Does this change if the client and server are swapped between these two hosts?

5.11 In Figure 5.15 we say that the client IP address is chosen by IP, based on routing. What does this mean?

6

I/O Multiplexing: The `select` and `poll` Functions

6.1 Introduction

We saw in Section 5.12 that our TCP client is handling two inputs at the same time: standard input and a TCP socket. The problem we encountered was when the client was blocked in a call to `fgets` (on standard input) and the server process was killed. The server TCP correctly sends a FIN to the client TCP, but since the client process is blocked reading from standard input, it never sees the end-of-file until it reads from the socket (possibly much later in time). What we need is the capability to tell the kernel that we want to be notified if one or more I/O conditions are ready (i.e., input is ready to be read, or the descriptor is capable of taking more output). This capability is called *I/O multiplexing* and is provided by the `select` and `poll` functions. We also cover a newer Posix.1g variation of the former, called `pselect`.

I/O multiplexing is typically used in networking applications in the following scenarios:

- When a client is handling multiple descriptors (normally interactive input and a network socket), I/O multiplexing should be used. This is the scenario we described in the previous paragraph.
- It is possible, but rare, for a client to handle multiple sockets at the same time. We show an example of this using `select` in Section 15.5 in the context of a Web client.
- If a TCP server handles both a listening socket and its connected sockets, I/O multiplexing is normally used as we show in Section 6.8.
- If a server handles both TCP and UDP, I/O multiplexing is normally used. We show an example of this in Section 8.15.

- If a server handles multiple services and perhaps multiple protocols (e.g., the inetd daemon that we describe in Section 12.5), I/O multiplexing is normally used.

I/O multiplexing is not limited to network programming. Any nontrivial application often finds a need to use this technique.

6.2 I/O Models

Before describing select and poll we need to step back and look at the bigger picture, examining the basic differences in the five I/O models that are available to us under Unix:

- blocking I/O,
- nonblocking I/O,
- I/O multiplexing (select and poll),
- signal driven I/O (SIGIO), and
- asynchronous I/O (the Posix.1 aio_ functions).

You may want to skim this section on your first reading and then refer back to it as you encounter the different I/O models described in more detail in later chapters.

As we show in all the examples in this section, there are normally two distinct phases for an input operation:

1. waiting for the data to be ready, and
2. copying the data from the kernel to the process.

For an input operation on a socket the first step normally involves waiting for data to arrive on the network. When the packet arrives, it is copied into a buffer within the kernel. The second step is copying this data from the kernel's buffer into our application buffer.

Blocking I/O Model

The most prevalent model for I/O is the *blocking I/O model*, which we have used for all our examples so far in the text. By default, all sockets are blocking. Using a datagram socket for our examples we have the scenario shown in Figure 6.1.

We use UDP for this example, instead of TCP, because with UDP the concept of data being "ready" to read is simple: either an entire datagram has been received or not. With TCP it gets more complicated, as additional variables, such as the socket's low-water mark, come into play.

In the examples in this section we also refer to recvfrom as a system call, because we are differentiating between our application and the kernel. Regardless of how recvfrom is implemented (as a system call on a Berkeley-derived kernel, or as a function that invokes the getmsg system call on a System V kernel), there is normally a

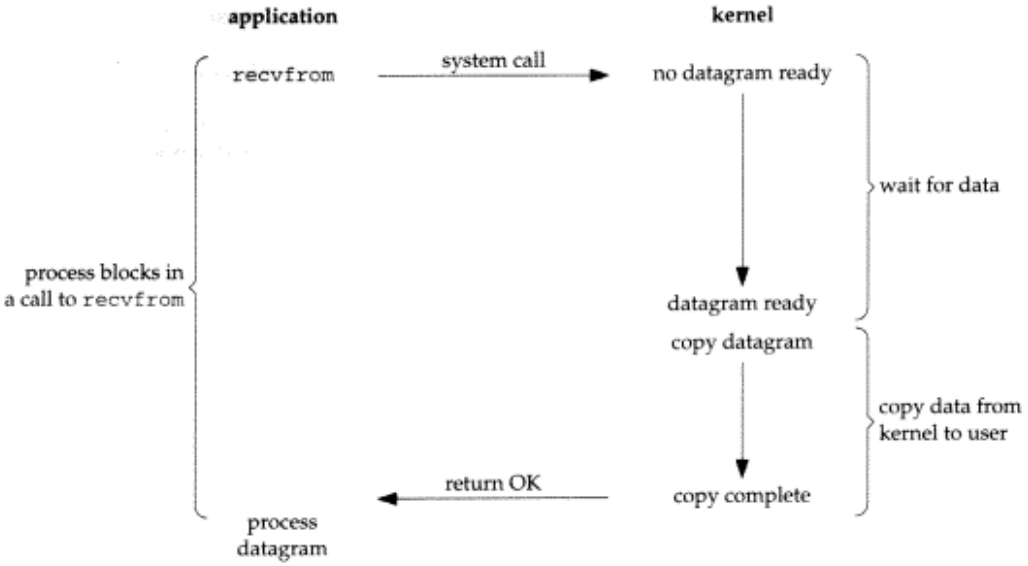

Figure 6.1 Blocking I/O model.

switch from running in the application to running in the kernel, followed at some time later by a return to the application.

In Figure 6.1 the process calls `recvfrom` and the system call does not return until the datagram arrives and is copied into our application buffer, or an error occurs. The most common error is the system call being interrupted by a signal, as we described in Section 5.9. We say that our process is *blocked* the entire time from when it calls `recvfrom` until it returns. When `recvfrom` returns OK, our application processes the datagram.

Nonblocking I/O Model

When we set a socket nonblocking, we are telling the kernel "when an I/O operation that I request cannot be completed without putting the process to sleep, do not put the process to sleep but return an error instead." We describe nonblocking I/O in Chapter 15 but show a summary in Figure 6.2 for the example we are considering.

The first three times that we call `recvfrom`, there is no data to return, so the kernel immediately returns an error of `EWOULDBLOCK` instead. The fourth time we call `recvfrom` a datagram is ready, it is copied into our application buffer, and `recvfrom` returns OK. We then process the data.

When an application sits in a loop calling `recvfrom` on a nonblocking descriptor like this, it is called *polling*. The application is continually polling the kernel to see if some operation is ready. This is often a waste of CPU time, but this model is occasionally encountered, normally on systems dedicated to one function.

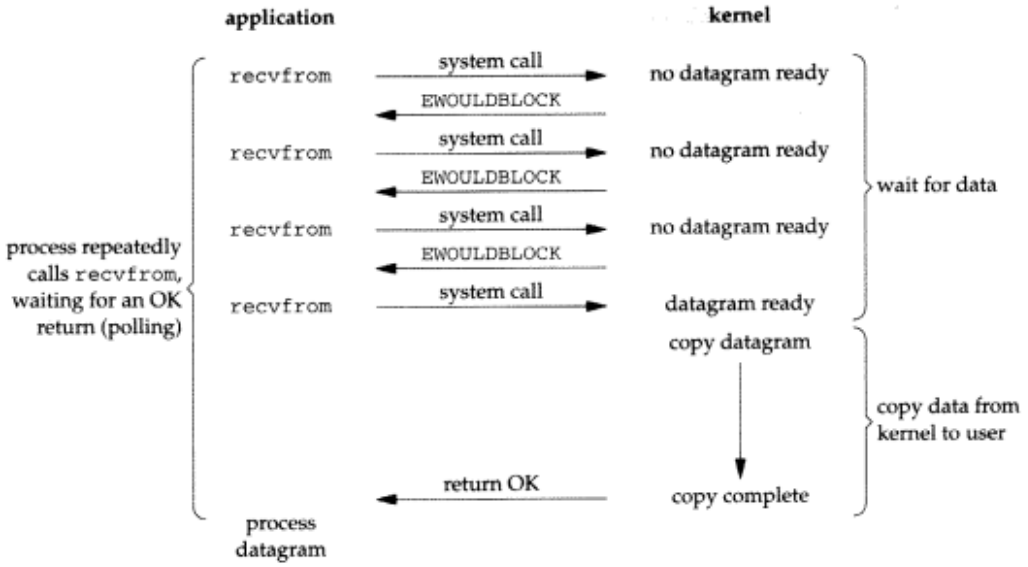

Figure 6.2 Nonblocking I/O model.

I/O Multiplexing Model

With *I/O multiplexing*, we call `select` or `poll` and block in one of these two system calls, instead of blocking in the actual I/O system call. Figure 6.3 is a summary of the I/O multiplexing model.

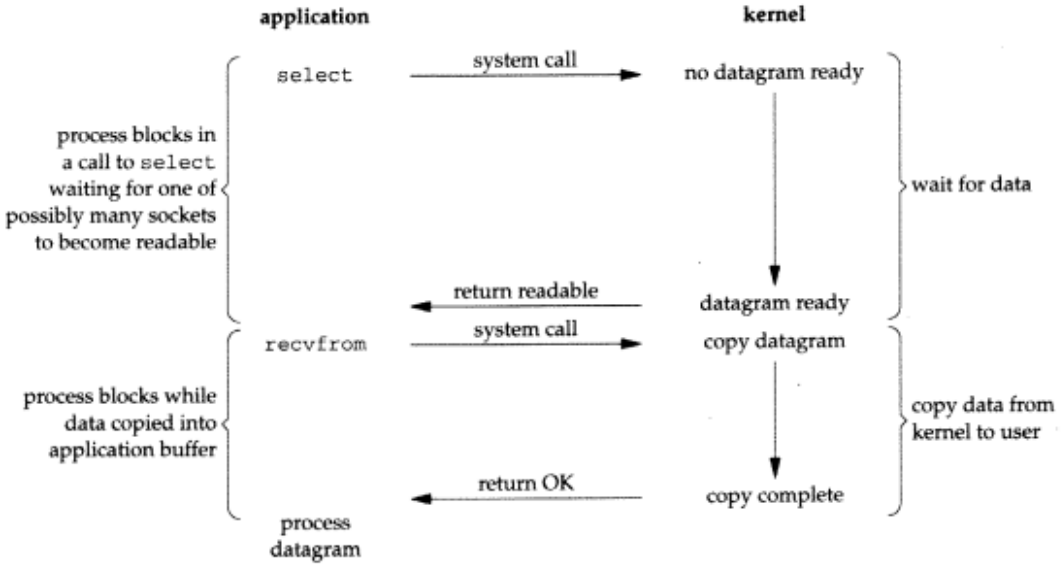

Figure 6.3 I/O multiplexing model.

We block in a call to `select`, waiting for the datagram socket to be readable. When `select` returns that the socket is readable, we then call `recvfrom` to copy the datagram into our application buffer.

Comparing Figure 6.3 to Figure 6.1, there does not appear to be any advantage, and in fact there is a slight disadvantage because using `select` requires two system calls instead of one. But the advantage in using `select`, which we will see later in this chapter, is that we can wait for more than one descriptor to be ready.

Signal Driven I/O Model

We can also use signals, telling the kernel to notify us with the `SIGIO` signal when the descriptor is ready. We call this *signal driven I/O* and show a summary of it in Figure 6.4.

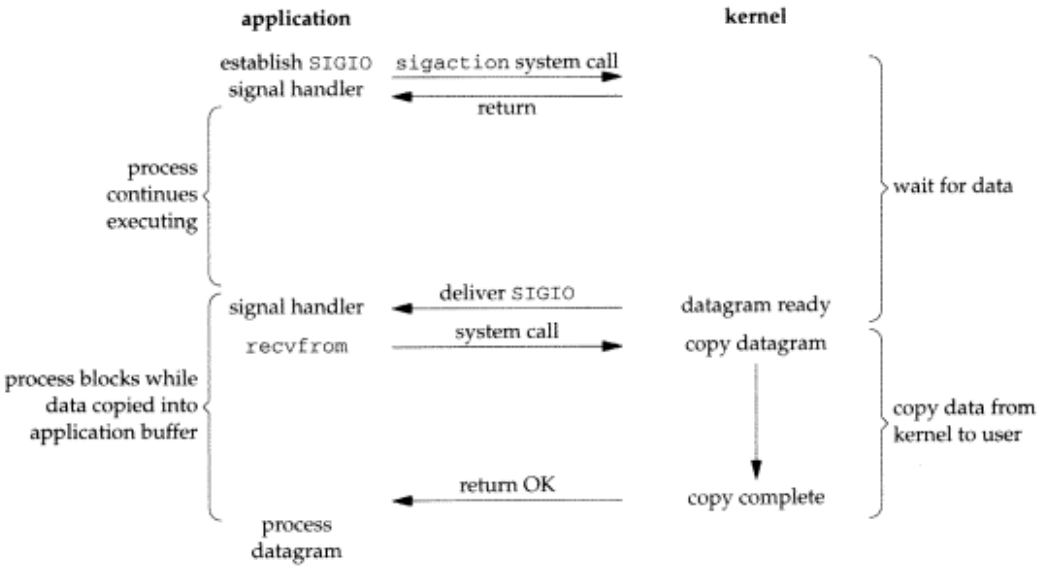

Figure 6.4 Signal Driven I/O model.

We first enable the socket for signal driven I/O (as we describe in Section 22.2) and install a signal handler using the `sigaction` system call. The return from this system call is immediate and our process continues; it is not blocked. When the datagram is ready to be read, the `SIGIO` signal is generated for our process. We can either read the datagram from the signal handler by calling `recvfrom` and then notify the main loop that the data is ready to be processed (this is what we do in Section 22.3), or we can notify the main loop and let it read the datagram.

Regardless of how we handle the signal, the advantage in this model is that we are not blocked while waiting for the datagram to arrive. The main loop can continue executing and just wait to be notified by the signal handler that either the data is ready to process or that the datagram is ready to be read.

Asynchronous I/O Model

Asynchronous I/O is new with the 1993 edition of Posix.1 (the "realtime" extensions). We tell the kernel to start the operation and to notify us when the entire operation (including the copy of the data from the kernel to our buffer) is complete. We do not discuss it in this book because it is not yet widespread. The main difference between this model and the signal driven I/O model in the previous section is that with signal driven I/O the kernel tells us when an I/O operation can be *initiated*, but with asynchronous I/O the kernel tells us when an I/O operation is *complete*. We show an example in Figure 6.5.

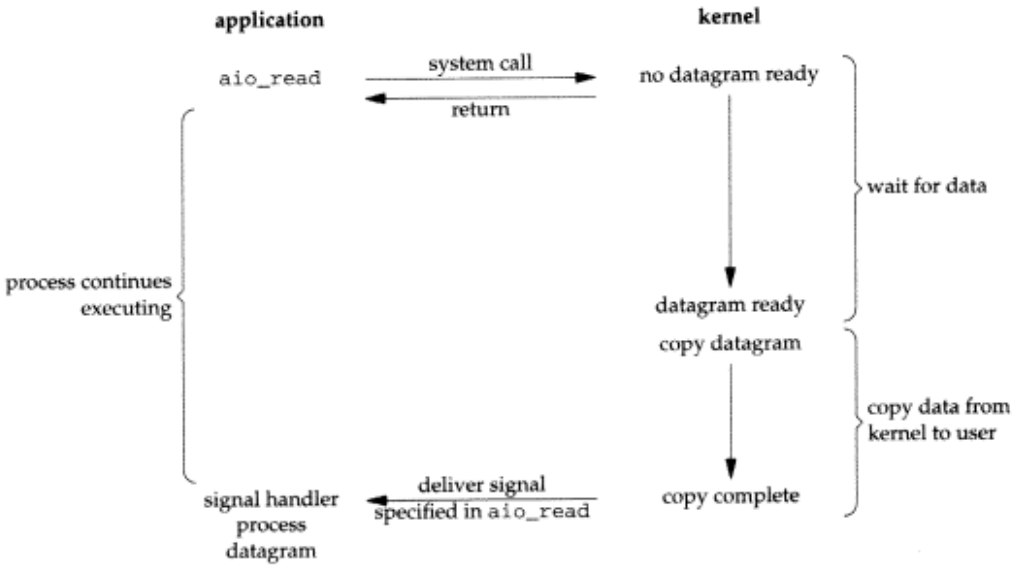

Figure 6.5 Asynchronous I/O model.

We call `aio_read` (the Posix asynchronous I/O functions begin with `aio_` or `lio_`) and pass the kernel the descriptor, buffer pointer, buffer size (the same three arguments for `read`), file offset (similar to `lseek`), and how to notify us when the entire operation is complete. This system call returns immediately and our process is not blocked waiting for the I/O to complete. We assume in this example that we ask the kernel to generate some signal when the operation is complete. This signal is not generated until the data has been copied into our application buffer, which is different from the signal driven I/O model.

> As of this writing, few systems support Posix.1 asynchronous I/O. We are not certain, for example, if systems will support it for sockets. Our use of it here is as an example to compare against the signal driven I/O model.

Comparison of the I/O Models

Figure 6.6 is a comparison of the five different I/O models. This shows that the main difference between the first four models is the first phase, as the second phase in the first four models is the same: the process is blocked in a call to `recvfrom` while the data is copied from the kernel to the caller's buffer. Asynchronous I/O, however, handles both phases and is different from the first four.

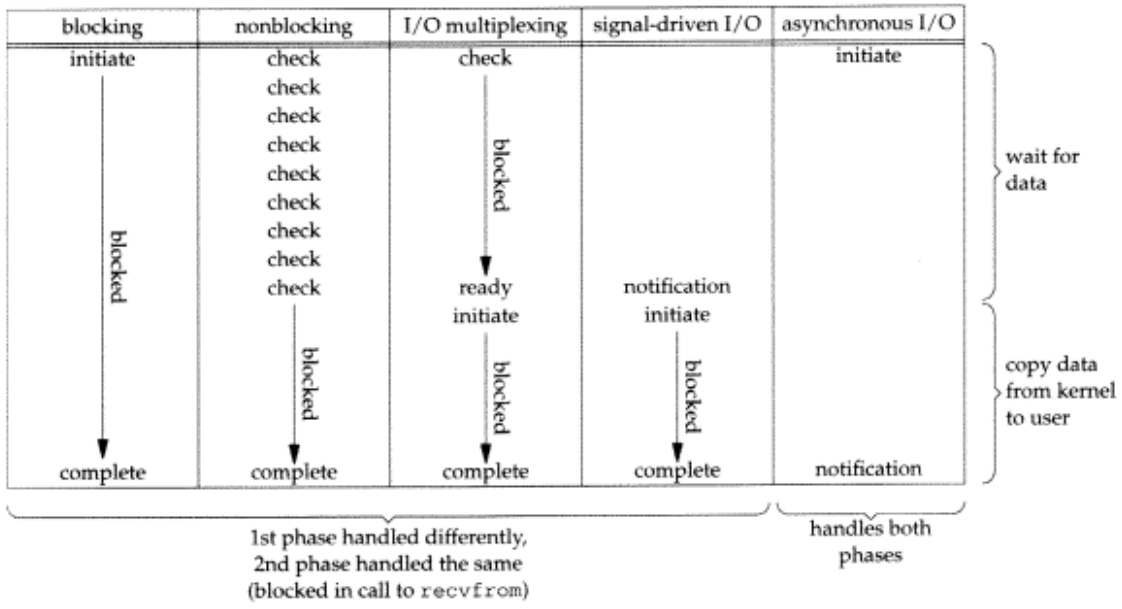

Figure 6.6 Comparison of the five I/O models.

Synchronous I/O versus Asynchronous I/O

Posix.1 defines these two terms as follows:

- A *synchronous I/O operation* causes the requesting process to be blocked until that I/O operation completes.
- An *asynchronous I/O operation* does not cause the requesting process to be blocked.

Using these definitions the first four I/O models—blocking, nonblocking, I/O multiplexing, and signal-driven I/O—are all synchronous because the actual I/O operation (`recvfrom`) blocks the process. Only the asynchronous I/O model matches the asynchronous I/O definition.

6.3 select Function

This function allows the process to instruct the kernel to wait for any one of multiple events to occur and to wake up the process only when one or more of these events occurs or when a specified amount of time has passed.

As an example, we can call `select` and tell the kernel to return only when

- any of the descriptors in the set {1, 4, 5} are ready for reading, or
- any of the descriptors in the set {2, 7} are ready for writing, or
- any of the descriptors in the set {1, 4} have an exception condition pending, or
- after 10.2 seconds have elapsed.

That is, we tell the kernel what descriptors we are interested in (for reading, writing, or an exception condition) and how long to wait. The descriptors in which we are interested are not restricted to sockets: any descriptor can be tested using `select`.

> Berkeley-derived implementations have always allowed I/O multiplexing with any descriptor. SVR3 originally limited I/O multiplexing to descriptors that were streams devices (Chapter 33), but this limitation was removed with SVR4.

```
#include <sys/select.h>
#include <sys/time.h>

int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
           const struct timeval *timeout);
```

Returns: positive count of ready descriptors, 0 on timeout, –1 on error

We start our description of this function with its final argument, which tells the kernel how long to wait for one of the specified descriptors to become ready. A `timeval` structure specifies the number of seconds and microseconds.

```
struct timeval {
  long    tv_sec;     /* seconds */
  long    tv_usec;    /* microseconds */
};
```

There are three possibilities.

1. Wait forever: return only when one of the specified descriptors is ready for I/O. For this, we specify the *timeout* argument as a null pointer.
2. Wait up to a fixed amount of time: return when one of the specified descriptors is ready for I/O, but do not wait beyond the number of seconds and microseconds specified in the `timeval` structure pointed to by the *timeout* argument.
3. Do not wait at all: return immediately after checking the descriptors. This is called *polling*. To specify this, the *timeout* argument must point to a `timeval`

structure, and the timer value (the number of seconds and microseconds specified by the structure) must be 0.

The wait in the first two scenarios is normally interrupted if the process catches a signal and returns from the signal handler.

> Berkeley-derived kernels never automatically restart `select` (p. 527 of TCPv2), while SVR4 will if the `SA_RESTART` flag is specified when the signal handler is installed. This means that for portability we must be prepared for `select` to return an error of `EINTR` if we are catching signals.

Although the `timeval` structure lets us specify a resolution in microseconds, the actual resolution supported by the kernel is often more coarse. For example, many Unix kernels round the timeout value up to a multiple of 10 ms. There is also a scheduling latency involved, meaning it takes some time after the timer expires before the kernel schedules this process to run.

The `const` qualifier on the *timeout* argument means it is not modified by `select` on return. For example, if we specify a time limit of 10 seconds, and `select` returns before the timer expires, with one or more of the descriptors ready or with an error of `EINTR`, the `timeval` structure is not updated with the number of seconds remaining when the function returns. If we wish to know this value, we must obtain the system time before calling `select`, and then again when it returns, and subtract the two.

> Current Linux systems modify the `timeval` structure. Therefore for portability, assume the `timeval` structure is undefined upon return, and initialize it before each call to `select`. Posix.1g specifies the `const` qualifier.

The three middle arguments *readset*, *writeset*, and *exceptset* specify the descriptors that we want the kernel to test for reading, writing, and exception conditions. There are only two exception conditions currently supported.

1. The arrival of out-of-band data for a socket. We describe this in more detail in Chapter 21.
2. The presence of control status information to be read from the master side of a pseudo terminal that has been put into packet mode. We do not talk about pseudo terminals in this volume.

A design problem is how to specify one or more descriptor values for each of these three arguments. `select` uses *descriptor sets*, typically an array of integers, with each bit in each integer corresponding to a descriptor. For example, using 32-bit integers, the first element of the array corresponds to descriptors 0 through 31, the second element of the array corresponds to descriptors 32 through 63, and so on. All the implementation details are irrelevant to the application and are hidden in the `fd_set` datatype and the following four macros:

```
void FD_ZERO(fd_set *fdset);          /* clear all bits in fdset */
void FD_SET(int fd, fd_set *fdset);   /* turn on the bit for fd in fdset */
void FD_CLR(int fd, fd_set *fdset);   /* turn off the bit for fd in fdset */
int  FD_ISSET(int fd, fd_set *fdset); /* is the bit for fd on in fdset ? */
```

We allocate a descriptor set of the `fd_set` datatype, we set and test the bits in the set using these macros, and we can also assign it to another descriptor set across an equals sign in C.

> What we are describing, an array of integers using one bit per descriptor, is just one possible way to implement `select`. Nevertheless, it is common to refer to the individual descriptors within a descriptor set as *bits*, as in "turn on the bit for the listening descriptor in the read set."
>
> We will see in Section 6.10 that the `poll` function uses a completely different representation: a variable-length array of structures with one structure per descriptor.

For example, to define a variable of type `fd_set` and then turn on the bits for descriptors 1, 4, and 5, we write

```
fd_set  rset;

FD_ZERO(&rset);       /* initialize the set: all bits off */
FD_SET(1, &rset);     /* turn on bit for fd 1 */
FD_SET(4, &rset);     /* turn on bit for fd 4 */
FD_SET(5, &rset);     /* turn on bit for fd 5 */
```

It is important to initialize the set, since unpredictable results can occur if the set is allocated as an automatic variable and not initialized.

Any of the middle three arguments to `select`, *readset*, *writeset*, or *exceptset*, can be specified as a null pointer, if we are not interested in that condition. Indeed, if all three pointers are null, then we have a higher precision timer than the normal Unix `sleep` function (which sleeps for multiples of a second). The `poll` function provides similar functionality. Figures C.9 and C.10 of APUE show a `sleep_us` function implemented using both `select` and `poll` that sleeps for multiples of a microsecond.

The *maxfdp1* argument specifies the number of descriptors to be tested. Its value is the maximum descriptor to be tested, plus one (hence our name of *maxfdp1*). The descriptors 0, 1, 2, up through and including *maxfdp1–1* are tested.

The constant `FD_SETSIZE`, defined by including `<sys/select.h>`, is the number of descriptors in the `fd_set` datatype. Its value is often 1024, but few programs use that many descriptors. The *maxfdp1* argument forces us to calculate the largest descriptor that we are interested in and then tell the kernel this value. For example, given the previous code that turns on the indicators for descriptors 1, 4, and 5, *maxfdp1* value is 6. The reason it is 6 and not 5 is that we are specifying the number of descriptors, not the largest value, and descriptors start at 0.

> The reason this argument exists along with the burden of calculating its value is purely for efficiency. Although each `fd_set` has room for many descriptors, typically 1024, this is much more than the number used by a typical process. The kernel gains efficiency by not copying unneeded portions of the descriptor set between the process and the kernel, and by not testing bits that are always 0 (Section 16.13 of TCPv2).

`select` modifies the descriptor sets pointed to by the *readset*, *writeset*, and *exceptset* pointers. These three arguments are value–result arguments. When we call the function, we specify the values of the descriptors that we are interested in and on return the result indicates which descriptors are ready. We use the `FD_ISSET` macro on return to

test a specific descriptor in an fd_set structure. Any descriptor that is not ready on return will have its corresponding bit cleared in the descriptor set. To handle this we turn on all the bits in which we are interested in all the descriptor sets each time we call select.

> The two most common programming errors when using select are to forget to add one to the largest descriptor number and to forget that the descriptor sets are value–result. The second error results in select being called with a bit set to 0 in the descriptor set, when we think that bit is 1. The author also wasted 2 hours debugging an example for this text that uses select by forgetting to add one to the first argument.

The return value from this function indicates the total number of bits that are ready across all the descriptor sets. If the timer value expires before any of the descriptors are ready, a value of 0 is returned. A return value of –1 indicates an error (which can happen, for example, if the function is interrupted by a caught signal).

> Early releases of SVR4 had a bug in their implementation of select: if the same bit was on in multiple sets, say a descriptor was ready for both reading and writing, it was counted only once. Current releases fix this bug.

Under What Conditions Is a Descriptor Ready?

We have been talking about waiting for a descriptor to become ready for I/O (reading or writing) or to have an exception condition pending on it (out-of-band data). While readability and writability are obvious for descriptors such as regular files, we must be more specific about the conditions that cause select to return "ready" for sockets (Figure 16.52 of TCPv2).

1. A socket is ready for reading if any of the following four conditions is true:
   a. The number of bytes of data in the socket receive buffer is greater than or equal to the current size of the low-water mark for the socket receive buffer. A read operation on the socket will not block and will return a value greater than 0 (i.e., the data that is ready to be read). We can set this low-water mark using the SO_RCVLOWAT socket option. It defaults to 1 for TCP and UDP sockets.
   b. The read-half of the connection is closed (i.e., a TCP connection that has received a FIN). A read operation on the socket will not block and will return 0 (i.e., end-of-file).
   c. The socket is a listening socket and the number of completed connections is nonzero. An accept on the listening socket will normally not block, although we describe a timing condition in Section 15.6 under which the accept can block.
   d. A socket error is pending. A read operation on the socket will not block and will return an error (–1) with errno set to the specific error condition. These *pending errors* can also be fetched and cleared by calling getsockopt specifying the SO_ERROR socket option.

2. A socket is ready for writing if any of the following three conditions is true:
   a. The number of bytes of available space in the socket send buffer is greater than or equal to the current size of the low-water mark for the socket send buffer *and* either (i) the socket is connected, or (ii) the socket does not require a connection (e.g., UDP). This means that if we set the socket nonblocking (Chapter 15), a write operation will not block and will return a positive value (e.g., the number of bytes accepted by the transport layer). We can set this low-water mark using the SO_SNDLOWAT socket option. This low-water mark normally defaults to 2048 for TCP and UDP sockets.
   b. The write-half of the connection is closed. A write operation on the socket will generate SIGPIPE (Section 5.12).
   c. A socket error is pending. A write operation on the socket will not block and will return an error (−1) with errno set to the specific error condition. These *pending errors* can also be fetched and cleared by calling getsockopt with the SO_ERROR socket option.
3. A socket has an exception condition pending if there exists out-of-band data for the socket or the socket is still at the out-of-band mark. (We describe out-of-band data in Chapter 21.)

> Our definitions of "readable" and "writable" are taken directly from the kernel's soreadable and sowriteable macros on pp. 530–531 of TCPv2. Similarly our definition of the "exception condition" for a socket is from the soo_select function on these same pages.

Notice that when an error occurs on a socket it is marked as both readable and writable by select.

The purpose of the receive and send low-water marks is to give the application control over how much data must be available for reading or how much space must be available for writing before select returns readable or writable. For example, if we know that our application has nothing productive to do unless at least 64 bytes of data are present, we can set the receive low-water mark to 64 to prevent select from waking us up if less than 64 bytes are ready for reading.

As long as the send low-water mark for a UDP socket is less than the send buffer size (which should always be the default relationship), the UDP socket is always writable, since a connection is not required.

Figure 6.7 summarizes the conditions just described that cause a socket to be ready for select.

Maximum Number of Descriptors for select?

We said earlier that most applications do not use lots of descriptors. It is rare, for example, to find an application that uses hundreds of descriptors. But these applications do exist, and they often use select to multiplex the descriptors. When select was originally designed, the operating system normally had an upper limit on the maximum number of descriptors per process (the 4.2BSD limit was 31), and select just used this same limit. But current versions of Unix allow for an unlimited number of descriptors

Condition	readable?	writable?	exception?
data to read	•		
read-half of the connection closed	•		
new connection ready for listening socket	•		
space available for writing		•	
write-half of the connection closed		•	
pending error	•	•	
TCP out-of-band data			•

Figure 6.7 Summary of conditions that cause a socket to be ready for `select`.

per process (often limited only by the amount of memory and any administrative limits), so the question is: how does this affect `select`?

Many implementations have declarations similar to the following, which are taken from the 4.4BSD `<sys/types.h>` header:

```
/*
 * Select uses bitmasks of file descriptors in longs.  These macros
 * manipulate such bit fields (the filesystem macros use chars).
 * FD_SETSIZE may be defined by the user, but the default here should
 * be enough for most uses.
 */
#ifndef FD_SETSIZE
#define FD_SETSIZE      256
#endif
```

This makes us think that we can just `#define FD_SETSIZE` to some larger value before including this header to increase the size of the descriptor sets used by `select`. Unfortunately, this normally does not work.

> To see what is wrong, notice that Figure 16.53 of TCPv2 declares three descriptor sets within the kernel and also uses the kernel's definition of `FD_SETSIZE` as the upper limit. The only way to increase the size of the descriptor sets is to increase the value of `FD_SETSIZE` and then recompile the kernel. Changing the value without recompiling the kernel is inadequate.

Some vendors are changing their implementation of `select` to allow the process to define `FD_SETSIZE` to a larger than default value. BSD/OS has changed the kernel implementation to allow larger descriptor sets, and it also provides four new `FD_`*xxx* macros to dynamically allocate and manipulate these larger sets. From a portability standpoint, however, beware of using large descriptor sets.

6.4 `str_cli` Function (Revisited)

We can now rewrite our `str_cli` function from Section 5.5, this time using `select`, so we are notified as soon as the server process terminates. The problem with that earlier version was that we could be blocked in the call to `fgets` when something happened on the socket. Our new version blocks in a call to `select` instead, waiting for either

standard input or the socket to be readable. Figure 6.8 shows the various conditions that are handled by our call to `select`.

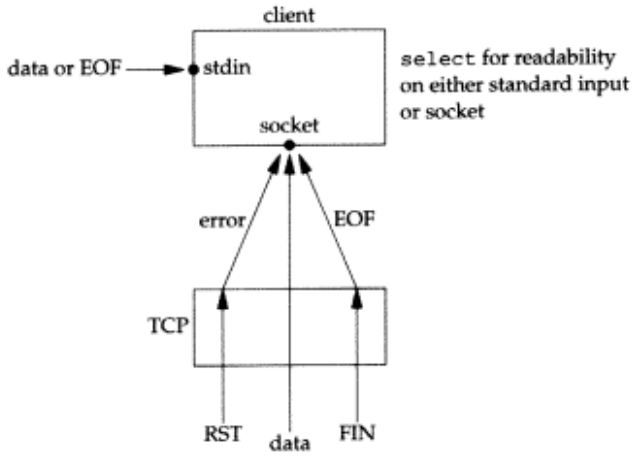

Figure 6.8 Conditions handled by `select` in `str_cli`.

Three conditions are handled with the socket.

1. If the peer TCP sends data, the socket becomes readable and `read` returns greater than 0 (i.e., the number of bytes of data).
2. If the peer TCP sends a FIN (the peer process terminates), the socket becomes readable and `read` returns 0 (end-of-file).
3. If the peer TCP sends an RST (the peer host has crashed and rebooted), the socket becomes readable and `read` returns –1 and `errno` contains the specific error code.

Figure 6.9 shows the source code for this new version.

Call `select`

8-13 We only need one descriptor set—to check for readability. This set is initialized by `FD_ZERO` and then two bits are turned on using `FD_SET`: the bit corresponding to the standard I/O file pointer `fp` and the bit corresponding to the socket `sockfd`. The function `fileno` converts a standard I/O file pointer into its corresponding descriptor. `select` (and `poll`) work only with descriptors.

`select` is called after calculating the maximum of the two descriptors. In the call the write-set pointer and the exception-set pointer are both null pointers. The final argument (the time limit) is also a null pointer since we want the call to block until something is ready.

Handle readable socket

14-18 If, on return from `select`, the socket is readable, the echoed line is read with `readline` and output by `fputs`.

select/strcliselect01.c

```
 1 #include    "unp.h"

 2 void
 3 str_cli(FILE *fp, int sockfd)
 4 {
 5     int     maxfdp1;
 6     fd_set  rset;
 7     char    sendline[MAXLINE], recvline[MAXLINE];

 8     FD_ZERO(&rset);
 9     for ( ; ; ) {
10         FD_SET(fileno(fp), &rset);
11         FD_SET(sockfd, &rset);
12         maxfdp1 = max(fileno(fp), sockfd) + 1;
13         Select(maxfdp1, &rset, NULL, NULL, NULL);

14         if (FD_ISSET(sockfd, &rset)) {  /* socket is readable */
15             if (Readline(sockfd, recvline, MAXLINE) == 0)
16                 err_quit("str_cli: server terminated prematurely");
17             Fputs(recvline, stdout);
18         }
19         if (FD_ISSET(fileno(fp), &rset)) {  /* input is readable */
20             if (Fgets(sendline, MAXLINE, fp) == NULL)
21                 return;          /* all done */
22             Writen(sockfd, sendline, strlen(sendline));
23         }
24     }
25 }
```

select/strcliselect01.c

Figure 6.9 Implementation of `str_cli` function using `select` (improved in Figure 6.13).

Handle readable input

19–23 If the standard input is readable, a line is read by `fgets` and written to the socket using `writen`.

Notice that the same four I/O functions are used as in Figure 5.5: `fgets`, `writen`, `readline`, and `fputs`, but the order of flow within the function has changed. Instead of the function flow being driven by the call to `fgets`, it is now driven by the call to `select`. With only a few additional lines of code in Figure 6.9, compared to Figure 5.5, we have added greatly to the robustness of our client.

6.5 Batch Input

Unfortunately, our `str_cli` function is still not correct. First let's go back to our original version, Figure 5.5. It operates in a stop-and-wait mode, which is fine for interactive use: it sends a line to the server and then waits for the reply. This amount of time is one RTT (round-trip time) plus the server's processing time (which is close to 0 for a simple echo server). We can therefore estimate how long it will take for a given number of lines to be echoed, if we know the RTT between the client and server.

The Ping program is an easy way to measure RTTs. If we run Ping to the host connix.com from our host solaris, the average RTT over 30 measurements is 175 ms. Page 89 of TCPv1 shows that these Ping measurements are for an IP datagram whose length is 84 bytes. If we take the first 2000 lines of the Solaris 2.5 termcap file, the resulting file size is 98,349 bytes, for an average of 49 bytes per line. If we add the sizes of the IP header (20 bytes) and the TCP header (20), the average TCP segment will be about 89 bytes, nearly the same as the Ping packet sizes. We can therefore estimate that the total clock time will be around 350 seconds for 2000 lines ($2000 \times 0.175 sec$). If we run our TCP echo client from Chapter 5, the actual time is about 354 seconds, which is very close to our estimate.

If we consider the network between the client and server as a full-duplex pipe, with requests going from the client to server, and replies in the reverse direction, then Figure 6.10 shows our stop-and-wait mode.

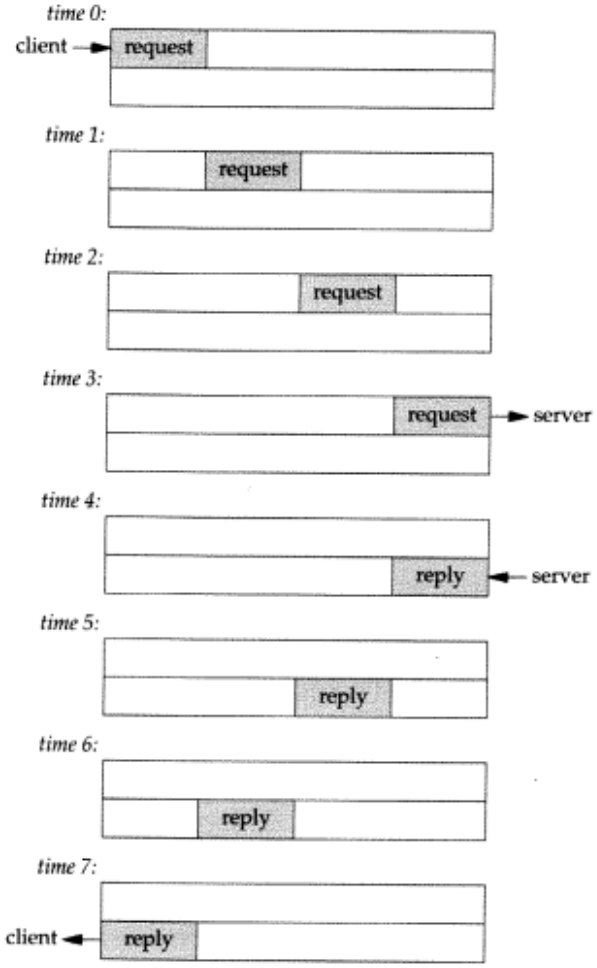

Figure 6.10 Time line of stop-and-wait mode: interactive input.

The request is sent by the client at time 0 and we assume an RTT of 8 units of time. The reply sent at time 4 is received at time 7. We also assume that there is no server processing time and that the size of the request is the same as the reply. We show only the data packets between the client and server, ignoring the TCP acknowledgments that are also going across the network.

But since there is a delay between sending a packet and that packet arriving at the other end of the pipe, and since the pipe is full-duplex, in this example we are only using one-eighth of the pipe capacity. This stop-and-wait mode is fine for interactive input, but since our client reads from standard input and writes to standard output, and since it is trivial under the Unix shells to redirect the input and output, we can easily run our client in a batch mode. When we redirect the input and output, however, the resulting output file is always smaller than the input file (and they should be identical for an echo server).

To see what's happening, realize that in a batch mode we can keep sending requests as fast as the network can accept them. The server processes them and sends back the replies at the same rate. This leads to the full pipe at time 7, as shown in Figure 6.11.

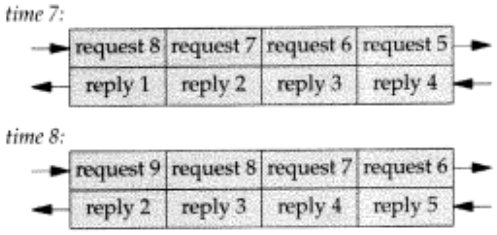

Figure 6.11 Filling the pipe between the client and server: batch mode.

Here we assume that after sending the first request, we immediately send another, and another. We also assume that we can keep sending them as fast as the network can accept them, along with processing the replies as fast as the network supplies them.

> There are numerous subtleties dealing with TCP's bulk data flow that we are ignoring here, such as its slow start algorithm, which limits the rate at which data is sent on a new or idle connection, and the returning ACKs. These are all covered in Chapter 20 of TCPv1.

To see the problem with our revised `str_cli` function in Figure 6.9, assume that the input file contains only nine lines. The last line is sent at time 8, as shown in Figure 6.11. But we cannot close the connection after writing this request, because there are still other requests and replies in the pipe. The cause of the problem is our handling of an end-of-file on input: the function returns to the `main` function, which then terminates. But in a batch mode, an end-of-file on the input does not imply that we have finished reading from the socket: there might still be requests on the way to the server, or replies on the way back from the server.

What we need is a way to close one-half of the TCP connection. That is, we want to send a FIN to the server, telling it we have finished sending data, but leave the socket descriptor open for reading. This is done with the `shutdown` function, described in the next section.

6.6 shutdown Function

The normal way to terminate a network connection is to call the close function. But there are two limitations with close that can be avoided with shutdown.

1. close decrements the descriptor's reference count and closes the socket only if the count reaches 0. We talked about this in Section 4.8. With shutdown we can initiate TCP's normal connection termination sequence (the four segments beginning with a FIN in Figure 2.5) regardless of the reference count.
2. close terminates both directions of data transfer, reading and writing. Since a TCP connection is full-duplex, there are times when we want to tell the other end that we have finished sending, even though that end might have more data to send us. This is the scenario we encountered in the previous section with batch input to our str_cli function. Figure 6.12 shows the typical function calls in this scenario.

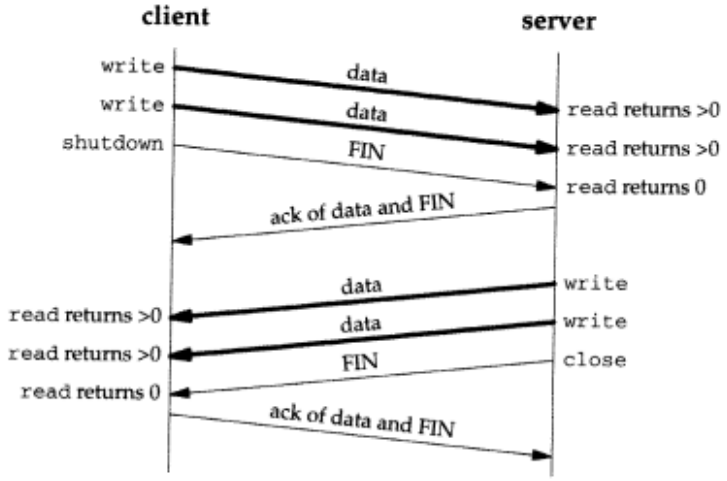

Figure 6.12 Calling shutdown to close half of a TCP connection.

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

int shutdown(int sockfd, int howto);
```

Returns: 0 if OK, –1 on error

The action of the function depends on the value of the *howto* argument.

SHUT_RD The read-half of the connection is closed: no more data can be received on the socket and any data currently in the socket receive

buffer is discarded. The process can no longer issue any of the read functions on the socket. Any data received after this call for a TCP socket is acknowledged and then silently discarded.

> By default, everything written to a routing socket (Chapter 17) loops back as possible input to all routing sockets on the host. Some programs call `shutdown` with a second argument of `SHUT_RD` to prevent the loopback copy. An alternative way to prevent this loopback copy is to clear the `SO_USELOOPBACK` socket option.

`SHUT_WR` The write-half of the connection is closed. In the case of TCP, this is called a *half-close* (Section 18.5 of TCPv1). Any data currently in the socket send buffer will be sent, followed by TCP's normal connection termination sequence. As we mentioned earlier, this closing of the write-half is done regardless whether or not the socket descriptor's reference count is currently greater than 0. The process can no longer issue any of the write functions on the socket.

`SHUT_RDWR` The read-half and the write-half of the connection are both closed. This is equivalent to calling `shutdown` twice: first with `SHUT_RD` and then with `SHUT_WR`.

Figure 7.10 summarizes the different possibilities available to the process by calling `shutdown` and `close`. The operation of `close` depends on the value of the `SO_LINGER` socket option.

> The three `SHUT_xxx` names are new with Posix.1g. Typical values for the *howto* argument that you will encounter will be 0 (close the read-half), 1 (close the write-half), and 2 (close the read-half and the write-half).

6.7 `str_cli` Function (Revisited Again)

Figure 6.13 shows our revised (and correct) version of the `str_cli` function. This version uses `select` and `shutdown`. The former notifies us as soon as the server closes its end of the connection and the latter lets us handle batch input correctly.

5–8 `stdineof` is a new flag that is initialized to 0. As long as this flag is 0, each time around the main loop we `select` on standard input for readability.

16–24 When we read the end-of-file on the socket, if we have already encountered an end-of-file on standard input, this is the normal termination and the function returns. But if we have not yet encountered an end-of-file on standard input, the server process has prematurely terminated.

25–33 When we encounter the end-of-file on standard input, our new flag `stdineof` is set and we call `shutdown` with a second argument of `SHUT_WR` to send the FIN.

If we measure our TCP client using the `str_cli` function from Figure 6.13 using the same 2000-line file, the clock time is now about 12.3 seconds, about 30 times faster than the stop-and-wait version.

select/strcliselect02.c

```
 1 #include    "unp.h"
 2 void
 3 str_cli(FILE *fp, int sockfd)
 4 {
 5     int     maxfdp1, stdineof;
 6     fd_set  rset;
 7     char    sendline[MAXLINE], recvline[MAXLINE];
 8     stdineof = 0;
 9     FD_ZERO(&rset);
10     for ( ; ; ) {
11         if (stdineof == 0)
12             FD_SET(fileno(fp), &rset);
13         FD_SET(sockfd, &rset);
14         maxfdp1 = max(fileno(fp), sockfd) + 1;
15         Select(maxfdp1, &rset, NULL, NULL, NULL);
16         if (FD_ISSET(sockfd, &rset)) {  /* socket is readable */
17             if (Readline(sockfd, recvline, MAXLINE) == 0) {
18                 if (stdineof == 1)
19                     return;     /* normal termination */
20                 else
21                     err_quit("str_cli: server terminated prematurely");
22             }
23             Fputs(recvline, stdout);
24         }
25         if (FD_ISSET(fileno(fp), &rset)) {  /* input is readable */
26             if (Fgets(sendline, MAXLINE, fp) == NULL) {
27                 stdineof = 1;
28                 Shutdown(sockfd, SHUT_WR);  /* send FIN */
29                 FD_CLR(fileno(fp), &rset);
30                 continue;
31             }
32             Writen(sockfd, sendline, strlen(sendline));
33         }
34     }
35 }
```

select/strcliselect02.c

Figure 6.13 str_cli function using select that handles end-of-file correctly.

We are not finished with our str_cli function. We develop a version using nonblocking I/O in Section 15.2, and a version using threads in Section 23.3.

6.8 TCP Echo Server (Revisited)

We can revisit our TCP echo server from Sections 5.2 and 5.3 and rewrite the server as a single process that uses select to handle any number of clients, instead of forking one child per client. Before showing the code, let's look at the data structures that we will use to keep track of the clients. Figure 6.14 shows the state of the server before the first client has established a connection.

Figure 6.14 TCP server before first client has established a connection.

The server has a single listening descriptor, which we show as a bullet.

The server maintains only a read descriptor set, which we show in Figure 6.15. We assume that the server is started in the foreground, so descriptors 0, 1, and 2 are set to standard input, output, and error. Therefore the first available descriptor for the listening socket is 3. We also show an array of integers named `client` that contains the connected socket descriptor for each client. All elements in this array are initialized to –1.

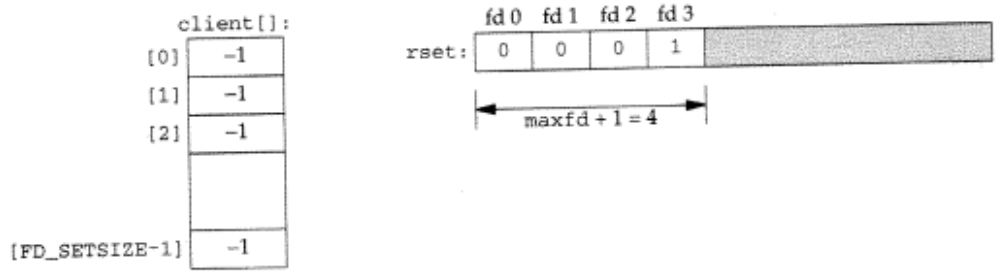

Figure 6.15 Data structures for TCP server with just listening socket.

The only nonzero entry in the descriptor set is the entry for the listening socket and the first argument to `select` will be 4.

When the first client establishes a connection with our server, the listening descriptor becomes readable and our server calls `accept`. The new connected descriptor returned by `accept` will be 4, given the assumptions of this example. Figure 6.16 shows the connection from the client to the server.

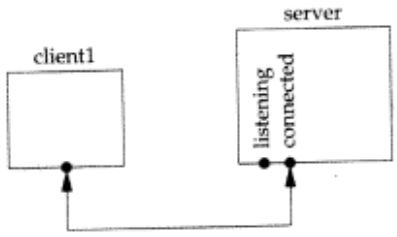

Figure 6.16 TCP server after first client establishes connection.

From this point on our server must remember the new connected socket in its `client` array, and the connected socket must be added to the descriptor set. These updated data structures are shown in Figure 6.17.

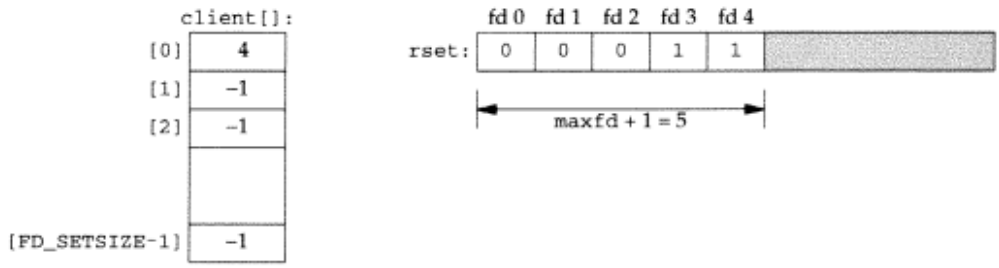

Figure 6.17 Data structures after first client connection is established.

Some time later a second client establishes a connection and we have the scenario shown in Figure 6.18.

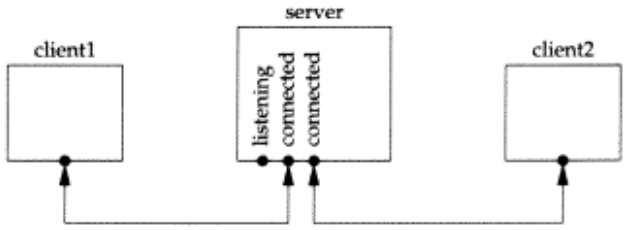

Figure 6.18 TCP server after second client connection is established.

The new connected socket (which we assume is 5) must be remembered, giving the data structures shown in Figure 6.19.

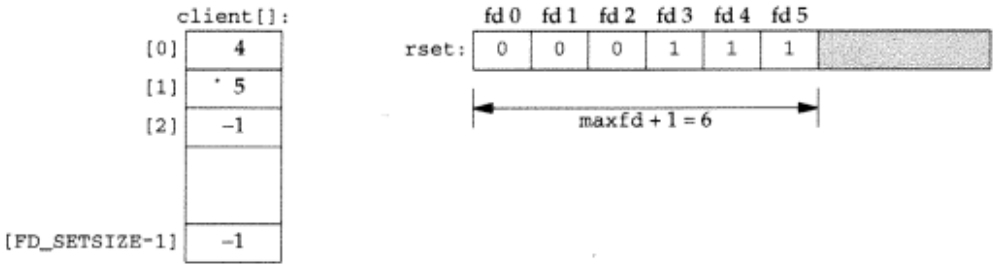

Figure 6.19 Data structures after second client connection is established.

Next we assume the first client terminates its connection. The client TCP sends a FIN, which makes descriptor 4 in the server readable. When our server reads this connected socket, `readline` returns 0. We then close this socket and update our data structures accordingly. The value of `client[0]` is set to –1 and descriptor 4 in the descriptor set is set to 0. This is shown in Figure 6.20. Notice that the value of `maxfd` does not change.

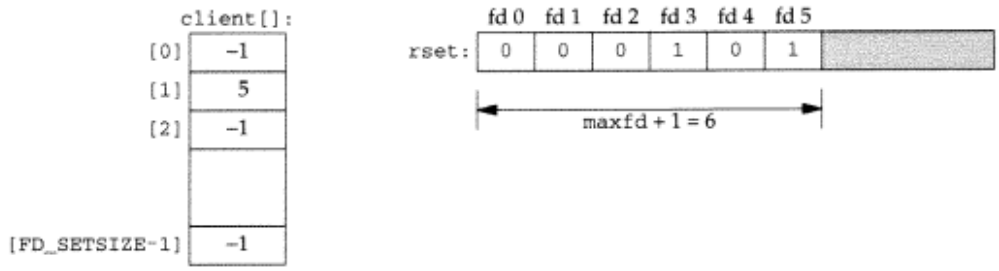

Figure 6.20 Data structures after first client terminates its connection.

In summary, as clients arrive we record their connected socket descriptor in the first available entry in the `client` array (i.e., the first entry with a value of –1). We must also add the connected socket to the read descriptor set. The variable `maxi` is the highest index in the `client` array that is currently in use and the variable `maxfd` (plus one) is the current value of the first argument to `select`. The only limit on the number of clients that this server can handle is the minimum of the two values `FD_SETSIZE` and the maximum number of descriptors allowed for this process by the kernel (which we talked about at the end of Section 6.3).

Figure 6.21 shows the first half of this version of the server.

tcpcliserv/tcpservselect01.c

```
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     i, maxi, maxfd, listenfd, connfd, sockfd;
 6     int     nready, client[FD_SETSIZE];
 7     ssize_t n;
 8     fd_set  rset, allset;
 9     char    line[MAXLINE];
10     socklen_t clilen;
11     struct sockaddr_in cliaddr, servaddr;

12     listenfd = Socket(AF_INET, SOCK_STREAM, 0);

13     bzero(&servaddr, sizeof(servaddr));
14     servaddr.sin_family = AF_INET;
15     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
16     servaddr.sin_port = htons(SERV_PORT);

17     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

18     Listen(listenfd, LISTENQ);

19     maxfd = listenfd;           /* initialize */
20     maxi = -1;                  /* index into client[] array */
21     for (i = 0; i < FD_SETSIZE; i++)
22         client[i] = -1;         /* -1 indicates available entry */
23     FD_ZERO(&allset);
24     FD_SET(listenfd, &allset);
```

tcpcliserv/tcpservselect01.c

Figure 6.21 TCP server using a single process and `select`: initialization.

Create listening socket and initialize for `select`

12-24 The steps to create the listening socket are the same as seen earlier: `socket`, `bind`, and `listen`. We initialize our data structures given that the only descriptor that we will `select` on initially is the listening socket.

The last half of the function is shown in Figure 6.22

tcpcliserv/tcpservselect01.c

```
25      for ( ; ; ) {
26          rset = allset;          /* structure assignment */
27          nready = Select(maxfd + 1, &rset, NULL, NULL, NULL);

28          if (FD_ISSET(listenfd, &rset)) {    /* new client connection */
29              clilen = sizeof(cliaddr);
30              connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);

31              for (i = 0; i < FD_SETSIZE; i++)
32                  if (client[i] < 0) {
33                      client[i] = connfd;     /* save descriptor */
34                      break;
35                  }
36              if (i == FD_SETSIZE)
37                  err_quit("too many clients");

38              FD_SET(connfd, &allset);    /* add new descriptor to set */
39              if (connfd > maxfd)
40                  maxfd = connfd; /* for select */
41              if (i > maxi)
42                  maxi = i;       /* max index in client[] array */

43              if (--nready <= 0)
44                  continue;       /* no more readable descriptors */
45          }
46          for (i = 0; i <= maxi; i++) {   /* check all clients for data */
47              if ( (sockfd = client[i]) < 0)
48                  continue;
49              if (FD_ISSET(sockfd, &rset)) {
50                  if ( (n = Readline(sockfd, line, MAXLINE)) == 0) {
51                          /* connection closed by client */
52                      Close(sockfd);
53                      FD_CLR(sockfd, &allset);
54                      client[i] = -1;
55                  } else
56                      Writen(sockfd, line, n);

57                  if (--nready <= 0)
58                      break;      /* no more readable descriptors */
59              }
60          }
61      }
62  }
```

tcpcliserv/tcpservselect01.c

Figure 6.22 TCP server using a single process and `select`: loop.

Block in `select`

26-27 `select` waits for something to happen: either the establishment of a new client connection or the arrival of data, a FIN, or an RST on an existing connection.

`accept` new connections

28-45 If the listening socket is readable, a new connection has been established. We call `accept` and update our data structures accordingly. We use the first unused entry in the `client` array to record the connected socket. The number of ready descriptors is decremented, and if it is 0, we can avoid the next `for` loop. This lets us use the return value from `select` to avoid checking descriptors that are not ready.

Check existing connections

46-60 A test is made for each existing client connection as to whether or not its descriptor is in the descriptor set returned by `select`. If so, a line is read from the client and echoed back to the client. If the client closes the connection, `readline` returns 0 and we update our data structures accordingly.

We never decrement the value of `maxi` but could check for this possibility each time a client closes its connection.

This server is more complicated than the one shown in Figures 5.2 and 5.3, but it avoids all the overhead of creating a new process for each client and it is a nice example of `select`. Nevertheless, in Section 15.6 we will describe a problem with this server that is easily fixed by making the listening socket nonblocking and then checking for, and ignoring, a few errors from `accept`.

Denial of Service Attacks

Unfortunately there is a problem with the server that we just showed. Consider what happens if a malicious client connects to the server, sends 1 byte of data (other than a newline), and then goes to sleep. The server will call `readline`, which will read the single byte of data from the client and then block in the next call to `read`, waiting for more data from this client. The server is then blocked ("hung" may be a better term) by this one client and will not service any other clients (either new client connections or existing client's data) until the malicious client either sends a newline or terminates.

The basic concept here is that when a server is handling multiple clients the server can *never* block in a function call related to a single client. Doing so can hang the server and deny service to all other clients. This is called a *denial of service* attack. It does something to the server that prevents it from servicing other legitimate clients. Possible solutions are to (a) use nonblocking I/O (Chapter 15), (b) have each client serviced by a separate thread of control (e.g., either spawn a process or a thread to service each client), or (c) place a timeout on the I/O operations (Section 13.2).

6.9 pselect Function

The pselect function was invented by Posix.1g.

```
#include <sys/select.h>
#include <signal.h>
#include <time.h>

int pselect(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
            const struct timespec *timeout, const sigset_t *sigmask);
```

Returns: count of ready descriptors, 0 on timeout, –1 on error

pselect contains two changes from the normal select function.

1. pselect uses the timespec structure, an invention of the Posix.1b realtime standard, instead of the timeval structure.

   ```
   struct timespec {
     time_t tv_sec;      /* seconds */
     long   tv_nsec;     /* nanoseconds */
   };
   ```

   The difference in these two structures is with the second member: the tv_nsec member of the newer structure specifies nanoseconds, whereas the tv_usec member of the older structure specifies microseconds.

2. pselect adds a sixth argument: a pointer to a signal mask. This allows the program to disable the delivery of certain signals, test some global variables that are set by the handlers for these now-disabled signals, and then call pselect, telling it to reset the signal mask.

With regard to the second point, consider the following example (discussed on pp. 308–309 of APUE). Our program's signal handler for SIGINT just sets the global intr_flag and returns. If our process is blocked in a call to select, the return from the signal handler causes the function to return with errno set to EINTR. But when select is called, the code looks like the following:

```
if (intr_flag)
    handle_intr();    /* handle the signal */
if ( (nready = select( ... )) < 0) {
    if (errno == EINTR) {
        if (intr_flag)
            handle_intr();
    }
    ...
}
```

The problem is that between the test of intr_flag and the call to select, if the signal occurs, it will be lost if select blocks forever. With pselect we can now code this example reliably as

```
sigset_t  newmask, oldmask, zeromask;

sigemptyset(&zeromask);
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);

sigprocmask(SIG_BLOCK, &newmask, &oldmask); /* block SIGINT */
if (intr_flag)
    handle_intr();     /* handle the signal */
if ( (nready = pselect( ... , &zeromask)) < 0) {
    if (errno == EINTR) {
        if (intr_flag)
            handle_intr();
    }
    ...
}
```

Before testing the `intr_flag` variable we block `SIGINT`. When `pselect` is called, it replaces the signal mask of the process with an empty set (i.e., `zeromask`) and then checks the descriptors, possibly going to sleep. But when `pselect` returns, the signal mask of the process is reset to its value before `pselect` was called (i.e., `SIGINT` is blocked).

We say more about `pselect` and show an example of it in Section 18.5. We use `pselect` in Figure 18.7 and show a simple, albeit incorrect, implementation of `pselect` in Figure 18.8.

> There is one other slight difference between the two `select` functions. The first member of the `timeval` structure is a signed long integer while the first member of the `timespec` structure is a `time_t`. The signed long in the former should also be a `time_t` but was not changed retroactively, to avoid breaking existing code. The brand new function, however, could make this change.

6.10 poll Function

The `poll` function originated with SVR3 and was originally limited to streams devices (Chapter 33). SVR4 removed this limitation, allowing `poll` to work with any descriptor. `poll` provides functionality that is similar to `select`, but `poll` provides additional information when dealing with streams devices.

```
#include <poll.h>

int poll(struct pollfd *fdarray, unsigned long nfds, int timeout);
```

Returns: count of ready descriptors, 0 on timeout, –1 on error

The first argument is a pointer to the first element of an array of structures. Each element of the array is a `pollfd` structure that specifies the conditions to be tested for a given descriptor `fd`.

```
struct pollfd {
  int    fd;       /* descriptor to check */
  short  events;   /* events of interest on fd */
  short  revents;  /* events that occurred on fd */
};
```

The conditions to be tested are specified by the `events` member, and the function returns the status for that descriptor in the corresponding `revents` member. (Having two variables per descriptor, one a value and one a result, avoids value–result arguments. Recall that the middle three arguments for `select` are value–result.) Each of these two members is composed of one or more bits that specify a certain condition. Figure 6.23 shows the constants used to specify the `events` flag and to test the `revents` flag against.

Constant	Input to *events* ?	Result from *revents* ?	Description
POLLIN	•	•	normal or priority band data can be read
POLLRDNORM	•	•	normal data can be read
POLLRDBAND	•	•	priority band data can be read
POLLPRI	•	•	high-priority data can be read
POLLOUT	•	•	normal data can be written
POLLWRNORM	•	•	normal data can be written
POLLWRBAND	•	•	priority band data can be written
POLLERR		•	an error has occurred
POLLHUP		•	hangup has occurred
POLLNVAL		•	descriptor is not an open file

Figure 6.23 Input *events* and returned *revents* for `poll`.

We have divided this figure into three sections: the first four constants deal with input, the next three deal with output, and the final three deal with errors. Notice that the final three cannot be set in `events` but are always returned in `revents` when the corresponding condition exists.

There are three classes of data identified by `poll`: *normal*, *priority band*, and *high priority*. These terms come from the streams-based implementations (Figure 33.5).

> `POLLIN` can be defined as the logical OR of `POLLRDNORM` and `POLLRDBAND`. The `POLLIN` constant exists from SVR3 implementations that predated the priority bands in SVR4, so the constant remains for backward compatibility. Similarly, `POLLOUT` is equivalent to `POLLWRNORM`, with the former predating the latter.

With regard to TCP and UDP sockets, the following conditions cause `poll` to return the specified *revent*. Unfortunately, Posix.1g leaves many holes (i.e., optional ways to return the same condition) in its definition of `poll`.

- All regular TCP data and all UDP data is considered normal.
- TCP's out-of-band data (Chapter 21) is considered priority band.
- When the read-half of a TCP connection is closed (e.g., a FIN is received), this is also considered normal data and a subsequent read operation will return 0.

- The presence of an error for a TCP connection can be considered either normal data or an error (POLLERR). In either case, a subsequent read will return –1 with errno set to the appropriate value. This handles conditions such as the receipt of an RST or a timeout.
- The availability of a new connection on a listening socket can be considered either normal data or priority data. Most implementations consider this normal data.

The number of elements in the array of structures is specified by the *nfds* argument.

> Historically this argument has been an unsigned long, which seems excessive. An unsigned int would be adequate. Unix 98 defines a new datatype for this argument: nfds_t.

The *timeout* argument specifies how long the function is to wait before returning. A positive value specifies the number of milliseconds to wait. Figure 6.24 shows the possible values for the *timeout* argument.

timeout value	Description
INFTIM	wait forever
0	return immediately, do not block
> 0	wait specified number of milliseconds

Figure 6.24 *timeout* values for poll.

The constant INFTIM is defined to be a negative value. If the system does not provide a timer with millisecond accuracy, the value is rounded up to the nearest supported value.

> Posix.1g requires that INFTIM be defined by including <poll.h>, but many systems still define it in <sys/stropts.h>.
>
> As with select, any timeout set for poll is limited by the implementation's clock resolution (often 10 ms).

The return value from poll is –1 if an error occurred, 0 if no descriptors are ready before the timer expires, otherwise it is the number of descriptors that have a nonzero revents member.

If we are no longer interested in a particular descriptor, we just set the fd member of the pollfd structure to a negative value. Then the events member is ignored and the revents member is set to 0 on return.

Recall our discussion at the end of Section 6.3 about FD_SETSIZE and the maximum number of descriptors per descriptor set versus the maximum number of descriptors per process. We do not have that problem with poll since it is the caller's responsibility to allocate an array of pollfd structures and then tell the kernel the number of elements in the array. There is no fixed-size datatype similar to fd_set that the kernel knows about.

Posix.1g requires both `select` and `poll`. But from a portability perspective today, more systems support `select` than support `poll`. Also, Posix.1g defines `pselect`, an enhanced version of `select` that handles signal blocking and provides increased time resolution but defines nothing similar for `poll`.

6.11 TCP Echo Server (Revisited Again)

We now redo our TCP echo server from Section 6.8 using `poll` instead of `select`. In the previous version using `select` we had to allocate a `client` array along with a descriptor set named `rset` (Figure 6.15). With `poll` we must allocate an array of `pollfd` structures so we use it to maintain the client information, instead of allocating another array. We handle the `fd` member of this array the same way we handled the `client` array in Figure 6.15: a value of –1 means the entry is not in use; otherwise it is the descriptor value. Recall from the previous section that any entry in the array of `pollfd` structures passed to `poll` with a negative value for the `fd` member is just ignored.

Figure 6.25 shows the first half of our server.

tcpcliserv/tcpservpoll01.c

```
 1 #include    "unp.h"
 2 #include    <limits.h>          /* for OPEN_MAX */

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     i, maxi, listenfd, connfd, sockfd;
 7     int     nready;
 8     ssize_t n;
 9     char    line[MAXLINE];
10     socklen_t clilen;
11     struct pollfd client[OPEN_MAX];
12     struct sockaddr_in cliaddr, servaddr;

13     listenfd = Socket(AF_INET, SOCK_STREAM, 0);

14     bzero(&servaddr, sizeof(servaddr));
15     servaddr.sin_family = AF_INET;
16     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
17     servaddr.sin_port = htons(SERV_PORT);

18     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

19     Listen(listenfd, LISTENQ);

20     client[0].fd = listenfd;
21     client[0].events = POLLRDNORM;
22     for (i = 1; i < OPEN_MAX; i++)
23         client[i].fd = -1;      /* -1 indicates available entry */
24     maxi = 0;                   /* max index into client[] array */
```

tcpcliserv/tcpservpoll01.c

Figure 6.25 First half of TCP server using `poll`.

Allocate array of `pollfd` structures

11 We declare `OPEN_MAX` elements in our array of `pollfd` structures. Determining the maximum number of descriptors that a process can have open at any one time is hard. We encounter this problem again in Figure 12.4. One way is to call the Posix.1 `sysconf` function with an argument of `_SC_OPEN_MAX` (as described on pp. 42–44 of APUE) and then dynamically allocate an array of the appropriate size. But one of the possible returns from `sysconf` is "indeterminate," meaning we still have to guess a value. Here we just use the Posix.1 `OPEN_MAX` constant.

Initialize

20–24 We use the first entry in the `client` array for the listening socket, and set the descriptor for the remaining entries to –1. We also set the `POLLRDNORM` event for this descriptor, to be notified by `poll` when a new connection is ready to be accepted. The variable `maxi` contains the largest index of the `client` array currently in use.

The second half of our function is shown in Figure 6.26.

Call `poll`; check for new connection

26–42 We call `poll` to wait for either a new connection or data on an existing connection. When a new connection is accepted, we find the first available entry in the `client` array by looking for the first one with a negative descriptor. Notice that we start the search with the index of 1, since `client[0]` is used for the listening socket. When an available entry is found, we save the descriptor and set the `POLLRDNORM` event.

Check for data on an existing connection

43–63 The two return events that we check for are `POLLRDNORM` and `POLLERR`. The second of these we did not set in the `event` member, because it is always returned when the condition is true. The reason we check for `POLLERR` is because some implementations return this event when an RST is received for a connection, while others just return `POLLRDNORM`. In either case we call `readline` and if an error has occurred, it will return an error. When an existing connection is terminated by the client, we just set the `fd` member to –1.

tcpcliserv/tcpservpoll01.c

```
25      for ( ; ; ) {
26          nready = Poll(client, maxi + 1, INFTIM);

27          if (client[0].revents & POLLRDNORM) {   /* new client connection */
28              clilen = sizeof(cliaddr);
29              connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);

30              for (i = 1; i < OPEN_MAX; i++)
31                  if (client[i].fd < 0) {
32                      client[i].fd = connfd;  /* save descriptor */
33                      break;
34                  }
35              if (i == OPEN_MAX)
36                  err_quit("too many clients");

37              client[i].events = POLLRDNORM;
38              if (i > maxi)
39                  maxi = i;       /* max index in client[] array */

40              if (--nready <= 0)
41                  continue;       /* no more readable descriptors */
42          }
43          for (i = 1; i <= maxi; i++) {   /* check all clients for data */
44              if ( (sockfd = client[i].fd) < 0)
45                  continue;
46              if (client[i].revents & (POLLRDNORM | POLLERR)) {
47                  if ( (n = readline(sockfd, line, MAXLINE)) < 0) {
48                      if (errno == ECONNRESET) {
49                              /* connection reset by client */
50                          Close(sockfd);
51                          client[i].fd = -1;
52                      } else
53                          err_sys("readline error");
54                  } else if (n == 0) {
55                              /* connection closed by client */
56                      Close(sockfd);
57                      client[i].fd = -1;
58                  } else
59                      Writen(sockfd, line, n);

60                  if (--nready <= 0)
61                      break;      /* no more readable descriptors */
62              }
63          }
64      }
65  }
```

tcpcliserv/tcpservpoll01.c

Figure 6.26 Second half of TCP server using poll.

6.12 Summary

There are five different models for I/O provided by Unix:

- blocking,
- nonblocking,
- I/O multiplexing,
- signal-driven I/O, and
- asynchronous I/O.

The default is blocking I/O, which is also the most commonly used. We cover nonblocking I/O and signal-driven I/O in later chapters and have covered I/O multiplexing in this chapter. True asynchronous I/O is defined by Posix.1, but few implementations exist.

The most commonly used function for I/O multiplexing is `select`. We tell the function what descriptors we are interested in (for reading, writing, and exceptions), the maximum amount of time to wait, the maximum descriptor number (plus one). Most calls to `select` specify readability, and we noted that the only exception condition when dealing with sockets is the arrival of out-of-band data (Chapter 21). Since `select` provides a time limit on how long the function blocks, we will use this feature in Figure 13.3 to place a time limit on an input operation.

We used our echo client in a batch mode using `select` and discovered that even though the end of the user input is encountered, data can still be in the pipe to or from the server. To handle this scenario requires the `shutdown` function, and it lets us take advantage of TCP's half-close feature.

Posix.1g defines the new function `pselect`, which increases the time precision from microseconds to nanoseconds and takes a new argument that is a pointer to a signal set. This lets us avoid race conditions when signals are being caught and we talk more about this in Section 18.5.

The `poll` function from System V provides functionality similar to `select` and provides additional information on streams devices. Posix.1g requires both `select` and `poll`, but the former is used more often.

Exercises

6.1 We said that a descriptor set can be assigned to another descriptor set across an equals sign in C. How is this done if a descriptor set is an array of integers? (*Hint*: Look at your system's `<sys/select.h>` or `<sys/types.h>` header.)

6.2 When describing the conditions for which `select` returns "writable" in Section 6.3, why did we need the qualifier that the socket had to be nonblocking in order for a write operation to return a positive value?

6.3 What happens in Figure 6.9 if we prepend the word `else` before the word `if` on line 19?

6.4 In our example in Figure 6.21 add code to allow the server to be able to use as many descriptors as currently allowed by the kernel. (*Hint*: Look at the `setrlimit` function.)

6.5 Let's see what happens when the second argument to `shutdown` is `SHUT_RD`. Start with the TCP client in Figure 5.4 and make the following changes: change the port number from `SERV_PORT` to 19, the `chargen` server (Figure 2.13); replace the call to `str_cli` with a call to the `pause` function. Run this program specifying the IP address of a local host that runs the `chargen` server. Watch the packets with a tool such as `tcpdump` (Section C.5). What happens?

6.6 Why would an application call `shutdown` with an argument of `SHUT_RDWR`, instead of just calling `close`?

6.7 What happens in Figure 6.22 when the client sends an RST to terminate the connection?

6.8 Recode Figure 6.25 to call `sysconf` to determine the maximum number of descriptors and allocate the `client` array accordingly.

7

Socket Options

7.1 Introduction

There are various ways to get and set the options that affect a socket:

- the `getsockopt` and `setsockopt` functions,
- the `fcntl` function, and
- the `ioctl` function.

This chapter starts by covering the `setsockopt` and `getsockopt` functions, followed by an example that prints the default value of all the options, followed by a detailed description of all of the socket options. We divide the detailed descriptions into the following categories: generic, IPv4, IPv6, and TCP. This detailed coverage can be skipped during a first reading of this chapter, and the individual sections referred to when needed. A few options are discussed in detail in a later chapter, such as the IPv4 and IPv6 multicasting options, which we describe with multicasting in Section 19.5.

We also describe the `fcntl` function, because it is the Posix way to set a socket for nonblocking I/O, signal-driven I/O, and to set the owner of a socket. We save the `ioctl` function for Chapter 16.

7.2 getsockopt and setsockopt Functions

These two functions apply only to sockets.

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

int getsockopt(int sockfd, int level, int optname, void *optval, socklen_t *optlen);

int setsockopt(int sockfd, int level, int optname, const void *optval,
               socklen_t optlen);
```

Both return: 0 if OK, –1 on error

sockfd must refer to an open socket descriptor. The *level* specifies the code in the system to interpret the option: the general socket code, or some protocol-specific code (e.g., IPv4, IPv6, or TCP).

optval is a pointer to a variable from which the new value of the option is fetched by `setsockopt`, or into which the current value of the option is stored by `getsockopt`. The size of this variable is specified by the final argument, as a value for `setsockopt` and as a value–result for `getsockopt`.

Figure 7.1 summarizes the options that can be queried by `getsockopt` or set by `setsockopt`. The "Datatype" column shows the datatype of what the *optval* pointer must point to for each option. We use the notation of two braces to indicate a structure, as in `linger{}` to mean a `struct linger`.

There are two basic types of options: binary options that enable or disable a certain feature (flags), and options that fetch and return specific values that we can either set or examine (values). The column labeled "Flag" specifies if the option is a flag option. When calling `getsockopt` for these flag options, `*optval` is an integer. The value returned in `*optval` is zero if the option is disabled, or nonzero if the option is enabled. Similarly, `setsockopt` requires a nonzero `*optval` to turn the option on, and a zero value to turn the option off. If the "Flag" column does not contain a "•" then the option is used to pass a value of the specified datatype between the user process and the system.

Following sections of this chapter give additional details on the options that affect a socket.

7.3 Checking If an Option Is Supported and Obtaining the Default

We now write a program to check whether most of the options defined in Figure 7.1 are supported, and if so, print their default value. Figure 7.2 contains the declarations for our program.

level	*optname*	get	set	Description	Flag	Datatype
SOL_SOCKET	SO_BROADCAST	•	•	permit sending of broadcast datagrams	•	int
	SO_DEBUG	•	•	enable debug tracing	•	int
	SO_DONTROUTE	•	•	bypass routing table lookup	•	int
	SO_ERROR	•		get pending error and clear		int
	SO_KEEPALIVE	•	•	periodically test if connection still alive	•	int
	SO_LINGER	•	•	linger on close if data to send		linger{}
	SO_OOBINLINE	•	•	leave received out-of-band data inline	•	int
	SO_RCVBUF	•	•	receive buffer size		int
	SO_SNDBUF	•	•	send buffer size		int
	SO_RCVLOWAT	•	•	receive buffer low-water mark		int
	SO_SNDLOWAT	•	•	send buffer low-water mark		int
	SO_RCVTIMEO	•	•	receive timeout		timeval{}
	SO_SNDTIMEO	•	•	send timeout		timeval{}
	SO_REUSEADDR	•	•	allow local address reuse	•	int
	SO_REUSEPORT	•	•	allow local address reuse	•	int
	SO_TYPE	•		get socket type		int
	SO_USELOOPBACK	•	•	routing socket gets copy of what it sends	•	int
IPPROTO_IP	IP_HDRINCL	•	•	IP header included with data	•	int
	IP_OPTIONS	•	•	IP header options		(see text)
	IP_RECVDSTADDR	•	•	return destination IP address	•	int
	IP_RECVIF	•	•	return received interface index	•	int
	IP_TOS	•	•	type-of-service and precedence		int
	IP_TTL	•	•	time-to-live		int
	IP_MULTICAST_IF	•	•	specify outgoing interface		in_addr{}
	IP_MULTICAST_TTL	•	•	specify outgoing TTL		u_char
	IP_MULTICAST_LOOP	•	•	specify loopback		u_char
	IP_ADD_MEMBERSHIP		•	join a multicast group		ip_mreq{}
	IP_DROP_MEMBERSHIP		•	leave a multicast group		ip_mreq{}
IPPROTO_ICMPV6	ICMP6_FILTER	•	•	specify ICMPv6 message types to pass		icmp6_filter{}
IPPROTO_IPV6	IPV6_ADDRFORM	•	•	change address format of socket		int
	IPV6_CHECKSUM	•	•	offset of checksum field for raw sockets		int
	IPV6_DSTOPTS	•	•	receive destination options	•	int
	IPV6_HOPLIMIT	•	•	receive unicast hop limit	•	int
	IPV6_HOPOPTS	•	•	receive hop-by-hop options	•	int
	IPV6_NEXTHOP	•	•	specify next-hop address	•	sockaddr{}
	IPV6_PKTINFO	•	•	receive packet information	•	int
	IPV6_PKTOPTIONS	•	•	specify packet options		(see text)
	IPV6_RTHDR	•	•	receive source route	•	int
	IPV6_UNICAST_HOPS	•	•	default unicast hop limit		int
	IPV6_MULTICAST_IF	•	•	specify outgoing interface		in6_addr{}
	IPV6_MULTICAST_HOPS	•	•	specify outgoing hop limit		u_int
	IPV6_MULTICAST_LOOP	•	•	specify loopback	•	u_int
	IPV6_ADD_MEMBERSHIP		•	join a multicast group		ipv6_mreq{}
	IPV6_DROP_MEMBERSHIP		•	leave a multicast group		ipv6_mreq{}
IPPROTO_TCP	TCP_KEEPALIVE	•	•	idle time in seconds before probing		int
	TCP_MAXRT	•	•	TCP maximum retransmit time		int
	TCP_MAXSEG	•	•	TCP maximum segment size		int
	TCP_NODELAY	•	•	disable Nagle algorithm	•	int
	TCP_STDURG	•	•	interpretation of urgent pointer	•	int

Figure 7.1 Summary of socket options for `getsockopt` and `setsockopt`.

sockopt/checkopts.c

```
 1 #include    "unp.h"
 2 #include    <netinet/tcp.h>     /* for TCP_xxx defines */

 3 union val {
 4   int               i_val;
 5   long              l_val;
 6   char              c_val[10];
 7   struct linger     linger_val;
 8   struct timeval    timeval_val;
 9 } val;

10 static char *sock_str_flag(union val *, int);
11 static char *sock_str_int(union val *, int);
12 static char *sock_str_linger(union val *, int);
13 static char *sock_str_timeval(union val *, int);

14 struct sock_opts {
15   char     *opt_str;
16   int       opt_level;
17   int       opt_name;
18   char   *(*opt_val_str)(union val *, int);
19 } sock_opts[] = {
20     "SO_BROADCAST",     SOL_SOCKET, SO_BROADCAST,    sock_str_flag,
21     "SO_DEBUG",         SOL_SOCKET, SO_DEBUG,        sock_str_flag,
22     "SO_DONTROUTE",     SOL_SOCKET, SO_DONTROUTE,    sock_str_flag,
23     "SO_ERROR",         SOL_SOCKET, SO_ERROR,        sock_str_int,
24     "SO_KEEPALIVE",     SOL_SOCKET, SO_KEEPALIVE,    sock_str_flag,
25     "SO_LINGER",        SOL_SOCKET, SO_LINGER,       sock_str_linger,
26     "SO_OOBINLINE",     SOL_SOCKET, SO_OOBINLINE,    sock_str_flag,
27     "SO_RCVBUF",        SOL_SOCKET, SO_RCVBUF,       sock_str_int,
28     "SO_SNDBUF",        SOL_SOCKET, SO_SNDBUF,       sock_str_int,
29     "SO_RCVLOWAT",      SOL_SOCKET, SO_RCVLOWAT,     sock_str_int,
30     "SO_SNDLOWAT",      SOL_SOCKET, SO_SNDLOWAT,     sock_str_int,
31     "SO_RCVTIMEO",      SOL_SOCKET, SO_RCVTIMEO,     sock_str_timeval,
32     "SO_SNDTIMEO",      SOL_SOCKET, SO_SNDTIMEO,     sock_str_timeval,
33     "SO_REUSEADDR",     SOL_SOCKET, SO_REUSEADDR,    sock_str_flag,
34 #ifdef  SO_REUSEPORT
35     "SO_REUSEPORT",     SOL_SOCKET, SO_REUSEPORT,    sock_str_flag,
36 #else
37     "SO_REUSEPORT",     0,          0,               NULL,
38 #endif
39     "SO_TYPE",          SOL_SOCKET, SO_TYPE,         sock_str_int,
40     "SO_USELOOPBACK",   SOL_SOCKET, SO_USELOOPBACK,  sock_str_flag,
41     "IP_TOS",           IPPROTO_IP, IP_TOS,          sock_str_int,
42     "IP_TTL",           IPPROTO_IP, IP_TTL,          sock_str_int,
43     "TCP_MAXSEG",       IPPROTO_TCP,TCP_MAXSEG,      sock_str_int,
44     "TCP_NODELAY",      IPPROTO_TCP,TCP_NODELAY,     sock_str_flag,
45     NULL,               0,          0,               NULL
46 };
```

sockopt/checkopts.c

Figure 7.2 Declarations for our program to check the socket options.

Declare `union` of possible values

3-9 Our `union` contains one member for each possible return value from `getsockopt`.

Define function prototypes

10-13 We define function prototypes for four functions that are called to print the value for a given socket option.

Define structure and initialize array

14-46 Our `sock_opts` structure contains all the information necessary to call `getsockopt` for each socket option and then print its current value. The final member, `opt_val_str`, is a pointer to one of our four functions that will print the option value. We allocate and initialize an array of these structures, one element for each socket option.

> Not all implementations support all socket options. The way to determine if a given option is supported is to use an `#ifdef` or a `#if defined`, as we show for `SO_REUSEPORT`. For completeness *every* element of the array should be compiled similar to what we show for `SO_REUSEPORT`, but we omit these because the `#ifdefs` just lengthen the code that we show and add nothing to the discussion.

Figure 7.3 shows our `main` function.

sockopt/checkopts.c

```
47 int
48 main(int argc, char **argv)
49 {
50     int     fd, len;
51     struct sock_opts *ptr;

52     fd = Socket(AF_INET, SOCK_STREAM, 0);

53     for (ptr = sock_opts; ptr->opt_str != NULL; ptr++) {
54         printf("%s: ", ptr->opt_str);
55         if (ptr->opt_val_str == NULL)
56             printf("(undefined)\n");
57         else {
58             len = sizeof(val);
59             if (getsockopt(fd, ptr->opt_level, ptr->opt_name,
60                            &val, &len) == -1) {
61                 err_ret("getsockopt error");
62             } else {
63                 printf("default = %s\n", (*ptr->opt_val_str) (&val, len));
64             }
65         }
66     }
67     exit(0);
68 }
```

sockopt/checkopts.c

Figure 7.3 `main` function to check all socket options.

Create TCP socket, go through all options

52-56 We create a TCP socket and then go through all elements in our array. If the `opt_val_str` pointer is null, the option is not defined by the implementation (which we showed is possible for `SO_REUSEPORT`).

Call `getsockopt`

57-61 We call `getsockopt` but do not terminate if an error is returned. Many implementations define some of the socket option names even though they do not support the option. Unsupported options should elicit an error of `ENOPROTOOPT`.

Print option's default value

62-63 If `getsockopt` returns success, we call our function to convert the option value to a string, and print the string.

In Figure 7.2 we showed four function prototypes, one for each type of option value that is returned. Figure 7.4 shows one of these four functions, `sock_str_flag`, which prints the value of a flag option. The other three functions are similar.

sockopt/checkopts.c

```
69 static char strres[128];

70 static char *
71 sock_str_flag(union val *ptr, int len)
72 {
73     if (len != sizeof(int))
74         snprintf(strres, sizeof(strres), "size (%d) not sizeof(int)", len);
75     else
76         snprintf(strres, sizeof(strres),
77                  "%s", (ptr->i_val == 0) ? "off" : "on");
78     return(strres);

79 }
```

sockopt/checkopts.c

Figure 7.4 `sock_str_flag` function: convert flag option to a string.

73-78 Recall that the final argument to `getsockopt` is a value–result argument. The first check we make is that the size of the value returned by `getsockopt` is the expected size. The string returned is `off` or `on`, depending whether the value of the flag option is 0 or nonzero, respectively.

Running this program under AIX 4.2 gives the following output:

```
aix % checkopts
SO_BROADCAST: default = off
SO_DEBUG: default = off
SO_DONTROUTE: default = off
SO_ERROR: default = 0
SO_KEEPALIVE: default = off
SO_LINGER: default = l_onoff = 0, l_linger = 0
SO_OOBINLINE: default = off
SO_RCVBUF: default = 16384
SO_SNDBUF: default = 16384
```

```
SO_RCVLOWAT: default = 1
SO_SNDLOWAT: default = 4096
SO_RCVTIMEO: default = 0 sec, 0 usec
SO_SNDTIMEO: default = 0 sec, 0 usec
SO_REUSEADDR: default = off
SO_REUSEPORT: (undefined)
SO_TYPE: default = 1
SO_USELOOPBACK: default = off
IP_TOS: default = 0
IP_TTL: default = 60
TCP_MAXSEG: default = 512
TCP_NODELAY: default = off
```

The value of 1 returned for the `SO_TYPE` option corresponds to `SOCK_STREAM` for this implementation.

7.4 Socket States

For some socket options there are timing considerations about when to set or fetch the option versus the state of the socket. We mention these with the affected options.

The following socket options are inherited by a connected TCP socket from the listening socket (pp. 462–463 of TCPv2): `SO_DEBUG`, `SO_DONTROUTE`, `SO_KEEPALIVE`, `SO_LINGER`, `SO_OOBINLINE`, `SO_RCVBUF`, and `SO_SNDBUF`. This is important with TCP because the connected socket is not returned to a server by `accept` until the three-way handshake is completed by the TCP layer. If we want to ensure that one of these socket options is set for the connected socket when the three-way handshake completes, we must set that option for the listening socket.

7.5 Generic Socket Options

We start with a discussion of the generic socket options. These options are protocol independent (that is, they are handled by the protocol-independent code within the kernel, not by one particular protocol module such as IPv4), but some of the options apply to only certain types of sockets. For example, even though the `SO_BROADCAST` socket option is called "generic," it applies only to datagram sockets.

`SO_BROADCAST` Socket Option

This option enables or disables the ability of the process to send broadcast messages. Broadcasting is supported for only datagram sockets and only on networks that support the concept of a broadcast message (e.g., Ethernet, token ring, etc.). You cannot broadcast on a point-to-point link. We talk more about broadcasting in Chapter 18.

Since an application must set this socket option before sending a broadcast datagram, it prevents a process from sending a broadcast when the application was never designed to broadcast. For example, a UDP application might take the destination IP

address as a command-line argument, but the application never intended for a user to type in a broadcast address. Rather than forcing the application to try to determine if a given address is a broadcast address or not, the test is in the kernel: if the destination address is a broadcast address and this socket option is not set, `EACCES` is returned (p. 233 of TCPv2).

`SO_DEBUG` Socket Option

This option is supported only by TCP. When enabled for a TCP socket, the kernel keeps track of detailed information about all the packets sent or received by TCP for the socket. These are kept in a circular buffer within the kernel that can be examined with the `trpt` program. Pages 916–920 of TCPv2 provide additional details and an example that uses this option.

`SO_DONTROUTE` Socket Option

This option specifies that outgoing packets are to bypass the normal routing mechanisms of the underlying protocol. For example, with IPv4, the packet is directed to the appropriate local interface, as specified by the network and subnet portions of the destination address. If the local interface cannot be determined from the destination address (e.g., the destination is not on the other end of a point-to-point link, or not on a shared network), `ENETUNREACH` is returned.

The equivalent of this option can also be applied to individual datagrams using the `MSG_DONTROUTE` flag with the `send`, `sendto`, or `sendmsg` functions.

This option is often used by the routing daemons (`routed` and `gated`) to bypass the routing table (in case the routing table is incorrect) and force a packet to be sent out a particular interface.

`SO_ERROR` Socket Option

When an error occurs on a socket, the protocol module in a Berkeley-derived kernel sets a variable named `so_error` for that socket to one of the standard Unix `Exxx` values. This is called the *pending error* for the socket. The process can be immediately notified of the error in one of two ways.

1. If the process is blocked in a call to `select` on the socket (Section 6.3), for either readability or writability, `select` returns with either or both conditions set.
2. If the process is using signal-driven I/O (Chapter 22), the `SIGIO` signal is generated for either the process or the process group.

The process can then obtain the value of `so_error` by fetching the `SO_ERROR` socket option. The integer value returned by `getsockopt` is the pending error for the socket. The value of `so_error` is then reset to 0 by the kernel (p. 547 of TCPv2).

If `so_error` is nonzero when the process calls `read` and there is no data to return, `read` returns –1 with `errno` set to the value of `so_error` (p. 516 of TCPv2). The value

of `so_error` is then reset to 0. If there is data queued for the socket, that data is returned by `read` instead of the error condition. If `so_error` is nonzero when the process calls `write`, −1 is returned with `errno` set to the value of `so_error` (p. 495 of TCPv2) and `so_error` is reset to 0.

> There is a bug in the code shown on p. 495 of TCPv2, in that `so_error` is not reset to 0. This has been fixed in the BSD/OS release. Anytime the pending error for a socket is returned, it must be reset to 0.

This is the first socket option that we have encountered that can be fetched but cannot be set.

SO_KEEPALIVE Socket Option

When the keepalive option is set for a TCP socket and no data has been exchanged across the socket in either direction for 2 hours, TCP automatically sends a *keepalive probe* to the peer. This probe is a TCP segment to which the peer must respond. One of three scenarios results.

1. The peer responds with the expected ACK. The application is not notified (since everything is OK). TCP will send another probe following another 2 hours of inactivity.
2. The peer responds with an RST, which tells the local TCP that the peer host has crashed and rebooted. The socket's pending error is set to `ECONNRESET` and the socket is closed.
3. There is no response from the peer to the keepalive probe. Berkeley-derived TCPs send eight additional probes, 75 seconds apart, trying to elicit a response. TCP will give up if there is no response within 11 minutes and 15 seconds after sending the first probe. If there is no response at all to TCP's keepalive probes, the socket's pending error is set to `ETIMEDOUT` and the socket is closed. But if the socket receives an ICMP error in response to one of the keepalive probes, the corresponding error (Figures A.15 and A.16) is returned instead (and the socket is still closed). A common ICMP error in this scenario is "host unreachable," indicating that the peer host has not crashed but is just unreachable, in which case the pending error is set to `EHOSTUNREACH`.

Chapter 23 of TCPv1 and pp. 828–831 of TCPv2 contain additional details on the keepalive option.

Undoubtedly the most common question regarding this option is whether the timing parameters can be modified (usually to reduce the 2-hour period of inactivity to some shorter value). We describe the new Posix.1g `TCP_KEEPALIVE` option in Section 7.9, but this is not widely implemented. Appendix E of TCPv1 discusses how to change these timing parameters for various kernels, but be aware that most kernels maintain these parameters on a per-kernel basis, not on a per-socket basis, so changing the inactivity period from 2 hours to 15 minutes, for example, will affect *all* sockets on the host that enable this option.

The purpose of this option is to detect if the peer *host* crashes. If the peer *process* crashes, its TCP will send a FIN across the connection, which we can easily detect with `select`. (This was why we used `select` in Section 6.4.) Also realize that if there is no response to any of the keepalive probes (scenario 3), we are not guaranteed that the peer host has crashed, and TCP may well terminate a valid connection. It could be that some intermediate router has crashed for 15 minutes, and that period of time just happens to completely overlap our host's 11-minute and 15-second keepalive probe period.

This option is normally used by servers, although clients can also use the option. Servers use the option because they spend most of their time blocked waiting for input across the TCP connection, that is, waiting for a client request. But if the client host crashes, the server process will never know about it, and the server will continually wait for input that can never arrive. This is called a *half-open connection*. The keepalive option will detect these half-open connections and terminate them.

> Most Rlogin and Telnet servers set this option to terminate the connection if the interactive client hangs up the phone line or powers off the terminal (for example) without logging out.
>
> Some servers, notably FTP servers, provide an application timeout, often on the order of minutes. This is done by the application itself, normally around a call to `read`, reading the next client command. This timeout does not involve this socket option.

Figure 7.5 summarizes the various methods that we have to detect when something happens on the other end of a TCP connection. When we say "using `select` for readability" we mean calling `select` to test whether the socket is readable.

Scenario	Peer process crashes	Peer host crashes	Peer host is unreachable
Our TCP is actively sending data	Peer TCP sends a FIN, which we can detect immediately using `select` for readability. If TCP sends another segment, peer TCP responds with RST. If TCP sends yet another segment, our TCP sends us `SIGPIPE`.	Our TCP will time out and our socket's pending error is set to `ETIMEDOUT`.	Our TCP will time out and our socket's pending error is set to `EHOSTUNREACH`.
Our TCP is actively receiving data	Peer TCP will send a FIN, which we will read as a (possibly premature) end-of-file.	We will stop receiving data.	We will stop receiving data.
Connection is idle, keepalive set	Peer TCP sends a FIN, which we can detect immediately using `select` for readability.	Nine keepalive probes are sent after 2 hours of inactivity and then our socket's pending error is set to `ETIMEDOUT`.	Nine keepalive probes are sent after 2 hours of inactivity and then our socket's pending error is set to `EHOSTUNREACH`.
Connection is idle, keepalive not set	Peer TCP sends a FIN, which we can detect immediately using `select` for readability.	(Nothing.)	(Nothing.)

Figure 7.5 Ways to detect various TCP conditions.

SO_LINGER Socket Option

This option specifies how the close function operates for a connection-oriented protocol (e.g., for TCP but not for UDP). By default, close returns immediately, but if there is any data still remaining in the socket send buffer, the system will try to deliver the data to the peer.

The SO_LINGER socket option lets us change this default. This option requires the following structure to be passed between the user process and the kernel. It is defined by including <sys/socket.h>.

```
struct linger {
  int   l_onoff;   /* 0=off, nonzero=on */
  int   l_linger;  /* linger time, Posix.1g specifies units as seconds */
};
```

Calling setsockopt leads to one of the following three scenarios depending on the values of the two structure members.

1. If l_onoff is 0, the option is turned off. The value of l_linger is ignored and the previously discussed TCP default applies: close returns immediately.
2. If l_onoff is nonzero and l_linger is 0, TCP aborts the connection when it is closed (pp. 1019–1020 of TCPv2). That is, TCP discards any data still remaining in the socket send buffer and sends an RST to the peer, not the normal four-packet connection termination sequence (Section 2.5). We show an example of this in Figure 15.21. This avoids TCP's TIME_WAIT state, but in doing so leaves open the possibility of another incarnation of this connection being created within 2MSL seconds and having old duplicate segments from the just-terminated connection being incorrectly delivered to the new incarnation (Section 2.6).

   Some implementations, notably Solaris 2.x where x ≤ 5, do not implement this feature of the SO_LINGER option.

   Occasional Usenet postings advocate the use of this feature just to avoid the TIME_WAIT state and to be able to restart a listening server even if connections are still in use with the server's well-known port. This should not be done and could lead to data corruption, as detailed in RFC 1337 [Braden 1992a]. Instead, the SO_REUSEADDR socket option should always be used in the server before the call to bind, as we describe shortly. The TIME_WAIT state is our friend and is there to help us (i.e., to let old duplicate segments expire in the network). Instead of trying to avoid the state, we should understand it (Section 2.6).
3. If l_onoff is nonzero and l_linger is nonzero, then the kernel will *linger* when the socket is closed (p. 472 of TCPv2). That is, if there is any data still remaining in the socket send buffer, the process is put to sleep until either (a) all the data is sent and acknowledged by the peer TCP, or (b) the linger time expires. If the socket has been set nonblocking (Chapter 15), it will not wait for the close to complete, even if the linger time is nonzero.

When using this feature of the SO_LINGER option it is important for the application to check the return value from close, because if the linger time expires before the remaining data is sent and acknowledged, close returns EWOULDBLOCK and any remaining data in the send buffer is discarded.

Unfortunately the interpretation of a nonzero l_linger member in the third case is implementation dependent. 4.4BSD assumes the units are clock ticks (one-hundredths of a second) but Posix.1g specifies the units as seconds. Another problem with existing Berkeley-derived implementations is that the l_linger member (an int) is copied into a kernel variable (so_linger) that is a 16-bit signed integer, limiting the linger time to 32767 clock ticks, or 327.67 seconds.

We now need to see exactly when a close on a socket returns, given the various scenarios that we have looked at. We assume that the client writes data to the socket and then calls close. Figure 7.6 shows the default situation.

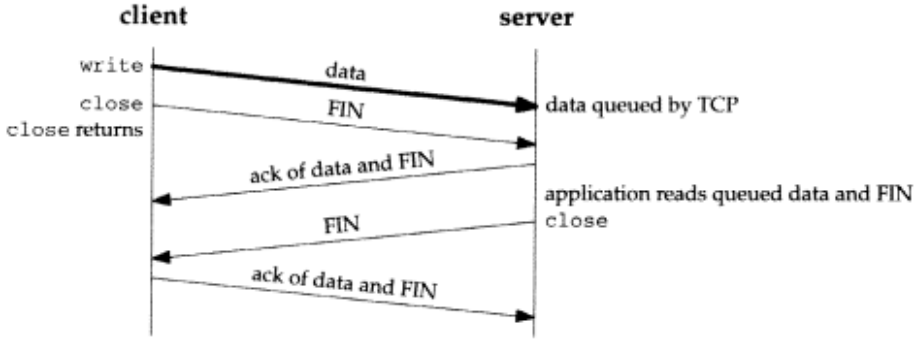

Figure 7.6 Default operation of close: it returns immediately.

We assume that when the client's data arrives, the server is temporarily busy, so the data is added to the socket receive buffer by its TCP. Similarly the next segment, the client's FIN, is also added to the socket receive buffer (in whatever manner the implementation records that a FIN has been received on the connection). But by default the client's close returns immediately. As we show in this scenario, the client's close can return before the server reads the remaining data in its socket receive buffer. It is possible for the server host to crash before the server application reads this remaining data, and the client application will never know.

The client can set the SO_LINGER socket option, specifying some positive linger time. When this occurs, the client's close does not return until all the client's data and its FIN have been acknowledged by the server TCP. We show this in Figure 7.7. But we still have the same problem as in Figure 7.6: the server host can crash before the server application reads its remaining data, and the client application will never know.

The basic principle here is that a successful return from close, with the SO_LINGER socket option set, only tells us that the data we sent (and our FIN) have been acknowledged by the peer TCP. This does *not* tell us whether the peer application

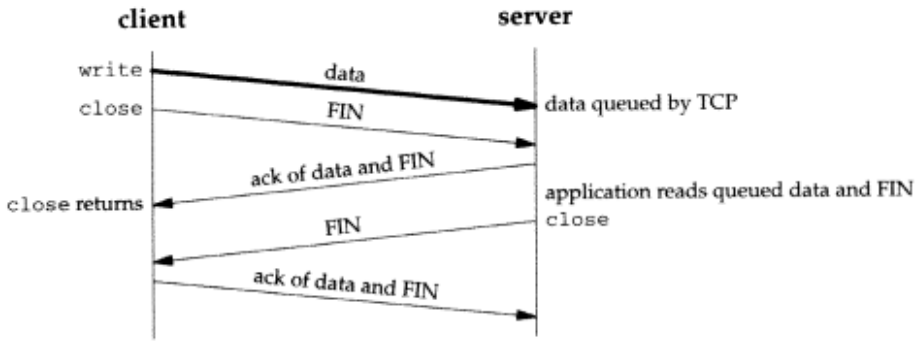

Figure 7.7 `close` with `SO_LINGER` socket option set and `l_linger` a positive value.

has read the data. If we do not set the `SO_LINGER` socket option, we do not know whether the peer TCP has acknowledged the data.

One way for the client to know that the server has read its data is to call `shutdown` (with a second argument of `SHUT_WR`) instead of `close` and wait for the peer to `close` its end of the connection. We show this scenario in Figure 7.8.

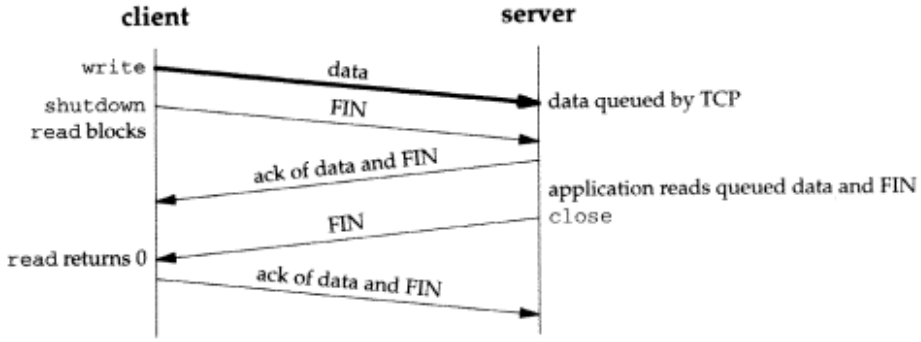

Figure 7.8 Using `shutdown` to know that peer has received our data.

Comparing this figure to Figures 7.6 and 7.7 we see that when we close our end of the connection, depending on the function called (`close` or `shutdown`) and whether the `SO_LINGER` socket option is set, the return can occur at three different times:

1. `close` returns immediately, without waiting at all (the default; Figure 7.6),
2. `close` lingers until the ACK of our FIN is received (Figure 7.7), or
3. `shutdown` followed by a `read` waits until we receive the peer's FIN (Figure 7.8).

Another way to know that the peer application has read our data is to use an *application-level acknowledgment* or *application ACK*. For example, the client sends its data to the server and then calls `read` for 1 byte of data:

```
char  ack;

Write(sockfd, data, nbytes);     /* data from client to server */
n = Read(sockfd, &ack, 1);       /* wait for application-level ACK */
```

The server reads the data from the client and then sends back the 1-byte application-level ACK:

```
nbytes = Read(sockfd, buff, sizeof(buff));  /* data from client */
         /* server verifies it received the correct
            amount of data from the client */
Write(sockfd, "", 1);              /* server's ACK back to client */
```

We are guaranteed that when the `read` in the client returns, the server process has read the data that we sent. (This assumes that either the server knows how much data the client is sending, or there is some application-defined end-of-record marker, which we do not show here.) Here the application-level ACK is a byte of 0, but the contents of this byte could be used to signal other conditions from the server to the client. Figure 7.9 shows the possible packet exchange.

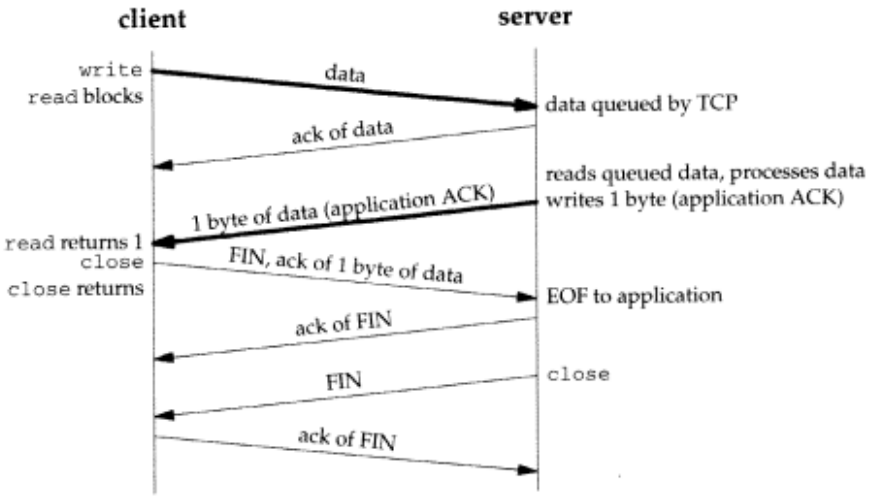

Figure 7.9 Application ACK.

Figure 7.10 summarizes the two possible calls to `shutdown` and the three possible calls to `close`, and the effect on a TCP socket.

Function	Description
`shutdown, SHUT_RD`	No more receives can be issued on socket; process can still send on socket; socket receive buffer discarded; any further data received is discarded by TCP (Exercise 6.5); no effect on socket send buffer.
`shutdown, SHUT_WR`	No more sends can be issued on socket; process can still receive on socket; contents of socket send buffer sent to other end, followed by normal TCP connection termination (FIN); no effect on socket receive buffer.
`close, l_onoff = 0` (default)	No more receives or sends can be issued on socket; contents of socket send buffer sent to other end. If descriptor reference count becomes 0: normal TCP connection termination (FIN) sent following data in send buffer and socket receive buffer discarded.
`close, l_onoff = 1` `l_linger = 0`	No more receives or sends can be issued on socket. If descriptor reference count becomes 0: RST sent to other end, connection state set to CLOSED (no TIME_WAIT state), socket send buffer and socket receive buffer discarded.
`close, l_onoff = 1` `l_linger != 0`	No more receives or sends can be issued on socket; contents of socket send buffer sent to other end. If descriptor reference count becomes 0: normal TCP connection termination (FIN) sent following data in send buffer, socket receive buffer discarded, and if linger time expires before connection CLOSED, `close` returns `EWOULDBLOCK`.

Figure 7.10 Summary of `shutdown` and `SO_LINGER` scenarios.

`SO_OOBINLINE` Socket Option

When this option is set, out-of-band data will be placed in the normal input queue (i.e., inline). When this occurs, the `MSG_OOB` flag to the receive functions cannot be used to read the out-of-band data. We discuss out-of-band data in more detail in Chapter 21.

`SO_RCVBUF` and `SO_SNDBUF` Socket Options

Every socket has a send buffer and a receive buffer. We described the operation of the send buffers with TCP and UDP in Figures 2.11 and 2.12.

The receive buffers are used by TCP and UDP to hold received data until it is read by the application. With TCP, the available room in the socket receive buffer is the window that TCP advertises to the other end. The TCP socket receive buffer cannot overflow because the peer is not allowed to send data beyond the advertised window. This is TCP's flow control and if the peer ignores the advertised window and sends data beyond the window, the receiving TCP discards it. With UDP, however, when a datagram arrives that will not fit in the socket receive buffer, that datagram is discarded. Recall that UDP has no flow control: it is easy for a fast sender to overwhelm a slower receiver, causing datagrams to be discarded by the receiver's UDP, as we show in Section 8.13.

These two socket options let us change the default sizes. The default values differ widely between implementations. Older Berkeley-derived implementations would default the TCP send and receive buffers to 4096 bytes, but newer systems use larger values, anywhere from 8192 to 61440 bytes. The UDP send buffer size often defaults to

a value around 9000 bytes if the host supports NFS, and the UDP receive buffer size often defaults to a value around 40000 bytes.

When setting the size of the TCP socket receive buffer, the ordering of the function calls is important. This is because of TCP's window scale option (Section 2.5), which is exchanged with the peer on the SYN segments when the connection is established. For a client, this means the `SO_RCVBUF` socket option must be set before calling `connect`. For a server, this means the socket option must be set for the listening socket before calling `listen`. Setting this option for the connected socket will have no effect whatsoever on the possible window scale option because `accept` does not return with the connected socket until TCP's three-way handshake is complete. That is why this option must be set for the listening socket. (The sizes of the socket buffers are always inherited from the listening socket by the newly created connected socket: pp. 462–463 of TCPv2.)

The TCP socket buffer sizes should be at least three times the MSS for the connection. If we are dealing with unidirectional data transfer, such as a file transfer in one direction, when we say "socket buffer sizes" we mean the socket send buffer size on the sending host and the socket receive buffer size on the receiving host. For bidirectional data transfer, we mean both socket buffer sizes on the sender and both socket buffer sizes on the receiver. With typical default buffer sizes of 8192 bytes or larger, and a typical MSS of 512 or 1460, this requirement is normally met. The problem has been seen on networks with large MTUs, which then provide a larger than normal MSS (e.g., ATM networks with an MTU of 9188 as described in [Comer and Lin 1994]).

The TCP socket buffer sizes should also be an even multiple of the MSS for the connection. Some implementations handle this detail for the application, rounding up the socket buffer size after the connection is established (p. 902 of TCPv2). This is another reason to set these two socket options before establishing a connection. For example, using the default 4.4BSD sizes of 8192 and assuming an Ethernet with an MSS of 1460, both socket buffers are rounded up to 8760 (6×1460) when the connection is established.

Another consideration in setting the socket buffer sizes deals with performance. Figure 7.11 shows a TCP connection between two endpoints (which we call a *pipe*) with a capacity of eight segments.

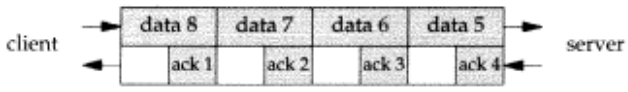

Figure 7.11 TCP connection (pipe) with a capacity of eight segments.

We show four data segments on the top, and four ACKs on the bottom. Even though there are only four segments of data in the pipe, the client must have a send buffer capacity of at least eight segments, because the client TCP must keep a copy of each segment until the ACK is received from the server.

> We are ignoring some details here. First, TCP's slow start algorithm limits the rate at which segments are initially sent on an idle connection. Next, TCP often acknowledges every other segment, not every segment as we show. All these details are covered in Chapters 20 and 24 of TCPv1.

What is important to understand is the concept of the full-duplex pipe, its capacity, and how that relates to the socket buffer sizes on both ends of the connection. The capacity of the pipe is called the *bandwidth-delay product* and we calculate this by multiplying the bandwidth (in bits/sec) times the RTT (round-trip time, in seconds), converting the result from bits to bytes. The RTT is easily measured with the Ping program. The bandwidth is the value corresponding to the slowest link between the two endpoints and must somehow be known. For example, a T1 line (1,536,000 bits/sec) with an RTT of 60 ms gives a bandwidth-delay product of 11,520 bytes. If the socket buffer sizes are less than this, the pipe will not stay full, and the performance will be less than expected. Large socket buffers are required when the bandwidth gets faster (e.g., T3 lines at 45 Mbits/sec) or when the RTT gets large (e.g., satellite links with an RTT around 500 ms). When the bandwidth-delay product exceeds TCP's maximum normal window size (65535 bytes), both endpoints also need the TCP *long fat pipe* options that we mentioned in Section 2.5.

Most implementations have an upper limit for the sizes of the socket send and receive buffers, and sometimes this limit can be modified by the administrator. Older Berkeley-derived implementations had a hard upper limit of around 52,000 bytes, but newer implementations have a default limit of 256,000 bytes or more, and this can usually be increased by the administrator. Unfortunately there is no simple way for an application to determine this limit. Posix.1 defines the `fpathconf` function, which most implementations support, and Posix.1g defines a new constant that can be used as the second argument to this function, `_PC_SOCK_MAXBUF`, which is the maximum size of the socket buffers.

`SO_RCVLOWAT` and `SO_SNDLOWAT` Socket Options

Every socket also has a receive low-water mark and a send low-water mark. These are used by the `select` function, as we described in Section 6.3. These two socket options let us change these two low-water marks.

The receive low-water mark is the amount of data that must be in the socket receive buffer for `select` to return "readable." It defaults to 1 for TCP and UDP sockets. The send low-water mark is the amount of available space that must exist in the socket send buffer for `select` to return "writable." This low-water mark normally defaults to 2048 for TCP sockets. With UDP the low-water mark is used, as we described in Section 6.3, but since the number of bytes of available space in the send buffer for a UDP socket never changes (since UDP does not keep a copy of the datagrams sent by the application), as long as the UDP socket send buffer size is greater than the socket's low-water mark, the UDP socket is always writable. Recall from Figure 2.12 that UDP does not have a send buffer; it has only a send buffer size.

Posix.1g does not require support for these two socket options.

`SO_RCVTIMEO` and `SO_SNDTIMEO` Socket Options

These two socket options allow us to place a timeout on socket receives and sends. Notice that the argument to the two `sockopt` functions is a pointer to a `timeval` structure, the same one used with `select` (Section 6.3). This lets us specify the timeouts in

seconds and microseconds. We disable a timeout by setting its value to 0 seconds and 0 microseconds. Both timeouts are disabled by default.

The receive timeout affects the five input functions: `read`, `readv`, `recv`, `recvfrom`, and `recvmsg`. The send timeout affects the five output functions: `write`, `writev`, `send`, `sendto`, and `sendmsg`. We talk more about socket timeouts in Section 13.2.

> These two socket options and the concept of inherent timeouts on socket receives and sends were added with 4.3BSD Reno. Posix.1g does not require support for these two socket options.
>
> In Berkeley-derived implementations these two values really implement an inactivity timer and not an absolute timer on the read or write system call. Pages 496 and 516 of TCPv2 talk about this in more detail.

`SO_REUSEADDR` and `SO_REUSEPORT` Socket Options

The `SO_REUSEADDR` socket option serves four different purposes.

1. `SO_REUSEADDR` allows a listening server to start and `bind` its well-known port even if previously established connections exist that use this port as their local port. This condition is typically encountered as follows:
   (a) A listening server is started.
   (b) A connection request arrives and a child process is spawned to handle that client.
   (c) The listening server terminates but the child continues to service the client on the existing connection.
   (d) The listening server is restarted.

   By default, when the listening server is restarted in (d) by calling `socket`, `bind`, and `listen`, the call to `bind` fails because the listening server is trying to bind a port that is part of an existing connection (the one being handled by the previously spawned child). But if the server sets the `SO_REUSEADDR` socket option between the calls to `socket` and `bind`, the latter function will succeed. *All* TCP servers should specify this socket option to allow the server to be restarted in this situation.

   > This scenario is one of the most frequently asked questions on Usenet.

2. `SO_REUSEADDR` allows multiple instances of the same server to be started on the same port, as long as each instance binds a different local IP address. This is common for a site hosting multiple HTTP servers using the IP alias technique (Section A.4). Assume the local host's primary IP address is 198.69.10.2 but it has two aliases of 198.69.10.128 and 198.69.10.129. Three HTTP servers are started. The first HTTP server would call `bind` with a local IP address of 198.69.10.128 and a local port of 80 (the well-known port for HTTP). The second

server would `bind` 198.69.10.129 and port 80. But this second call to `bind` fails unless `SO_REUSEADDR` is set before the call. The third server would call `bind` with the wildcard as the local IP address and a local port of 80. Again, `SO_REUSEADDR` is required for this final call to succeed. Assuming `SO_REUSEADDR` is set and the three servers are started, incoming TCP connection requests with a destination IP address of 198.69.10.128 and a destination port of 80 are delivered to the first server, incoming requests with a destination IP address of 198.69.10.129 and a destination port of 80 are delivered to the second server, and all other TCP connection requests with a destination port of 80 are delivered to the third server. This final server handles requests destined for 198.69.10.2 in addition to any other IP aliases that the host may have configured. The wildcard means "everything that doesn't have a better (more specific) match." Note that this scenario of allowing multiple servers for a given service is handled automatically if the server always sets the `SO_REUSEADDR` socket option (as we recommend).

With TCP we are never able to start multiple servers that `bind` the same IP address and the same port: a *completely duplicate binding*. That is, we cannot start one server that binds 198.69.10.2 port 80 and start another that also binds 198.69.10.2 port 80, even if we set the `SO_REUSEADDR` socket option for the second server.

3. `SO_REUSEADDR` allows a single process to bind the same port to multiple sockets, as long as each bind specifies a different local IP address. This is common for UDP servers that need to know the destination IP address of client requests on systems that do not provide the `IP_RECVDSTADDR` socket option, and we develop an example using this technique in Section 19.11. This technique is normally not used with TCP servers since a TCP server can always determine the destination IP address by calling `getsockname` after the connection is established.

4. `SO_REUSEADDR` allows *completely duplicate bindings*: a `bind` of an IP address and port, when that same IP address and port are already bound to another socket. Normally this feature is supported only on systems that support multicasting, when that system does not support the `SO_REUSEPORT` socket option (which we describe shortly), and only for UDP sockets (multicasting does not work with TCP).

   This feature is used with multicasting to allow the same application to be run multiple times on the same host. When a UDP datagram is received for one of these multiply bound sockets, the rule is that if the datagram is destined for either a broadcast address or a multicast address, one copy of the datagram is delivered to each matching socket. But if the datagram is destined for a unicast address, the datagram is delivered to only one socket. If, in the case of a unicast datagram, there are multiple sockets that match the datagram, which one of the sockets will receive the datagram is implementation dependent. Pages 777–779 of TCPv2 talks more about this feature. We talk more about broadcasting and multicasting in Chapters 18 and 19.

Exercises 7.5 and 7.6 show some examples of this socket option.

4.4BSD introduced the SO_REUSEPORT socket option when support for multicasting was added. Instead of overloading SO_REUSEADDR with the desired multicast semantics that allow completely duplicate bindings, this new socket option was introduced with the following semantics:

1. This option allows completely duplicate bindings but only if each socket that wants to bind the same IP address and port specify this socket option.
2. SO_REUSEADDR is considered equivalent to SO_REUSEPORT if the IP address being bound is a multicast address (p. 731 of TCPv2).

The problem with this socket option is that not all systems support it, and on those that do not support the option but do support multicasting, SO_REUSEADDR is used instead of SO_REUSEPORT to allow completely duplicate bindings when it makes sense (i.e., a UDP server that can be run multiple times on the same host at the same time and that expects to receive either broadcast or multicast datagrams).

We can summarize our discussion of these socket options with the following recommendations:

1. Set the SO_REUSEADDR socket option before calling bind in all TCP servers.
2. When writing a multicast application that can be run multiple times on the same host at the same time, set the SO_REUSEADDR socket option and bind the group's multicast address as the local IP address.

Chapter 22 of TCPv2 talks about these two socket options in more detail.

There is a potential security problem with SO_REUSEADDR. If a socket exists that is bound to, say, the wildcard address and port 5555, if we specify SO_REUSEADDR, we can bind that same port to a different IP address, say the primary IP address of the host. Any future datagrams that arrive destined to port 5555 and the IP address that we bound to our socket are delivered to our socket, not to the other socket bound to the wildcard address. These could be TCP SYN segments or UDP datagrams. (Exercise 11.3 shows this feature with UDP.) For most well-known services, HTTP, FTP, and Telnet, for example, this is not a problem because these servers all bind a reserved port. Hence, any process that comes along later and tries to bind a more-specific instance of that port (i.e., steal the port) requires superuser privileges. NFS, however, can be a problem, since its normal port (2049) is not reserved.

> One underlying problem with the sockets API is that the setting of the socket pair is done with two function calls (bind and connect), instead of one. [Torek 1994] proposes a single function that solves this problem:
>
> ```
> int bind_connect_listen(int sockfd,
>                         const struct sockaddr *laddr, int laddrlen,
>                         const struct sockaddr *faddr, int faddrlen,
>                         int listen);
> ```
>
> *laddr* specifies the local IP address and local port, *faddr* specifies the foreign IP address and foreign port, and *listen* specifies a client (0) or a server (nonzero; same as the backlog argument to

`listen`). Then `bind` would be a library function that calls this function with *faddr* a null pointer and *faddrlen* 0, and `connect` would be a library function that calls this function with *laddr* a null pointer and *laddrlen* 0. There are a few applications, notably FTP, that need to specify both the local pair and the foreign pair, and they could call `bind_connect_listen` directly. With such a function, the need for `SO_REUSEADDR` disappears, other than for multicast UDP servers that explicitly need to allow completely duplicate bindings of the same IP address and port. Another benefit of this new function is that a TCP server could restrict itself to servicing connection requests that arrive from one specific IP address and port, something which RFC 793 [Postel 1981c] specifies but is impossible with the existing sockets API.

A similar proposal for a function named `set_addresses` was made in 1993 to the Posix 1003.12 working group by Keith Sklower. The proposal was, however, rejected.

`SO_TYPE` Socket Option

This option returns the socket type. The integer value returned is a value such as `SOCK_STREAM` or `SOCK_DGRAM`. This option is typically used by a process that inherits a socket when it is started.

`SO_USELOOPBACK` Socket Option

This option applies only to sockets in the routing domain (`AF_ROUTE`). This option defaults on for these sockets (the only one of the `SO_`*xxx* socket options that defaults on instead of off). When this option is enabled, the socket receives a copy of everything sent on the socket.

Another way to disable these loopback copies is to call `shutdown` with a second argument of `SHUT_RD`.

7.6 IPv4 Socket Options

These socket options are processed by IPv4 and have a *level* of `IPPROTO_IP`. We defer discussion of the five multicasting socket options until Section 19.5.

All of the socket options that we describe in this section are specified by Posix.1g, with the exception of `IP_RECVIF`.

`IP_HDRINCL` Socket Option

If this option is set for a raw IP socket (Chapter 25), we must build our own IP header for all the datagrams that we send on the raw socket. Normally the kernel builds the IP header for datagrams sent on a raw socket, but there are some applications (notably Traceroute) that build their own IP header to override values that IP would place into certain header fields.

When this option is set, we build a complete IP header, with the following exceptions:

- IP always calculates and stores the IP header checksum.
- If we set the IP identification field to 0, the kernel will set the field.
- If the source IP address is `INADDR_ANY`, IP sets it to the primary IP address of the outgoing interface.
- How to set IP options is implementation dependent. Some implementations take any IP options that were set using the `IP_OPTIONS` socket option and append these to the header that we build, while others require our header to also contain any desired IP options.

We show an example of this option in Section 26.6. Pages 1056–1057 of TCPv2 provide additional details on this socket option.

`IP_OPTIONS` Socket Option

Setting this option allows us to set IP options in the IPv4 header. This requires intimate knowledge of the format of the IP options in the IP header. We discuss this option with regard to IPv4 source routes in Section 24.3.

`IP_RECVDSTADDR` Socket Option

This socket option causes the destination IP address of a received UDP datagram to be returned as ancillary data by `recvmsg`. We show an example of this option in Section 20.2.

`IP_RECVIF` Socket Option

This socket option causes the index of the interface on which a UDP datagram is received to be returned as ancillary data by `recvmsg`. We show an example of this option in Section 20.2.

> This is a new socket option that was developed by Bill Fenner for FreeBSD and NetBSD for the DARTNet testbed [Fenner 1997]. This is an experimental research network used for testing new protocols and applications. The socket option was supposed to be in 4.4BSD, but never made it into the release. The author took the FreeBSD implementation and added it to BSD/OS 3.0.

`IP_TOS` Socket Option

This option lets us set the type-of-service field (Figure A.1) in the IP header for a TCP or UDP socket. If we call `getsockopt` for this option, the current value that would be placed into the TOS field in the IP header (which defaults to 0) is returned. There is no way to fetch the value from a received IP datagram.

We can set the TOS to one of the constants shown in Figure 7.12, which are defined by including `<netinet/ip.h>`.

Constant	Description
IPTOS_LOWDELAY	minimize delay
IPTOS_THROUGHPUT	maximize throughput
IPTOS_RELIABILITY	maximize reliability
IPTOS_LOWCOST	minimize cost

Figure 7.12 IPv4 type-of-service constants.

RFC 1349 [Almquist 1992] contains a detailed description of the TOS field and how this field should be set for the standard Internet applications. For example, Telnet and Rlogin should specify `IPTOS_LOWDELAY` while the data portion of an FTP transfer should specify `IPTOS_THROUGHPUT`.

`IP_TTL` Socket Option

With this option we can set and fetch the default TTL (time-to-live field, Figure A.1) that the system will use for a given socket. 4.4BSD, for example, uses the default of 64 for both TCP and UDP sockets (which is specified in RFC 1700 [Reynolds and Postel 1994]), and 255 for raw sockets. As with the TOS field, calling `getsockopt` returns the default value of the field that the system will use in outgoing datagrams—there is no way to obtain the value from a received datagram. We set this socket option with our Traceroute program in Figure 25.18.

7.7 ICMPv6 Socket Option

This socket option is processed by ICMPv6 and has a *level* of `IPPROTO_ICMPV6`.

`ICMP6_FILTER` Socket Option

This option lets us fetch and set an `icmp6_filter` structure that specifies which of the 256 possible ICMPv6 message types are passed to the process on a raw socket. We discuss this option in Section 25.4.

7.8 IPv6 Socket Options

These socket options are processed by IPv6 and have a *level* of `IPPROTO_IPV6`. We defer discussion of the five multicasting socket options until Section 19.5. We note that many of these options make use of *ancillary data* with the `recvmsg` function, and we describe this in Section 13.6. All the IPv6 socket options are defined in RFC 2133 [Gilligan et al. 1997] and [Stevens and Thomas 1997].

Posix.1g says nothing about IPv6.

IPV6_ADDRFORM Socket Option

This option allows a socket to be converted from IPv4 to IPv6 or vice versa. We describe this option in Section 10.5.

IPV6_CHECKSUM Socket Option

This socket option specifies the byte offset into the user data of where the checksum field is located. If this value is nonnegative, the kernel will (1) compute and store a checksum for all outgoing packets, and (2) verify the received checksum on input, discarding packets with an invalid checksum. This option affects all IPv6 raw sockets other than ICMPv6 raw sockets. (The kernel always calculates and stores the checksum for ICMPv6 raw sockets.) If a value of −1 is specified (the default), the kernel will not calculate and store the checksum for outgoing packets on this raw socket and will not verify the checksum for received packets.

> All protocols that use IPv6 should have a checksum in their own protocol header. These checksums include a pseudoheader (RFC 1883 [Deering and Hinden 1995]) that includes the source IPv6 address as part of the checksum (which differs from all the other protocols that are normally implemented using a raw socket with IPv4). Rather than forcing the application using the raw socket to perform source address selection, the kernel will do this and then calculate and store the checksum incorporating the standard IPv6 pseudoheader.

IPV6_DSTOPTS Socket Option

Setting this option specifies that any received IPv6 destination options are to be returned as ancillary data by `recvmsg`. This option defaults off. We describe the functions that are used to build and process these options in Section 24.5.

IPV6_HOPLIMIT Socket Option

Setting this option specifies that the received hop limit field be returned as ancillary data by `recvmsg`. This option defaults off. We describe this option in Section 20.8.

> There is no way with IPv4 to obtain the received time-to-live field.

IPV6_HOPOPTS Socket Option

Setting this option specifies that any received IPv6 hop-by-hop options are to be returned as ancillary data by `recvmsg`. This option defaults off. We describe the functions that are used to build and process these options in Section 24.5.

IPV6_NEXTHOP

This is not a socket option but the type of an ancillary data object that can be specified to `sendmsg`. This object specifies the next-hop address for a datagram as a socket address structure. We say more about this feature in Section 20.8.

IPV6_PKTINFO Socket Option

Setting this option specifies that the following two pieces of information about a received IPv6 datagram are to be returned as ancillary data by `recvmsg`: the destination IPv6 address and the arriving interface index. We describe this option in Section 20.8.

IPV6_PKTOPTIONS Socket Option

Most of the IPv6 socket options assume a UDP socket with the information being passed between the kernel and the application using ancillary data with `recvmsg` and `sendmsg`. A TCP socket fetches and stores these values using the `IPV6_PKTOPTIONS` socket option.

The buffer pointed to by `getsockopt` and `setsockopt` contains the same information as would be passed using ancillary data using either `recvmsg` or `sendmsg`. We discuss this option in Section 24.7.

IPV6_RTHDR Socket Option

Setting this option specifies that a received IPv6 routing header is to be returned as ancillary data by `recvmsg`. This option defaults off. We describe the functions that are used to build and process an IPv6 routing header in Section 24.6.

IPV6_UNICAST_HOPS Socket Option

This IPv6 option is similar to the IPv4 `IP_TTL` socket option. Setting the socket option specifies the default hop limit for outgoing datagrams sent on the socket, while fetching the socket option returns the value for the hop limit that the kernel will use for the socket. To obtain the actual hop limit field from a received IPv6 datagram requires using the `IPV6_HOPLIMIT` socket option. We set this socket option with our Traceroute program in Figure 25.18.

7.9 TCP Socket Options

There are five socket options for TCP, but three are new with Posix.1g and not widely supported. We specify the *level* as `IPPROTO_TCP`.

TCP_KEEPALIVE Socket Option

This option is new with Posix.1g. It specifies the idle time in seconds for the connection before TCP starts sending keepalive probes. The default value must be at least 7200 seconds, which is 2 hours. This option is effective only when the `SO_KEEPALIVE` socket option is enabled.

TCP_MAXRT Socket Option

This option is new with Posix.1g. It specifies the amount of time in seconds before a connection is broken once TCP starts retransmitting data. A value of 0 means to use the system default, and a value of −1 means to retransmit forever. If a positive value is specified, it may be rounded up to the implementation's next retransmission time.

TCP_MAXSEG Socket Option

This socket option allows us to fetch or set the *maximum segment size* (MSS) for a TCP connection. The value returned is the maximum amount of data that our TCP will send to the other end; often it is the MSS announced by the other end with its SYN, unless our TCP chooses to use a smaller value than the peer's announced MSS. If this value is fetched before the socket is connected, the value returned is the default value that will be used if an MSS option is not received from the other end. Also be aware that a value smaller than the returned value can actually be used for the connection if the timestamp option, for example, is in use, because this option occupies 12 bytes of TCP options in each segment.

The maximum amount of data that our TCP will send per segment can also change during the life of a connection if TCP supports path MTU discovery. If the route to the peer changes, this value can go up or down.

We note in Figure 7.1 that this socket option can also be set by the application. Before 4.4BSD this was not possible: it was a read-only option. 4.4BSD limits the application to *decreasing* the value: we cannot increase the value (p. 1023 of TCPv2). Since this option controls the amount of data that TCP sends per segment, it makes sense to forbid the application from increasing the value. Once the connection is established, this value is the MSS option that was announced by the peer, and we cannot exceed that value. Our TCP, however, can always send less than the peer's announced MSS.

TCP_NODELAY Socket Option

If set, this option disables TCP's *Nagle algorithm* (Section 19.4 of TCPv1 and pp. 858–859 of TCPv2). By default this algorithm is enabled.

The purpose of the Nagle algorithm is to reduce the number of small packets on a WAN. The algorithm states that if a given connection has outstanding data (that is, data that our TCP has sent, and for which it is currently awaiting an acknowledgment), then no small packets will be sent on the connection until the existing data is acknowledged. The definition of a "small" packet is any packet smaller than the MSS. TCP will always send a full-sized packet if possible; the purpose of the Nagle algorithm is to prevent a connection from having multiple small packets outstanding at any time.

The two common generators of small packets are the Rlogin and Telnet clients, since they normally send each keystroke as a separate packet. On a fast LAN we normally do not notice the Nagle algorithm with these clients, because the time required for a small packet to be acknowledged is typically a few milliseconds, far less than the time between two successive characters that we type. But on a WAN, where it can take a

second for a small packet to be acknowledged, we can notice a delay in the character echoing, and this delay is often exaggerated by the Nagle algorithm.

Consider the following example. We type the six-character string "hello!" to either the Rlogin or Telnet client, with exactly 250 ms between each character. The RTT to the server is 600 ms and the server immediately sends back the echo of the character. We assume the ACK of the client's character is sent back to the client along with the character echo and we ignore the ACKs that the client sends for the server's echo. (We talk about delayed ACKs shortly.) Assuming the Nagle algorithm is disabled, we have the 12 packets shown in Figure 7.13.

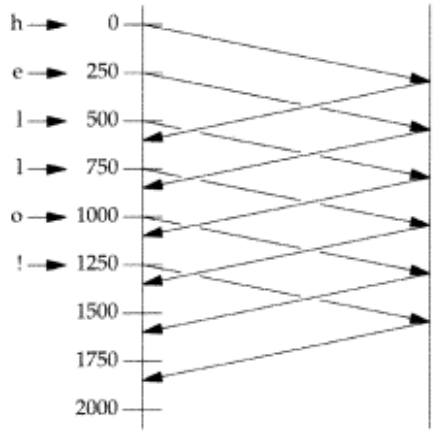

Figure 7.13 Six characters echoed by server with Nagle algorithm disabled.

Each character is sent in a packet by itself: the data segments from the left to right, and the ACKs from the right to left.

But if the Nagle algorithm is enabled (the default), we have the eight packets shown in Figure 7.14. The first character is sent as a packet by itself, but the next two characters are not sent, since the connection has a small packet outstanding. At time 600, when the ACK of the first packet is received, along with the echo of the first character, these two characters are sent. Until this packet is ACKed at time 1200, no more small packets are sent.

The Nagle algorithm often interacts with another TCP algorithm: the *delayed ACK* algorithm. This algorithm causes TCP to not send an ACK immediately when it receives data; instead TCP will wait some small amount of time (typically 50–200 ms) and only then send the ACK. The hope is that in this small amount of time there will be data to send back to the peer, and the ACK can piggyback with the data, saving one TCP segment. This is normally the case with the Rlogin and Telnet clients, because the servers typically echo each character sent by the client, so the ACK of the client's character piggybacks with the server's echo of that character.

The problem is with other clients whose servers do not generate traffic in the reverse direction on which ACKs can piggyback. These clients can detect noticeable

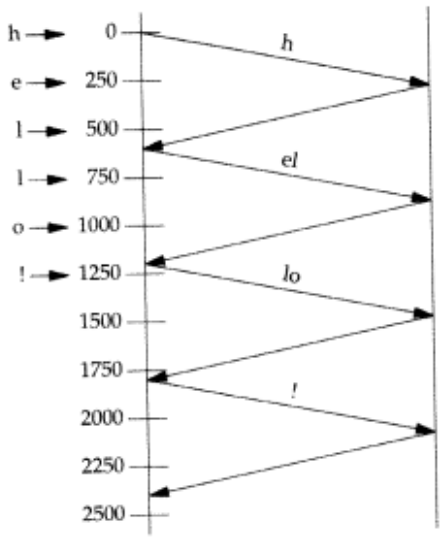

Figure 7.14 Six characters echoed by server with Nagle algorithm enabled.

delays because the client TCP will not send any data to the server until the server's delayed-ACK timer expires. These clients need a way to disable the Nagle algorithm, hence the `TCP_NODELAY` option.

Another type of client that interacts badly with the Nagle algorithm and TCP's delayed ACKs is a client that sends a single logical request to its server in small pieces. For example, assume a client sends a 400-byte request to its server, but this is a 4-byte request type followed by 396 bytes of request data. If the client performs a 4-byte `write` followed by a 396-byte `write`, the second write will not be sent by the client TCP until the server TCP acknowledges the 4-byte write. Also, since the server application cannot operate on the 4 bytes of data until it receives the remaining 396 bytes of data, the server TCP will delay the ACK of the 4 bytes of data (i.e., there will not be any data from the server to the client on which to piggyback the ACK). There are three ways to fix this type of client.

1. Use `writev` (Section 13.4) instead of two calls to `write`. A single call to `writev` ends up with one call to TCP output, instead of two calls, resulting in one TCP segment for our example. This is the preferred solution.
2. Copy the 4 bytes of data and the 396 bytes of data into a single buffer and call `write` once for this buffer.
3. Set the `TCP_NODELAY` socket option and continue to call `write` two times. This is the least desirable solution.

Exercises 7.8 and 7.9 continue this example.

TCP_STDURG Socket Option

This option is new with Posix.1g and it affects the interpretation of TCP's urgent pointer (which we encounter with out-of-band data in Chapter 21). There are two possible interpretations about where TCP's urgent pointer points (pp. 292–293 of TCPv1). By default the urgent pointer points to the data byte following the byte sent with the MSG_OOB flag. This is how almost all implementations interoperate today. But if this socket option is set nonzero, the urgent pointer will point to the data byte sent with the MSG_OOB flag.

> This socket option should never need to be set, and it is questionable why Posix.1g even defines it.

7.10 fcntl Function

fcntl stands for "file control" and this function performs various descriptor control operations. Before describing the function, and how it affects a socket, we need to look at the bigger picture. Figure 7.15 summarizes the different operations performed by fcntl, ioctl, and routing sockets.

Operation	fcntl	ioctl	Routing socket	Posix.1g
set socket for nonblocking I/O	F_SETFL, O_NONBLOCK	FIONBIO		fcntl
set socket for signal-driven I/O	F_SETFL, O_ASYNC	FIOASYNC		fcntl
set socket owner	F_SETOWN	SIOCSPGRP or FIOSETOWN		fcntl
get socket owner	F_GETOWN	SIOCGPGRP or FIOGETOWN		fcntl
get #bytes in socket receive buffer		FIONREAD		
test for socket at out-of-band mark		SIOCATMARK		sockatmark
obtain interface list		SIOCGIFCONF	sysctl	
interface operations		SIOC[GS]IF*xxx*		
ARP cache operations		SIOC*x*ARP	RTM_*xxx*	
routing table operations		SIOC*xxx*RT	RTM_*xxx*	

Figure 7.15 Summary of fcntl, ioctl, and routing socket operations.

The first six operations can be applied to sockets by any process, while many of the latter four (interface, ARP, and routing table) are issued by administrative programs such as ifconfig and route. We talk more about the various ioctl operations in Chapter 16 and routing sockets in Chapter 17.

There are multiple ways to perform the first four operations, but we note in the final column that Posix.1g specifies that fcntl is the preferred way. We also note that Posix.1g provides the sockatmark function (Section 21.3) as the preferred way to test for the out-of-band mark. The remaining operations, with a blank final column, have not been standardized by Posix.

We also note that the first two operations, setting a socket for nonblocking I/O and for signal-driven I/O, have been set historically using the FNDELAY and FASYNC commands with fcntl. Posix defines the O_*xxx* constants.

The fcntl function provides the following features related to network programming:

- Nonblocking I/O. We can set the O_NONBLOCK file status flag using the F_SETFL command to set a socket nonblocking. We describe nonblocking I/O in Chapter 15.
- Signal-driven I/O. We can set the O_ASYNC file status flag using the F_SETFL command which causes the SIGIO signal to be generated when the status of a socket changes. We discuss this in Chapter 22.

  This flag is new with Posix.1g.

- The F_SETOWN command lets us set the socket owner (the process ID or process group ID) to receive the SIGIO and SIGURG signals. The former signal is generated when signal-driven I/O is enabled for a socket (Chapter 22) and the latter is generated when new out-of-band data arrives for a socket (Chapter 21). The F_GETOWN command returns the current owner of the socket.

  The term "socket owner" is new with Posix.1g. Historically Berkeley-derived implementations have called this "the process group ID of the socket" because the variable that stores this ID is the so_pgid member of the socket structure (p. 438 of TCPv2).

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* int arg */ );
```

Returns: depends on *cmd* if OK, –1 on error

Each descriptor (including a socket) has a set of file flags that are fetched with the F_GETFL command and set with the F_SETFL command. The two flags that affect a socket are

O_NONBLOCK	nonblocking I/O
O_ASYNC	signal-driven I/O notification

We describe both of these features in more detail later. For now we note that typical code to enable nonblocking I/O, using fcntl, would be:

```
int    flags;

    /* Set socket nonblocking */
if ( (flags = fcntl(fd, F_GETFL, 0)) < 0)
    err_sys("F_GETFL error");
flags |= O_NONBLOCK;
if (fcntl(fd, F_SETFL, flags) < 0)
    err_sys("F_SETFL error");
```

Beware of code that you may encounter that simply sets the desired flag:

```
        /* Wrong way to set socket nonblocking */
    if (fcntl(fd, F_SETFL, O_NONBLOCK) < 0)
        err_sys("F_SETFL error");
```

While this sets the nonblocking flag, it also clears all the other file status flags. The only correct way to set one of the file status flags is to fetch the current flags, logically OR in the new flag, and then set the flags.

The following code turns off the nonblocking flag, assuming `flags` was set by the call to `fcntl` shown above:

```
flags &= ~O_NONBLOCK;
if (fcntl(fd, F_SETFL, flags) < 0)
    err_sys("F_SETFL error");
```

The two signals `SIGIO` and `SIGURG` are different from other signals in that these two are generated for a socket only if the socket has been assigned an owner with the `F_SETOWN` command. The integer *arg* value for the `F_SETOWN` command can be either a positive integer, specifying the process ID to receive the signal, or a negative integer whose absolute value is the process group ID to receive the signal. The `F_GETOWN` command returns the socket owner as the return value from the `fcntl` function, either the process ID (a positive return value) or the process group ID (a negative value other than −1). The difference between specifying a process or a process group to receive the signal is that the former causes only a single process to receive the signal, while the latter causes all processes in the process group (perhaps more than one) to receive the signal.

SVR4 only allows the socket owner to be set to a process ID and not to a process group ID.

When a new socket is created by `socket`, it has no owner. But when a new socket is created from a listening socket, the socket owner is inherited from the listening socket by the connected socket (as are many socket options, pp. 462–463 of TCPv2).

7.11 Summary

Socket options run the gamut from the very general (`SO_ERROR`) to the very specific (IP header options). The most commonly used options that we might encounter are `SO_KEEPALIVE`, `SO_RCVBUF`, `SO_SNDBUF`, and `SO_REUSEADDR`. The latter should always be set for a TCP server before it calls `bind` (Figure 11.8). The `SO_BROADCAST` option and the 10 multicast socket options are only for applications that broadcast or multicast, respectively.

The `SO_KEEPALIVE` socket option is set by many TCP servers and automatically terminates a half-open connection. The nice feature of this option is that it is handled by the TCP layer, without requiring an application-level inactivity timer, but its downside is that it cannot tell the difference between a crashed client and a temporary loss of connectivity to the client.

The `SO_LINGER` socket option gives us more control over when `close` returns and also lets us force an RST to be sent instead of TCP's four-packet connection termination

sequence. We must be careful sending RSTs, because this avoids TCP's TIME_WAIT state. There are also instances where this socket option does not provide the information that we need, in which case an application-level ACK is required.

Every TCP socket has a send buffer and a receive buffer, and every UDP socket has a receive buffer. The `SO_SNDBUF` and `SO_RCVBUF` socket options let us change the sizes of these buffers. The most common use of these options is for bulk data transfer across long fat pipes: TCP connections with either a high bandwidth or a long delay, often using the RFC 1323 extensions. UDP sockets, on the other hand, might want to increase the size of the receive buffer to allow the kernel to queue more datagrams if the application is busy.

Exercises

7.1 Write a program that prints the default TCP and UDP send and receive buffer sizes and run it on the systems to which you have access.

7.2 Modify Figure 1.5 as follows. Before calling `connect`, call `getsockopt` to obtain the socket receive buffer size and the MSS. Print both values. After `connect` returns success, fetch these same two socket options and print their values. Have the values changed? Why? Run the program connecting to a server on your local network and also run the program connecting to a server on a remote network. Does the MSS change? Why? You should also run the program on any different hosts to which you have access.

7.3 Start with our TCP server from Figures 5.2 and 5.3 and our TCP client from Figures 5.4 and 5.5. Modify the client `main` function to set the `SO_LINGER` socket option before calling `exit`, setting `l_onoff` to 1 and `l_linger` to 0. Start the server and then start the client. Type in a line or two at the client to verify the operation, and then terminate the client by entering your end-of-file character. What happens? After you terminate the client, run `netstat` on the client host and see if the socket goes through the TIME_WAIT state.

7.4 Assume two TCP clients start at about the same time. Both set the `SO_REUSEADDR` socket option and then call `bind` with the same local IP address and the same local port (say 1500). But one client `connects` to 198.69.10.2 port 7000 and the second `connects` to 198.69.10.2 (same peer IP address) but port 8000. Describe the race condition that occurs.

7.5 Obtain the source code for the examples in this book (see the Preface) and compile the `sock` program (Section C.3). First classify your host as (1) no multicast support, (2) multicast support but `SO_REUSEPORT` not provided, or (3) multicast support and `SO_REUSEPORT` provided. Try to start multiple instances of the `sock` program as a TCP server (`-s` command-line option) on the same port, binding the wildcard address, one of your host's interface addresses, and the loopback address. Do you need to specify the `SO_REUSEADDR` option (the `-A` command-line option)? Use `netstat` to see the listening sockets.

7.6 Continue the previous example, but start a UDP server (`-u` command-line option) and try to start two instances, both binding the same local IP address and port. If your implementation supports `SO_REUSEPORT`, try using it (`-T` command-line option).

7.7 Many versions of the Ping program have a `-d` flag to enable the `SO_DEBUG` socket option. What does this do?

7.8 Continuing the example at the end of our discussion of the `TCP_NODELAY` socket option, assume that a client performs two `writes`: the first of 4 bytes and the second of 396 bytes. Also assume that the server's delayed-ACK time is 100 ms, the RTT between the client and server is 100 ms, and the server's processing time for the client's request is 50 ms. Draw a time line that shows the interaction of the Nagle algorithm with delayed-ACKs.

7.9 Redo the previous exercise, assuming the `TCP_NODELAY` socket option is set.

7.10 Redo Exercise 7.8 assuming the process calls `writev` one time, for both the 4-byte buffer and the 396-byte buffer.

7.11 Read RFC 1122 [Braden 1989] to determine the recommended interval for delayed ACKs.

7.12 Where does our server in Figures 5.2 and 5.3 spend most of its time? If the server sets the `SO_KEEPALIVE` socket option, there is no data being exchanged across the connection, and the client host crashes and does not reboot, what happens?

7.13 Where does our client in Figures 5.4 and 5.5 spend most of its time? If the client sets the `SO_KEEPALIVE` socket option, there is no data being exchanged across the connection, and the server host crashes and does not reboot, what happens?

7.14 Where does our client in Figures 5.4 and 6.13 spend most of its time? If the client sets the `SO_KEEPALIVE` socket option, there is no data being exchanged across the connection, and the server host crashes and does not reboot, what happens?

7.15 Assume both a client and server set the `SO_KEEPALIVE` socket option. Connectivity is maintained between the two peers but there is no application data exchanged across the connection. When the keepalive timer expires every 2 hours, how many TCP segments are exchanged across the connection?

7.16 Almost all implementations define the constant `SO_ACCEPTCON` in the `<sys/socket.h>` header, but we have not described this option. Read [Lanciani 1996] to find out why this option exists.

8

Elementary UDP Sockets

8.1 Introduction

There are some fundamental differences between applications written using TCP versus those that use UDP. These are because of the differences in the two transport layers: UDP is a connectionless, unreliable, datagram protocol, quite unlike the connection-oriented, reliable byte stream provided by TCP. Nevertheless, there are instances when it makes sense to use UDP instead of TCP and we go over this design choice in Section 20.4. Some popular applications are built using UDP: DNS (the Domain Name System), NFS (the Network File System), and SNMP (Simple Network Management Protocol), for example.

Figure 8.1 shows the function calls for a typical UDP client–server. The client does not establish a connection with the server. Instead, the client just sends a datagram to the server using the `sendto` function (described in the next section), which requires the address of the destination (the server) as a parameter. Similarly, the server does not accept a connection from a client. Instead, the server just calls the `recvfrom` function, which waits until data arrives from some client. `recvfrom` returns the protocol address of the client, along with the datagram, so the server can send a response to the correct client.

Figure 8.1 shows a time line of the typical scenario that takes place for a UDP client–server exchange. We can compare this to the typical TCP exchange, Figure 4.1.

In this chapter we describe the new functions that we use with UDP sockets, `recvfrom` and `sendto`, and redo our echo client–server to use UDP. We also describe the use of the `connect` function with a UDP socket, and the concept of asynchronous errors.

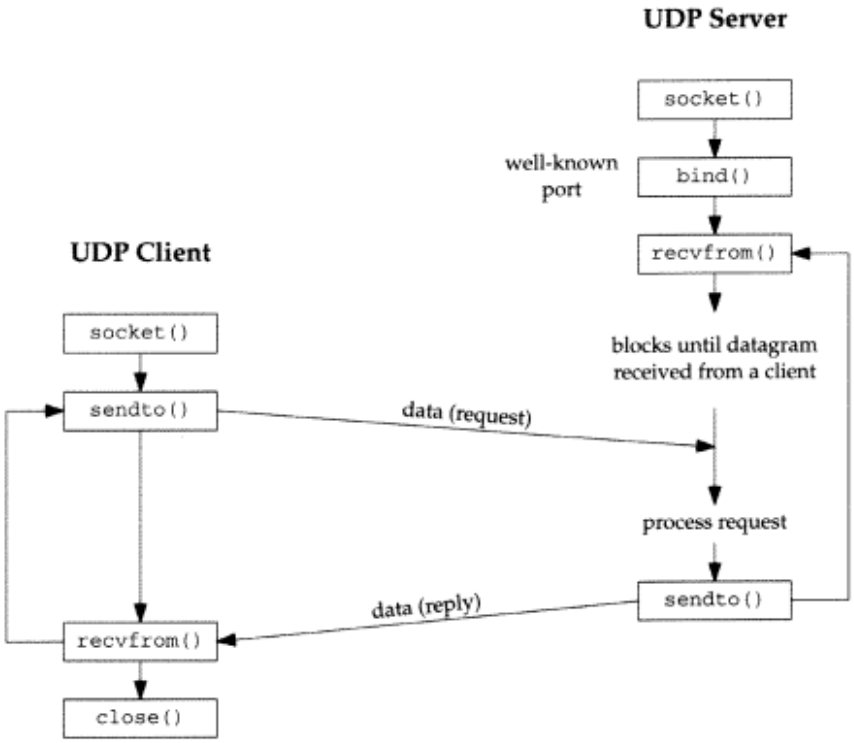

Figure 8.1 Socket functions for UDP client–server.

8.2 `recvfrom` and `sendto` Functions

These two functions are similar to the standard `read` and `write` functions, but three additional arguments are required.

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

ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags,
                 struct sockaddr *from, socklen_t *addrlen);

ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags,
               const struct sockaddr *to, socklen_t addrlen);
```

Both return: number of bytes read or written if OK, –1 on error

The first three arguments, *sockfd*, *buff*, and *nbytes*, are identical to the first three arguments for `read` and `write`: descriptor, pointer to buffer to read into or write from, and number of bytes to read or write.

We describe the *flags* argument in Chapter 13 when we discuss the `recv`, `send`, `recvmsg`, and `sendmsg` functions, since we do not need them with our simple UDP client–server example in this chapter. For now we always set the *flags* to 0.

The *to* argument for `sendto` is a socket address structure containing the protocol address (e.g., IP address and port number) of where the data is to be sent. The size of this socket address structure is specified by *addrlen*. The `recvfrom` function fills in the socket address structure pointed to by *from* with the protocol address of who sent the datagram. The number of bytes stored in this socket address structure is also returned to the caller in the integer pointed to by *addrlen*. Note that the final argument to `sendto` is an integer value, while the final argument to `recvfrom` is a pointer to an integer value (a value–result argument).

The final two arguments to `recvfrom` are similar to the final two arguments to `accept`: the contents of the socket address structure upon return tell us who sent the datagram (in the case of UDP) or who initiated the connection (in the case of TCP). The final two arguments to `sendto` are similar to the final two arguments to `connect`: we fill in the socket address structure with the protocol address of where to send the datagram (in the case of UDP) or with whom to establish a connection (in the case of TCP).

Both functions return the length of the data that was read or written as the value of the function. In the typical use of `recvfrom`, with a datagram protocol, the return value is the amount of user data in the datagram that was received.

Writing a datagram of length 0 is OK. In the case of UDP, this results in an IP datagram containing an IP header (normally 20 bytes for IPv4 and 40 bytes for IPv6), an 8-byte UDP header, and no data. This also means that a return value of 0 from `recvfrom` is OK for a datagram protocol: it does not mean that the peer has closed the connection, as does a return value of 0 from `read` on a TCP socket. Since UDP is connectionless, there is no such thing as closing a UDP connection.

If the *from* argument to `recvfrom` is a null pointer, then the corresponding length argument (*addrlen*) must also be a null pointer, and this indicates that we are not interested in knowing the protocol address of who sent us data.

Both `recvfrom` and `sendto` can be used with TCP, although there is normally no reason to do so.

T/TCP, TCP for Transactions, uses `sendto` as we describe in Section 13.9.

8.3 UDP Echo Server: `main` Function

We now redo our simple echo client–server from Chapter 5 using UDP. Our UDP client and server programs follow the function call flow that we diagramed in Figure 8.1. Figure 8.2 depicts the functions that are used. Figure 8.3 shows the server `main` function.

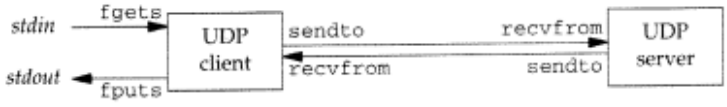

Figure 8.2 Simple echo client–server using UDP.

udpcliserv/udpserv01.c

```
 1 #include    "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd;
 6     struct sockaddr_in servaddr, cliaddr;
 7     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
 8     bzero(&servaddr, sizeof(servaddr));
 9     servaddr.sin_family = AF_INET;
10     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
11     servaddr.sin_port = htons(SERV_PORT);
12     Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));
13     dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
14 }
```

udpcliserv/udpserv01.c

Figure 8.3 UDP echo server.

Create UDP socket, bind server's well-known port

7–12 We create a UDP socket by specifying the second argument to `socket` as `SOCK_DGRAM` (a datagram socket in the IPv4 protocol). As with the TCP server example, the IPv4 address for the `bind` is specified as `INADDR_ANY` and the server's well-known port is the constant `SERV_PORT` from the `unp.h` header.

13 The function `dg_echo` is then called to perform the server processing.

8.4 UDP Echo Server: `dg_echo` Function

Figure 8.4 shows the `dg_echo` function.

lib/dg_echo.c

```
 1 #include    "unp.h"
 2 void
 3 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
 4 {
 5     int     n;
 6     socklen_t len;
 7     char    mesg[MAXLINE];
 8     for ( ; ; ) {
 9         len = clilen;
10         n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
11         Sendto(sockfd, mesg, n, 0, pcliaddr, len);
12     }
13 }
```

lib/dg_echo.c

Figure 8.4 dg_echo function: echo lines on a datagram socket.

Read datagram, echo back to sender

8-12 This function is a simple loop that reads the next datagram arriving at the server's port using `recvfrom` and sends it back using `sendto`.

Despite the simplicity of this function, there are numerous details to consider. First, this function never terminates. Since UDP is a connectionless protocol, there is nothing like an end-of-file as we have with TCP.

Next, this function provides an *iterative server*, not a concurrent server as we had with TCP. There is no call to `fork`, so a single server process handles any and all clients. In general, most TCP servers are concurrent and most UDP servers are iterative.

There is implied queueing taking place in the UDP layer for this socket. Indeed, each UDP socket has a receive buffer and each datagram that arrives for this socket is placed in that socket receive buffer. When the process calls `recvfrom`, the next datagram from the buffer is returned to the process in a FIFO (first-in, first-out) order. This way, if multiple datagrams arrive for the socket before the process can read what's already queued for the socket, the arriving datagrams are just added to the socket receive buffer. But this buffer has a limited size. We discussed this size, and how to increase it, with the `SO_RCVBUF` socket option in Section 7.5.

Figure 8.5 summarizes our TCP client–server from Chapter 5 when two clients establish connections with the server.

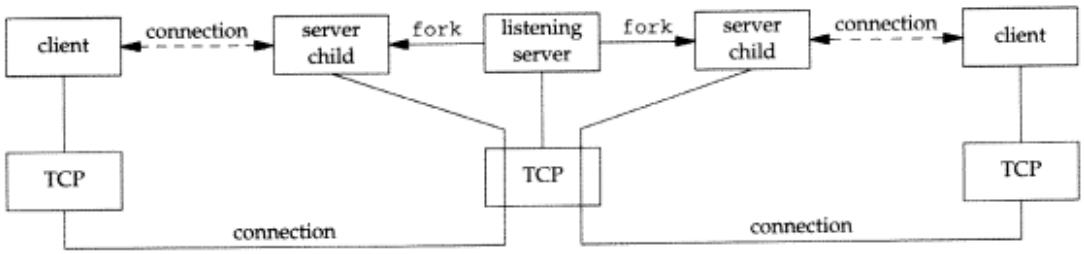

Figure 8.5 Summary of TCP client–server with two clients.

There are two connected sockets and each of the two connected sockets on the server host has its own socket receive buffer.

Figure 8.6 shows the scenario when two clients send datagrams to our UDP server.

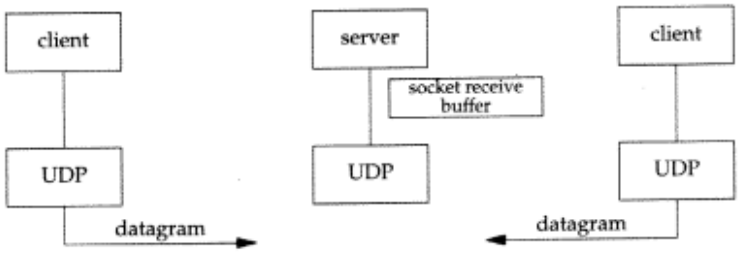

Figure 8.6 Summary of UDP client–server with two clients.

There is only one server process and it has a single socket on which it receives all arriving datagrams and sends all responses. That socket has a receive buffer into which all arriving datagrams are placed.

The `main` function in Figure 8.3 is *protocol dependent* (it creates a socket of protocol `AF_INET` and allocates and initializes an IPv4 socket address structure), but the `dg_echo` function is *protocol independent*. The reason `dg_echo` is protocol independent is because the caller (the `main` function in our case) must allocate a socket address structure of the correct size, and a pointer to this structure, along with its size, are passed as arguments to `dg_echo`. The function `dg_echo` never looks inside this protocol-dependent structure: it simply passes a pointer to the structure to `recvfrom` and `sendto`. It is `recvfrom` that fills in this structure with the IP address and port number of the client, and since the same pointer (`pcliaddr`) is then passed to `sendto` as the destination address, this is how the datagram is echoed back to the client that sent the datagram.

8.5 UDP Echo Client: `main` Function

The UDP client `main` function is shown in Figure 8.7.

udpcliserv/udpcli01.c

```
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd;
 6     struct sockaddr_in servaddr;

 7     if (argc != 2)
 8         err_quit("usage: udpcli <IPaddress>");

 9     bzero(&servaddr, sizeof(servaddr));
10     servaddr.sin_family = AF_INET;
11     servaddr.sin_port = htons(SERV_PORT);
12     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

13     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

14     dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr));

15     exit(0);
16 }
```

udpcliserv/udpcli01.c

Figure 8.7 UDP echo client.

Fill in socket address structure with server's address

9-12 An IPv4 socket address structure is filled in with the IP address and port number of the server. This structure will be passed to `dg_cli`, specifying where to send the datagrams.

13-14 A UDP socket is created and the function `dg_cli` is called.

8.6 UDP Echo Client: dg_cli Function

Figure 8.8 shows the function dg_cli, which performs most of the client processing.

lib/dg_cli.c

```
 1 #include    "unp.h"

 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5     int     n;
 6     char    sendline[MAXLINE], recvline[MAXLINE + 1];

 7     while (Fgets(sendline, MAXLINE, fp) != NULL) {

 8         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

 9         n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);

10         recvline[n] = 0;        /* null terminate */
11         Fputs(recvline, stdout);
12     }
13 }
```

lib/dg_cli.c

Figure 8.8 dg_cli function: client processing loop.

7–12 There are four steps in the client processing loop: read a line from standard input using fgets, send the line to the server using sendto, read back the server's echo using recvfrom, and print the echoed line to standard output using fputs.

Our client has not asked the kernel to assign an ephemeral port to its socket. (With a TCP client we said the call to connect is where this takes place.) With a UDP socket, the first time the process calls sendto, if the socket has not yet had a local port bound to it, that is when an ephemeral port is chosen by the kernel for the socket. As with TCP, the client can call bind explicitly, but this is rarely done.

Notice that the call to recvfrom specifies a null pointer as the fifth and sixth arguments. This tells the kernel that we are not interested in knowing who sent the reply. There is a risk that any process, on either the same host or some other host, can send a datagram to the client's IP address and port, and that datagram will be read by the client who will think it is the server's reply. We will address this in Section 8.8.

As with the server function dg_echo, the client function dg_cli is protocol independent, but the client main function is protocol dependent. The main function allocates and initializes a socket address structure of some protocol type and then passes a pointer to this structure along with its size to dg_cli.

8.7 Lost Datagrams

Our UDP client–server example is not reliable. If a client datagram is lost (say it is discarded by some router between the client and server), the client will block forever in its call to recvfrom in the function dg_cli, waiting for a server reply that will never arrive. Similarly, if the client datagram arrives at the server but the server's reply is lost,

the client will again block forever in its call to `recvfrom`. The only way to prevent this is to place a timeout on the client's call to `recvfrom`. We discuss this in Section 13.2.

Just placing a timeout on the `recvfrom` is not the entire solution. For example, if we do time out, we cannot tell whether our datagram never made it to the server, or if the server's reply never made it back. If the client's request were something like "transfer a certain amount of money from account A to account B" (instead of our simple echo server), it makes a big difference whether the request is lost or whether the reply is lost. We talk more about adding reliability to a UDP client–server in Section 20.5.

8.8 Verifying Received Response

At the end of Section 8.6 we mentioned that any process that knows the client's ephemeral port number could send datagrams to our client, and these will be intermixed with the normal server replies. What we can do is change the call to `recvfrom` in Figure 8.8 to return the IP address and port of who sent the reply and ignore any received datagrams that are not from the server to whom we sent the datagram. There are a few pitfalls with this, however, as we will see.

First, we change the client `main` function (Figure 8.7) to use the standard echo server (Figure 2.13). We just replace the assignment

```
servaddr.sin_port = htons(SERV_PORT);
```

with

```
servaddr.sin_port = htons(7);
```

We do this so we can use any host running the standard echo server with our client.

We then recode the `dg_cli` function to allocate another socket address structure to hold the structure returned by `recvfrom`. We show this in Figure 8.9.

Allocate another socket address structure

9 We allocate another socket address structure by calling `malloc`. Notice that the `dg_cli` function is still protocol independent; as we do not care what type of socket address structure we are dealing with, we use only its size in the call to `malloc`.

Compare returned address

12–18 In the call to `recvfrom` we tell the kernel to return the address of the sender of the datagram. We first compare the length returned by `recvfrom` in the value–result argument and then compare the socket address structures themselves using `memcmp`.

This new version of our client works fine if the server is on a host with just a single IP address. But this program can fail if the server is multihomed. We run this program to our host `bsdi`, which has two interfaces and two IP addresses.

```
solaris % host bsdi
bsdi.kohala.com has address 206.62.226.35
bsdi.kohala.com has address 206.62.226.66
```

udpcliserv/dgcliaddr.c

```
 1 #include    "unp.h"

 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5     int     n;
 6     char    sendline[MAXLINE], recvline[MAXLINE + 1];
 7     socklen_t len;
 8     struct sockaddr *preply_addr;

 9     preply_addr = Malloc(servlen);

10     while (Fgets(sendline, MAXLINE, fp) != NULL) {

11         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

12         len = servlen;
13         n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
14         if (len != servlen || memcmp(pservaddr, preply_addr, len) != 0) {
15             printf("reply from %s (ignored)\n",
16                     Sock_ntop(preply_addr, len));
17             continue;
18         }
19         recvline[n] = 0;        /* null terminate */
20         Fputs(recvline, stdout);
21     }
22 }
```

udpcliserv/dgcliaddr.c

Figure 8.9 Version of `dg_cli` that verifies returned socket address.

```
solaris % udpcli02 206.62.226.66
hello
reply from 206.62.226.35.7 (ignored)
goodbye
reply from 206.62.226.35.7 (ignored)
```

From Figure 1.16 we see that we specified the IP address that does not share the same subnet as the client.

> This is normally allowed. Most IP implementations accept an arriving IP datagram that is destined for *any* of the host's IP addresses, regardless of the interface on which the datagram arrives (pp. 217–219 of TCPv2). RFC 1122 [Braden 1989] calls this the *weak end system model*. If a system were to implement what this RFC calls the *strong end system model*, it would accept an arriving datagram only if that datagram arrives on the interface to which it is addressed.

The IP address returned by `recvfrom` (the source IP address of the UDP datagram) is not the IP address to which we sent the datagram. When the server sends its reply, the destination IP address is 206.62.226.33. The routing function within the kernel on `bsdi` chooses 206.62.226.35 as the outgoing interface. Since the server has not bound an IP address to its socket (the server has bound the wildcard address to its socket, something we can verify by running `netstat` on `bsdi`), the kernel chooses the source

address for the IP datagram. It is chosen to be the primary IP address of the outgoing interface (pp. 232–233 of TCPv2). Also, since it is the primary IP address of the interface, if we send our datagram to a nonprimary IP address of the interface (i.e., an alias), this will also cause our test in Figure 8.9 to fail.

One solution is for the client to verify the responding host's domain name instead of its IP address by looking up the server's name in the DNS (Chapter 9), given the IP address returned by `recvfrom`. Another solution is for the UDP server to create one socket for every IP address that is configured on the host, `bind` that IP address to the socket, use `select` across all these sockets (waiting for any one to become readable), and then reply from the socket that is readable. Since the socket used for the reply was bound to the IP address that was the destination address of the client's request (or the datagram would not have been delivered to the socket), this guarantees that the source address of the reply is the same as the destination address of the request. We show examples of this in Sections 19.11 and 20.6.

> On a multihomed Solaris system, the source IP address for the server's reply is the destination IP address of the client's request. The scenario described in this section is for Berkeley-derived implementations that choose the source IP address based on the outgoing interface.

8.9 Server Not Running

The next scenario to examine is if we start the client without starting the server. If we do so and type in a single line to the client, nothing happens. The client blocks forever in its call to `recvfrom`, waiting for a server reply that will never appear. But this is an example where we need to understand more about the underlying protocols to understand what is happening to our networking application.

First we start `tcpdump` on the host `bsdi` and then we start the client on the same host, specifying the host `solaris` as the server host. We then type a single line, but the line is not echoed.

```
bsdi % udpcli01 206.62.226.33
hello, world                          we type this line
                                      but nothing is echoed back
```

Figure 8.10 shows the `tcpdump` output.

```
1  0.0                  arp who-has solaris tell bsdi
2  0.002526 (0.0025)    arp reply solaris is-at 8:0:20:78:e3:e3
3  0.002932 (0.0004)    bsdi.1105 > solaris.9877: udp 13
4  0.006932 (0.0040)    solaris > bsdi: icmp: solaris udp port 9877 unreachable
```

Figure 8.10 `tcpdump` output when server process not started on server host.

First we notice that an ARP request and reply are needed before the client host can send the UDP datagram to the server host. (We left this exchange in the output to reiterate the potential for an ARP request–reply before an IP datagram can be sent to another host or router on the local network.)

In line 3 we see the client datagram sent but the server host responds in line 4 with an ICMP port unreachable. (The length of 13 accounts for the 12 characters and the newline.) This ICMP error, however, is not returned to the client process, for reasons that we describe shortly. Instead, the client blocks forever in the call to `recvfrom` in Figure 8.8. We also note that ICMPv6 has a "port unreachable" error, similar to ICMPv4 (Figures A.15 and A.16), so the results described here are similar for IPv6.

We call this ICMP error an *asynchronous error*. The error was caused by the `sendto`, but `sendto` returned OK. Recall from Section 2.9 that an OK return from a UDP output operation only means there was room for the resulting IP datagram on the interface output queue. The ICMP error is not returned until later (4 ms later in Figure 8.10), which is why it is called asynchronous.

The basic rule is that asynchronous errors are not returned for UDP sockets unless the socket has been connected. We describe how to call `connect` for a UDP socket in Section 8.11. Why this design decision was made when sockets were first implemented is rarely understood. (The implementation implications are discussed on pp. 748–749 of TCPv2.) Consider a UDP client that sends three datagrams in a row to three different servers (i.e., three different IP addresses) on a single UDP socket. The client then enters a loop that calls `recvfrom` to read the replies. Two of the datagrams are correctly delivered (that is, the server was running on two of the three hosts) but the third host was not running the server. This third host responds with an ICMP port unreachable. This ICMP error message contains the IP header and the UDP header of the datagram that caused the error. (ICMPv4 and ICMPv6 error messages always contain the IP header and all of the UDP header or part of the TCP header to allow the receiver of the ICMP error to determine which socket caused the error. We show this in Figures 25.20 and 25.21.) The client that sent the three datagrams needs to know the destination of the datagram that caused the error to distinguish which of the three datagrams caused the error. But how can the kernel return this information to the process? The only piece of information that `recvfrom` can return is an `errno` value; `recvfrom` has no way of returning the destination IP address and destination UDP port number of the datagram in error. The decision was made, therefore, that these asynchronous errors are returned to the process only if the process has connected the UDP socket to exactly one peer.

> Linux returns most ICMP destination unreachable errors even for an unconnected socket, as long as the `SO_BSDCOMPAT` socket option is not enabled. All the destination unreachable errors from Figure A.15 are returned, other than codes 0, 1, 4, 5, 11, and 12.
>
> The XTI interface provides a way of returning this additional information to the process: the `t_rcvuderr` can return this information (Section 31.4). Unfortunately, many XTI implementations do not return this information.
>
> We return to this problem of asynchronous errors with a UDP socket in Section 25.7 and show an easy way to obtain these errors on an unconnected socket using a daemon of our own.

8.10 Summary of UDP example

Figure 8.11 shows as bullets the four values that must be specified or chosen when the client sends a UDP datagram.

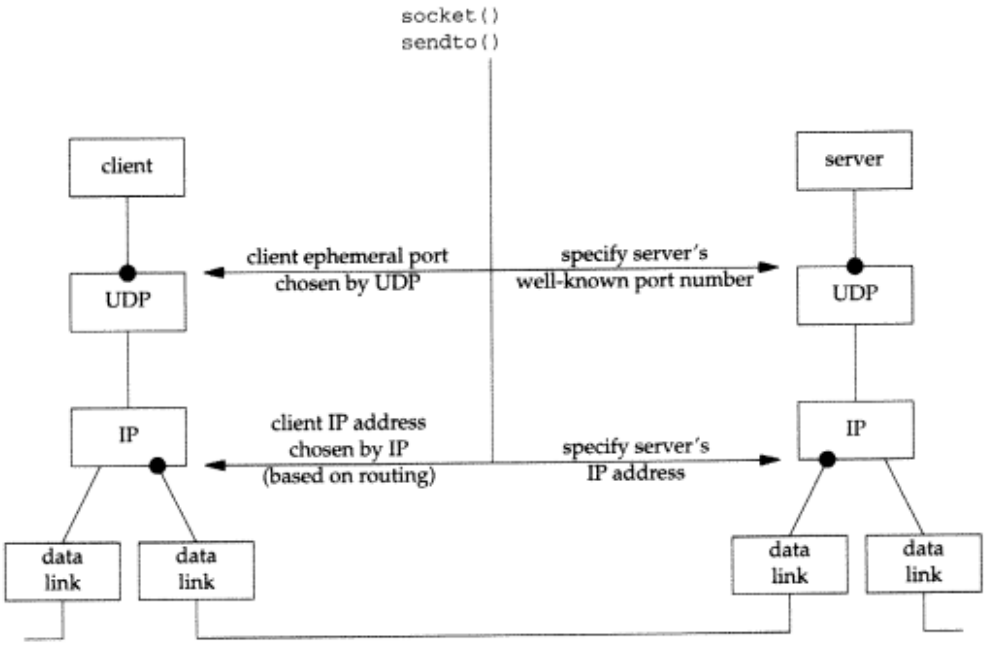

Figure 8.11 Summary of UDP client–server from client's perspective.

The client must specify the server's IP address and port number for the call to `sendto`. Normally the client's IP address and port are chosen automatically by the kernel, although we mentioned that the client can call `bind` if it so chooses. If these two values for the client are chosen by the kernel, we also mentioned that the client ephemeral port is chosen once, on the first `sendto`, and then never changes. The client's IP address, however, can change for every UDP datagram that the client sends, assuming the client does not `bind` a specific IP address to the socket. The reason is shown in Figure 8.11: if the client host is multihomed, the client could alternate between two destinations, one going out the datalink on the left, and the other going out the datalink on the right. In this worst case scenario, the client's IP address, as chosen by the kernel based on the outgoing datalink, would change for every datagram.

What happens if the client `binds` an IP address to its socket, but the kernel decides that an outgoing datagram must be sent out some other datalink? In this case the IP datagram will contain a source IP address that is different from the IP address of the outgoing datalink. (See Exercise 8.6.)

Figure 8.12 shows the same four values, but from the server's perspective.

There are four pieces of information that a server might want to know from an IP datagram that arrives: the source IP address, destination IP address, source port number, and destination port number. Figure 8.13 shows the function calls that return this information for a TCP server and a UDP server.

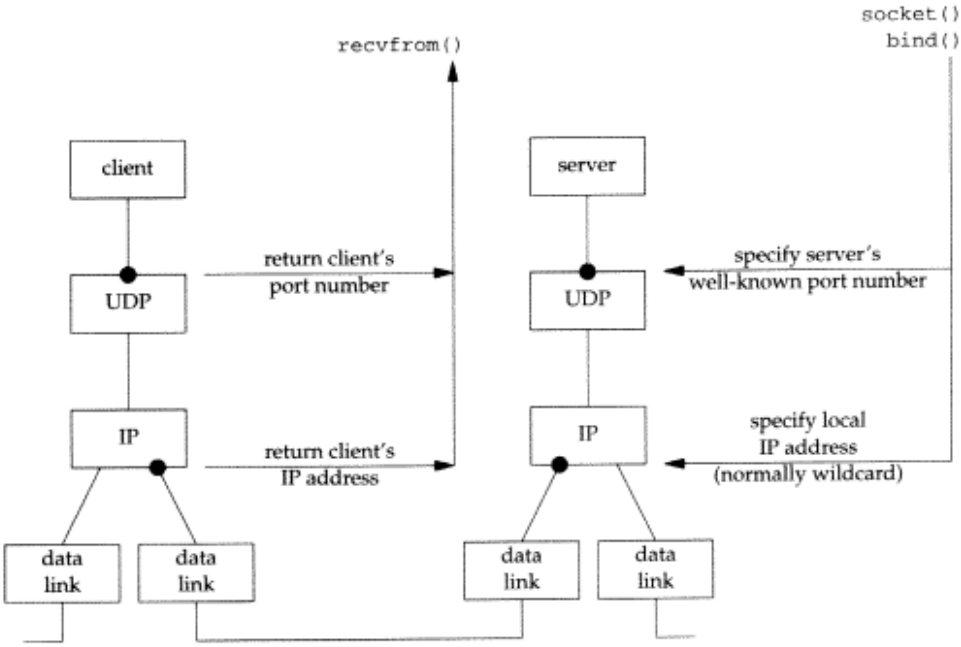

Figure 8.12 Summary of UDP client–server from server's perspective.

From client's IP datagram	TCP server	UDP server
source IP address	`accept`	`recvfrom`
source port number	`accept`	`recvfrom`
destination IP address	`getsockname`	`recvmsg`
destination port number	`getsockname`	`getsockname`

Figure 8.13 Information available to server from arriving IP datagram.

A TCP server always has easy access to all four pieces of information for a connected socket, and these four values remain constant for the lifetime of a connection. With a UDP socket, however, the destination IP address can only be obtained by setting the `IP_RECVDSTADDR` socket option for IPv4 or the `IPV6_PKTINFO` socket option for IPv6 and then calling `recvmsg` instead of `recvfrom`. Since UDP is connectionless, the destination IP address can change for each datagram that is sent to the server. A UDP server can also receive datagrams destined for one of the host's broadcast addresses or for a multicast address, as we discuss in Chapters 18 and 19. We will show how to determine the destination address of a UDP datagram in Section 20.2, after we cover the `recvmsg` function.

8.11 `connect` Function with UDP

We mentioned at the end of Section 8.9 that asynchronous errors are not returned on UDP sockets unless the socket has been connected. Indeed, we are able to call `connect` (Section 4.3) for a UDP socket. But this does not result in anything like a TCP connection: there is no three-way handshake. Instead, the kernel just records the IP address and port number of the peer, which are contained in the socket address structure passed to `connect`, and returns immediately to the calling process.

> Overloading the `connect` function with this capability for UDP sockets is confusing. If the convention that `sockname` is the local protocol address and `peername` is the foreign protocol address is used, then a better name would have been `setpeername`. Similarly, a better name for the `bind` function would be `setsockname`.

With this capability we must now distinguish between

- an *unconnected UDP socket*, the default when we create a UDP socket, and
- a *connected UDP socket*, the result of calling `connect` on a UDP socket.

With a connected UDP socket three things change, compared to the default unconnected UDP socket.

1. We can no longer specify the destination IP address and port for an output operation. That is, we do not use `sendto` but use `write` or `send` instead. Anything written to a connected UDP socket is automatically sent to the protocol address (e.g., IP address and port) specified by the `connect`.

   > Similar to TCP, we can call `sendto` for a connected UDP socket, but we cannot specify a destination address. The fifth argument to `sendto` (the pointer to the socket address structure) must be a null pointer, and the sixth argument (the size of the socket address structure) should be 0. Posix.1g specifies that when the fifth argument is a null pointer, the sixth argument is ignored.

2. We do not use `recvfrom` but use `read` or `recv` instead. The only datagrams returned by the kernel for an input operation on a connected UDP socket are those arriving from the protocol address specified in the `connect`. Datagrams destined to the connected UDP socket's local protocol address (e.g., IP address and port) but arriving from a protocol address other than the one to which the socket was `connect`ed, are not passed to the connected socket. This limits a connected UDP socket to exchanging datagrams with one and only one peer.

   > Technically, a connected UDP socket exchanges datagrams with only one IP address, because it is possible to `connect` to a multicast or broadcast address.

3. Asynchronous errors are returned to the process for a connected UDP socket. The corollary, as we previously described, is that an unconnected UDP socket does not receive any asynchronous errors.

Figure 8.14 summarizes the first point in the list with respect to 4.4BSD.

Type of socket	write or send	sendto that does not specify a destination	sendto that specifies a destination
TCP socket	OK	OK	EISCONN
UDP socket, connected	OK	OK	EISCONN
UDP socket, unconnected	EDESTADDRREQ	EDESTADDRREQ	OK

Figure 8.14 TCP and UDP sockets: can a destination protocol address be specified?

Posix.1g specifies that an output operation that does not specify a destination address on an unconnected UDP socket should return ENOTCONN, not EDESTADDRREQ.

Solaris 2.5 allows a sendto that specifies a destination address for a connected UDP socket. Posix.1g specifies that EISCONN should be returned instead.

Figure 8.15 summarizes the three points that we made about a connected UDP socket.

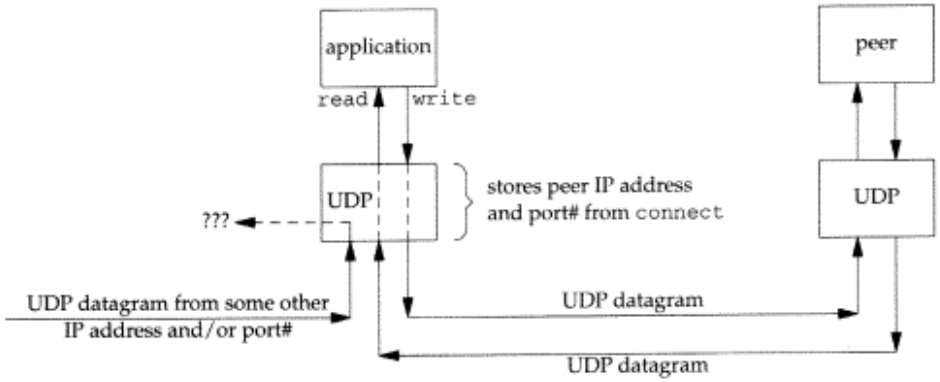

Figure 8.15 Connected UDP socket.

The application calls connect, specifying the IP address and port number of its peer. It then uses read and write to exchange data with the peer.

Datagrams arriving from any other IP address or port (which we show as "???" in Figure 8.15) are not passed to the connected socket because either the source IP address or source UDP port does not match the protocol address to which the socket is connected. These datagrams could be delivered to some other UDP socket on the host. If there is no other matching socket for the arriving datagram, UDP will discard it and generate an ICMP port unreachable error.

In summary we can say that a UDP client or server can call connect only if that process uses the UDP socket to communicate with exactly one peer. Normally it is a UDP client that calls connect, but there are applications in which the UDP server communicates with a single client for a long duration (e.g., TFTP), and in this case both the client and server can call connect.

The DNS provides another example, as shown in Figure 8.16.

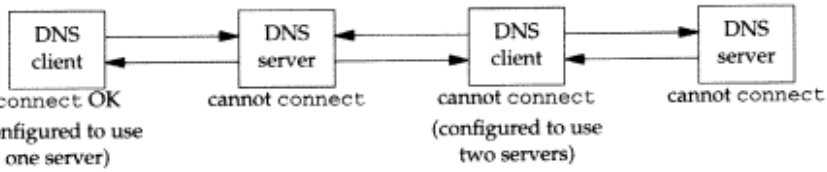

Figure 8.16 Example of DNS clients and servers and the `connect` function.

A DNS client can be configured to use one or more servers, normally by listing the IP addresses of the servers in the file `/etc/resolv.conf`. If a single server is listed (the leftmost box in the figure), the client can call `connect`, but if multiple servers are listed (the second box from the right in the figure), the client cannot call `connect`. Also, a DNS server normally handles any client's request, so the servers cannot call `connect`.

Calling `connect` Multiple Times for a UDP Socket

A process with a connected UDP socket can call `connect` again for that socket, to either

- specify a new IP address and port, or to
- unconnect the socket.

The first case, specifying a new peer for a connected UDP socket, differs from the use of `connect` with a TCP socket: `connect` can be called only one time for a TCP socket.

To unconnect a UDP socket we call `connect` but set the family member of the socket address structure (`sin_family` for IPv4 or `sin6_family` for IPv6) to `AF_UNSPEC`. This might return an error of `EAFNOSUPPORT` (p. 736 of TCPv2) but that is OK. It is the process of calling `connect` on an already connected UDP socket that causes the socket to become unconnected (pp. 787–788 of TCPv2).

> The BSD manual page for `connect` has traditionally said "Datagram sockets may dissolve the association by connecting to an invalid address, such as the null address." Unfortunately the manual page never defines what a "null address" is, and doesn't mention that an error results (which is OK). Posix.1g explicitly states that the address family must be set to `AF_UNSPEC` but then waffles by saying that this call to `connect` may or may not return the error of `EAFNOSUPPORT`.

Performance

When an application calls `sendto` on an unconnected UDP socket, Berkeley-derived kernels temporarily connect the socket, send the datagram, and then unconnect the socket (pp. 762–763 of TCPv2). Calling `sendto` for two datagrams on an unconnected UDP socket then involves the following six steps by the kernel:

- connect the socket,
- output the first datagram,
- unconnect the socket,

- connect the socket,
- output the second datagram, and
- unconnect the socket.

> Another consideration is the number of searches of the routing table. The first temporary connect searches the routing table for the destination IP address and saves (caches) that information. The second temporary connect notices that the destination address equals the destination of the cached routing table information (we are assuming two `sendto`s to the same destination) and need not search the routing table again (pp. 737–738 of TCPv2).

When the application knows it will be sending multiple datagrams to the same peer, it is more efficient to connect the socket explicitly. Calling `connect` and then calling `write` two times now involves the following steps by the kernel:

- connect the socket,
- output first datagram, and
- output second datagram.

In this case the kernel copies only the socket address structure containing the destination IP address and port one time, versus two times when `sendto` is called twice. [Partridge and Pink 1993] note that the temporary connecting of an unconnected UDP socket accounts for nearly one-third of the cost of each UDP transmission.

8.12 `dg_cli` Function (Revisited)

We now return to the `dg_cli` function from Figure 8.8 and recode it to call `connect`. Figure 8.17 shows the new function.

udpcliserv/dgcliconnect.c

```
 1 #include    "unp.h"

 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5     int     n;
 6     char    sendline[MAXLINE], recvline[MAXLINE + 1];

 7     Connect(sockfd, (SA *) pservaddr, servlen);

 8     while (Fgets(sendline, MAXLINE, fp) != NULL) {

 9         Write(sockfd, sendline, strlen(sendline));

10         n = Read(sockfd, recvline, MAXLINE);

11         recvline[n] = 0;        /* null terminate */
12         Fputs(recvline, stdout);
13     }
14 }
```

udpcliserv/dgcliconnect.c

Figure 8.17 `dg_cli` function that calls `connect`.

The changes are the new call to `connect` and replacing the calls to `sendto` and `recvfrom` with calls to `write` and `read`. This function is still protocol independent since it doesn't look inside the socket address structure that is passed to `connect`. Our client `main` function, Figure 8.7, remains the same.

If we run this program on the host `bsdi`, specifying the IP address of the host `solaris` (which is not running our server on port 9877), we have the following output:

```
bsdi % udpcli04 206.62.226.33
hello, world
read error: Connection refused
```

The first point we notice is that we do *not* receive the error when we start the client process. The error occurs only after we send the first datagram to the server. It is sending this datagram that elicits the ICMP error from the server host. But when a TCP client calls `connect`, specifying a server host that is not running the server process, `connect` returns the error because the call to `connect` causes the first packet of TCP's three-way handshake to be sent, and it is this packet that elicits the RST from the server TCP (Section 4.3).

Figure 8.18 shows the `tcpdump` output.

```
bsdi % tcpdump
1  0.0                    bsdi.1318 > solaris.9877: udp 13
2  0.000628 ( 0.0006)     solaris > bsdi: icmp: solaris udp port 9877 unreachable
```

Figure 8.18 `tcpdump` output when running Figure 8.17.

We also see in Figure A.15 that this ICMP error is mapped by the kernel into the error `ECONNREFUSED`, which corresponds to the message string output by our `err_sys` function: "Connection refused."

> Unfortunately, not all kernels return ICMP messages to a connected UDP socket, as we have shown in this section. Normally Berkeley-derived kernels return the error, while System V kernels do not. For example, if we run the same client on a Solaris 2.4 host and `connect` to a host that is not running our server, we can watch with `tcpdump` and verify that the ICMP port unreachable error is returned by the server host, but the client's call to `read` never returns. This bug was fixed in Solaris 2.5. UnixWare does not return the error while AIX, Digital Unix, HP-UX, and Linux all return the error.

8.13 Lack of Flow Control with UDP

We now examine the effect of UDP not having any flow control. First we modify our `dg_cli` function to send a fixed number of datagrams. It no longer reads from standard input. Figure 8.19 shows the new version. This function writes 2000 1400-byte UDP datagrams to the server.

We then modify the server to receive the datagrams and count the number received. This server no longer echoes the datagrams back to the client. Figure 8.20 shows the new `dg_echo` function. When we terminate the server with our terminal interrupt key (`SIGINT`), it prints the number of received datagrams and terminates.

udpcliserv/dgcliloop1.c

```
1 #include    "unp.h"

2 #define NDG     2000            /* #datagrams to send */
3 #define DGLEN   1400            /* length of each datagram */

4 void
5 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
6 {
7     int     i;
8     char    sendline[DGLEN];

9     for (i = 0; i < NDG; i++) {
10        Sendto(sockfd, sendline, DGLEN, 0, pservaddr, servlen);
11    }
12 }
```

udpcliserv/dgcliloop1.c

Figure 8.19 dg_cli function that writes a fixed number of datagrams to server.

udpcliserv/dgecholoop1.c

```
1 #include    "unp.h"

2 static void recvfrom_int(int);
3 static int count;

4 void
5 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
6 {
7     socklen_t len;
8     char    mesg[MAXLINE];

9     Signal(SIGINT, recvfrom_int);

10    for ( ; ; ) {
11        len = clilen;
12        Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

13        count++;
14    }
15 }

16 static void
17 recvfrom_int(int signo)
18 {
19    printf("\nreceived %d datagrams\n", count);
20    exit(0);
21 }
```

udpcliserv/dgecholoop1.c

Figure 8.20 dg_echo function that counts received datagrams.

We now run the server on the host bsdi, a slow 80386. We run the client on the much faster SparcStation 4. Additionally, we run netstat -s on the server, both before and after, as the statistics that are output tell us how many datagrams were lost. Figure 8.21 shows the output on the server.

```
bsdi % netstat -s | tail
udp:    80300 datagrams received
        0 with incomplete header
        0 with bad data length field
        0 with bad checksum
        12 dropped due to no socket
        77725 broadcast/multicast datagrams dropped due to no socket
        1970 dropped due to full socket buffers
        593 delivered
        70592 datagrams output
bsdi % udpserv06                        start our server
                                        we run the client here
^?                                      type our interrupt key after client is finished
received 82 datagrams
bsdi % netstat -s | tail
udp:    82294 datagrams received
        0 with incomplete header
        0 with bad data length field
        0 with bad checksum
        12 dropped due to no socket
        77725 broadcast/multicast datagrams dropped due to no socket
        3882 dropped due to full socket buffers
        675 delivered
        70592 datagrams output
```

Figure 8.21 Output on server host.

The client sent 2000 datagrams, but the server application received only 82 of these, for a 96% loss rate. There is *no* indication whatsoever to the server application or to the client application that these datagrams are lost. As we have said, UDP has no flow control and it is unreliable. It is trivial, as we have shown, for a UDP sender to overrun the receiver.

If we look at the `netstat` output, the total number of datagrams received by the server host (not the server application) is 1994 (82294 – 80300). Six datagrams were never received by the interface, either because the interface's buffers were full or they could have been discarded by the sending host. The counter "dropped due to full socket buffers" indicates how many datagrams were received by UDP but were discarded because the receiving socket's receive queue was full (p. 775 of TCPv2). This value is 1912 (3882 – 1970), which when added to the counter output by the application (82) equals the 1994 datagrams that the host received. Unfortunately, the `netstat` counter of the number dropped due to full socket buffer is systemwide. There is no way to determine which applications (e.g., which UDP ports) are affected.

> Notice that 97% of all the received UDP datagrams (77725 + 80300) on this particular host are broadcast or multicast datagrams that are then discarded because there is no application with a socket bound to the destination port. We return to this phenomenon when we talk about broadcasting in Chapter 18.

The number of datagrams received by the server in this example is nondeterministic. It depends on many factors, such as the network load, the processing load on the

client host, and the processing load on the server host. If we run this example five more times the count of received datagrams is 37, 108, 30, 108, and 114.

If we run the same client and server, but this time with the client on the slow 80386 and the server on the faster SparcStation, no datagrams are lost.

```
solaris % udpserv06
^?                                    type our interrupt key after client is finished
received 2000 datagrams
```

If we run `netstat -s` under Solaris, the output format differs from the classical Berkeley output shown in Figure 8.21. The Solaris format mimics the SNMP counters (Simple Network Management Protocol, described in Chapter 25 of TCPv1). The `netstat` counter `udpInDatagrams`, the number of UDP datagrams delivered to user processes, is 139 before and 2139 after, accounting for all 2000 datagrams. The counter `udpInOverflows`, which is not an official SNMP counter, counts the number of received UDP datagrams that are discarded because the receiving socket's receive queue has no room. Its value is 0 both before and after, as we expect.

UDP Socket Receive Buffer

The number of UDP datagrams that are queued by UDP for a given socket is limited by the size of that socket's receive buffer. We can change this with the `SO_RCVBUF` socket option, as we described in Section 7.5. The default size of the UDP socket receive buffer under BSD/OS is 41,600 bytes, which allows room for only 29 of our 1400-byte datagrams. If we increase the size of the socket receive buffer, we expect the server to receive additional datagrams. Figure 8.22 shows a modification to the `dg_echo` function from Figure 8.20 that sets the socket receive buffer to 240 Kbytes. If we run this server five times on the 80386 and the client on the SparcStation 4, the count of received datagrams is 115, 168, 179, 145, and 133. While this is slightly better than the earlier example with the default socket receive buffer, it is no panacea.

> Why do we set the receive socket buffer size to 240 × 1024 in Figure 8.22? The maximum size of a socket receive buffer in BSD/OS 2.1 defaults to 262,144 bytes (256 × 1024) but due to the buffer allocation policy (described in Chapter 2 of TCPv2) the actual limit is 246,723 bytes. Many earlier systems based on 4.3BSD restricted the size of a socket buffer to around 52,000 bytes.

8.14 Determining Outgoing Interface with UDP

A connected UDP socket can also be used to determine the outgoing interface that will be used to a particular destination. This is because of a side effect of the `connect` function when applied to a UDP socket: the kernel chooses the local IP address (assuming the process has not already called `bind` to explicitly assign this). This local IP address is chosen by searching the routing table for the destination IP address, and then using the primary IP address for the resulting interface.

udpcliserv/dgecholoop2.c

```
1 #include    "unp.h"
2 static void recvfrom_int(int);
3 static int count;
4 void
5 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
6 {
7     int     n;
8     socklen_t len;
9     char    mesg[MAXLINE];
10    Signal(SIGINT, recvfrom_int);
11    n = 240 * 1024;
12    Setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &n, sizeof(n));
13    for ( ; ; ) {
14        len = clilen;
15        Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
16        count++;
17    }
18 }
19 static void
20 recvfrom_int(int signo)
21 {
22    printf("\nreceived %d datagrams\n", count);
23    exit(0);
24 }
```

udpcliserv/dgecholoop2.c

Figure 8.22 dg_echo function that increases the size of the socket receive queue.

Figure 8.23 shows a simple UDP program that connects to a specified IP address and then calls getsockname, printing the local IP address and port.

If we run the program on the multihomed host bsdi, we have the following output:

```
bsdi % udpcli09 206.62.226.42
local address 206.62.226.35.1331

bsdi % udpcli09 206.62.226.65
local address 206.62.226.66.1332

bsdi % udpcli09 127.0.0.1
local address 127.0.0.1.1335
```

We see from Figure 1.16 that the first two times we run the program the command-line argument is an IP address on a different Ethernet. The kernel assigns the local IP address to the primary address of the interface on that Ethernet. That is, the .42 host is on the top Ethernet so the outgoing interface address has the .35 address. The .65 host is on the lower Ethernet so the outgoing interface has the .66 address. Calling connect on a UDP socket does not send anything to that host; it is entirely a local operation that saves the peer's IP address and port. We also see that calling connect on an unbound UDP socket also assigns an ephemeral port to the socket.

udpcliserv/udpcli09.c

```
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd;
 6     socklen_t len;
 7     struct sockaddr_in cliaddr, servaddr;

 8     if (argc != 2)
 9         err_quit("usage: udpcli <IPaddress>");

10     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

11     bzero(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_port = htons(SERV_PORT);
14     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

15     Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));

16     len = sizeof(cliaddr);
17     Getsockname(sockfd, (SA *) &cliaddr, &len);
18     printf("local address %s\n", Sock_ntop((SA *) &cliaddr, len));

19     exit(0);
20 }
```

udpcliserv/udpcli09.c

Figure 8.23 UDP program that uses `connect` to determine outgoing interface.

Unfortunately this technique does not work on all implementations, notably SVR4-derived kernels. For example this does not work on HP-UX, Solaris 2.5, and UnixWare, but it works on AIX, Digital Unix, Linux, and Solaris 2.6.

8.15 TCP and UDP Echo Server Using `select`

We now combine our concurrent TCP echo server from Chapter 5 with our iterative UDP echo server from this chapter into a single server that uses `select` to multiplex a TCP and UDP socket. Figure 8.24 is the first half of this server.

Create listening TCP socket

14–22 A listening TCP socket is created that is bound to the server's well-known port. We set the `SO_REUSEADDR` socket option in case connections exist on this port.

Create UDP socket

23–29 A UDP socket is also created and bound to the same port. Even though the same port is used for the TCP and UDP sockets, there is no need to set the `SO_REUSEADDR` socket option before this call to `bind`, because TCP ports are independent of UDP ports.

Figure 8.25 shows the second half of our server.

udpcliserv/udpservselect01.c

```
 1 #include    "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     listenfd, connfd, udpfd, nready, maxfdp1;
 6     char    mesg[MAXLINE];
 7     pid_t   childpid;
 8     fd_set  rset;
 9     ssize_t n;
10     socklen_t len;
11     const int on = 1;
12     struct sockaddr_in cliaddr, servaddr;
13     void    sig_chld(int);
14         /* create listening TCP socket */
15     listenfd = Socket(AF_INET, SOCK_STREAM, 0);
16     bzero(&servaddr, sizeof(servaddr));
17     servaddr.sin_family = AF_INET;
18     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
19     servaddr.sin_port = htons(SERV_PORT);
20     Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
21     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
22     Listen(listenfd, LISTENQ);
23         /* create UDP socket */
24     udpfd = Socket(AF_INET, SOCK_DGRAM, 0);
25     bzero(&servaddr, sizeof(servaddr));
26     servaddr.sin_family = AF_INET;
27     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
28     servaddr.sin_port = htons(SERV_PORT);
29     Bind(udpfd, (SA *) &servaddr, sizeof(servaddr));
```

udpcliserv/udpservselect01.c

Figure 8.24 First half of echo server that handles TCP and UDP using `select`.

Establish signal handler for `SIGCHLD`

30 A signal handler is established for `SIGCHLD` because TCP connections will be handled by a child process. We showed this signal handler in Figure 5.11.

Prepare for `select`

31-32 We initialize a descriptor set for `select` and calculate the maximum of the two descriptors for which we will wait.

Call `select`

34-41 We call `select` waiting only for readability on the listening TCP socket or readability on the UDP socket. Since our `sig_chld` handler can interrupt our call to `select`, we handle an error of `EINTR`.

udpcliserv/udpservselect01.c

```
30      Signal(SIGCHLD, sig_chld);  /* must call waitpid() */

31      FD_ZERO(&rset);
32      maxfdp1 = max(listenfd, udpfd) + 1;
33      for ( ; ; ) {
34          FD_SET(listenfd, &rset);
35          FD_SET(udpfd, &rset);
36          if ( (nready = select(maxfdp1, &rset, NULL, NULL, NULL)) < 0) {
37              if (errno == EINTR)
38                  continue;       /* back to for() */
39              else
40                  err_sys("select error");
41          }
42          if (FD_ISSET(listenfd, &rset)) {
43              len = sizeof(cliaddr);
44              connfd = Accept(listenfd, (SA *) &cliaddr, &len);

45              if ( (childpid = Fork()) == 0) {     /* child process */
46                  Close(listenfd);    /* close listening socket */
47                  str_echo(connfd);   /* process the request */
48                  exit(0);
49              }
50              Close(connfd);      /* parent closes connected socket */
51          }
52          if (FD_ISSET(udpfd, &rset)) {
53              len = sizeof(cliaddr);
54              n = Recvfrom(udpfd, mesg, MAXLINE, 0, (SA *) &cliaddr, &len);

55              Sendto(udpfd, mesg, n, 0, (SA *) &cliaddr, len);
56          }
57      }
58  }
```

udpcliserv/udpservselect01.c

Figure 8.25 Second half of echo server that handles TCP and UDP using `select`.

Handle new client connection

42–51 We `accept` a new client connection when the listening TCP socket is readable, `fork` a child, and call our `str_echo` function in the child. This is the same sequence of steps that we used in Chapter 5.

Handle arrival of datagram

52–57 If the UDP socket is readable, a datagram has arrived. We read it with `recvfrom` and send it back to the client with `sendto`.

8.16 Summary

Converting our echo client–server to use UDP instead of TCP was simple. But lots of features provided by TCP are missing: detecting lost packets and retransmitting,

verifying responses as being from the correct peer, and the like. We return to this topic in Section 20.5 and see what it takes to add some reliability to a UDP application.

UDP sockets can generate asynchronous errors, that is, errors that are reported some time after the packet was sent. TCP sockets always report these errors to the application, but with UDP the socket must be connected to receive these errors.

UDP has no flow control, and this is easy to demonstrate. Normally this is not a problem, because many UDP applications are built using a request–reply model, and not for transferring bulk data.

There are still more points to consider when writing UDP applications, but we save these until Chapter 20, after covering the interface functions, broadcasting, and multicasting.

Exercises

8.1 We have two applications, one using TCP and the other using UDP. 4096 bytes are in the receive buffer for the TCP socket and two 2048-byte datagrams are in the receive buffer for the UDP socket. The TCP application calls `read` with a third argument of 4096 and the UDP application calls `recvfrom` with a third argument of 4096. Is there any difference?

8.2 What happens in Figure 8.4 if we replace the final argument to `sendto` (which we show as `len`) with `clilen`?

8.3 Compile and run the UDP server in Figures 8.3 and 8.4 and then the UDP client in Figures 8.7 and 8.8. Verify that the client and server work together.

8.4 Run the `ping` program in one window, specifying the `-i 60` option (send one packet every 60 seconds; some systems use `-I` instead of `-i`), the `-v` option (print all received ICMP errors), and specifying the loopback address (normally 127.0.0.1). We will use this program to see the ICMP port unreachable returned by the server host. Then run our client from the previous exercise in another window, specifying the IP address of some host that is not running the server. What happens?

8.5 We said with Figure 8.5 that each connected TCP socket has its own socket receive buffer. What about the listening socket; do you think it has its own socket receive buffer?

8.6 Use the `sock` program (Section C.3) and a tool such as `tcpdump` (Section C.5) to test what we claimed in Section 8.10: if the client `bind`s an IP address to it socket but sends a datagram that goes out some other interface, the resulting IP datagram still contains the IP address that was bound to the socket, even though this does not correspond to the outgoing interface.

8.7 Compile the programs from Section 8.13 and run the client and server on different hosts. Put a `printf` in the client each time a datagram is written to the socket. Does this change the percentage of received packets? Why? Put a `printf` in the server each time a datagram is read from the socket. Does this change the percentage of received packets? Why?

8.8 What is the largest length that we can pass to `sendto` for a UDP/IPv4 socket, that is, what is the largest amount of data that can fit into a UDP/IPv4 datagram? What changes with UDP/IPv6?

Modify Figure 8.8 to send one maximum size UDP datagram, read it back, and print the number of bytes returned by `recvfrom`.

9

Elementary Name and Address Conversions

9.1 Introduction

All the examples so far in this text have used numeric addresses for the hosts (e.g., 206.6.226.33) and numeric port numbers to identify the servers (e.g., port 13 for the standard daytime server and port 9877 for our echo server). We should, however, use names instead of numbers for numerous reasons: names are easier to remember, the numeric address can change but the name can remain the same, and with the move to IPv6 numeric addresses become much longer making it much more error prone to enter an address by hand. This chapter describes the functions that convert between names and numeric values: `gethostbyname` and `gethostbyaddr` to convert between hostnames and IP addresses, and `getservbyname` and `getservbyport` to convert between service names and port numbers.

The hostname functions have recently been enhanced to work with IPv6, in addition to IPv4, and we also describe these changes. This is the beginning of our move toward protocol independence, which we continue in Chapter 11. Indeed, Chapter 11 takes the functions that we describe in this chapter and develops numerous functions that can make our applications protocol independent.

9.2 Domain Name System

The *Domain Name System*, or DNS, is used primarily to map between hostnames and IP addresses. A hostname can be either a *simple name* , such as `solaris` or `bsdi`, or a *fully qualified domain name* (FQDN) such as `solaris.kohala.com`.

> Technically, an FQDN is also called an *absolute name* and must end with a period, but users often omit the ending period.

In this section we cover only the basics of the DNS that we need for network programming. Readers interested in additional details should consult Chapter 14 of TCPv1 and [Albitz and Liu 1997]. The additions required for IPv6 are in RFC 1886 [Thomson and Huitema 1995].

Resource Records

Entries in the DNS are known as *resource records* (RRs). There are only a few types of RRs that affect us.

A An A record maps a hostname into a 32-bit IPv4 address. For example, here are the four DNS records for the host `solaris` in the `kohala.com` domain, the first of which is an A record:

```
solaris    IN    A       206.62.226.33
           IN    AAAA    5f1b:df00:ce3e:e200:0020:0800:2078:e3e3
           IN    MX      5  solaris.kohala.com.
           IN    MX      10 mailhost.kohala.com.
```

AAAA A AAAA record, called a "quad A" record, maps a hostname into a 128-bit IPv6 address. The term "quad A" was chosen because a 128-bit address is four times larger than a 32-bit address.

PTR PTR records (called "pointer records") map IP addresses into hostnames. For an IPv4 address the 4 bytes of the 32-bit address are reversed, each byte is converted to its decimal ASCII value (0–255), and `in-addr.arpa` is then appended. The resulting string is used in the PTR query.

For an IPv6 address the 32 4-bit nibbles of the 128-bit address are reversed, each nibble is converted to its corresponding hexadecimal ASCII value (`0-9a-f`), and `ip6.int` is appended.

For example, the two PTR records for our host `solaris` would be `33.226.62.206.in-addr.arpa` and `3.e.3.e.8.7.0.2.0.0.8.0.0.2.0.0.0.0.2.e.e.3.e.c.0.0.f.d.b.1.f.5.ip6.int`.

MX An MX record specifies a host to act as a "mail exchanger" for the specified host. In the example for the host `solaris` above, two MX records are provided. The first has a preference value of 5 and the second has a preference value of 10. When multiple MX records exist, they are used in order of preference, starting with the smallest value.

> We do not use MX records in this text, but we mention them because they are used extensively in the real world.

CNAME CNAME stands for "canonical name." A common use is to assign CNAME records for common services, such as `ftp` and `www`. If people

use these service names, instead of the actual hostname, it is transparent if the service is moved to another host. For example, the following could be CNAMEs for our host `bsdi`:

```
ftp       IN   CNAME  bsdi.kohala.com.
www       IN   CNAME  bsdi.kohala.com.
mailhost  IN   CNAME  bsdi.kohala.com.
```

It is too early in the deployment of IPv6 to know what conventions administrators will use for hosts that support both IPv4 and IPv6. In our example earlier in this section we specified both an A record and a AAAA record for our host `solaris`. Some administrators place all AAAA records into their own subdomain, often named `ipv6`. For example the hostname associated with the AAAA record would then be `solaris.ipv6.kohala.com`. Sometimes this is done because the administrator of the dual-stack host does not have domain name responsibility for the entire domain but obtains responsibility for the separate `ipv6` subdomain.

Instead, the author places both the A record and the AAAA record under the host's normal name (as shown earlier) and creates another RR whose name ends in `-4` containing the A record, another RR whose name ends in `-6` containing the AAAA record, and another RR whose name ends in `-6ll` containing a AAAA record with the host's link-local address (which is sometimes handy for debugging purposes). All the records for another of our hosts are then

```
aix-4      IN   A      206.62.226.43
aix        IN   A      206.62.226.43
           IN   MX     5  aix.kohala.com.
           IN   MX     10 mailhost.kohala.com.
           IN   AAAA   5f1b:df00:ce3e:e200:0020:0800:5afc:2b36
aix-6      IN   AAAA   5f1b:df00:ce3e:e200:0020:0800:5afc:2b36
aix-6ll    IN   AAAA                       fe80::0800:5afc:2b36
```

This gives us additional control over the protocol chosen by some applications, as we will see in the next chapter.

Resolvers and Name Servers

Organizations run one or more *name servers*, often the program known as BIND (Berkeley Internet Name Domain). Applications such as the clients and servers that we are writing in this text contact a DNS server by calling functions in a library known as the *resolver*. The common resolver functions are `gethostbyname` and `gethostbyaddr`, both of which are described in this chapter. The former maps a hostname into its IP addresses, and the latter does the reverse mapping.

Figure 9.1 shows a typical arrangement of applications, resolvers, and name servers. We write the application code. The resolver code is contained in a system library and is link-edited into the application when the application is built. The application code calls the resolver code using normal function calls, typically calling the functions `gethostbyname` and `gethostbyaddr`.

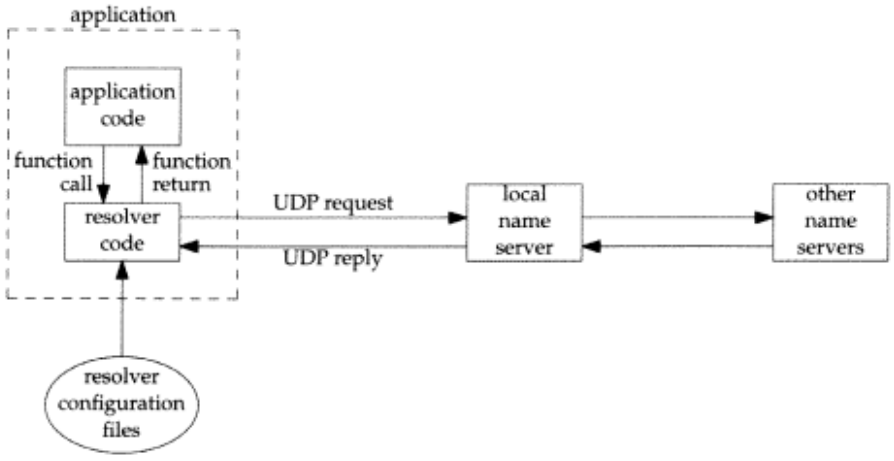

Figure 9.1 Typical arrangement of clients, resolvers, and name servers.

The resolver code reads its system-dependent configuration files to determine the location of the organization's name servers. (We use the plural "name servers" because most organizations run multiple name servers, even though we show only one local server in the figure.) The file `/etc/resolv.conf` normally contains the IP addresses of the local name servers.

The resolver sends the query to the local name server using UDP. If the local name server does not know the answer, it will normally query other name servers across the Internet, also using UDP.

DNS Alternatives

It is possible to obtain the name and address information without using the DNS and common alternatives are static hosts files (normally the file `/etc/hosts` as we describe in Figure 9.9) or NIS (Network Information System). Unfortunately it is implementation dependent how an administrator configures a host to use the different types of name service. Solaris 2.x and HP-UX 10.30 use the file `/etc/nsswitch.conf`, Digital Unix uses the file `/etc/svc.conf`, and AIX uses the file `/etc/netsvc.conf`. BIND 8.1 supplies its own version named IRS (Information Retrieval Service) that uses the file `/etc/irs.conf`. If a name server is to be used for hostname lookups, then all these systems use the file `/etc/resolv.conf` to specify the IP addresses of the name servers. Fortunately, these differences are normally hidden to the application programmer, so we just call the resolver functions such as `gethostbyname` and `gethostbyaddr`.

9.3 `gethostbyname` Function

Host computers are normally known by human-readable names. All the examples that we have shown so far in this book have intentionally used IP addresses instead of

names, so we know exactly what goes into the socket address structures, for functions such as `connect` and `sendto`, and what is returned by functions such as `accept` and `recvfrom`. But most applications should deal with names and not addresses. This is especially true as we move to IPv6, since IPv6 addresses (hex strings) are much longer than IPv4 dotted-decimal numbers. (The example AAAA record and `ip6.int` PTR record in the previous section should make this obvious.)

The most basic function that looks up a hostname is `gethostbyname`. If successful, it returns a pointer to a `hostent` structure that contains all the IPv4 addresses or all the IPv6 addresses for the host.

```
#include <netdb.h>

struct hostent *gethostbyname(const char *hostname);
```

Returns: nonnull pointer if OK, `NULL` on error with `h_errno` set

The nonnull pointer returned by this function points to the following `hostent` structure:

```
struct hostent {
  char  *h_name;       /* official (canonical) name of host */
  char **h_aliases;    /* pointer to array of pointers to alias names */
  int    h_addrtype;   /* host address type: AF_INET or AF_INET6 */
  int    h_length;     /* length of address: 4 or 16 */
  char **h_addr_list;  /* ptr to array of ptrs with IPv4 or IPv6 addrs */
};

#define h_addr  h_addr_list[0]  /* first address in list */
```

In terms of the DNS, `gethostbyname` performs a query for an A record or for a AAAA record. This function can return either IPv4 addresses or IPv6 addresses. We summarize in Figure 9.5 the conditions under which it returns these two types of addresses.

> The definition of `h_addr` is for backward compatibility and new code should not use `h_addr`. 4.2BSD did not have the `h_addr_list` member, having a `char *h_addr` that pointed only to one IP address.

Figure 9.2 shows the arrangement of the `hostent` structure and the information that it points to assuming the hostname that is looked up has two alias names and three IPv4 addresses. Of these fields, the official hostname and all of the aliases are null-terminated C strings.

The returned `h_name` is called the *canonical* name of the host. For example, given the CNAME records shown in the previous section, the canonical name of the host `ftp.kohala.com` would be `bsdi.kohala.com`. Also, if we call `gethostbyname` from the host `solaris` with an unqualified hostname, say `solaris`, the FQDN (`solaris.kohala.com`) is returned as the canonical name.

When IPv6 addresses are returned, the `h_addrtype` member of the `hostent` structure is set to `AF_INET6` and the `h_length` member is set to 16. Figure 9.3 shows these changes, with the shaded fields having changed from Figure 9.2.

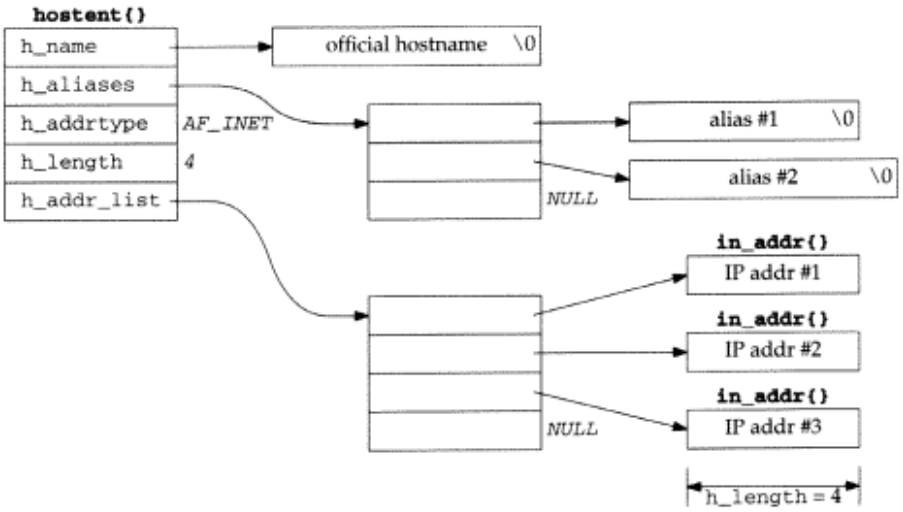

Figure 9.2 `hostent` structure and the information it contains.

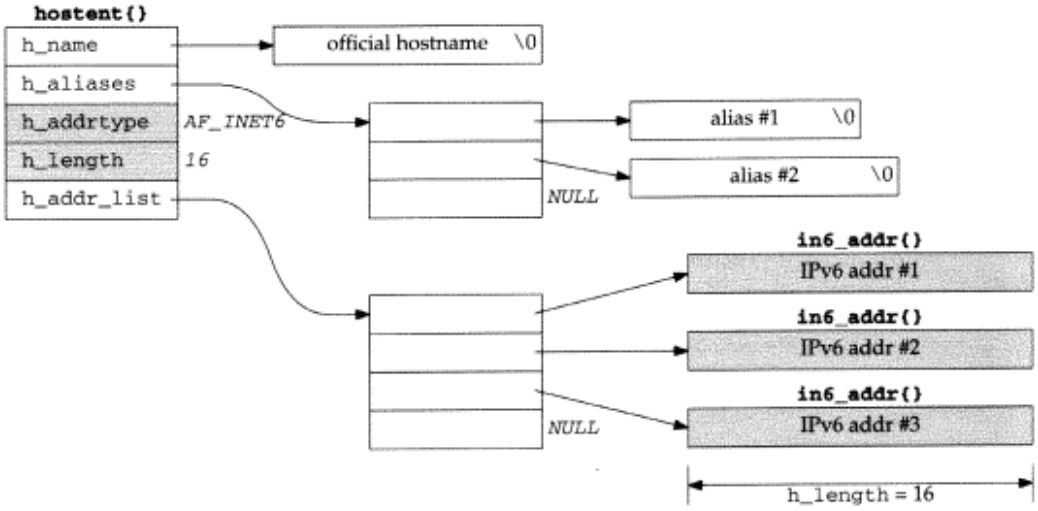

Figure 9.3 Changes in information returned in `hostent` structure with IPv6 addresses.

Recent versions of `gethostbyname`, starting around BIND release 4.9.2, allow the *hostname* argument to be a dotted-decimal string. That is, a call of the form

```
hptr = gethostbyname("206.62.226.33");
```

will work. This code was added because the Rlogin client accepts only a hostname, calling `gethostbyname`, and will not accept a dotted-decimal string [Vixie 1996].

gethostbyname differs from the other socket functions that we have described in that it does not set errno when an error occurs. Instead, it sets the global integer h_errno to one of the following constants defined by including <netdb.h>:

- HOST_NOT_FOUND
- TRY_AGAIN
- NO_RECOVERY
- NO_DATA (identical to NO_ADDRESS)

The NO_DATA error means the specified name is valid, but it does not have either an A record or a AAAA record. An example of this is a hostname with only an MX record.

Current releases of BIND provide the function hstrerror that takes an h_errno value as its only argument and returns a const char * pointer to a description of the error. We show some examples of the strings returned by this function in the next example.

Example

Figure 9.4 shows a simple program that calls gethostbyname for any number of command-line arguments and prints all the returned information.

8-14 gethostbyname is called for each command-line argument.

15-17 The official hostname is output followed by the list of alias names.

20-22 For this program to support both IPv4 and IPv6 addresses we allow the returned address type to be either AF_INET or AF_INET6. But we do not allow the latter unless it is defined (i.e., the host supports IPv6).

23-26 pptr points to the array of pointers to the individual addresses. For each address we call inet_ntop and print the returned string. Note that inet_ntop handles both IPv4 and IPv6 addresses, based on its first argument. Also notice that we defined str of length INET6_ADDRSTRLEN, which we said in Section 3.7 is large enough for the longest possible IPv6 address string. In our unp.h file we define this constant, even if the host does not support IPv6, so that we can always count on it being defined (avoiding yet another #ifdef within our code).

We first execute the program with the name of our host solaris, which has just one IPv4 address.

```
solaris % hostent solaris
official hostname: solaris.kohala.com
        address: 206.62.226.33
```

Notice that the official hostname is the FQDN. Also notice that even though this host has an IPv6 address, only the IPv4 address is returned. Next is a host with multiple IPv4 addresses.

```
solaris % hostent gemini.tuc.noao.edu
official hostname: gemini.tuc.noao.edu
        address: 140.252.1.11
        address: 140.252.3.54
        address: 140.252.4.54
        address: 140.252.8.54
```

names/hostent.c

```
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     char   *ptr, **pptr;
 6     char    str[INET6_ADDRSTRLEN];
 7     struct hostent *hptr;

 8     while (--argc > 0) {
 9         ptr = *++argv;
10         if ( (hptr = gethostbyname(ptr)) == NULL) {
11             err_msg("gethostbyname error for host: %s: %s",
12                     ptr, hstrerror(h_errno));
13             continue;
14         }
15         printf("official hostname: %s\n", hptr->h_name);

16         for (pptr = hptr->h_aliases; *pptr != NULL; pptr++)
17             printf("\talias: %s\n", *pptr);

18         switch (hptr->h_addrtype) {
19         case AF_INET:
20 #ifdef  AF_INET6
21         case AF_INET6:
22 #endif
23             pptr = hptr->h_addr_list;
24             for ( ; *pptr != NULL; pptr++)
25                 printf("\taddress: %s\n",
26                     Inet_ntop(hptr->h_addrtype, *pptr, str, sizeof(str)));
27             break;

28         default:
29             err_ret("unknown address type");
30             break;
31         }
32     }
33     exit(0);
34 }
```

names/hostent.c

Figure 9.4 Call `gethostbyname` and print returned information.

Next is a name that we showed in Section 9.2 as having a CNAME record.

```
solaris % hostent www
official hostname: bsdi.kohala.com
        alias: www.kohala.com
        address: 206.62.226.35
```

As expected, the official hostname differs from our command-line argument.

To see the error strings returned by the `hstrerror` function we first specify a nonexistent hostname, and then a name that has only an MX record.

```
solaris % hostent nosuchname
gethostbyname error for host: nosuchname: Unknown host

solaris % hostent uunet.uu.net
gethostbyname error for host: uunet.uu.net: No address associated with name
```

9.4 RES_USE_INET6 Resolver Option

Newer releases of BIND (4.9.4 and later) provide a resolver option named RES_USE_INET6 that we can set in three different ways. We can use this option to tell the resolver that we want IPv6 addresses returned by gethostbyname, instead of IPv4 addresses.

1. An application can set this option itself by first calling the resolver's res_init function and then enabling the option:

   ```
   #include <resolv.h>

   res_init();
   _res.options |= RES_USE_INET6;
   ```

   This must be done before the first call to gethostbyname or gethostbyaddr. The effect of this option is only on the application that sets the option.

2. If the environment variable RES_OPTIONS contains the string inet6, the option is enabled. The effect of this option depends on the scope of the environment variable. If we set it in our .profile file for example (assuming a Korn-Shell) with the export attribute, as in

   ```
   export RES_OPTIONS=inet6
   ```

   then it affects every program that we run from our login shell. But if we just set the variable on a command line (as we show shortly), then it affects only that command.

3. The resolver configuration file (normally /etc/resolv.conf) can contain the line

   ```
   options inet6
   ```

   Be aware, however, that setting this option in the resolver configuration file affects *all* applications on the host that call the resolver functions. Therefore this technique should not be used until all applications on the host are capable of handling IPv6 addresses returned in a hostent structure.

The first method sets the option on a per-application basis, the second method on a per-user basis, and the third method on a per-system basis.

We now run our example program from Figure 9.4 setting the environment variable RES_OPTIONS to the value inet6.

```
solaris % RES_OPTIONS=inet6 hostent solaris      a name with a AAAA record
official hostname: solaris.kohala.com
        address: 5f1b:df00:ce3e:e200:20:800:2078:e3e3

solaris % RES_OPTIONS=inet6 hostent bsdi         a name without a AAAA record
official hostname: bsdi.kohala.com
        address: ::ffff:206.62.226.35
        address: ::ffff:206.62.226.66
```

The first time we execute our program it returns the IPv6 address of the host (recall its AAAA record in Section 9.2). The second time we execute our program we specify a hostname that does not have a AAAA record. Still IPv6 addresses are returned: the IPv4-mapped IPv6 addresses (Section A.5).

We talk more about the IPv6 support in the resolver in the next two sections.

9.5 gethostbyname2 Function and IPv6 Support

When support for IPv6 was added to BIND 4.9.4, the function `gethostbyname2` was added, which has two arguments, allowing us to specify the address family.

```
#include <netdb.h>

struct hostent *gethostbyname2(const char *hostname, int family);
```

Returns: nonnull pointer if OK, NULL on error with h_errno set

The return value is the same as with `gethostbyname`, a pointer to a `hostent` structure, and this structure remains the same. The logic of the function depends on the *family* argument and on the `RES_USE_INET6` resolver option (which we mentioned at the end of the previous section).

Before describing the details, Figure 9.5 summarizes the operation of `gethostbyname` and `gethostbyname2` with regard to the new `RES_USE_INET6` option. We show in a bolder font the values that can change:

- whether the `RES_USE_INET6` option is **off** or **on**,
- whether the second argument to `gethostbyname2` is **`AF_INET`** or **`AF_INET6`**,
- whether the resolver searches for **A** records or **AAAA** records, and
- whether the returned addresses are of length **4** or **16**.

The operation of `gethostbyname2` is as follows:

- If the *family* argument is `AF_INET`, a query is made for A records. If unsuccessful, the function returns a null pointer. If successful, the type and size of the returned addresses depends on the new `RES_USE_INET6` resolver option: if the option is not set (the default), IPv4 addresses are returned and the `h_length` member of the `hostent` structure will be 4; if the option is set, IPv4-mapped IPv6 addresses are returned and the `h_length` member of the `hostent` structure will be 16.

	RES_USE_INET6 option	
	off	**on**
gethostbyname (*host*)	Search for **A** records. If found, return IPv4 addresses (h_length = **4**). Else error. This provides backward compatibility for all existing IPv4 applications.	Search for **AAAA** records. If found, return IPv6 addresses (h_length = **16**). Else search for **A** records. If found, return IPv4-mapped IPv6 addresses (h_length = **16**). Else error.
gethostbyname2 (*host*, **AF_INET**)	Search for **A** records. If found, return IPv4 addresses (h_length = **4**). Else error.	Search for **A** records. If found, return IPv4-mapped IPv6 addresses (h_length = **16**). Else error.
gethostbyname2 (*host*, **AF_INET6**)	Search for **AAAA** records. If found, return IPv6 addresses (h_length = **16**). Else error.	Search for **AAAA** records. If found, return IPv6 addresses (h_length = **16**). Else error.

Figure 9.5 gethostbyname and gethostbyname2 with resolver RES_USE_INET6 option.

- If the *family* argument is AF_INET6, a query is made for AAAA records. If successful, IPv6 addresses are returned and the h_length member of the hostent structure will be 16; otherwise the function returns a null pointer.

This function can be used if the application wants to force a search for one specific type of address, either IPv4 or IPv6. But it is more common for applications to call gethostbyname, and newer versions of this function can return either IPv4 or IPv6 addresses.

One way to describe the actions of gethostbyname and the RES_USE_INET6 option is to look at its source code, which we show in Figure 9.6.

If the resolver has not yet been initialized (the RES_INIT flag is not set), res_init is called. This initialization function examines and processes the RES_OPTIONS environment variable. If this variable contains the string inet6 or if the resolver configuration file contains the options inet6 line, then the flag RES_USE_INET6 is set by res_init. The res_init function is normally called automatically by gethostbyname (as we show here) the first time it is called by the application, or by gethostbyaddr. Alternately, we showed that the application can also call res_init and then set the RES_USE_INET6 flag explicitly.

If the RES_USE_INET6 option is *not* set, the last line of the function is executed and gethostbyname2 is called with an address family argument of AF_INET. We saw in Figure 9.5 that this call searches for only A records. This provides backward compatibility for all existing applications.

If the RES_USE_INET6 option is enabled, gethostbyname2 is called with an address family argument of AF_INET6 to search for AAAA records (Figure 9.5). If this succeeds, gethostbyname returns. If this fails, gethostbyname2 is called with an address family argument of AF_INET to search for A records. If this succeeds, what is

```
struct hostent *
gethostbyname(const char *name)
{
    struct hostent *hp;

    if ((_res.options & RES_INIT) == 0 && res_init() == -1) {
            h_errno = NETDB_INTERNAL;
            return (NULL);
    }

    if (_res.options & RES_USE_INET6) {
        hp = gethostbyname2(name, AF_INET6);
        if (hp)
            return (hp);
    }
    return (gethostbyname2(name, AF_INET));
}
```

Figure 9.6 `gethostbyname` function and IPv6 support.

not apparent in Figure 9.6 is that these 4-byte addresses are automatically mapped into 16-byte IPv4-mapped IPv6 addresses.

In summary, when the `RES_USE_INET6` option is enabled and the application calls `gethostbyname`, the application is telling the resolver "I want only IPv6 addresses returned, period. Search for AAAA records first, but if none are found then search for A records and if they are found, return the addresses as IPv4-mapped IPv6 addresses."

9.6 `gethostbyaddr` Function

The function `gethostbyaddr` takes a binary IP address and tries to find the hostname corresponding to that address. This is the reverse of `gethostbyname`.

```
#include <netdb.h>

struct hostent *gethostbyaddr(const char *addr, size_t len, int family);
```

Returns: nonnull pointer if OK, `NULL` on error with `h_errno` set

This function returns a pointer to the same `hostent` structure that we described with `gethostbyname`. The field of interest in this structure is normally `h_name`, the canonical hostname.

The *addr* argument is not a `char*` but is really a pointer to an `in_addr` or `in6_addr` structure containing the IPv4 or IPv6 address. *len* is the size of this structure: 4 for an IPv4 addresses, or 16 for an IPv6 address. The *family* argument is either `AF_INET` or `AF_INET6`.

In terms of the DNS, `gethostbyaddr` queries a name server for a PTR record in the `in-addr.arpa` domain for an IPv4 address, or a PTR record in the `ip6.int` domain for an IPv6 address.

gethostbyaddr Function and IPv6 Support

gethostbyaddr has always had an address family argument, so when IPv6 support was added to BIND there was no need to invent another function (similar to gethostbyname2). But there are a few differences with gethostbyaddr when the argument is an IPv6 address. The following three tests are applied in the order listed:

1. If the *family* is AF_INET6, the *len* is 16, and the address is an IPv4-mapped IPv6 address, then the low-order 32 bits of the address (the IPv4 portion) are looked up in the in-addr.arpa domain.
2. If the *family* is AF_INET6, the *len* is 16, and the address is an IPv4-compatible IPv6 address, then the low-order 32 bits of the address (the IPv4 portion) are looked up in the in-addr.arpa domain.
3. If an IPv4 address was looked up (either the *family* argument was AF_INET or one of the two cases above were true) and the RES_USE_INET6 resolver option is set, then the one returned address (a copy of the *addr* argument) is converted to an IPv4-mapped address: h_addrtype is AF_INET6 and h_length is 16.

The third point is usually of little importance because few applications examine the IP address returned by gethostbyaddr, since it is just a copy of the argument. Applications normally call this function to examine the h_name member of the returned hostent structure (and possibly the aliases too).

9.7 uname Function

The uname function returns the name of the current host. This function is not part of the resolver library, but we cover it here because it is often used along with gethostbyname to determine the local host's IP addresses.

```
#include <sys/utsname.h>

int uname(struct utsname *name);
```

Returns: nonnegative value if OK, –1 on error

This function fills in a utsname structure whose address is passed by the caller:

```
#define _UTS_NAMESIZE   16
#define _UTS_NODESIZE   256

struct utsname {
  char  sysname[_UTS_NAMESIZE];  /* name of this operating system */
  char  nodename[_UTS_NODESIZE]; /* name of this node */
  char  release[_UTS_NAMESIZE];  /* O.S. release level */
  char  version[_UTS_NAMESIZE];  /* O.S. version level */
  char  machine[_UTS_NAMESIZE];  /* hardware type */
};
```

Unfortunately all that Posix.1 specifies is the names of the five structure members that we show, and that each array is a null-terminated array of characters. Nothing is said about the size of each array or its contents. The sizes that we show are from 4.4BSD. Other operating systems use different sizes.

The most important omission, from our network programming perspective, is a definition of the size and contents of the `nodename` array. Some systems store only the hostname in this array (e.g., `gemini`) while others store the FQDN (e.g., `gemini.tuc.noao.edu`). On some operating systems, such as Solaris 2.x, it can contain either, depending on how the operating system was installed by the administrator.

Example: Determine Local Host's IP addresses

To determine the local host's IP addresses we call `uname` to obtain the host's name, and then `gethostbyname` to obtain all of its IP addresses. The `my_addrs` function shown in Figure 9.7 performs these steps.

lib/my_addrs.c

```
 1 #include    "unp.h"
 2 #include    <sys/utsname.h>

 3 char **
 4 my_addrs(int *addrtype)
 5 {
 6     struct hostent *hptr;
 7     struct utsname myname;

 8     if (uname(&myname) < 0)
 9         return (NULL);

10     if ( (hptr = gethostbyname(myname.nodename)) == NULL)
11         return (NULL);

12     *addrtype = hptr->h_addrtype;
13     return (hptr->h_addr_list);
14 }
```

lib/my_addrs.c

Figure 9.7 Function to return all of a host's IP addresses.

The return value of the function is the `h_addr_list` member of the `hostent` structure, the array of pointers to the IP addresses. We also return the address family through the pointer argument.

Another way to determine the local host's IP addresses is with the `SIOCGIFCONF` command of `ioctl`. We discuss this in Chapter 16.

9.8 `gethostname` Function

The `gethostname` function also returns the name of the current host.

```
#include <unistd.h>

int gethostname(char *name, size_t namelen);
```

Returns: 0 if OK, –1 on error

name is a pointer to where the hostname is stored, and *namelen* is the size of this array. If there is room, the hostname is null terminated. The maximum size of the hostname is normally the MAXHOSTNAMELEN constant that is defined by including the <sys/param.h> header.

Historically uname was defined by System V and gethostname by Berkeley. Posix.1 specifies uname but Unix 98 requires both.

9.9 getservbyname and getservbyport Functions

Services, like hosts, are often known by names too. If we refer to a service in our code by its name, instead of by its port number, and if the mapping from the name to port number is contained in a file (normally /etc/services), then if the port number changes, all we need modify is one line in the /etc/services file, instead of having to recompile the applications. The next function, getservbyname, looks up a service given its name.

```
#include <netdb.h>

struct servent *getservbyname(const char *servname, const char *protoname);
```

Returns: nonnull pointer if OK, NULL on error

This function returns a pointer to the following structure:

```
struct servent {
  char   *s_name;      /* official service name */
  char  **s_aliases;   /* alias list */
  int     s_port;      /* port number, network-byte order */
  char   *s_proto;     /* protocol to use */
};
```

The service name *servname* must be specified. If a protocol is also specified (if *protoname* is a nonnull pointer), then the entry must also have a matching protocol. Some Internet services are provided using either TCP or UDP (for example, the DNS and all the services in Figure 2.13), while others support only a single protocol (FTP requires TCP.) If *protoname* is not specified and the service supports multiple protocols, it is implementation dependent which port number is returned. Normally this does not matter, because services that support multiple protocols often use the same TCP and UDP port number, but this is not guaranteed.

The main field of interest in the servent structure is the port number. Since the port number is returned in network-byte order, we must not call htons when storing this into a socket address structure.

Typical calls to this function could be

```
struct servent  *sptr;

sptr = getservbyname("domain", "udp"); /* DNS using UDP */
sptr = getservbyname("ftp", "tcp");    /* FTP using TCP */
sptr = getservbyname("ftp", NULL);     /* FTP using TCP */
sptr = getservbyname("ftp", "udp");    /* this call will fail */
```

Since FTP supports only TCP, the second and third calls are the same, and the fourth call will fail. Typical lines from the `/etc/services` file are

```
solaris % grep -e ftp -e domain /etc/services
ftp-data        20/tcp
ftp             21/tcp
domain          53/udp
domain          53/tcp
tftp            69/udp
```

The next function, `getservbyport`, looks up a service given its port number and an optional protocol.

```
#include <netdb.h>

struct servent *getservbyport(int port, const char *protname);
```

Returns: nonnull pointer if OK, NULL on error

The *port* value must be network byte ordered. Typical calls to this function could be

```
struct servent  *sptr;

sptr = getservbyport(htons(53), "udp");  /* DNS using UDP */
sptr = getservbyport(htons(21), "tcp");  /* FTP using TCP */
sptr = getservbyport(htons(21), NULL);   /* FTP using TCP */
sptr = getservbyport(htons(21), "udp");  /* this call will fail */
```

The last call fails because there is no service that uses port 21 with UDP.

Be aware that a few port numbers are used with TCP for one service, but the same port number is used with UDP for a totally different service. For example,

```
solaris % grep 514 /etc/services
shell           514/tcp
syslog          514/udp
```

shows that port 514 is used by the `rsh` command with TCP but with the `syslog` daemon with UDP. Ports 512–514 have this property.

Example: Using `gethostbyname` and `getservbyname`

We can now modify our TCP daytime client from Figure 1.5 to use `gethostbyname` and `getservbyname` and take two command-line arguments: a hostname and a service name. Figure 9.8 shows our program. This program also shows the desired behavior of

attempting to connect to all the IP addresses for a multihomed server, until one succeeds or all the addresses have been tried.

names/daytimetcpcli1.c

```
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd, n;
 6     char    recvline[MAXLINE + 1];
 7     struct sockaddr_in servaddr;
 8     struct in_addr **pptr;
 9     struct hostent *hp;
10     struct servent *sp;

11     if (argc != 3)
12         err_quit("usage: daytimetcpcli1 <hostname> <service>");

13     if ( (hp = gethostbyname(argv[1])) == NULL)
14         err_quit("hostname error for %s: %s", argv[1], hstrerror(h_errno));

15     if ( (sp = getservbyname(argv[2], "tcp")) == NULL)
16         err_quit("getservbyname error for %s", argv[2]);

17     pptr = (struct in_addr **) hp->h_addr_list;
18     for ( ; *pptr != NULL; pptr++) {
19         sockfd = Socket(AF_INET, SOCK_STREAM, 0);

20         bzero(&servaddr, sizeof(servaddr));
21         servaddr.sin_family = AF_INET;
22         servaddr.sin_port = sp->s_port;
23         memcpy(&servaddr.sin_addr, *pptr, sizeof(struct in_addr));
24         printf("trying %s\n",
25                Sock_ntop((SA *) &servaddr, sizeof(servaddr)));

26         if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) == 0)
27             break;              /* success */
28         err_ret("connect error");
29         close(sockfd);
30     }
31     if (*pptr == NULL)
32         err_quit("unable to connect");

33     while ( (n = Read(sockfd, recvline, MAXLINE)) > 0) {
34         recvline[n] = 0;        /* null terminate */
35         Fputs(recvline, stdout);
36     }
37     exit(0);
38 }
```

names/daytimetcpcli1.c

Figure 9.8 Our daytime client that uses `gethostbyname` and `getservbyname`.

Call `gethostbyname` and `getservbyname`

13-16 The first command-line argument is a hostname, which we pass as an argument to `gethostbyname` and the second is a service name, which we pass as an argument to

`getservbyname`. Our code assumes TCP, and that is what we code as the second argument to `getservbyname`.

Try each server address

18-25 We now code the calls to `socket` and `connect` in a loop that is executed for every server address until a `connect` succeeds or the list of IP addresses is exhausted. After calling `socket`, we fill in an Internet socket address structure with the IP address and port of the server. While we could move the call to `bzero` and the subsequent two assignments out of the loop, for efficiency, the code is easier to read as shown. Establishing the connection with the server is rarely the performance bottleneck of a network client.

Call `connect`

26-30 `connect` is called, and if it succeeds, `break` terminates the loop. If the connection establishment fails, we print an error and close the socket. Recall that a descriptor that fails a call to `connect` must be closed and is no longer usable.

Check for failure

31-32 If the loop terminates because no call to `connect` succeeded, the program terminates.

Read server's reply

33-37 Otherwise we read the server's response, terminating when the server closes the connection.

If we run this program specifying one of our hosts that is running the daytime server we get the expected output:

```
solaris % daytimetcpcli1 aix daytime
trying 206.62.226.35.13
Thu May 22 19:28:11 1997
```

What is more interesting is to run the program to a multihomed router that is not running the daytime server:

```
solaris % daytimetcpcli1 gateway.tuc.noao.edu daytime
trying 140.252.1.4.13
connect error: Connection refused
trying 140.252.101.4.13
connect error: Connection refused
trying 140.252.102.1.13
connect error: Connection refused
trying 140.252.104.1.13
connect error: Connection refused
trying 140.252.3.6.13
connect error: Connection refused
trying 140.252.4.100.13
connect error: Connection refused
unable to connect
```

9.10 Other Networking Information

Our focus in this chapter has been on hostnames and IP addresses and service names and their port numbers. But looking at the bigger picture, there are four types of information (related to networking) that an application might want to look up: hosts, networks, protocols, and services. Most lookups are for hosts (`gethostbyname` and `gethostbyaddr`), with a smaller number for services (`getservbyname` and `getservbyaddr`), and an even smaller number for networks and protocols.

All four types of information can be stored in a file and three functions are defined for each of the four types:

1. A `getXXXent` function that reads the next entry in the file, opening the file if necessary.
2. A `setXXXent` function that opens (if not already open) and rewinds the file.
3. A `endXXXent` function that closes the file.

Each of the four types of information defines its own structure, and these definitions are defined by including the `<netdb.h>` header: the `hostent`, `netent`, `protoent`, and `servent` structures.

In addition to the three `get`, `set`, and `end` functions, which allow sequential processing of the file, each of the four types of information provides some *keyed lookup* functions. These functions go through the file sequentially (calling the `getXXXent` function to read each line) but instead of returning each line to the caller, these functions look for an entry that matches an argument. These keyed lookup functions have names of the form `getXXXbyYYY`. For example, the two keyed lookup functions for the host information are `gethostbyname` (look for an entry that matches a hostname) and `gethostbyaddr` (look for an entry that matches an IP address). Figure 9.9 summarizes this information.

Information	Data file	Structure	Keyed lookup functions
hosts	`/etc/hosts`	`hostent`	`gethostbyaddr`, `gethostbyname`
networks	`/etc/networks`	`netent`	`getnetbyaddr`, `getnetbyname`
protocols	`/etc/protocols`	`protoent`	`getprotobyname`, `getprotobynumber`
services	`/etc/services`	`servent`	`getservbyname`, `getservbyport`

Figure 9.9 Four types of network-related information.

How does this apply when the DNS is being used? First, only the host and network information is available through the DNS. The protocol and service information is always read from the corresponding file. We mentioned earlier in this chapter (with Figure 9.1) that different implementations use different ways for the administrator to specify whether to use the DNS or a file for the host and network information.

Second, if the DNS is being used for the host and network information, then only the keyed lookup functions make sense. You cannot, for example, use `gethostent` and expect to sequence through all entries in the DNS! If `gethostent` is called, it reads only the hosts file and avoids the DNS.

> Although the network information can be made available through the DNS, few people set this up. Pages 346–348 of [Albitz and Liu 1997] describe this feature. Typically, however, administrators build and maintain an `/etc/networks` file and it is used instead of the DNS. The `netstat` program with the `-i` option uses this file, if present, and prints the name for each network.

9.11 Summary

The set of functions that an application calls to convert a hostname into an IP address and vice versa is called the resolver. The two functions `gethostbyname` and `gethostbyaddr` are the common entry points. With the move to IPv6 the `hostent` structure filled in by these two functions remains the same, but some of the information within this structure changes. A new function, `gethostbyname2`, and a new resolver option, `RES_USE_INET6`, are also needed for IPv6 support.

The commonly used function dealing with service names and port numbers is `getservbyname`, which takes a service name and returns a structure containing the port number. This mapping is normally contained in a text file. Additional functions exist to map protocol names into protocol numbers, and network names into network numbers, but these are rarely used.

Another alternative that we have not mentioned is calling the resolver functions directly, instead of using `gethostbyname` and `gethostbyaddr`. One program that invokes the DNS this way is `sendmail` to search for an MX record, something that the `gethostbyXXX` functions cannot do. The resolver functions have names that begin with `res_` and the `res_init` function that we described in Section 9.4 is an example. A description of these functions and an example program that calls them is in Chapter 14 of [Albitz and Liu 1997] and typing `man resolver` should display the manual pages for these functions.

We continue the topic of name and address conversions in Chapter 11 when we look at a protocol-independent interface to `gethostbyname` and `gethostbyaddr`: the `getaddrinfo` and `getnameinfo` functions. These two new functions were designed to work with IPv4 and IPv6, but we first need to look at some interoperability aspects of IPv4 and IPv6 in the next chapter, before discussing the newer functions.

Exercises

9.1 Modify the program in Figure 9.4 to call `gethostbyaddr` for each returned address, and then print the `h_name` that is returned. First run the program specifying a hostname with just one IP address and then run the program specifying a hostname that has more than one IP address. What happens?

9.2 Fix the problem shown in the preceding exercise.

9.3 Modify Figure 9.7 to call `gethostname` instead of `uname`. Write a `main` function to call `my_addrs` and then print the IP addresses.

9.4 In Figure 9.8 what can happen if we change the third argument to `memcpy` (when filling in the socket address structure) to be `hp->h_length`? (*Hint*: Consider what happens if we set `RES_OPTIONS=inet6` when executing the program and specify a hostname that has an IPv6 address.)

9.5 Run Figure 9.8 specifying a service name of `chargen`.

9.6 Run Figure 9.8 specifying a dotted-decimal IP address as the hostname. Does your resolver allow this? Modify Figure 9.8 to allow a dotted-decimal IP address as the hostname and to allow a decimal port number string as the service name. In testing the IP address for either a dotted-decimal string or a hostname, in what order should these two tests be performed?

9.7 Modify Figure 9.8 to work with either IPv4 of IPv6.

9.8 Modify Figure 8.9 to query the DNS and compare the returned IP address with all of the destination host's IP addresses. That is, call `gethostbyaddr` using the IP address returned by `recvfrom`, followed by `gethostbyname` to find all the IP addresses for the host.

Part 3

Advanced Sockets

10

IPv4 and IPv6 Interoperability

10.1 Introduction

Over the coming years there will probably be a gradual transition of the Internet from IPv4 to IPv6. During this transition phase it is important that existing IPv4 applications continue to work with newer IPv6 applications. For example, a vendor cannot provide a Telnet client that works only with IPv6 Telnet servers but must provide one that works with IPv4 servers and one that works with IPv6 servers. Better yet would be one IPv6 Telnet client that can work with both IPv4 and IPv6 servers, along with one Telnet server that can work with both IPv4 and IPv6 clients. We will see how this is done in this chapter.

We assume throughout this chapter that the hosts are running *dual stacks*, that is both an IPv4 protocol stack and an IPv6 protocol stack. Our example in Figure 2.1 is a dual-stack host. Hosts and routers will probably run like this for many years into the transition to IPv6. At some point many systems will be able to turn off their IPv4 stack, but only time will tell when (and if) that will occur.

In this chapter we discuss how IPv4 applications and IPv6 applications can communicate with each other. There are four combinations of clients and servers using either IPv4 or IPv6 and we show these in Figure 10.1.

	IPv4 server	IPv6 server
IPv4 client	Almost all existing clients and servers.	Discussed in Section 10.2.
IPv6 client	Discussed in Section 10.3.	Simple modifications to most existing clients and servers (e.g., Figure 1.5 to Figure 1.6).

Figure 10.1 Combinations of clients and servers using IPv4 or IPv6.

We will not say much more about the two scenarios where the client and server use the same protocol. The interesting cases are when the client and server use different protocols.

10.2 IPv4 Client, IPv6 Server

A general property of a dual-stack host is that IPv6 servers can handle both IPv4 and IPv6 clients. This is done using IPv4-mapped IPv6 addresses (Figure A.10). Figure 10.2 shows an example of this.

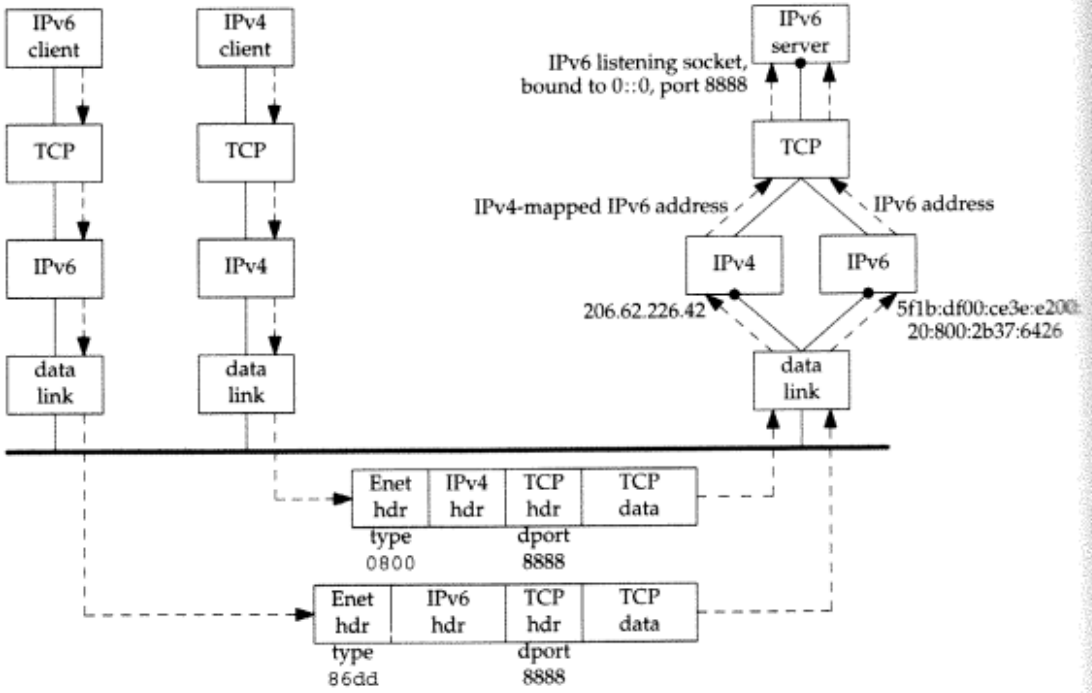

Figure 10.2 IPv6 server on dual-stack host serving IPv4 and IPv6 clients.

We have an IPv4 client and an IPv6 client on the left. The server on the right is written using IPv6 and it is running on a dual-stack host. The server has created an IPv6 listening TCP socket that is bound to the IPv6 wildcard address and TCP port 8888.

We assume the clients and server are on the same Ethernet. They could also be connected by routers, as long as all the routers support IPv4 and IPv6, but that adds nothing to this discussion. Section B.3 discusses a different case where IPv6 clients and servers are connected by IPv4-only routers.

We assume both clients send SYN segments to establish a connection with the server. The IPv4 client host will send the SYN in an IPv4 datagram and the IPv6 client host will send the SYN in an IPv6 datagram. The TCP segment from the IPv4 client

appears on the wire as an Ethernet header followed by an IPv4 header, a TCP header, and the TCP data. The Ethernet header contains a type field of `0x0800`, which identifies the frame as an IPv4 frame. The TCP header contains the destination port of 8888. (Appendix A talks more about the formats and contents of these headers.) The destination IP address in the IPv4 header, which we do not show, would be 206.62.226.42.

The TCP segment from the IPv6 client appears on the wire as an Ethernet header followed by an IPv6 header, a TCP header, and the TCP data. The Ethernet header contains a type field of `0x86dd`, which identifies the frame as an IPv6 frame. The TCP header has the same format as the TCP header in the IPv4 packet and contains the destination port of 8888. The destination IP address in the IPv6 header, which we do not show, would be `5f1b:df00:ce3e:e200:20:800:2b37:6426`.

The receiving datalink looks at the Ethernet type field and passes each frame to the appropriate IP module. The IPv4 module, probably in conjunction with the TCP module, detects that the destination socket is an IPv6 socket, and the source IPv4 address in the IPv4 header is converted into the equivalent IPv4-mapped IPv6 address. That mapped address is returned to the IPv6 socket as the client's IPv6 address when `accept` returns to the server with the IPv4 client connection. All remaining datagrams for this connection are IPv4 datagrams.

When `accept` returns to the server with the IPv6 client connection, the client's IPv6 address does not change from whatever source address appears in the IPv6 header. All remaining datagrams for this connection are IPv6 datagrams.

We can summarize the steps that allow an IPv4 TCP client to communicate with an IPv6 server.

1. The IPv6 server starts, creates an IPv6 listening socket, and we assume it `binds` the wildcard address to the socket.
2. The IPv4 client calls `gethostbyname` and finds an A record for the server (Figure 9.5). The server host will have both an A record and a AAAA record, since it supports both protocols but the IPv4 client asks for only an A record.
3. The client calls `connect` and the client's host sends an IPv4 SYN to the server.
4. The server host receives the IPv4 SYN directed to the IPv6 listening socket, sets a flag indicating that this connection is using IPv4-mapped IPv6 addresses, and responds with an IPv4 SYN/ACK. When the connection is established, the address returned to the server by `accept` is the IPv4-mapped IPv6 address.
5. All communication between this client and server takes place using IPv4 datagrams.
6. Unless the server explicitly checks whether this IPv6 address is an IPv4-mapped IPv6 address (using the `IN6_IS_ADDR_V4MAPPED` macro described in Section 10.4), the server never knows that it is communicating with an IPv4 client. The dual protocol stack handles this detail. Similarly, the IPv4 client has no idea that it is communicating with an IPv6 server.

An underlying assumption in this scenario is that the dual-stack server host has both an IPv4 address and an IPv6 address. This will work until all the IPv4 addresses are taken.

The scenario is similar for an IPv6 UDP server, but the address format can change for each datagram. For example, if the IPv6 server receives a datagram from an IPv4 client, the address returned by `recvfrom` will be the client's IPv4-mapped IPv6 address. The server responds to this client's request by calling `sendto` with the IPv4-mapped IPv6 address as the destination. This address format tells the kernel to send an IPv4 datagram to the client. But the next datagram received for the server could be an IPv6 datagram, and `recvfrom` will return the IPv6 address. If the server responds, the kernel will generate an IPv6 datagram.

Figure 10.3 summarizes how a received IPv4 or IPv6 datagram is processed, depending on the type of the receiving socket, for TCP and UDP, assuming a dual-stack host.

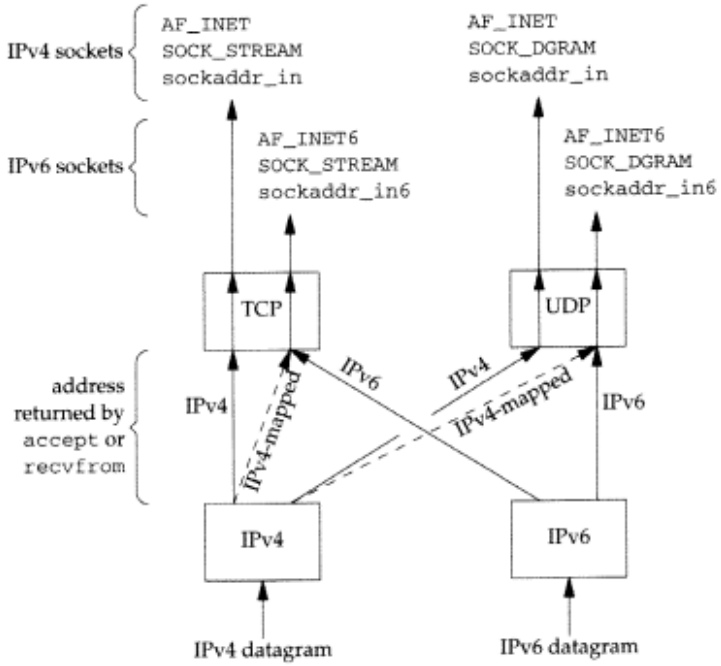

Figure 10.3 Processing of received IPv4 or IPv6 datagrams, depending on type of receiving socket.

- If an IPv4 datagram is received for an IPv4 socket, nothing special is done. These are the two arrows labeled "IPv4" in the figure, one to TCP and one to UDP. IPv4 datagrams are exchanged between the client and server.
- If an IPv6 datagram is received for an IPv6 socket, nothing special is done. These are the two arrows labeled "IPv6" in the figure, one to TCP and one to UDP. IPv6 datagrams are exchanged between the client and server.
- But when an IPv4 datagram is received for an IPv6 socket, the kernel returns the corresponding IPv4-mapped IPv6 address as the address returned by `accept` (TCP) or `recvfrom` (UDP). These are the two dashed arrows in the figure. This mapping is possible because an IPv4 address can always be represented as an

IPv6 address. IPv4 datagrams are exchanged between the client and server.

- The converse of the previous bullet is false: in general an IPv6 address cannot be represented as an IPv4 address; therefore there are no arrows from the IPv6 protocol box to the two IPv4 sockets.

Most dual-stack hosts should use the following rules in dealing with listening sockets:

1. A listening IPv4 socket can accept incoming connections from only IPv4 clients.
2. If a server has a listening IPv6 socket that has bound the wildcard address, that socket can accept incoming connections from either IPv4 clients or IPv6 clients. For a connection from an IPv4 client the server's local address for the connection will be the corresponding IPv4-mapped IPv6 address.
3. If a server has a listening IPv6 socket that has bound an IPv6 address other than an IPv4-mapped IPv6 address, that socket can accept incoming connections from IPv6 clients only.

10.3 IPv6 Client, IPv4 Server

We now swap the protocols used by the client and server from the example in the previous section. First consider an IPv6 TCP client running on a dual-stack host.

1. An IPv4 server starts on an IPv4-only host and creates an IPv4 listening socket.
2. The IPv6 client starts, calls `gethostbyname` asking for only IPv6 addresses (it enables the `RES_USE_INET6` option). Since the IPv4-only server host has only A records, we see from Figure 9.5 that an IPv4-mapped IPv6 address is returned to the client.
3. The IPv6 client calls `connect` with the IPv4-mapped IPv6 address in the IPv6 socket address structure. The kernel detects the mapped address and automatically sends an IPv4 SYN to the server.
4. The server responds with an IPv4 SYN/ACK, and the connection is established using IPv4 datagrams.

We can summarize this scenario in Figure 10.4.

- If an IPv4 TCP client calls `connect` specifying an IPv4 address, or if an IPv4 UDP client calls `sendto` specifying an IPv4 address, nothing special is done. These are the two arrows labeled "IPv4" in the figure.
- If an IPv6 TCP client calls `connect` specifying an IPv6 address, or if an IPv6 UDP client calls `sendto` specifying an IPv6 address, nothing special is done. These are the two arrow labeled "IPv6" in the figure.

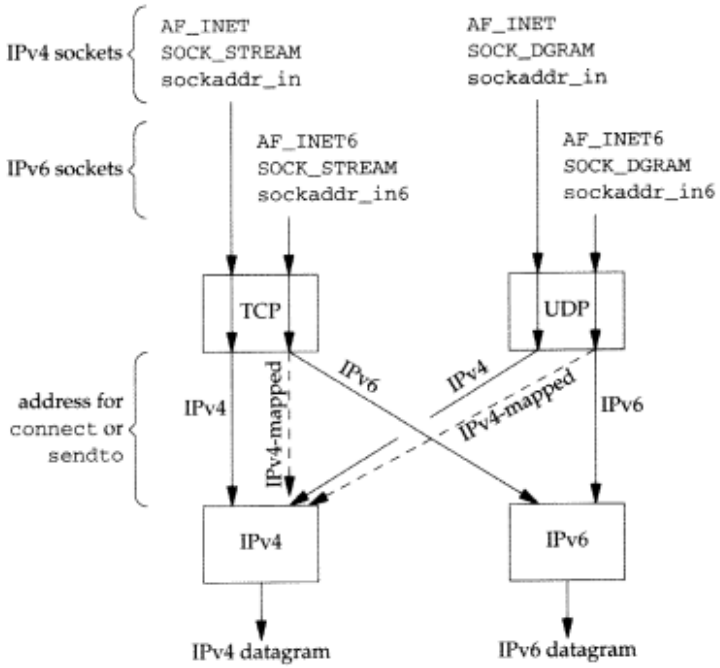

Figure 10.4 Processing of client requests, depending on address type and socket type.

- If an IPv6 TCP client specifies an IPv4-mapped IPv6 address to `connect` or if an IPv6 UDP client specifies an IPv4-mapped IPv6 address to `sendto`, the kernel detects the mapped address and causes an IPv4 datagram to be sent, instead of an IPv6 datagram. These are the two dashed arrows in the figure.
- An IPv4 client cannot specify an IPv6 address to either `connect` or `sendto` because a 16-byte IPv6 address does not fit in the 4-byte `in_addr` structure within the IPv4 `sockaddr_in` structure. Therefore there are no arrows from the IPv4 sockets to the IPv6 protocol box in the figure.

In the previous section (an IPv4 datagram arriving for an IPv6 server socket) the conversion of the received address to the IPv4-mapped IPv6 address is done by the kernel and returned transparently to the application by `accept` or `recvfrom`. In this section (an IPv4 datagram needing to be sent on an IPv6 socket) the conversion of the IPv4 address to the IPv4-mapped IPv6 address is done by the resolver according to the rules in Figure 9.5, and the mapped address is then passed transparently by the application to `connect` or `sendto`.

Summary of Interoperability

Figure 10.5 summarizes this section and the previous section and the combinations of clients and servers.

	IPv4 server IPv4-only host (A only)	IPv6 server IPv6-only host (AAAA only)	IPv4 server dual-stack host (A and AAAA)	IPv6 server dual-stack host (A and AAAA)
IPv4 client, IPv4-only host	IPv4	(no)	IPv4	IPv4
IPv6 client, IPv6-only host	(no)	IPv6	(no)	IPv6
IPv4 client, dual-stack host	IPv4	(no)	IPv4	IPv4
IPv6 client, dual-stack host	IPv4	IPv6	(no*)	IPv6

Figure 10.5 Summary of interoperability between IPv4 and IPv6 clients and servers.

Each box contains "IPv4" or "IPv6" if the combination is OK, indicating which protocol is used, or "(no)" if the combination is invalid. The third column on the final row is marked with an asterisk because interoperability depends on the address chosen by the client. Choosing the AAAA record and sending an IPv6 datagram will not work. But choosing the A record, which is returned to the client as an IPv4-mapped IPv6 address, causes an IPv4 datagram to be sent, which will work.

Although it appears that one-fourth of the table will not interoperate, in the real world for the foreseeable future, most implementations of IPv6 will be on dual-stack hosts and will not be IPv6-only implementations. If we then remove the second row and second column, all of the "(no)" entries disappear and the only problem is the entry with the asterisk.

10.4 IPv6 Address Testing Macros

There are a small class of IPv6 applications that must know whether they are talking to an IPv4 peer. These applications need to know if the peer's address is an IPv4-mapped IPv6 address. Twelve macros are defined to test an IPv6 address for certain properties.

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

int IN6_IS_ADDR_UNSPECIFIED(const struct in6_addr *aptr);
int IN6_IS_ADDR_LOOPBACK(const struct in6_addr *aptr);
int IN6_IS_ADDR_MULTICAST(const struct in6_addr *aptr);
int IN6_IS_ADDR_LINKLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_SITELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_V4MAPPED(const struct in6_addr *aptr);
int IN6_IS_ADDR_V4COMPAT(const struct in6_addr *aptr);

int IN6_IS_ADDR_MC_NODELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_LINKLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_SITELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_ORGLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_GLOBAL(const struct in6_addr *aptr);
```

All return: nonzero if IPv6 address is of specified type, 0 otherwise

The first seven macros test the basic type of IPv6 address. We show these various address types in Section A.5. The final five macros test the scope of an IPv6 multicast address (Section 19.2).

An IPv6 client could call the `IN6_IS_ADDR_V4MAPPED` macro to test the IPv6 address returned by the resolver. An IPv6 server could call this macro to test the IPv6 address returned by `accept` or `recvfrom`.

As an example of an application that needs this macro, consider FTP and its PORT command. If we start an FTP client, log in to an FTP server, and issue an FTP `dir` command, the FTP client sends a PORT command to the FTP server across the control connection. This tells the server the client's IP address and port to which the server then creates a data connection. (Chapter 27 of TCPv1 contains all the details on the FTP application protocol.) But an IPv6 FTP client must know whether the server is an IPv4 server or an IPv6 server, because the former requires a command of the form "PORT *a1,a2,a3,a4,p1,p2*" where the first four numbers (each between 0 and 255) form the 4-byte IPv4 address and the last two numbers form the 2-byte port number. An IPv6 server, however, requires an LPRT command (documented in RFC 1639 [Piscitello 1994]) containing 21 numbers. Exercise 10.1 gives an example of both commands.

10.5 `IPV6_ADDRFORM` Socket Option

The `IPV6_ADDRFORM` socket option can change a socket from one type to another, subject to the following restrictions:

1. An IPv4 socket can always be changed to an IPv6 socket. Any IPv4 addresses already associated with the socket are converted to IPv4-mapped IPv6 addresses.
2. An IPv6 socket can be changed to an IPv4 socket only if any addresses already associated with the socket are IPv4-mapped IPv6 addresses.

The reason for wanting to change the address format of a socket is that descriptors can be passed between processes easily under Unix. The most common way is across a `fork`, but we will see how a descriptor can be passed between related processes in Section 14.7 and between unrelated processes in Section 25.7.

As an example, consider a process that creates a listening IPv4 socket and then accepts a connection from an IPv4 client. This server then calls `fork` and `exec`, starting a new program to handle the client. Assume that the convention with this application is that the connected socket is passed to the new program as standard input, standard output, and standard error (this is similar to what `inetd` does, Section 12.5). We could have the pseudocode shown in Figure 10.6. The only difference from our concurrent server in Section 4.8 is duplicating the connected socket to the agreed on descriptors and then calling `exec`.

But the program that is `exec`ed expects an IPv6 socket. We can use the `IPV6_ADDRFORM` socket option to convert the socket's address format, as shown in Figure 10.7.

```
int                 listenfd, connfd;
socklen_t           clilen;
struct sockaddr_in  serv, cli;                  /* IPv4 structs */

listenfd = Socket(AF_INET, SOCK_STREAM, 0);     /* IPv4 socket */

/* fill in serv{} with well-known port */
Bind(listenfd, &serv, sizeof(serv));
Listen(listenfd, LISTENQ);

for ( ; ; ) {
    clilen = sizeof(cli);
    connfd = Accept(listenfd, &cli, &clilen);

    if (Fork() == 0) {
        Close(listenfd);            /* child */
        Dup2(connfd, STDIN_FILENO);
        Dup2(connfd, STDOUT_FILENO);
        Dup2(connfd, STDERR_FILENO);
        Close(connfd);
        Exec( ... );    /* start new program */
    }
    Close(connfd);      /* parent */
}
```

Figure 10.6 Server that accepts incoming connection and execs new program.

```
int                 af;
socklen_t           clilen;
struct sockaddr_in6 cli;      /* IPv6 struct */
struct hostent      *ptr;

af = AF_INET6;
Setsockopt(STDIN_FILENO, IPPROTO_IPV6, IPV6_ADDRFORM, &af, sizeof(af));

clilen = sizeof(cli);
Getpeername(0, &cli, &clilen);

ptr = gethostbyaddr(&cli.sin6_addr, 16, AF_INET6);
```

Figure 10.7 Converting an IPv4 socket to an IPv6 socket.

The call to `setsockopt` changes the address format of the socket from IPv4 to IPv6, and the call to `getpeername` will return an IPv4-mapped IPv6 address, assuming the socket was an IPv4 socket. But if this program is `execed` with an IPv6 socket on standard input, the call to `setsockopt` does nothing, as the address format is already IPv6.

One scenario where this socket option can be used is when the program that accepts the incoming IPv4 connection is provided by someone else (i.e., we do not have the source code to modify the program to use IPv6, or better still to be protocol independent), but our program that is `execed` handles IPv6.

If `getsockopt` is called for `IPV6_ADDRFORM`, the returned value is either `AF_INET` or `AF_INET6`, depending on the address format of the socket. The second

argument to `getsockopt` or `setsockopt` can be either `IPPROTO_IP` or `IPPROTO_IPV6`.

10.6 Source Code Portability

Most existing network applications are written assuming IPv4. `sockaddr_in` structures are allocated and filled in and the calls to `socket` specify `AF_INET` as the first argument. We saw in the conversion from Figure 1.5 to Figure 1.6 that these IPv4 applications can be converted to use IPv6 without too much effort. Many of the changes that we showed could be done automatically using some editing scripts. Programs that are more dependent on IPv4, using features such as multicasting, IP options, or raw sockets, will take more work to convert.

If we convert an application to use IPv6 and distribute it in source code, we now have to worry whether or not the recipient's system supports IPv6. The typical way to handle this is with `#ifdefs` throughout the code, using IPv6 when possible (since we have seen in this chapter that an IPv6 client can still communicate with IPv4 servers, and vice versa). The problem with this approach is that the code becomes littered with `#ifdefs` very quickly, and harder to follow and maintain.

A better approach is to look upon the move to IPv6 as a chance to make the program protocol independent. The first step is to remove calls to `gethostbyname` and `gethostbyaddr` and use the `getaddrinfo` and `getnameinfo` functions that we describe in the next chapter. This lets us deal with socket address structures as opaque objects, referenced by a pointer and size, which is exactly what the basic socket functions do: `bind`, `connect`, `recvfrom`, and so on. Our `sock_XXX` functions from Section 3.8 can help manipulate these, independent of IPv4 or IPv6. Obviously these functions contain `#ifdefs` to handle IPv4 and IPv6, but hiding all of this protocol dependency in a few library functions makes our code simpler. We develop a set of `mcast_XXX` functions in Section 19.6 that can make multicast applications independent of IPv4 or IPv6.

Another point to consider is what happens if we compile our source code on a system that supports both IPv4 and IPv6, distribute either executable code or object files (but not the source code), and someone runs our application on a system that does not support IPv6. There is a chance that the local name server supports AAAA records and returns both AAAA records and A records for some peer with which our application tries to connect. If our application, which is IPv6-capable, calls `socket` to create an IPv6 socket, it will fail if the host does not support IPv6. We handle this in the functions described in the next chapter by ignoring the error from `socket` and trying the next address on the list returned by the name server. Assuming the peer has an A record, and that the name server returns the A record in addition to any AAAA records, the creation of an IPv4 socket will succeed. This is the type of functionality that belongs in a library function, and not in the source code of every application.

10.7 Summary

An IPv6 server on a dual-stack host can service both IPv4 clients and IPv6 clients. An IPv4 client still sends IPv4 datagrams to the server, but the server's protocol stack converts the client's address into an IPv4-mapped IPv6 address, since the IPv6 server is dealing with IPv6 socket address structures.

Similarly an IPv6 client on a dual-stack host can communicate with an IPv4 server. The client's resolver will return IPv4-mapped IPv6 addresses for all of the server's A records, and calling `connect` for one of these addresses results in the dual-stack sending an IPv4 SYN segment. Only a few special clients and servers need to know the protocol being used by the peer (e.g., FTP) and the `IN6_IS_ADDR_V4MAPPED` macro can be used to see if the peer is using IPv4. The `IPV6_ADDRFORM` socket option can be used by a program that expects one type of socket (normally an IPv6 socket).

Exercises

10.1 Start an IPv6 FTP client on a dual-stack host running IPv4 and IPv6. Connect to an IPv4 FTP server, issue the `debug` command, and then the `dir` command. Then perform the same operations but to an IPv6 server, and compare the PORT commands issued as a result of the `dir` commands.

10.2 Write a program that requires one command-line argument that is an IPv4 dotted-decimal address. Create an IPv4 TCP socket and `bind` this address to the socket along with some port, say 8888. Call `listen` and then `pause`. Write a similar program that takes an IPv6 hex string as the command-line argument and creates a listening IPv6 TCP socket. Start the IPv4 program, specifying the wildcard address as the argument. Then go to another window and start the IPv6 program, specifying the IPv6 wildcard address as the argument. Can you start the IPv6 program, since the IPv4 program has already bound that port? Does the `SO_REUSEADDR` socket option make a difference? What if you start the IPv6 program first, and then try to start the IPv4 program?

11

Advanced Name and Address Conversions

11.1 Introduction

The two functions described in Chapter 9, `gethostbyname` and `gethostbyaddr`, are protocol dependent. When using the former, we must know which member of the socket address structure to move the result into (e.g., the `sin_addr` member for IPv4 or the `sin6_addr` member for IPv6), and when calling the latter, we must know which member contains the binary address. This chapter begins with the new Posix.1g `getaddrinfo` function that provides protocol independence for our applications. We cover its complement, `getnameinfo`, later in the chapter.

We then use this function and develop six functions of our own that handle the typical scenarios for TCP and UDP clients and servers. We use these functions throughout the remainder of the text instead of calling `getaddrinfo` directly.

The functions `gethostbyname` and `gethostbyaddr` are also nice examples of functions that are nonreentrant. We show why this is so and describe some replacement functions that avoid this problem. Reentrancy is a problem that we come back to in Chapter 23, but we are able to show and explain the problem now, without having to understand the details of threads.

We finish the chapter showing our complete implementation of `getaddrinfo`. This lets us understand more about the function: how it operates, what it returns, and its interaction with IPv4 and IPv6.

11.2 `getaddrinfo` Function

The `getaddrinfo` function hides all of the protocol dependencies in the library function, which is where they belong. The application deals only with the socket address structures that are filled in by `getaddrinfo`. This function is defined in Posix.1g.

The Posix.1g definition of this function comes from an earlier proposal by Keith Sklower for a function named `getconninfo`. This function was the result of discussions with Eric Allman, William Durst, Michael Karels, and Steven Wise and from an early implementation written by Eric Allman. The observation that specifying a hostname and a service name would suffice for connecting to a service independent of protocol details was made by Marshall Rose in a proposal to X/Open.

```
#include <netdb.h>

int getaddrinfo(const char *hostname, const char *service,
                const struct addrinfo *hints, struct addrinfo **result);
```

Returns: 0 if OK, nonzero on error (see Figure 11.3)

This function returns, through the *result* pointer, a pointer to a linked list of `addrinfo` structures, which is defined by including `<netdb.h>`:

```
struct addrinfo {
  int     ai_flags;          /* AI_PASSIVE, AI_CANONNAME */
  int     ai_family;         /* AF_xxx */
  int     ai_socktype;       /* SOCK_xxx */
  int     ai_protocol;       /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
  size_t  ai_addrlen;        /* length of ai_addr */
  char   *ai_canonname;      /* ptr to canonical name for host */
  struct sockaddr  *ai_addr; /* ptr to socket address structure */
  struct addrinfo  *ai_next; /* ptr to next structure in linked list */
};
```

The *hostname* is either a hostname or an address string (dotted-decimal for IPv4 or a hex string for IPv6). The *service* is either a service name or a decimal port number string. (Recall our solution to Exercise 9.6 where we allowed an address string for the host or a port number string for the service.)

hints is either a null pointer or a pointer to an `addrinfo` structure that the caller fills in with hints about the types of information that the caller wants returned. For example, if the specified service is provided for both TCP and UDP (e.g., the `domain` service, which refers to a DNS server), the caller can set the `ai_socktype` member of the *hints* structure to `SOCK_DGRAM`. The only information returned will be for datagram sockets.

The members of the *hints* structure that can be set by the caller are:

- `ai_flags` (`AI_PASSIVE`, `AI_CANONNAME`),
- `ai_family` (an `AF_`*xxx* value),
- `ai_socktype` (a `SOCK_`*xxx* value), and
- `ai_protocol`.

The `AI_PASSIVE` flag indicates that the socket will be used for a passive open, and the `AI_CANONNAME` flag tells the function to return the canonical name of the host.

If the *hints* argument is a null pointer, the function assumes a value of 0 for `ai_flags`, `ai_socktype`, and `ai_protocol`, and a value of `AF_UNSPEC` for `ai_family`.

If the function returns success (0), the variable pointed to by the *result* argument is filled in with a pointer to a linked list of `addrinfo` structures, linked through the `ai_next` pointer. There are two ways that multiple structures can be returned.

1. If there are multiple addresses associated with the *hostname*, one structure is returned for each address that is usable with the requested address family (the `ai_family` hint, if specified).
2. If the service is provided for multiple socket types, one structure can be returned for each socket type, depending on the `ai_socktype` hint.

For example, if no hints are provided and if the `domain` service is looked up for a host with two IP addresses, four `addrinfo` structures are returned:

- one for the first IP address and a socket type of `SOCK_STREAM`,
- one for the first IP address and a socket type of `SOCK_DGRAM`,
- one for the second IP address and a socket type of `SOCK_STREAM`, and
- one for the second IP address and a socket type of `SOCK_DGRAM`.

We show a picture of this example in Figure 11.1. There is no guaranteed order of the structures when multiple items are returned; that is, we cannot assume that TCP services are returned before UDP services.

> Although not guaranteed, an implementation should return the IP addresses in the same order as they are returned by the DNS. For example, many DNS servers sort the returned addresses so that if the host sending the query and the name server are on the same network, then addresses on that shared network are returned first. Also, newer versions of BIND allow the resolver to specify an address sorting order in the `/etc/resolv.conf` file.

The information returned in the `addrinfo` structures is ready for a call to `socket` and then either a call to `connect`, or `sendto` (for a client) or `bind` (for a server). The arguments to `socket` are the members `ai_family`, `ai_socktype`, and `ai_protocol`. The second and third arguments to either `connect` or `bind` are `ai_addr` (a pointer to a socket address structure of the appropriate type, filled in by `getaddrinfo`) and `ai_addrlen` (the length of this socket address structure).

If the `AI_CANONNAME` flag is set in the *hints* structure, the `ai_canonname` member of the first returned structure points to the canonical name of the host. In terms of the DNS this is normally the FQDN.

Figure 11.1 shows the returned information if we execute

```
struct addrinfo     hints, *res;

bzero(&hints, sizeof(hints));
hints.ai_flags = AI_CANONNAME;
hints.ai_family = AF_INET;

getaddrinfo("bsdi", "domain", &hints, &res);
```

In this figure everything other than the `res` variable is dynamically allocated memory (e.g., from `malloc`). We assume that the canonical name of the host `bsdi` is `bsdi.kohala.com` and that this host has two IPv4 addresses in the DNS (Figure 1.16).

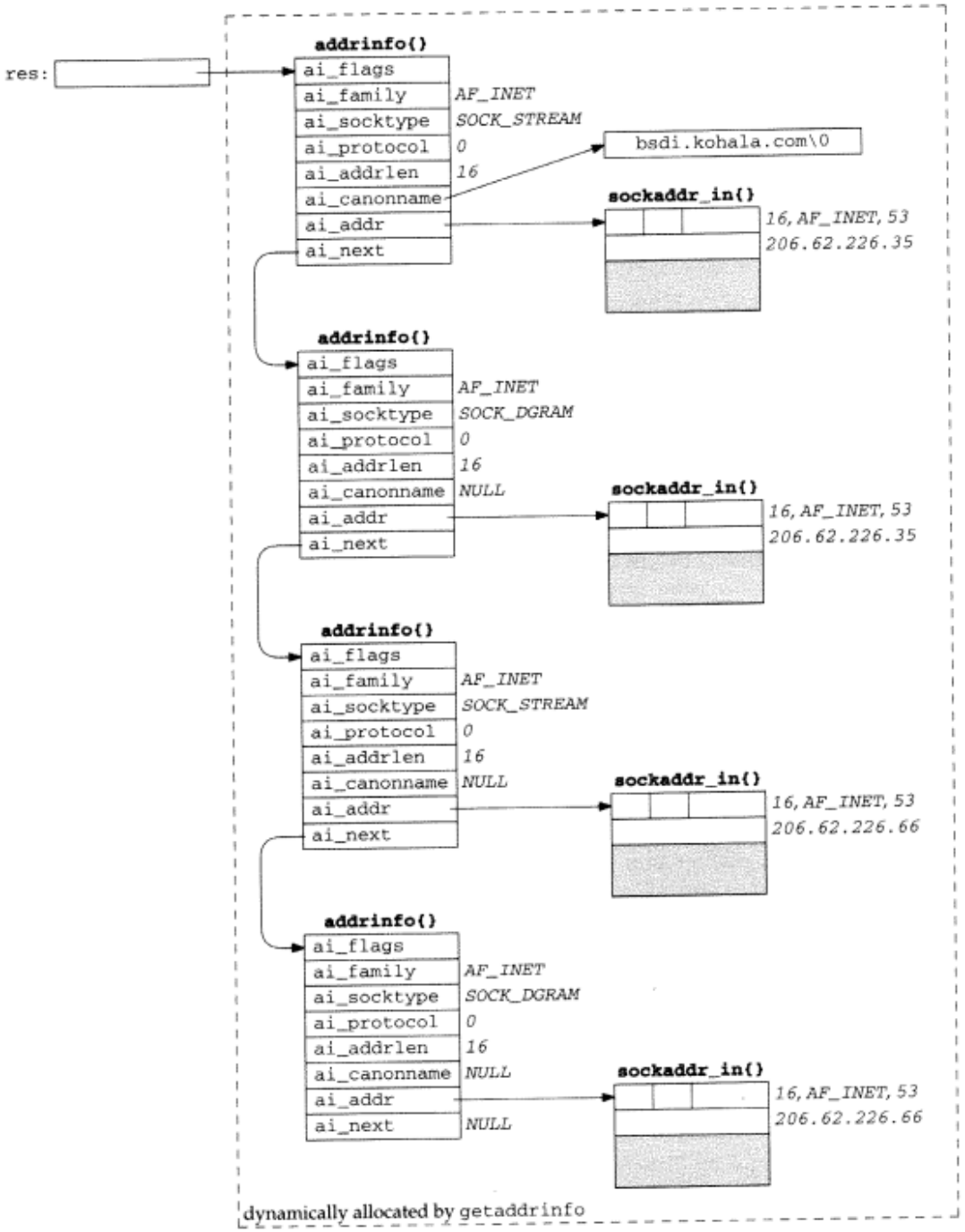

Figure 11.1 Example of information returned by `getaddrinfo`.

Port 53 is for the domain service, and realize that this port number will be in network byte order in the socket address structures. We also show the returned ai_protocol values as 0 since the combination of the ai_family and ai_socktype completely specifies the protocol for TCP and UDP. It would also be OK for getaddrinfo to return an ai_protocol of IPPROTO_TCP for the two SOCK_STREAM structures, and an ai_protocol of IPPROTO_UDP for the two SOCK_DGRAM structures.

Figure 11.2 summarizes the number of addrinfo structures returned for each address that is being returned, based on the specified service name (which can be a decimal port number) and any ai_socktype hint.

ai_socktype hint	Service is a name, service provided by:			Service is a port number
	TCP only	UDP only	TCP and UDP	
0	1	1	2	2
SOCK_STREAM	1	error	1	1
SOCK_DGRAM	error	1	1	1

Figure 11.2 Number of addrinfo structures returned per IP address.

Multiple addrinfo structures are returned for each IP address only when no ai_socktype hint is provided and either the service name is supported by TCP and UDP (as indicated in the /etc/services file) or a port number is specified.

If we enumerated all 64 possible inputs to getaddrinfo (there are six input variables), many would be invalid and some make little sense. Instead we will look at the common cases.

- Specify the *hostname* and *service*. This is normal for a TCP or UDP client. On return a TCP client loops through all returned IP addresses, calling socket and connect for each one, until the connection succeeds or until all addresses have been tried. We show an example of this with our tcp_connect function in Figure 11.6.

  For a UDP client, the socket address structure filled in by getaddrinfo would be used in a call to sendto or connect. If the client can tell that the first address doesn't appear to work (either an error on a connected UDP socket or a timeout on an unconnected socket), additional addresses can be tried.

  If the client knows it handles only one type of socket (e.g., Telnet and FTP clients handle only TCP, TFTP clients handle only UDP), then the ai_socktype member of the *hints* structure should be specified as either SOCK_STREAM or SOCK_DGRAM.

- A typical server specifies *service* but not the *hostname*, and specifies the AI_PASSIVE flag in the *hints* structure. The socket address structures returned should contain an IP address of INADDR_ANY (for IPv4) or IN6ADDR_ANY_INIT (for IPv6). A TCP server then calls socket, bind, and listen. If the server wants to malloc another socket address structure to obtain the client's address from accept, the returned ai_addrlen value specifies this size.

A UDP server would call `socket`, `bind`, and then `recvfrom`. If the server wants to `malloc` another socket address structure to obtain the client's address from `recvfrom`, the returned `ai_addrlen` value specifies this size.

As with the typical client code, if the server knows it only handles one type of socket, the `ai_socktype` member of the *hints* structure should be set to either `SOCK_STREAM` or `SOCK_DGRAM`. This avoids having multiple structures returned, possibly with the wrong `ai_socktype` value.

- The TCP servers that we have shown so far create one listening socket, and the UDP servers create one datagram socket. That is what we assume in the previous item. An alternate server design is for the server to handle multiple sockets using `select`. In this scenario the server would go through the entire list of structures returned by `getaddrinfo`, create one socket per structure, and use `select`.

  The problem with this technique is that one reason for `getaddrinfo` returning multiple structures is when a service can be handled by IPv4 and IPv6 (Figure 11.4). But these two protocols are not completely independent, as we saw in Section 10.2. That is, if we create a listening IPv6 socket for a given port, there is no need to also create a listening IPv4 socket for that same port, because connections arriving from IPv4 clients are automatically handled by the protocol stack and by the IPv6 listening socket.

Despite the fact that `getaddrinfo` is "better" than the `gethostbyname` and `getservbyname` functions (it makes it easier to write protocol-independent code, one function handles both the hostname and the service, and all the returned information is dynamically allocated, not statically allocated), it is still not as easy to use as it could be. The problem is that we must allocate a *hints* structure, initialize it to 0, fill in the desired fields, call `getaddrinfo`, and then traverse a linked list trying each one. In the next sections we provide some simpler interfaces for the typical TCP and UDP clients and servers that we write in the remainder of this text.

`getaddrinfo` solves the problem of converting hostnames and service names into socket address structures. In Section 11.13 we describe the reverse function, `getnameinfo`, which converts socket address structures into hostnames and service names. In Section 11.16 we provide an implementation of `getaddrinfo`, `getnameinfo`, and `freeaddrinfo`.

11.3 `gai_strerror` Function

The nonzero error return values from `getaddrinfo` have the names and meanings shown in Figure 11.3. The function `gai_strerror` takes one of these values as an argument and returns a pointer to the corresponding error string.

```
#include <netdb.h>

char *gai_strerror(int error);
```

Returns: pointer to string describing error message

Constant	Description
EAI_ADDRFAMILY	address family for *hostname* not supported
EAI_AGAIN	temporary failure in name resolution
EAI_BADFLAGS	invalid value for ai_flags
EAI_FAIL	unrecoverable failure in name resolution
EAI_FAMILY	ai_family not supported
EAI_MEMORY	memory allocation failure
EAI_NODATA	no address associated with *hostname*
EAI_NONAME	*hostname* or *service* not provided, or not known
EAI_SERVICE	*service* not supported for ai_socktype
EAI_SOCKTYPE	ai_socktype not supported
EAI_SYSTEM	system error returned in errno

Figure 11.3 Nonzero error return constants from getaddrinfo.

11.4 freeaddrinfo Function

All of the storage returned by getaddrinfo, the addrinfo structures, the ai_addr structures, and the ai_canonname string are obtained dynamically from malloc. This storage is returned by calling freeaddrinfo.

```
#include <netdb.h>

void freeaddrinfo(struct addrinfo *ai);
```

ai should point to the first of the addrinfo structures returned by getaddrinfo. All the structures in the linked list are freed, along with any dynamic storage pointed to by those structures (e.g., socket address structures and canonical hostnames).

Assume we call getaddrinfo, traverse the linked list of addrinfo structures, and find the desired structure. If we then try to save a copy of the information by copying just the addrinfo structure and then call freeaddrinfo, we have a lurking bug. The reason is that the addrinfo structure itself points to dynamically allocated memory (for the socket address structure and possibly the canonical name) and memory pointed to by our saved structure is returned to the system when freeaddrinfo is called and can be used for something else.

> Making a copy of just the addrinfo structure and not the structures that it in turn points to is called a *shallow copy*. Copying the addrinfo structure and all the structures that it points to is called a *deep copy*.

11.5 getaddrinfo Function: IPv6 and Unix Domain

Although Posix.1g defines the getaddrinfo function, it says nothing about IPv6 at all. The interaction between this function, the resolver (especially the RES_USE_INET6 option; recall Figure 9.5), and IPv6 is nontrivial. We note the following points before summarizing these interactions in Figure 11.4.

- `getaddrinfo` is dealing with two different inputs: what type of socket address structure does the caller want back and what type of records should be searched for in the DNS.
- The address family in the hints structure provided by the caller specifies the type of socket address structure that the caller expects to be returned. If the caller specifies `AF_INET`, the function must not return any `sockaddr_in6` structures and if the caller specifies `AF_INET6`, the function must not return any `sockaddr_in` structures.
- Posix.1g says that specifying `AF_UNSPEC` shall return addresses that can be used with *any* protocol family that can be used with the hostname and service name. This implies that if a host has both AAAA records and A records, the AAAA records are returned as `sockaddr_in6` structures and the A records are returned as `sockaddr_in` structures. It makes no sense to also return the A records as IPv4-mapped IPv6 addresses in `sockaddr_in6` structures as no additional information is being returned: these addresses are already being returned in `sockaddr_in` structures.
- This statement in Posix.1g also implies that if the `AI_PASSIVE` flag is specified without a hostname, then the IPv6 wildcard address (`IN6ADDR_ANY_INIT` or 0::0) should be returned as a `sockaddr_in6` structure along with the IPv4 wildcard address (`INADDR_ANY` or 0.0.0.0), returned as a `sockaddr_in` structure. It also makes sense to return the IPv6 wildcard address first because we saw in Section 10.2 that an IPv6 server socket can handle both IPv6 and IPv4 clients on a dual-stack host.
- The resolver's `RES_USE_INET6` option along with which function is called (`gethostbyname` or `gethostbyname2`) dictates the type of records that are searched for in the DNS (A or AAAA) and what type of addresses are returned (IPv4, IPv6, or IPv4-mapped IPv6). We summarized this in Figure 9.5.
- The hostname can also be either an IPv6 hex string or an IPv4 dotted-decimal string. The validity of this string depends on the address family specified by the caller. An IPv6 hex string is not acceptable if `AF_INET` is specified, and an IPv4 dotted-decimal string is not acceptable if `AF_INET6` is specified. But either is acceptable if `AF_UNSPEC` is specified, and the appropriate type of socket address structure is returned.

  One could argue that if `AF_INET6` is specified, then a dotted-decimal string should be returned as an IPv4-mapped IPv6 address in a `sockaddr_in6` structure. But another way to obtain this result is to prefix the dotted-decimal string with `0::ffff:`.

Figure 11.4 summarizes how we expect `getaddrinfo` to handle IPv4 and IPv6 addresses. The "result" column is what we want returned to the caller, given the variables in the first three columns. The "action" column is how we obtain this result and we show the code that performs this action in our implementation of `getaddrinfo` in Section 11.16.

Hostname specified by caller	Address family specified by caller	Hostname string contains	Result	Action
nonnull hostname string; active or passive	AF_UNSPEC	hostname	all AAAA records returned as sockaddr_in6{}s *and* all A records returned as sockaddr_in{}s	two DNS searches (note 1): gethostbyname2(AF_INET6) with RES_USE_INET6 off gethostbyname2(AF_INET) with RES_USE_INET6 off
		hex string	one sockaddr_in6{}	inet_pton(AF_INET6)
		dotted decimal	one sockaddr_in{}	inet_pton(AF_INET)
	AF_INET6	hostname	all AAAA records returned as sockaddr_in6{}s, *else* all A records returned as IPv4-mapped IPv6 sockaddr_in6{}s	gethostbyname() with RES_USE_INET6 on (note 2)
		hex string	one sockaddr_in6{}	inet_pton(AF_INET6)
		dotted decimal	error: EAI_ADDRFAMILY	
	AF_INET	hostname	all A records returned as sockaddr_in{}s	gethostbyname() with RES_USE_INET6 off
		hex string	error: EAI_ADDRFAMILY	
		dotted decimal	one sockaddr_in{}	inet_pton(AF_INET)
null hostname string; passive	AF_UNSPEC	implied 0::0 implied 0.0.0.0	one sockaddr_in6{} and one sockaddr_in{}	inet_pton(AF_INET6) inet_pton(AF_INET)
	AF_INET6	implied 0::0	one sockaddr_in6{}	inet_pton(AF_INET6)
	AF_INET	implied 0.0.0.0	one sockaddr_in{}	inet_pton(AF_INET)
null hostname string; active	AF_UNSPEC	implied 0::1 implied 127.0.0.1	one sockaddr_in6{} and one sockaddr_in{}	inet_pton(AF_INET6) inet_pton(AF_INET)
	AF_INET6	implied 0::1	one sockaddr_in6{}	inet_pton(AF_INET6)
	AF_INET	implied 127.0.0.1	one sockaddr_in{}	inet_pton(AF_INET)

Figure 11.4 Summary of getaddrinfo and its actions and results.

Note 1 is that when the two DNS searches are performed, either can fail (i.e., find no records of the desired type for the hostname) but at least one must succeed. But if both searches succeed (the hostname has both AAAA and A records), then both types of socket address structures are returned.

Note 2 is that this DNS search must succeed, or an error is returned. But since the RES_USE_INET6 option is enabled, gethostbyname first looks for the AAAA records, and if nothing is found, then looks for A records (Figure 9.6).

The setting and clearing of the resolver's RES_USE_INET6 option with the scenarios in notes 1 and 2 is to force the desired DNS search, given the rules in Figure 9.5.

We note that Figure 11.4 specifies only how getaddrinfo handles IPv4 and IPv6; that is, the number of addresses returned to the caller. The actual number of addrinfo structures returned to the caller also depends on the socket type specified and the service name, as summarized earlier in Figure 11.2.

Posix.1g says nothing specific about `getaddrinfo` and Unix domain sockets (which we describe in detail in Chapter 14). Nevertheless, adding support for Unix domain sockets to our implementation of `getaddrinfo` and testing applications with these protocols is a good test for protocol independence.

Our implementation makes the following assumption: if the hostname argument for `getaddrinfo` is either `/local` or `/unix` and the service name argument is an absolute pathname (one that begins with a slash), Unix domain socket structures are returned. Valid DNS hostnames cannot contain a slash and no existing IANA service names begin with a slash (Exercise 11.5). The socket address structures returned contain this absolute pathname, ready for a call to either `bind` or `connect`. If the caller specifies the `AI_CANONNAME` flag, the host's name (Section 9.7) is returned as the canonical name.

11.6 `getaddrinfo` Function: Examples

We will now show some examples of `getaddrinfo` using a test program that lets us enter all the parameters: the hostname, service name, address family, socket type, and the `AI_CANONNAME` and `AI_PASSIVE` flags. (We do not show this test program, as it is about 350 lines of uninteresting code. It is provided with the source code for the book, as described in the Preface.) The test program outputs information on the variable number of `addrinfo` structures that are returned, showing the arguments for a call to `socket` and the address in each socket address structure.

We first show the same example as in Figure 11.1.

```
solaris % testga  -f inet  -c  -h bsdi  -s domain

socket(AF_INET, SOCK_STREAM, 0), ai_canonname = bsdi.kohala.com
        address: 206.62.226.35.53

socket(AF_INET, SOCK_DGRAM, 0)
        address: 206.62.226.35.53

socket(AF_INET, SOCK_STREAM, 0)
        address: 206.62.226.66.53

socket(AF_INET, SOCK_DGRAM, 0)
        address: 206.62.226.66.53
```

The `-f inet` option specifies the address family, `-c` says to return the canonical name, `-h bsdi` specifies the hostname, and `-s domain` specifies the service name.

The common client scenario is to specify the address family, the socket type (the `-t` option), the hostname, and the service name. The following example shows this, for a multihomed host with six IPv4 addresses.

```
solaris % testga  -f inet  -t stream  -h gateway.tuc.noao.edu  -s daytime

socket(AF_INET, SOCK_STREAM, 0)
        address: 140.252.101.4.13

socket(AF_INET, SOCK_STREAM, 0)
        address: 140.252.102.1.13
```

```
socket(AF_INET, SOCK_STREAM, 0)
        address: 140.252.104.1.13

socket(AF_INET, SOCK_STREAM, 0)
        address: 140.252.3.6.13

socket(AF_INET, SOCK_STREAM, 0)
        address: 140.252.4.100.13

socket(AF_INET, SOCK_STREAM, 0)
        address: 140.252.1.4.13
```

Next we specify our host `alpha`, which has both a AAAA record and an A record, without specifying the address family, and a service name of `ftp`, which is provided by TCP only.

```
solaris % testga  -h alpha  -s ftp

socket(AF_INET6, SOCK_STREAM, 0)
        address: 5f1b:df00:ce3e:e200:20:800:2b37:6426.21

socket(AF_INET, SOCK_STREAM, 0)
        address: 206.62.226.42.21
```

Since we did not specify the address family, and since we ran this example on a host that supports both IPv4 and IPv6, two structures are returned: one for IPv6 and one for IPv4.

Next we specify the `AI_PASSIVE` flag (the `-p` option), do not specify an address family, do not specify a hostname (implying the wildcard address), specify a port number of 8888, and do not specify a socket type.

```
solaris % testga  -p  -s 8888

socket(AF_INET6, SOCK_STREAM, 0)
        address: ::.8888

socket(AF_INET6, SOCK_DGRAM, 0)
        address: ::.8888

socket(AF_INET, SOCK_STREAM, 0)
        address: 0.0.0.0.8888

socket(AF_INET, SOCK_DGRAM, 0)
        address: 0.0.0.0.8888
```

Four structures are returned. Since we ran this on a host that supports IPv6 and IPv4, without specifying an address family, `getaddrinfo` returns the IPv6 wildcard address and the IPv4 wildcard address. Since we specified a port number without a socket type, `getaddrinfo` returns one structure for each address specifying TCP and another structure for each address specifying UDP. The two IPv6 structures are returned before the two IPv4 structures, because we saw in Chapter 10 that an IPv6 client or server on a dual-stack host can communicate with either IPv6 or IPv4 peers.

As an example of Unix domain sockets, we specify `/local` as the hostname and `/tmp/test.1` as the service name.

```
solaris % testga  -c  -p  -h /local  -s /tmp/test.1

socket(AF_LOCAL, SOCK_STREAM, 0), ai_canonname = solaris.kohala.com
        address: /tmp/test.1

socket(AF_LOCAL, SOCK_DGRAM, 0)
        address: /tmp/test.1
```

Since we do not specify the socket type, two structures are returned: the first for a stream socket and the second for a datagram socket.

11.7 host_serv Function

Our first interface to getaddrinfo does not require the caller to allocate a hints structure and fill it in. Instead, the two fields of interest, the address family and the socket type, are arguments to our host_serv function.

```
#include "unp.h"

struct addrinfo *host_serv(const char *hostname, const char *service,
                           int family, int socktype);
```

Returns: pointer to addrinfo structure if OK, NULL on error

Figure 11.5 shows the source code for this function.

lib/host_serv.c

```
 1 #include    "unp.h"

 2 struct addrinfo *
 3 host_serv(const char *host, const char *serv, int family, int socktype)
 4 {
 5     int     n;
 6     struct addrinfo hints, *res;

 7     bzero(&hints, sizeof(struct addrinfo));
 8     hints.ai_flags = AI_CANONNAME;  /* always return canonical name */
 9     hints.ai_family = family;    /* AF_UNSPEC, AF_INET, AF_INET6, etc. */
10     hints.ai_socktype = socktype;    /* 0, SOCK_STREAM, SOCK_DGRAM, etc. */

11     if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
12         return (NULL);

13     return (res);                /* return pointer to first on linked list */
14 }
```

lib/host_serv.c

Figure 11.5 host_serv function.

7–13 The function initializes a hints structure, calls getaddrinfo, and returns a null pointer if an error occurs.

We call this function from Figure 15.17 when we want to use getaddrinfo to obtain the host and service information, but we want to establish the connection ourself.

11.8 tcp_connect Function

We now write two functions that use getaddrinfo to handle most scenarios for the TCP clients and servers that we write. The first function, tcp_connect, performs the normal client steps: create a TCP socket and connect to a server.

```
#include "unp.h"

int tcp_connect(const char *hostname, const char *service);
```

Returns: connected socket descriptor if OK, no return on error

Figure 11.6 shows the source code.

lib/tcp_connect.c

```
 1 #include    "unp.h"
 2 int
 3 tcp_connect(const char *host, const char *serv)
 4 {
 5     int     sockfd, n;
 6     struct addrinfo hints, *res, *ressave;
 7     bzero(&hints, sizeof(struct addrinfo));
 8     hints.ai_family = AF_UNSPEC;
 9     hints.ai_socktype = SOCK_STREAM;
10     if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
11         err_quit("tcp_connect error for %s, %s: %s",
12                  host, serv, gai_strerror(n));
13     ressave = res;
14     do {
15         sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
16         if (sockfd < 0)
17             continue;           /* ignore this one */
18         if (connect(sockfd, res->ai_addr, res->ai_addrlen) == 0)
19             break;              /* success */
20         Close(sockfd);          /* ignore this one */
21     } while ( (res = res->ai_next) != NULL);
22     if (res == NULL)            /* errno set from final connect() */
23         err_sys("tcp_connect error for %s, %s", host, serv);
24     freeaddrinfo(ressave);
25     return (sockfd);
26 }
```

lib/tcp_connect.c

Figure 11.6 tcp_connect function: perform normal client steps.

Call getaddrinfo

7-13 getaddrinfo is called once and we specify the address family as AF_UNSPEC and the socket type as SOCK_STREAM.

Try each addrinfo structure until success or end of list

14-25 Each returned IP address is then tried: socket and connect are called. It is not a fatal error for socket to fail, as this could happen if an IPv6 address were returned but the host kernel does not support IPv6. If connect succeeds, a break is made out of the loop. Otherwise, when all the addresses have been tried, the loop also terminates. freeaddrinfo returns all the dynamic memory.

This function (and our other functions that provide a simpler interface to getaddrinfo in the following sections) terminates if either getaddrinfo fails, or if no call to connect succeeds. The only return is upon success. It would be hard to return an error code (one of the EAI_*xxx* constants) without adding another argument. This means that our wrapper function is trivial:

```
int
Tcp_connect(const char *host, const char *serv)
{
    return(tcp_connect(host, serv));
}
```

Nevertheless, we still call our wrapper function, instead of tcp_connect, to maintain consistency with the remainder of the text.

> The problem with the return value is that descriptors are nonnegative but we do not know whether the EAI_*xxx* values are positive or negative. If these values were positive, we could return the negative of these values if getaddrinfo fails, but we also have to return some other negative value to indicate that all the structures were tried without success.

Example: Daytime Client

Figure 11.7 shows our daytime client from Figure 1.5 recoded to use tcp_connect.

Command-line arguments

9-10 We now require a second command-line argument to specify either the service name or the port number, which allows our program to connect to other ports.

Connect to server

11 All of the socket code for this client is now performed by tcp_connect.

Print server's address

12-15 We call getpeername to fetch the server's protocol address and print it. We do this to verify the protocol being used in the examples we are about to show.

Note that tcp_connect does not return the size of the socket address structure that was used for the connect. We could have added a pointer argument to return this value, but one design goal for this function was to reduce the number of arguments, compared to getaddrinfo. What we do instead is define the constant MAXSOCKADDR in our unp.h header to be the size of the largest socket address structure. This is normally the size of a Unix domain socket address structure (Section 14.2), just over 100 bytes. We allocate room for a structure of this size and this is what getpeername fills in.

names/daytimetcpcli.c

```
1 #include    "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int     sockfd, n;
6     char    recvline[MAXLINE + 1];
7     socklen_t len;
8     struct sockaddr *sa;

9     if (argc != 3)
10        err_quit("usage: daytimetcpcli <hostname/IPaddress> <service/port#>");

11    sockfd = Tcp_connect(argv[1], argv[2]);

12    sa = Malloc(MAXSOCKADDR);
13    len = MAXSOCKADDR;
14    Getpeername(sockfd, sa, &len);
15    printf("connected to %s\n", Sock_ntop_host(sa, len));

16    while ( (n = Read(sockfd, recvline, MAXLINE)) > 0) {
17        recvline[n] = 0;        /* null terminate */
18        Fputs(recvline, stdout);
19    }
20    exit(0);
21 }
```

names/daytimetcpcli.c

Figure 11.7 Daytime client recoded to use `tcp_connect`.

We call `malloc` for this structure, instead of allocating it as

```
char  sockaddr[MAXSOCKADDR];
```

for alignment reasons. `malloc` always returns a pointer with the strictest alignment required by the system, while a `char` array could be allocated on an odd-byte boundary, which could be a problem for the IP address or port number fields in the socket address structure. Another way to handle this potential alignment problem was shown in Figure 4.19 using a `union`.

This version of our client works with both IPv4 and IPv6, while the version in Figure 1.5 worked only with IPv4 and the version in Figure 1.6 worked only with IPv6. You should also compare our new version with Figure E.14, which we coded to use `gethostbyname` and `getservbyname` to support both IPv4 and IPv6.

We first specify the name of a host that supports only IPv4.

```
solaris % daytimetcpcli bsdi daytime
connected to 206.62.226.35
Fri May 30 12:33:32 1997
```

Next we specify the name of a host that supports both IPv4 and IPv6.

```
solaris % daytimetcpcli aix daytime
connected to 5f1b:df00:ce3e:e200:20:800:5afc:2b36
Fri May 30 12:43:43 1997
```

The IPv6 address is used because the host has both a AAAA record and an A record, and as noted in Figure 11.4, since `tcp_connect` sets the address family to `AF_UNSPEC`, AAAA records are searched for first, and only if this fails is a search made for an A record.

In the next example we force the use of the IPv4 address by specifying the hostname with our `-4` suffix, which we noted in Section 9.2 is our convention for the hostname with only A records.

```
solaris % daytimetcpcli aix-4 daytime
connected to 206.62.226.43
Fri May 30 12:43:48 1997
```

11.9 `tcp_listen` Function

Our next function, `tcp_listen`, performs the normal TCP server steps: create a TCP socket, `bind` the server's well-known port, and allow incoming connection requests to be accepted. Figure 11.8 shows the source code.

```
#include "unp.h"

int tcp_listen(const char *hostname, const char *service, socklen_t *lenptr);
```

Returns: connected socket descriptor if OK, no return on error

Call `getaddrinfo`

8-15 We initialize an `addrinfo` structure with our hints: `AI_PASSIVE`, since this function is for a server, `AF_UNSPEC` for the address family, and `SOCK_STREAM`. Recall from Figure 11.4 that if a hostname is not specified (which is common for a server that wants to bind the wildcard address), the `AI_PASSIVE` and `AF_UNSPEC` hints will cause two socket address structures to be returned: the first for IPv6 and the next for IPv4 (assuming a dual-stack host).

Create socket and bind address

16-24 The `socket` and `bind` functions are called. If either call fails we just ignore this `addrinfo` structure and move on to the next one. As stated in Section 7.5, we always set the `SO_REUSEADDR` socket option for a TCP server.

Check for failure

25-26 If all the calls to `socket` and `bind` failed, we print an error and terminate. As with our `tcp_connect` function in the previous section, we do not try to return an error from this function.

27 The socket is turned into a listening socket by `listen`.

Return size of socket address structure

28-31 If the *addrlenp* argument is nonnull, we return the size of the protocol addresses through this pointer. This allows the caller to allocate memory for a socket address structure to obtain the client's protocol address from `accept`. (See Exercise 11.1 also.)

lib/tcp_listen.c

```
 1 #include    "unp.h"
 2 int
 3 tcp_listen(const char *host, const char *serv, socklen_t *addrlenp)
 4 {
 5     int     listenfd, n;
 6     const int on = 1;
 7     struct addrinfo hints, *res, *ressave;
 8     bzero(&hints, sizeof(struct addrinfo));
 9     hints.ai_flags = AI_PASSIVE;
10     hints.ai_family = AF_UNSPEC;
11     hints.ai_socktype = SOCK_STREAM;
12     if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
13         err_quit("tcp_listen error for %s, %s: %s",
14                  host, serv, gai_strerror(n));
15     ressave = res;
16     do {
17         listenfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
18         if (listenfd < 0)
19             continue;           /* error, try next one */
20         Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
21         if (bind(listenfd, res->ai_addr, res->ai_addrlen) == 0)
22             break;              /* success */
23         Close(listenfd);        /* bind error, close and try next one */
24     } while ( (res = res->ai_next) != NULL);
25     if (res == NULL)            /* errno from final socket() or bind() */
26         err_sys("tcp_listen error for %s, %s", host, serv);
27     Listen(listenfd, LISTENQ);
28     if (addrlenp)
29         *addrlenp = res->ai_addrlen;    /* return size of protocol address */
30     freeaddrinfo(ressave);
31     return (listenfd);
32 }
```

lib/tcp_listen.c

Figure 11.8 `tcp_listen` function: perform normal server steps.

Example: Daytime Server

Figure 11.9 shows our daytime server from Figure 4.11 recoded to use `tcp_listen`.

Require service name or port number as command-line argument

11-12 We require a command-line argument to specify either the service name or the port number. This makes it easier to test our server, since binding port 13 for the daytime server requires superuser privileges.

names/daytimetcpsrv1.c

```
 1 #include    "unp.h"
 2 #include    <time.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     listenfd, connfd;
 7     socklen_t addrlen, len;
 8     char    buff[MAXLINE];
 9     time_t  ticks;
10     struct sockaddr *cliaddr;

11     if (argc != 2)
12         err_quit("usage: daytimetcpsrv1 <service or port#>");

13     listenfd = Tcp_listen(NULL, argv[1], &addrlen);

14     cliaddr = Malloc(addrlen);

15     for ( ; ; ) {
16         len = addrlen;
17         connfd = Accept(listenfd, cliaddr, &len);
18         printf("connection from %s\n", Sock_ntop(cliaddr, len));

19         ticks = time(NULL);
20         snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
21         Write(connfd, buff, strlen(buff));

22         Close(connfd);
23     }
24 }
```

names/daytimetcpsrv1.c

Figure 11.9 Daytime server recoded to use `tcp_listen` (see also Figure 11.10).

Create listening socket

13–14 `tcp_listen` creates the listening socket and `malloc` allocates a buffer to hold the client's address.

Server loop

15–23 `accept` waits for each client connection. We print the client address by calling `sock_ntop`. In the case of either IPv4 or IPv6, this function prints the IP address and port number. We could use the function `getnameinfo` (described in Section 11.13) to try to obtain the hostname of the client, but that involves a PTR query in the DNS, which can take some time, especially if the PTR query fails. Section 14.8 of TCPv3 notes that on a busy Web server almost 25% of all clients connecting to that server did not have PTR records in the DNS. Since we do not want a server (especially an iterative server) to wait seconds for a PTR query, we just print the IP address and port.

Example: Daytime Server with Protocol Specification

There is a slight problem with Figure 11.9: the first argument to `tcp_listen` is a null pointer, which combined with the address family of `AF_UNSPEC` that `tcp_listen`

specifies might cause `getaddrinfo` to return a socket address structure with an address family other than what is desired. For example, the first socket address structure returned will be for IPv6 on a dual-stack host (Figure 11.4) but we might want our server to handle only IPv4.

Clients do not have this problem, since the client must always specify either an IP address or a hostname. Client applications normally allow the user to enter this as a command-line argument. This gives us the opportunity to specify a hostname that is associated with a particular type of IP address (recall our `-4` and `-6` hostnames in Section 9.2), or to specify either an IPv4 dotted-decimal string (forcing IPv4) or an IPv6 hex string (forcing IPv6).

But there is a simple technique for servers that lets us force a given protocol upon a server, either IPv4 or IPv6: allow the user to enter either an IP address or a hostname as a command-line argument to the program and pass this to `getaddrinfo`. In the case of an IP address, an IPv4 dotted-decimal string differs from an IPv6 hex string. The following calls to `inet_pton` either fail or succeed, as indicated.

```
inet_pton(AF_INET,  "0.0.0.0", &foo);      /* succeeds */
inet_pton(AF_INET,  "0::0",    &foo);      /* fails */
inet_pton(AF_INET6, "0.0.0.0", &foo);      /* fails */
inet_pton(AF_INET6, "0::0",    &foo);      /* succeeds */
```

Therefore, if we change our servers to accept an optional argument, then if we enter

```
% server
```

it defaults to IPv6 on a dual-stack host, but entering

```
% server 0.0.0.0
```

explicitly specifies IPv4 and

```
% server 0::0
```

explicitly specifies IPv6.

Figure 11.10 shows this final version of our daytime server.

Handle command-line arguments

11-16 The only change from Figure 11.9 is the handling of the command-line arguments, allowing the user to specify either a hostname or an IP address for the server to bind, in addition to a service name or port.

We first start this server with an IPv4 socket and then connect to the server from clients on two other hosts on the local subnet.

```
solaris % daytimetcpsrv2 0.0.0.0 9999
connection from 206.62.226.36.32789
connection from 206.62.226.35.1389
```

But now we start the server with an IPv6 socket.

```
solaris % daytimetcpsrv2 0::0 9999
connection from 5f1b:df00:ce3e:e200:20:800:2003:f642.32799
connection from 5f1b:df00:ce3e:e200:20:800:2b37:6426.1026
```

names/daytimetcpsrv2.c

```
 1 #include    "unp.h"
 2 #include    <time.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     listenfd, connfd;
 7     socklen_t addrlen, len;
 8     struct sockaddr *cliaddr;
 9     char    buff[MAXLINE];
10     time_t  ticks;

11     if (argc == 2)
12         listenfd = Tcp_listen(NULL, argv[1], &addrlen);
13     else if (argc == 3)
14         listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
15     else
16         err_quit("usage: daytimetcpsrv2 [ <host> ] <service or port>");

17     cliaddr = Malloc(addrlen);

18     for ( ; ; ) {
19         len = addrlen;
20         connfd = Accept(listenfd, cliaddr, &len);
21         printf("connection from %s\n", Sock_ntop(cliaddr, len));

22         ticks = time(NULL);
23         snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
24         Write(connfd, buff, strlen(buff));

25         Close(connfd);
26     }
27 }
```

names/daytimetcpsrv2.c

Figure 11.10 Protocol-independent daytime server that uses `tcp_listen`.

```
connection from ::ffff:206.62.226.36.32792
connection from ::ffff:206.62.226.35.1390
```

The first connection is from the host `sunos5` using IPv6 and the second is from the host `alpha` using IPv6. The next two connections are from the hosts `sunos5` and `bsdi`, but using IPv4, not IPv6. We can tell this because the client's addresses returned by `accept` are both IPv4-mapped IPv6 addresses.

What we have just shown is that an IPv6 server running on a dual-stack host can handle either IPv4 or IPv6 clients. The IPv4 client addresses are passed to the IPv6 server as IPv4-mapped IPv6 address, as we discussed in Section 10.2.

This server, along with the client in Figure 11.7, also work with Unix domain sockets (Chapter 14) since our implementation of `getaddrinfo` in Section 11.16 supports Unix domain sockets. For example, we start the server as

```
solaris % daytimetcpsrv2 /local /tmp/rendezvous
```

where the pathname /tmp/rendezvous is an arbitrary pathname we choose for the server to bind and to which the client connects. We then start the client on the same host, specifying /local as the hostname and /tmp/rendezvous as the service name.

```
solaris % daytimetcpcli /local /tmp/rendezvous
connected to /tmp/rendezvous
Fri May 30 16:31:37 1997
```

11.10 udp_client Function

Our functions that provide a simpler interface to getaddrinfo change with UDP because we provide one client function that creates an unconnected UDP socket, and another in the next section that creates a connected UDP socket.

```
#include "unp.h"

int udp_client(const char *hostname, const char *service,
               void **saptr, socklen_t *lenp);
```

Returns: unconnected socket descriptor if OK, no return on error

This function creates an unconnected UDP socket, returning three items. First, the return value is the socket descriptor. Second, *saptr* is the address of a pointer (declared by the caller) to a socket address structure (allocated dynamically by udp_client) and in that structure the function stores the destination IP address and port for future calls to sendto. The size of that socket address structure is returned in the variable pointed to by *lenp*. This final argument cannot be a null pointer (as we allowed for the final argument to tcp_listen) because the length of the socket address structure is required in any calls to sendto and recvfrom.

> saptr should be declared as struct sockaddr **. We use the void ** datatype because we define another version of this function that uses XTI in Section 31.3 and it uses this argument to contain the address of a pointer to a different type of structure. This means our calls to this function must contain the cast (void **).

Figure 11.11 shows the source code for this function.

getaddrinfo converts the *hostname* and *service* arguments. A datagram socket is created. Memory is allocated for one socket address structure and the socket address structure corresponding to the socket that was created is copied into the memory.

Example: Protocol-Independent Daytime Client

We now recode our daytime client from Figure 11.7 to use UDP and our udp_client function. Figure 11.12 shows the protocol-independent source code.

11-16 We call our udp_client function and then print the IP address and port of the server to which we will send the UDP datagram. We send a 1-byte datagram and then read and print the reply.

lib/udp_client.c

```
 1 #include    "unp.h"

 2 int
 3 udp_client(const char *host, const char *serv, void **saptr, socklen_t *lenp)
 4 {
 5     int     sockfd, n;
 6     struct addrinfo hints, *res, *ressave;

 7     bzero(&hints, sizeof(struct addrinfo));
 8     hints.ai_family = AF_UNSPEC;
 9     hints.ai_socktype = SOCK_DGRAM;

10     if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
11         err_quit("udp_client error for %s, %s: %s",
12                  host, serv, gai_strerror(n));
13     ressave = res;

14     do {
15         sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
16         if (sockfd >= 0)
17             break;              /* success */
18     } while ( (res = res->ai_next) != NULL);

19     if (res == NULL)            /* errno set from final socket() */
20         err_sys("udp_client error for %s, %s", host, serv);

21     *saptr = Malloc(res->ai_addrlen);
22     memcpy(*saptr, res->ai_addr, res->ai_addrlen);
23     *lenp = res->ai_addrlen;

24     freeaddrinfo(ressave);

25     return (sockfd);
26 }
```

lib/udp_client.c

Figure 11.11 `udp_client` function: create an unconnected UDP socket.

We need to send only a 0-byte UDP datagram, as what triggers the daytime server's response is just the arrival of a datagram, regardless of its length and contents. But many SVR4 implementations do not allow a 0-length UDP datagram.

We run our client specifying a hostname that has a AAAA record and an A record. Since the structure with the AAAA record is returned first by `getaddrinfo`, an IPv6 socket is created.

```
solaris % daytimeudpcli1 aix daytime
sending to 5f1b:df00:ce3e:e200:20:800:5afc:2b36
Sat May 31 08:13:34 1997
```

Next we specify the dotted-decimal address of the same host, resulting in an IPv4 socket.

```
solaris % daytimeudpcli1 206.62.226.43 daytime
sending to 206.62.226.43
Sat May 31 08:14:02 1997
```

names/daytimeudpcli1.c

```
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd, n;
 6     char    recvline[MAXLINE + 1];
 7     socklen_t salen;
 8     struct sockaddr *sa;

 9     if (argc != 3)
10         err_quit("usage: daytimeudpcli1 <hostname/IPaddress> <service/port#>");

11     sockfd = Udp_client(argv[1], argv[2], (void **) &sa, &salen);

12     printf("sending to %s\n", Sock_ntop_host(sa, salen));

13     Sendto(sockfd, "", 1, 0, sa, salen);    /* send 1-byte datagram */

14     n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
15     recvline[n] = 0;            /* null terminate */
16     Fputs(recvline, stdout);

17     exit(0);
18 }
```

names/daytimeudpcli1.c

Figure 11.12 UDP daytime client using our `udp_client` function.

11.11 `udp_connect` Function

Our `udp_connect` function creates a connected UDP socket.

```
#include "unp.h"

int udp_connect(const char *hostname, const char *service);
```

Returns: connected socket descriptor if OK, no return on error

With a connected UDP socket the final two arguments required by `udp_client` are no longer needed. The caller can call `write` instead of `sendto`, so our function need not return a socket address structure and its length.

Figure 11.13 shows the source code.

This function is nearly identical to `tcp_connect`. One difference, however, is that the call to `connect` with a UDP socket does not send anything to the peer. If something is wrong (the peer is unreachable or there is no server at the specified port), the caller does not discover that until it sends a datagram to the peer.

lib/udp_connect.c

```
 1 #include    "unp.h"
 2 int
 3 udp_connect(const char *host, const char *serv)
 4 {
 5     int     sockfd, n;
 6     struct addrinfo hints, *res, *ressave;
 7     bzero(&hints, sizeof(struct addrinfo));
 8     hints.ai_family = AF_UNSPEC;
 9     hints.ai_socktype = SOCK_DGRAM;
10     if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
11         err_quit("udp_connect error for %s, %s: %s",
12                  host, serv, gai_strerror(n));
13     ressave = res;
14     do {
15         sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
16         if (sockfd < 0)
17             continue;           /* ignore this one */
18         if (connect(sockfd, res->ai_addr, res->ai_addrlen) == 0)
19             break;              /* success */
20         Close(sockfd);          /* ignore this one */
21     } while ( (res = res->ai_next) != NULL);
22     if (res == NULL)            /* errno set from final connect() */
23         err_sys("udp_connect error for %s, %s", host, serv);
24     freeaddrinfo(ressave);
25     return (sockfd);
26 }
```

lib/udp_connect.c

Figure 11.13 udp_connect function: create a connected UDP socket.

11.12 udp_server Function

Our final UDP function that provides a simpler interface to getaddrinfo is udp_server.

```
#include "unp.h"

int udp_server(const char *hostname, const char *service, socklen_t *lenptr);
```

Returns: unconnected socket descriptor if OK, no return on error

The arguments are the same as for tcp_listen: an optional *hostname*, a required *service* (so its port number can be bound), and an optional pointer to a variable in which the size of the socket address structure is returned.

Figure 11.14 shows the source code.

lib/udp_server.c

```
 1 #include    "unp.h"

 2 int
 3 udp_server(const char *host, const char *serv, socklen_t *addrlenp)
 4 {
 5     int     sockfd, n;
 6     struct addrinfo hints, *res, *ressave;

 7     bzero(&hints, sizeof(struct addrinfo));
 8     hints.ai_flags = AI_PASSIVE;
 9     hints.ai_family = AF_UNSPEC;
10     hints.ai_socktype = SOCK_DGRAM;

11     if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
12         err_quit("udp_server error for %s, %s: %s",
13                  host, serv, gai_strerror(n));
14     ressave = res;

15     do {
16         sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
17         if (sockfd < 0)
18             continue;           /* error, try next one */

19         if (bind(sockfd, res->ai_addr, res->ai_addrlen) == 0)
20             break;              /* success */

21         Close(sockfd);          /* bind error, close and try next one */
22     } while ( (res = res->ai_next) != NULL);

23     if (res == NULL)            /* errno from final socket() or bind() */
24         err_sys("udp_server error for %s, %s", host, serv);

25     if (addrlenp)
26         *addrlenp = res->ai_addrlen;    /* return size of protocol address */

27     freeaddrinfo(ressave);

28     return (sockfd);
29 }
```

lib/udp_server.c

Figure 11.14 udp_server function: create an unconnected socket for a UDP server.

This function is nearly identical to tcp_listen, but without the call to listen. We set the address family to AF_UNSPEC but the caller can use the same technique that we described with Figure 11.10 to force a particular protocol (IPv4 or IPv6).

We do not set the SO_REUSEADDR socket option for the UDP socket because this socket option can allow multiple sockets to bind the same UDP port on hosts that support multicasting, as we described in Section 7.5. Since there is nothing like TCP's TIME_WAIT state for a UDP socket, there is no need to set this socket option when the server is started.

Example: Protocol-Independent Daytime Server

Figure 11.15 shows our daytime server, modified from Figure 11.10 to use UDP.

names/daytimeudpsrv2.c

```
1 #include    "unp.h"
2 #include    <time.h>

3 int
4 main(int argc, char **argv)
5 {
6     int     sockfd;
7     ssize_t n;
8     char    buff[MAXLINE];
9     time_t  ticks;
10    socklen_t addrlen, len;
11    struct sockaddr *cliaddr;

12    if (argc == 2)
13        sockfd = Udp_server(NULL, argv[1], &addrlen);
14    else if (argc == 3)
15        sockfd = Udp_server(argv[1], argv[2], &addrlen);
16    else
17        err_quit("usage: daytimeudpsrv [ <host> ] <service or port>");

18    cliaddr = Malloc(addrlen);

19    for ( ; ; ) {
20        len = addrlen;
21        n = Recvfrom(sockfd, buff, MAXLINE, 0, cliaddr, &len);
22        printf("datagram from %s\n", Sock_ntop(cliaddr, len));

23        ticks = time(NULL);
24        snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
25        Sendto(sockfd, buff, strlen(buff), 0, cliaddr, len);
26    }
27 }
```

names/daytimeudpsrv2.c

Figure 11.15 Protocol independent UDP daytime server.

11.13 `getnameinfo` Function

This function is the complement of `getaddrinfo`: it takes a socket address and returns a character string describing the host and another character string describing the service. This function provides this information in a protocol-independent fashion; that is, the caller does not care what type of protocol address is contained in the socket address structure, as that detail is handled by the function.

```
#include <netdb.h>

int getnameinfo(const struct sockaddr *sockaddr, socklen_t addrlen,
                char *host, size_t hostlen,
                char *serv, size_t servlen, int flags);
```

Returns: 0 if OK, –1 on error

sockaddr points to the socket address structure containing the protocol address to be converted into a human-readable string, and *addrlen* is the length of this structure. This structure and its length are normally returned by either `accept`, `recvfrom`, `getsockname`, or `getpeername`.

The caller allocates space for the two human-readable strings: *host* and *hostlen* specify the host string, and *serv* and *servlen* specify the service string. If the caller does not want the host string returned, a *hostlen* of 0 is specified. Similarly a *servlen* of 0 specifies not to return information on the service. To help allocate arrays to hold these two strings, the constants shown in Figure 11.16 are defined by including the `<netdb.h>` header.

Constant	Description	Value
NI_MAXHOST	maximum size of returned host string	1025
NI_MAXSERV	maximum size of returned service string	32

Figure 11.16 Constants for returned string sizes from `getnameinfo`.

The difference between `sock_ntop` and `getnameinfo` is that the former does not involve the DNS and just returns a printable version of the IP address and port number. The latter normally tries to obtain a name for both the host and service.

Figure 11.17 shows the five *flags* that can be specified to change the operation of `getnameinfo`.

Constant	Description
NI_DGRAM	datagram service
NI_NAMEREQD	return an error if name cannot be resolved from address
NI_NOFQDN	return only hostname portion of FQDN
NI_NUMERICHOST	return numeric string for hostname
NI_NUMERICSERV	return numeric string for service name

Figure 11.17 *flags* for `getnameinfo`.

`NI_DGRAM` should be specified when the caller knows it is dealing with a datagram socket. The reason is that given only the IP address and port number in the socket address structure, `getnameinfo` cannot determine the protocol (TCP or UDP). There exist a few port numbers that are used for one service with TCP and a completely different service with UDP. An example is port 514, which is the `rsh` service with TCP, but the `syslog` service with UDP.

`NI_NAMEREQD` causes an error to be returned if the hostname cannot be resolved using the DNS. This can be used by servers that require the client's IP address be mapped into a hostname. These servers then take this returned hostname and call `gethostbyname` and verify that one of the returned addresses is the address in the socket address structure.

`NI_NOFQDN` causes the returned hostname to be truncated at the first period. For example, if the IP address in the socket address structure were 206.62.226.42, `gethostbyaddr` would return a name of `alpha.kohala.com`. But if this flag is specified to `getnameinfo`, it returns the hostname as just `alpha`.

NI_NUMERICHOST tells getnameinfo not to call the DNS (which can take time). Instead the numeric representation of the IP address is returned, probably by calling inet_ntop. Similarly NI_NUMERICSERV specifies that the decimal port number is to be returned, instead of looking up the service name. Servers should normally specify NI_NUMERICSERV because the client port numbers normally have no associated service name—they are ephemeral ports.

The logical OR of multiple flags can be specified if they make sense together (e.g., I_DGRAM and NI_NUMERICHOST), while other combinations make no sense (e.g., NI_NAMEREQD and NI_NUMERICHOST).

getnameinfo was overlooked by Posix.1g but is specified in RFC 2133 [Gilligan et al. 1997].

11.14 Reentrant Functions

The gethostbyname function from Section 9.3 presents an interesting problem that we have not yet examined in the text: it is not *reentrant*. We will encounter this problem in general when we deal with threads in Chapter 23, but it is interesting to examine the problem now (without having to deal with the concept of threads) and to see how to fix it.

First let us look at how the function works. If we look at its source code (which is easy since the source code for the entire BIND release is publicly available), we see that one file contains both gethostbyname and gethostbyaddr, and the file has the following general outline:

```
static struct hostent  host;     /* result stored here */

struct hostent *
gethostbyname(const char *hostname)
{
    return(gethostbyname2(hostname, family));  /* Figure 9.6 */
}

struct hostent *
gethostbyname2(const char *hostname, int family)
{
    /* call DNS functions for A or AAAA query */

    /* fill in host structure */

    return(&host);
}

struct hostent *
gethostbyaddr(const char *addr, size_t len, int family)
{
    /* call DNS functions for PTR query in in-addr.arpa domain */

    /* fill in host structure */

    return(&host);
}
```

We highlight the `static` storage class specifier of the result structure, because that is the basic problem. The fact that these three functions share a single `host` variable presents yet another problem that we discussed in Exercise 9.1. (Recall from Figure 9.6 that `gethostbyname2` is new with the IPv6 support in BIND 4.9.4. We will ignore the fact that `gethostbyname2` is involved when we call `gethostbyname`, as that doesn't affect this discussion.)

The reentrancy problem can occur in a normal Unix process that calls `gethostbyname` or `gethostbyaddr` from both the main flow of control and from a signal handler. When the signal handler is called (say it is a `SIGALRM` signal that is generated once a second), the main flow of control of the process is temporarily stopped and the signal handling function is called. Consider the following.

```
main()
{
    struct hostent  *hptr;

    ...
    signal(SIGALRM, sig_alrm);

    ...
    hptr = gethostbyname( ... );
    ...
}

void
sig_alrm(int signo)
{
    struct hostent  *hptr;

    ...
    hptr = gethostbyname( ... );
    ...
}
```

If the main flow of control is in the middle of `gethostbyname` when it is temporarily stopped (say the function has filled in the `host` variable and is about to return), and the signal handler then calls `gethostbyname`, since only one copy of the variable `host` exists in the process, it is reused. This overwrites the values that were calculated for the call from the main flow of control with the values calculated for the call from the signal handler.

If we look at the name and address conversion functions presented in this chapter and Chapter 9, along with the `inet_XXX` functions from Chapter 4, we note the following:

- Historically, `gethostbyname`, `gethostbyname2`, `gethostbyaddr`, `getservbyname`, and `getservbyport` are not reentrant because all return a pointer to a static structure.

  Some implementations that support threads (Solaris 2.x) provide reentrant versions of these four functions with names ending with the `_r` suffix, which we describe in the next section.

Alternately, some implementations that support threads (Digital Unix 4.0 and HP-UX 10.30) provide reentrant versions of these functions using thread-specific data.

- `inet_pton` and `inet_ntop` are always reentrant.
- Historically `inet_ntoa` is not reentrant but some implementations that support threads provide a reentrant version that uses thread-specific data.
- `getaddrinfo` is reentrant only if it calls reentrant functions itself; that is, if it calls reentrant versions of `gethostbyname` for the hostname, and `getservbyname` for the service name. One reason that all the memory for the results is dynamically allocated is to allow it to be reentrant.
- `getnameinfo` is reentrant only if it calls reentrant functions itself; that is, if it calls reentrant versions of `gethostbyaddr` to obtain the hostname, and `getservbyport` to obtain the service name. Notice that both result strings (for the hostname and the service name) are allocated by the caller, to allow this reentrancy.

A similar problem occurs with the variable `errno`. Historically there has been a single copy of this integer variable per process. If the process makes a system call that returns an error, an integer error code is stored into this variable. For example, when the function named `close` in the standard C library is called, it might execute something like the following pseudocode:

- put the argument to the system call (an integer descriptor) into a register
- put a value in another register indicating the `close` system call is being called
- invoke the system call (switch to the kernel with a special instruction)
- test the value of a register to see if an error occurred
- if no error, `return(0)`
- store the value of some other register into `errno`
- `return(-1)`

First notice that if an error does not occur, the value of `errno` is not changed. That is why we cannot look at the value of `errno` unless we know that an error has occurred (normally indicated by the function returning –1).

Assume the program tests the return value of the `close` function and then prints the value of `errno` if an error occurred, as in the following:

```
if (close(fd) < 0) {
    fprintf(stderr, "close error, errno = %d\n", errno)
    exit(1);
}
```

There is a small window of time between the storing of the error code into `errno` when the system call returns, and the printing of this value by the program, during which another thread of execution within this process (i.e., a signal handler) can change the value of `errno`. For example, if, when the signal handler is called, the main flow of control is between the `close` and the `fprintf` and the signal handler calls some other system call that returns an error (say `write`), then the `errno` value stored from the

write system call overwrites the value stored by the close system call.

In looking at these two problems with regard to signal handlers, one solution to the problem with gethostbyname (returning a pointer to a static variable) is to *not* call nonreentrant functions from a signal handler. The problem with errno (a single global variable that can be changed by the signal handler) can be avoided by coding the signal handler to save and restore the value of errno in the signal handler, as follows:

```
void
sig_alrm(int signo)
{
    int  errno_save;

    errno_save = errno;        /* save its value on entry */
    if (write( ... ) != nbytes)
        fprintf(stderr, "write error, errno = %d\n", errno);
    errno = errno_save;        /* restore its value on return */
}
```

In this example code we also call fprintf, a standard I/O function, from the signal handler. This is yet another reentrancy problem because many versions of the standard I/O library are nonreentrant: standard I/O functions should not be called from signal handlers.

We revisit this problem of reentrancy in Chapter 23 and we will see how threads handle the problem of the errno variable. The next section describes some reentrant versions of the hostname functions.

11.15 gethostbyname_r and gethostbyaddr_r Functions

There are two ways to make a nonreentrant function such as gethostbyname reentrant.

1. Instead of filling in and returning a static structure, the caller allocates the structure and the reentrant function fills in the caller's structure. This is the technique used in going from the nonreentrant gethostbyname to the reentrant gethostbyname_r. But this solution gets more complicated because not only must the caller provide the hostent structure to fill in, but this structure also points to other information: the canonical name, the array of alias pointers, the alias strings, the array of address pointers, and the addresses (e.g., Figure 9.2). The caller must provide one large buffer that is used for this additional information and the hostent structure that is filled in then contains numerous pointers into this other buffer. This adds at least three arguments to the function: a pointer to the hostent structure to fill in, a pointer to the buffer to use for all the other information, and the size of this buffer. A fourth additional argument is also required, a pointer to an integer in which an error code can be stored, since the global integer h_errno can no longer be used. (The global integer h_errno presents the same reentrancy problem that we described with errno.)

   This technique is also used by getnameinfo and inet_ntop.

2. The reentrant function calls `malloc` and dynamically allocates the memory. This is the technique used by `getaddrinfo`. The problem with this approach is that the application calling this function must also call `freeaddrinfo` to free the dynamic memory. If the free function is not called, a *memory leak* occurs: each time the process calls the function that allocates the memory, the memory use of the process increases. If the process runs for a long time (a common trait of network servers), the memory usage just grows and grows over time.

We now discuss the Solaris 2.x reentrant functions for name-to-address and address-to-name resolution.

```
#include <netdb.h>

struct hostent *gethostbyname_r(const char *hostname,
                                struct hostent *result,
                                char *buf, int buflen, int *h_errnop);

struct hostent *gethostbyaddr_r(const char *addr, int len, int type,
                                struct hostent *result,
                                char *buf, int buflen, int *h_errnop);
```

Both return: nonnull pointer if OK, NULL on error

Four additional arguments are required for each function. *result* is a `hostent` structure allocated by the caller which is filled in by the function. On success this pointer is also the return value of the function.

buf is a buffer allocated by the caller and *buflen* is its size. This buffer will contain the canonical hostname, the alias pointers, the alias strings, the address pointers, and the actual addresses. All the pointers in the structure pointed to by *result* point into this buffer. How big should this buffer be? Unfortunately all that most manual pages say is something vague like "The buffer must be large enough to hold all of the data associated with the host entry." Current implementations of `gethostbyname` can return up to 35 alias pointers, 35 address pointers, and internally use an 8192-byte buffer to hold the alias names and addresses. So a buffer size of 8192 bytes should be adequate.

If an error occurs, the error code is returned through the *h_errnop* pointer, and not through the global `h_errno`.

Unfortunately this problem of reentrancy is even worse than it appears. First, there is no standard regarding reentrancy and `gethostbyname` and `gethostbyaddr`. Posix.1g specifies both functions but says nothing about thread safety. Unix 98 just says that these two functions need not be thread-safe.

Second, there is no standard for the `_r` functions. What we have shown in this section (for example purposes) are two of the `_r` functions provided by Solaris 2.x. But Digital Unix 4.0 and HP-UX 10.30 have versions of these functions with different arguments. The first two arguments for `gethostbyname_r` are the same as the Solaris version, but the remaining three arguments for the Solaris version are combined into a new `hostent_data` structure (which must be allocated by the caller), and a pointer to this structure is the third and final argument. The normal functions `gethostbyname` and `gethostbyaddr` in Digital Unix 4.0 and HP-UX

10.30 are reentrant, by using thread-specific data (Section 23.5). An interesting history of the development of the Solaris 2.x _r functions is in [Maslen 1997].

Lastly, while a reentrant version of gethostbyname may provide safety from different threads calling it at the same time, this says nothing about the reentrancy of the underlying resolver functions. As of this writing, the resolver functions in BIND are not reentrant.

11.16 Implementation of getaddrinfo and getnameinfo Functions

We now look at an implementation of getaddrinfo and getnameinfo. Developing an implementation of the former will let us look at how it operates in more detail. Our implementation also supports Unix domain sockets, as we mentioned in Section 11.5.

Note: All of the appropriate portions of the code that we look at in this section that are dependent on IPv4, IPv6, or Unix domain support, are bounded by an #ifdef and #endif of the appropriate constant: IPV4, IPV6, or UNIXDOMAIN. This allows the code to be compiled on a system that supports any combination of these three protocols. But we have removed all these preprocessor statements from the code that we show because they add nothing to our discussion and make the code harder to follow.

We also note that we do not cover Unix domain sockets in detail until Chapter 14.

Figure 11.18 shows the functions that are called by getaddrinfo. All begin with the ga_ prefix.

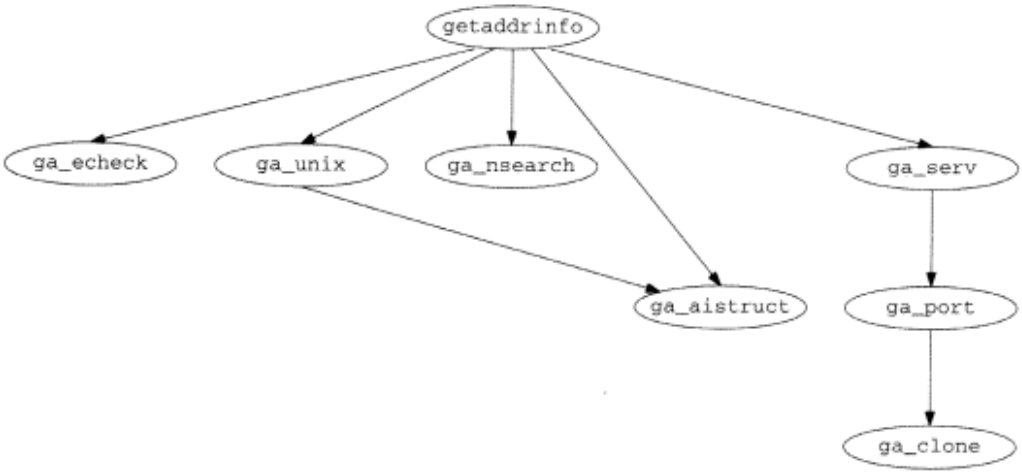

Figure 11.18 Functions called by our implementation of getaddrinfo.

The first file is our gai_hdr.h header, shown in Figure 11.19, which is included by all our source files.

We include our normal unp.h header and one additional header. We will see the use of our AI_CLONE flag and our search structure shortly. The remainder of the header defines the function prototypes for the various functions we show in this section.

libgai/gai_hdr.c

```
1 #include    "unp.h"
2 #include    <ctype.h>           /* isxdigit(), etc. */

3         /* following internal flag cannot overlap with other AI_xxx flags */
4 #define AI_CLONE         4      /* clone this entry for other socket types */

5 struct search {
6     const char *host;           /* hostname or address string */
7     int     family;             /* AF_xxx */
8 };

9             /* function prototypes for our own internal functions */
10 int     ga_aistruct(struct addrinfo ***, const struct addrinfo *,
11                     const void *, int);
12 struct addrinfo *ga_clone(struct addrinfo *);
13 int     ga_echeck(const char *, const char *, int, int, int, int);
14 int     ga_nsearch(const char *, const struct addrinfo *, struct search *);
15 int     ga_port(struct addrinfo *, int, int);
16 int     ga_serv(struct addrinfo *, const struct addrinfo *, const char *);
17 int     ga_unix(const char *, struct addrinfo *, struct addrinfo **);

18 int     gn_ipv46(char *, size_t, char *, size_t, void *, size_t,
19                  int, int, int);
```

libgai/gai_hdr.c

Figure 11.19 `gai_hdr.h` header.

Figure 11.20 shows the first part of the `getaddrinfo` function.

Define `error` macro

13–17 At more than a dozen points throughout this function if we encounter an error, we want to free all the memory that we have allocated and return the appropriate return code. To simplify the code we define this macro that stores the return code in the variable `error` and branches to the label `bad` at the end of the function (Figure 11.26).

Initialize automatic variables

18–20 Some automatic variables are initialized. We describe the `aihead` and `aipnext` pointers in Figure 11.34.

Copy caller's `hints` structure

21–25 If the caller provides a *hints* structure, we copy it into our own local variable, so we can modify it later. Otherwise we start with a structure that is all zero, other than `ai_family`, which is initialized to `AF_UNSPEC`. The latter is normally defined to be 0, but this is not required by Posix.1g.

Check arguments

26–29 We call our `ga_echeck` function, shown in Figure 11.39, to validate some of the arguments.

Check for Unix domain pathname

30–34 If the hostname is either `/local` or `/unix` and the service name begins with a slash, we process this argument as a Unix domain pathname. Our function `ga_unix` (Figure 11.33) completely processes the pathname.

libgai/getaddrinfo.c

```
 1 #include    "gai_hdr.h"
 2 #include    <arpa/nameser.h>    /* needed for <resolv.h> */
 3 #include    <resolv.h>          /* res_init, _res */

 4 int
 5 getaddrinfo(const char *hostname, const char *servname,
 6             const struct addrinfo *hintsp, struct addrinfo **result)
 7 {
 8     int     rc, error, nsearch;
 9     char  **ap, *canon;
10     struct hostent *hptr;
11     struct search search[3], *sptr;
12     struct addrinfo hints, *aihead, **aipnext;

13     /*
14      * If we encounter an error we want to free() any dynamic memory
15      * that we've allocated.  This is our hack to simplify the code.
16      */
17 #define error(e) { error = (e); goto bad; }

18     aihead = NULL;              /* initialize automatic variables */
19     aipnext = &aihead;
20     canon = NULL;

21     if (hintsp == NULL) {
22         bzero(&hints, sizeof(hints));
23         hints.ai_family = AF_UNSPEC;
24     } else
25         hints = *hintsp;        /* struct copy */

26         /* first some basic error checking */
27     if ( (rc = ga_echeck(hostname, servname, hints.ai_flags, hints.ai_family,
28                          hints.ai_socktype, hints.ai_protocol)) != 0)
29         error(rc);

30         /* special case Unix domain first */
31     if (hostname != NULL &&
32      (strcmp(hostname, "/local") == 0 || strcmp(hostname, "/unix") == 0) &&
33         (servname != NULL && servname[0] == '/'))
34         return (ga_unix(servname, &hints, result));
```

libgai/getaddrinfo.c

Figure 11.20 getaddrinfo function: first part, initialization.

The remainder of our getaddrinfo function (which continues in Figure 11.24) deals with IPv4 and IPv6 sockets. Our function ga_nsearch, the first part of which is shown in Figure 11.21, calculates the number of times that we look up a hostname. If the caller specifies an address family of AF_INET or AF_INET6, then we look up the hostname only one time. But if the address family is unspecified, AF_UNSPEC, then we do two lookups: once for an IPv6 hostname, and again for an IPv4 hostname. We show the function in three parts:

- no hostname and AI_PASSIVE specified,
- no hostname and AI_PASSIVE not specified (i.e., active), and
- hostname specified.

These three parts correspond to the three major portions of Figure 11.4.

libgai/ga_nsearch.c

```
 6 int
 7 ga_nsearch(const char *hostname, const struct addrinfo *hintsp,
 8             struct search *search)
 9 {
10     int     nsearch = 0;

11     if (hostname == NULL || hostname[0] == '\0') {
12         if (hintsp->ai_flags & AI_PASSIVE) {
13                 /* no hostname and AI_PASSIVE: implies wildcard bind */
14             switch (hintsp->ai_family) {
15             case AF_INET:
16                 search[nsearch].host = "0.0.0.0";
17                 search[nsearch].family = AF_INET;
18                 nsearch++;
19                 break;
20             case AF_INET6:
21                 search[nsearch].host = "0::0";
22                 search[nsearch].family = AF_INET6;
23                 nsearch++;
24                 break;
25             case AF_UNSPEC:
26                 search[nsearch].host = "0::0";  /* IPv6 first, then IPv4 */
27                 search[nsearch].family = AF_INET6;
28                 nsearch++;
29                 search[nsearch].host = "0.0.0.0";
30                 search[nsearch].family = AF_INET;
31                 nsearch++;
32                 break;
33             }
```

libgai/ga_nsearch.c

Figure 11.21 `ga_nsearch` function: no hostname and passive.

No hostname and passive socket

11-33 If the caller does not specify a hostname and specifies `AI_PASSIVE`, we return information to create one or more passive sockets that will bind the wildcard address. A `switch` is made based on the address family: an IPv4 socket needs to bind 0.0.0.0 (`INADDR_ANY`), and an IPv6 socket needs to bind 0::0 (`IN6ADDR_ANY_INIT`). If the family is `AF_UNSPEC`, we must return information to create two sockets: the first one for IPv6 and the second for IPv4. The reason for the ordering of IPv6 first, and then IPv4 is because an IPv6 socket on a dual-stack host can handle both IPv6 and IPv4 clients. In this scenario, if the caller creates only one socket from the returned list of `addrinfo` structures, it should be the IPv6 socket.

This function creates an array of `search` structures (Figure 11.19) with each entry specifying the hostname to look up and the address family. The pointer to the caller's array of `search` structures is the last argument to this function. The return value is the number of these structures that are created, and this will always be one or two.

The next part of this function, shown in Figure 11.22, handles the case of no hostname and AI_PASSIVE not set. This implies that the caller wants to create an active socket to the local host.

libgai/ga_nsearch.c

```
34          } else {
35                  /* no host and not AI_PASSIVE: connect to local host */
36              switch (hintsp->ai_family) {
37              case AF_INET:
38                  search[nsearch].host = "localhost";      /* 127.0.0.1 */
39                  search[nsearch].family = AF_INET;
40                  nsearch++;
41                  break;
42              case AF_INET6:
43                  search[nsearch].host = "0::1";
44                  search[nsearch].family = AF_INET6;
45                  nsearch++;
46                  break;
47              case AF_UNSPEC:
48                  search[nsearch].host = "0::1";  /* IPv6 first, then IPv4 */
49                  search[nsearch].family = AF_INET6;
50                  nsearch++;
51                  search[nsearch].host = "localhost";
52                  search[nsearch].family = AF_INET;
53                  nsearch++;
54                  break;
55              }
56          }
```

libgai/ga_nsearch.c

Figure 11.22 ga_nsearch function: no hostname and not passive.

34-56 For IPv4 we assume the hostname localhost will return the loopback address, normally 127.0.0.1. There is no common hostname for the local host with IPv6, so we return the loopback address of 0::1. As with the passive case, if no address family is specified we return two structures: first one for IPv6 and then one for IPv4.

Figure 11.23 shows the final part of this function, the else clause of the original if statement. This code is executed when a hostname is specified.

57-82 The AI_PASSIVE flag does not matter in this scenario; the hostname needs to be looked up. If the caller creates a passive socket, then the resulting socket address structure will be used in a call to bind, but if the caller creates an active socket, the socket address structure will be used in a call to connect. We create one or two search structures: one if the address family is specified and two if it is not specified. As with the previous two scenarios, if two structures are returned, the first is for IPv6 and the second for IPv4.

We now return to our getaddrinfo function, in Figure 11.24, which starts with a call to ga_nsearch.

libgai/ga_nsearch.c

```
57      } else {                        /* host is specified */
58          switch (hintsp->ai_family) {
59          case AF_INET:
60              search[nsearch].host = hostname;
61              search[nsearch].family = AF_INET;
62              nsearch++;
63              break;
64          case AF_INET6:
65              search[nsearch].host = hostname;
66              search[nsearch].family = AF_INET6;
67              nsearch++;
68              break;
69          case AF_UNSPEC:
70              search[nsearch].host = hostname;
71              search[nsearch].family = AF_INET6;  /* IPv6 first */
72              nsearch++;
73              search[nsearch].host = hostname;
74              search[nsearch].family = AF_INET;   /* then IPv4 */
75              nsearch++;
76              break;
77          }
78      }
79      if (nsearch < 1 || nsearch > 2)
80          err_quit("nsearch = %d", nsearch);
81      return (nsearch);
82  }
```

libgai/ga_nsearch.c

Figure 11.23 `ga_nsearch` function: hostname specified.

Call `ga_nsearch`

36 We call our `ga_nsearch` function, filling in our `search` array, and returning the number of structures in the array: one or two.

Loop through all the `search` structures

37 We loop through each `search` structure that was created by `ga_nsearch`.

Check for IPv4 dotted-decimal string

39–44 If the first character of the hostname is a digit, we check whether or not the hostname is really a dotted-decimal string. We call `inet_pton` to do this check and conversion. If it succeeds but the caller specifies an address family other than `AF_INET`, this is an error.

45–46 We check that the `family` of the `search` structure is also `AF_INET`, but a mismatch here only causes this `search` structure to be ignored. This scenario can happen, for example, if the caller specifies a hostname of 192.3.4.5 but no address family. `ga_nsearch` creates two `search` structures: one for IPv6 and one for IPv4. The first time through the `for` loop the call to `inet_pton` succeeds, but since the `family` of the `search` structure is `AF_INET6`, we want to ignore this structure, and not generate an error.

libgai/getaddrinfo.c

```
35          /* remainder of function for IPv4/IPv6 */
36      nsearch = ga_nsearch(hostname, &hints, &search[0]);
37      for (sptr = &search[0]; sptr < &search[nsearch]; sptr++) {
38              /* check for an IPv4 dotted-decimal string */
39          if (isdigit(sptr->host[0])) {
40              struct in_addr inaddr;

41              if (inet_pton(AF_INET, sptr->host, &inaddr) == 1) {
42                  if (hints.ai_family != AF_UNSPEC &&
43                      hints.ai_family != AF_INET)
44                      error(EAI_ADDRFAMILY);
45                  if (sptr->family != AF_INET)
46                      continue;   /* ignore */
47                  rc = ga_aistruct(&aipnext, &hints, &inaddr, AF_INET);
48                  if (rc != 0)
49                      error(rc);
50                  continue;
51              }
52          }
53              /* check for an IPv6 hex string */
54          if ((isxdigit(sptr->host[0]) || sptr->host[0] == ':') &&
55              (strchr(sptr->host, ':') != NULL)) {
56              struct in6_addr in6addr;

57              if (inet_pton(AF_INET6, sptr->host, &in6addr) == 1) {
58                  if (hints.ai_family != AF_UNSPEC &&
59                      hints.ai_family != AF_INET6)
60                      error(EAI_ADDRFAMILY);
61                  if (sptr->family != AF_INET6)
62                      continue;   /* ignore */
63                  rc = ga_aistruct(&aipnext, &hints, &in6addr, AF_INET6);
64                  if (rc != 0)
65                      error(rc);
66                  continue;
67              }
68          }
```

libgai/getaddrinfo.c

Figure 11.24 `getaddrinfo` function: check for IPv4 or IPv6 address string.

Create `addrinfo` structure

47–52 Our function `ga_aistruct` creates an `addrinfo` structure and adds it to the linked list that is being built (the `aipnext` pointer).

Check for IPv6 address string

53–60 If the first character of the hostname is either a hexadecimal digit or a colon and the string contains a colon, we check whether the hostname is an IPv6 address string by calling `inet_pton`. If it succeeds but the caller specifies an address family other than `AF_INET6`, this is an error.

61–62 We check that the `family` of the `search` structure is also `AF_INET6`, but a mismatch here only causes this `search` structure to be ignored.

Create addrinfo structure

63-68 Our function ga_aistruct creates an addrinfo structure and adds it to the linked list that is being built.

The first two tests in the loop (Figure 11.24) handle an IPv4 dotted-decimal string or an IPv6 address string. The remainder of the loop, shown in Figure 11.25, looks up the hostname by calling either gethostbyname or gethostbyname2.

Initialize resolver first time

70-71 We call the resolver's res_init function if it has not been called before.

Call gethostbyname2 if two searches are being performed

72-74 If nsearch is 2, then we are going through the for loop twice: once for IPv6 and again for IPv4. If the hostname argument has an address in only one of the two families, we want to return only that address. For example, our host solaris in Section 9.2 has a AAAA record and an A record in the DNS. The first time around the loop we want to find the AAAA record, and the second time the A record. But if the hostname has only an A record, we do not want to process that record the first time around the loop when the family member of the search structure is AF_INET6. That is, since we know that we will be searching for an A record for this host, do not search for a AAAA record using gethostbyname and possibly return the IPv4-mapped IPv6 address corresponding to the A record. Looking at Figure 9.5 the way to search for only A records when the family is AF_INET and to search for only AAAA records when the family is AF_INET6 is to call gethostbyname2 instead of gethostbyname, with the RES_USE_INET6 option off.

Call gethostbyname if one search is being performed

75-81 If only one search is being performed, we call gethostbyname with the RES_USE_INET6 option set if the family is AF_INET6 or the option cleared if the family is AF_INET. For example, if the caller specifies a hostname that has only an A record, but specifies a family of AF_INET6, we want to return the IPv4-mapped IPv6 address.

Handle resolver failure

82-97 If the call to the resolver failed, but nsearch is two, this is not an error, as one of the passes through the loop may succeed. (We check at the end of the loop that at least one addrinfo structure is being returned.) But if this was the only call to the resolver we return an error corresponding to the resolver's h_errno.

Check for address family mismatch

98-100 If the caller specifies an address family, but the family returned by the resolver differs, this is an error.

Save canonical name

101-106 If the caller specifies a hostname and the AI_CANONNAME flag, we save the first canonical name returned by the resolver. (Recall from Figure 11.22 that we call the resolver for the name localhost even if the caller does not specify a hostname.) We duplicate the string returned by the resolver and save its pointer in canon.

libgai/getaddrinfo.c

```
69              /* remainder of for() to look up hostname */
70          if ((_res.options & RES_INIT) == 0)
71              res_init();         /* need this to set _res.options */
72          if (nsearch == 2) {
73              _res.options &= ~RES_USE_INET6;
74              hptr = gethostbyname2(sptr->host, sptr->family);
75          } else {
76              if (sptr->family == AF_INET6)
77                  _res.options |= RES_USE_INET6;
78              else
79                  _res.options &= ~RES_USE_INET6;
80              hptr = gethostbyname(sptr->host);
81          }
82          if (hptr == NULL) {
83              if (nsearch == 2)
84                  continue;       /* failure OK if multiple searches */
85              switch (h_errno) {
86              case HOST_NOT_FOUND:
87                  error(EAI_NONAME);
88              case TRY_AGAIN:
89                  error(EAI_AGAIN);
90              case NO_RECOVERY:
91                  error(EAI_FAIL);
92              case NO_DATA:
93                  error(EAI_NODATA);
94              default:
95                  error(EAI_NONAME);
96              }
97          }
98              /* check for address family mismatch if one specified */
99          if (hints.ai_family != AF_UNSPEC && hints.ai_family != hptr->h_addrtype)
100             error(EAI_ADDRFAMILY);
101             /* save canonical name first time */
102         if (hostname != NULL && hostname[0] != '\0' &&
103             (hints.ai_flags & AI_CANONNAME) && canon == NULL) {
104             if ( (canon = strdup(hptr->h_name)) == NULL)
105                 error(EAI_MEMORY);
106         }
107             /* create one addrinfo() for each returned address */
108         for (ap = hptr->h_addr_list; *ap != NULL; ap++) {
109             rc = ga_aistruct(&aipnext, &hints, *ap, hptr->h_addrtype);
110             if (rc != 0)
111                 error(rc);
112         }
113     }
114     if (aihead == NULL)
115         error(EAI_NONAME);      /* nothing found */
```

libgai/getaddrinfo.c

Figure 11.25 getaddrinfo function: lookup hostname.

Create one addrinfo structure per address

107-112 For each address returned by the resolver in the h_addr_list array, we call our ga_aistruct function to create an addrinfo structure and append it to the linked list of structures being created.

Check for no matches

114-115 If the head of the linked list of addrinfo structures is still a null pointer, all iterations through the for loop failed.

Figure 11.26 shows the final part of the getaddrinfo function.

libgai/getaddrinfo.c

```
116         /* return canonical name */
117     if (hostname != NULL && hostname[0] != '\0' &&
118         hints.ai_flags & AI_CANONNAME) {
119         if (canon != NULL)
120             aihead->ai_canonname = canon;    /* strdup'ed earlier */
121         else {
122             if ( (aihead->ai_canonname = strdup(search[0].host)) == NULL)
123                 error(EAI_MEMORY);
124         }
125     }
126         /* now process the service name */
127     if (servname != NULL && servname[0] != '\0') {
128         if ( (rc = ga_serv(aihead, &hints, servname)) != 0)
129             error(rc);
130     }
131     *result = aihead;           /* pointer to first structure in linked list */
132     return (0);

133   bad:
134     freeaddrinfo(aihead);       /* free any alloc'ed memory */
135     return (error);
136 }
```

libgai/getaddrinfo.c

Figure 11.26 getaddrinfo function: process service name.

Return canonical name

116-125 If the caller specifies a hostname and the AI_CANONNAME flag, and if we saved a copy to the canonical name in our canon pointer, that pointer is returned in the ai_canonname member of the first addrinfo structure. If no canonical name was found by the resolver (perhaps the hostname was an address string), then a copy of the hostname argument is returned instead.

Process service name

126-130 If the caller specifies a service name, it is now processed by calling our ga_serv function.

Return pointer to linked list

131-132 The pointer to the head of the linked list of addrinfo structures that have been created is returned, along with a function return value of 0.

Error return

133-135 If an error was encountered, freeaddrinfo is called to free all the memory that was allocated, and the return value is the EAI_*xxx* value.

Our ga_serv function, which was called from Figure 11.26 to process the service name argument, is shown in Figure 11.27.

Check for port number string

12-27 If the first character of the service name is a digit, we assume the service name is a port number and call atoi to convert it to binary. If the caller specifies a socket type (SOCK_STREAM or SOCK_DGRAM), then our ga_port function is called once for that socket type. But if no socket type is specified, our ga_port function is called twice, once for TCP and once for UDP. (Recall Figure 11.2.) We keep a counter of the number of times that ga_port returns success and return an error only if this is 0 at the end of the function.

Try `getservbyname` for TCP

28-36 If no socket type is specified, or a TCP socket is specified, getservbyname is called with a second argument of "tcp". If this succeeds, our ga_port function is called. If this call fails, that is OK, as the service name could be valid for UDP.

Try `getservbyname` for UDP

37-44 If no socket type is specified, or a UDP socket is specified, we call getservbyname with a second argument of "udp". If this succeeds, we call our ga_port function.

Check for error

45-51 If our nfound counter is nonzero, we had success. Otherwise an error is returned.

Our ga_port function, which we show in Figure 11.28, was called from Figure 11.27 each time a port number was found.

Loop through all `addrinfo` structures

33 We loop through all the addrinfo structures that were created by the calls to ga_aistruct in Figures 11.24 and 11.25. The AI_CLONE flag is always set by ga_aistruct when no socket type is specified by the caller. That is an indication that this addrinfo structure might need to be cloned for both TCP and UDP.

Check `AI_CLONE` flag

34-42 If the AI_CLONE flag is set and if the socket type is nonzero, another addrinfo structure is cloned from this one by our ga_clone function. We show an example of this shortly.

Set port number in socket address structure

44-47 The port number in the socket address structure is set and our counter nfound is incremented.

libgai/ga_serv.c

```
 5 int
 6 ga_serv(struct addrinfo *aihead, const struct addrinfo *hintsp,
 7         const char *serv)
 8 {
 9     int     port, rc, nfound;
10     struct servent *sptr;

11     nfound = 0;
12     if (isdigit(serv[0])) {     /* check for port number string first */
13         port = htons(atoi(serv));
14         if (hintsp->ai_socktype) {
15                 /* caller specifies socket type */
16             if ( (rc = ga_port(aihead, port, hintsp->ai_socktype)) < 0)
17                 return (EAI_MEMORY);
18             nfound += rc;
19         } else {
20                 /* caller does not specify socket type */
21             if ( (rc = ga_port(aihead, port, SOCK_STREAM)) < 0)
22                 return (EAI_MEMORY);
23             nfound += rc;
24             if ( (rc = ga_port(aihead, port, SOCK_DGRAM)) < 0)
25                 return (EAI_MEMORY);
26             nfound += rc;
27         }
28     } else {
29             /* try service name, TCP then UDP */
30         if (hintsp->ai_socktype == 0 || hintsp->ai_socktype == SOCK_STREAM) {
31             if ( (sptr = getservbyname(serv, "tcp")) != NULL) {
32                 if ( (rc = ga_port(aihead, sptr->s_port, SOCK_STREAM)) < 0)
33                     return (EAI_MEMORY);
34                 nfound += rc;
35             }
36         }
37         if (hintsp->ai_socktype == 0 || hintsp->ai_socktype == SOCK_DGRAM) {
38             if ( (sptr = getservbyname(serv, "udp")) != NULL) {
39                 if ( (rc = ga_port(aihead, sptr->s_port, SOCK_DGRAM)) < 0)
40                     return (EAI_MEMORY);
41                 nfound += rc;
42             }
43         }
44     }

45     if (nfound == 0) {
46         if (hintsp->ai_socktype == 0)
47             return (EAI_NONAME);    /* all calls to getservbyname() failed */
48         else
49             return (EAI_SERVICE);   /* service not supported for socket type */
50     }
51     return (0);
52 }
```

libgai/ga_serv.c

Figure 11.27 ga_serv function.

libgai/ga_port.c

```
27 int
28 ga_port(struct addrinfo *aihead, int port, int socktype)
29         /* port must be in network byte order */
30 {
31     int     nfound = 0;
32     struct addrinfo *ai;

33     for (ai = aihead; ai != NULL; ai = ai->ai_next) {
34         if (ai->ai_flags & AI_CLONE) {
35             if (ai->ai_socktype != 0) {
36                 if ( (ai = ga_clone(ai)) == NULL)
37                     return (-1);    /* memory allocation error */
38                 /* ai points to newly cloned entry, which is what we want */
39             }
40         } else if (ai->ai_socktype != socktype)
41             continue;           /* ignore if mismatch on socket type */

42         ai->ai_socktype = socktype;

43         switch (ai->ai_family) {
44         case AF_INET:
45             ((struct sockaddr_in *) ai->ai_addr)->sin_port = port;
46             nfound++;
47             break;
48         case AF_INET6:
49             ((struct sockaddr_in6 *) ai->ai_addr)->sin6_port = port;
50             nfound++;
51             break;
52         }
53     }
54     return (nfound);
55 }
```

libgai/ga_port.c

Figure 11.28 ga_port function.

Consider an example. In Figure 11.1 we assumed a call to getaddrinfo for a host with two IP addresses, a service name of domain (port 53 for both TCP and UDP), and no specification of the socket type. The loop in our getaddrinfo function (Figure 11.25) creates two addrinfo structures, one for each IP address returned by gethostbyname. The AI_CLONE flag is also set in each structure, because no socket type is specified. We show the resulting linked list in Figure 11.29.

ga_serv is called from Figure 11.26. Since the domain service name is valid for both TCP and UDP, getservbyname is called two times, and ga_port is called two times: first with a final argument of SOCK_STREAM and again with a final argument of SOCK_DGRAM. The first time ga_port is called it starts with the linked list shown in Figure 11.29. In Figure 11.28 the AI_CLONE flag is set for both structures, but the socket type is 0. Therefore all that happens to each addrinfo structure the first time ga_port is called is to set the ai_socktype member to SOCK_STREAM and the port number in the socket address structure to 53. The AI_CLONE flag remains set. This gives us the linked list shown in Figure 11.30.

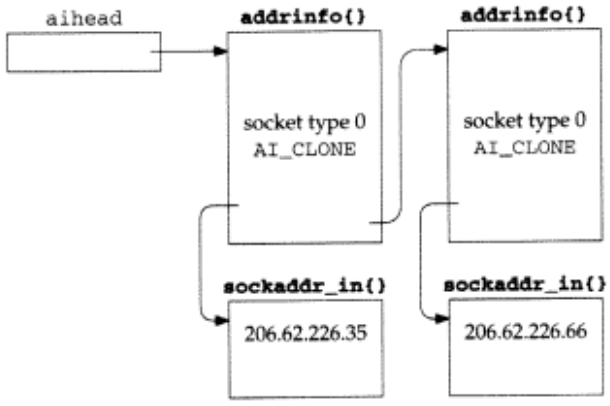

Figure 11.29 `addrinfo` structures when `ga_port` is called first time.

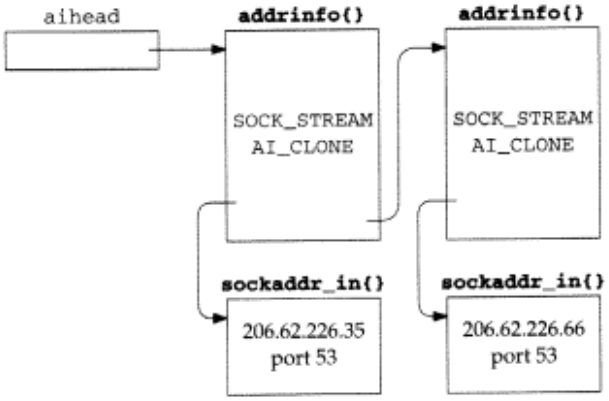

Figure 11.30 `addrinfo` structures after first call to `ga_port`.

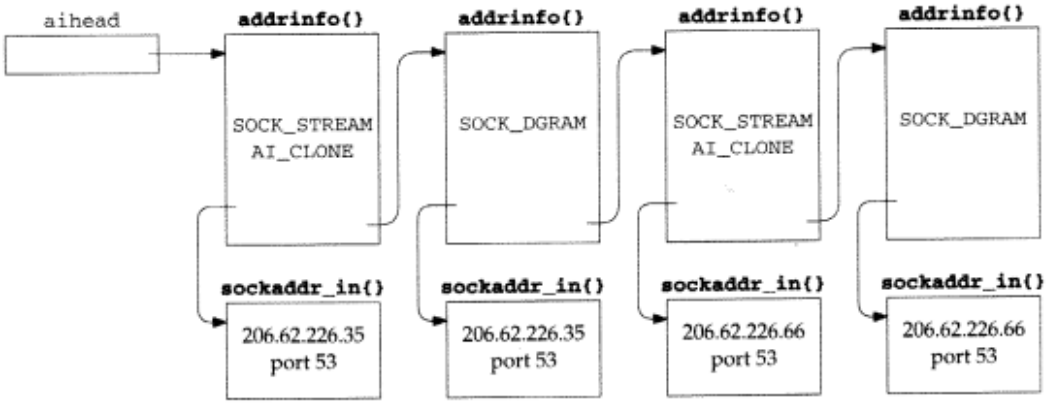

Figure 11.31 `addrinfo` structures after second call to `ga_port`.

But the second time ga_port is called (with a final argument of SOCK_DGRAM), since the AI_CLONE flag is set and the socket type is not 0, ga_clone is called for each addrinfo structure. The ai_socktype member in each of the newly cloned structures is set to SOCK_DGRAM and we end up with the linked list shown in Figure 11.31. In this figure the second addrinfo structure and its socket address structure are cloned from the first set of structures, and the fourth addrinfo structure and its socket address structure are cloned from the third set of structures.

Figure 11.32 shows our ga_clone function, which was called from Figure 11.28 to clone a new addrinfo structure and its socket address structure from an existing set of structures.

libgai/ga_clone.c

```
 5 struct addrinfo *
 6 ga_clone(struct addrinfo *ai)
 7 {
 8     struct addrinfo *new;

 9     if ( (new = calloc(1, sizeof(struct addrinfo))) == NULL)
10                 return (NULL);

11     new->ai_next = ai->ai_next;
12     ai->ai_next = new;

13     new->ai_flags = 0;          /* make sure AI_CLONE is off */
14     new->ai_family = ai->ai_family;
15     new->ai_socktype = ai->ai_socktype;
16     new->ai_protocol = ai->ai_protocol;
17     new->ai_canonname = NULL;
18     new->ai_addrlen = ai->ai_addrlen;
19     if ( (new->ai_addr = malloc(ai->ai_addrlen)) == NULL)
20         return (NULL);
21     memcpy(new->ai_addr, ai->ai_addr, ai->ai_addrlen);

22     return (new);
23 }
```

libgai/ga_clone.c

Figure 11.32 ga_clone function.

Allocate addrinfo structure and insert into linked list

9–12 A new addrinfo structure is allocated and its ai_next pointer is set to the ai_next pointer of the entry being cloned (i.e., what will be the previous entry on the list). The next pointer of the entry being cloned becomes the new structure just allocated.

Initialize from cloned entry

13–22 All the fields in the new addrinfo structure are copied from the entry being cloned with the exception of ai_flags, which is set to 0, and ai_canonname, which is set to a null pointer. A pointer to the newly created structure is the return value of the function.

Our ga_unix function, which we show in Figure 11.33, was called from Figure 11.20 to completely process a Unix domain pathname.

libgai/ga_unix.c

```
 3 int
 4 ga_unix(const char *path, struct addrinfo *hintsp, struct addrinfo **result)
 5 {
 6     int     rc;
 7     struct addrinfo *aihead, **aipnext;

 8     aihead = NULL;
 9     aipnext = &aihead;

10     if (hintsp->ai_family != AF_UNSPEC && hintsp->ai_family != AF_LOCAL)
11         return (EAI_ADDRFAMILY);

12     if (hintsp->ai_socktype == 0) {
13             /* no socket type specified: return stream then dgram */
14         hintsp->ai_socktype = SOCK_STREAM;
15         if ( (rc = ga_aistruct(&aipnext, hintsp, path, AF_LOCAL)) != 0)
16             return (rc);
17         hintsp->ai_socktype = SOCK_DGRAM;
18     }
19     if ( (rc = ga_aistruct(&aipnext, hintsp, path, AF_LOCAL)) != 0)
20         return (rc);

21     if (hintsp->ai_flags & AI_CANONNAME) {
22         struct utsname myname;

23         if (uname(&myname) < 0)
24             return (EAI_SYSTEM);
25         if ( (aihead->ai_canonname = strdup(myname.nodename)) == NULL)
26             return (EAI_MEMORY);
27     }
28     *result = aihead;           /* pointer to first structure in linked list */
29     return (0);
30 }
```

libgai/ga_unix.c

Figure 11.33 ga_unix function.

ga_aistruct creates structures

10-20 If a socket type is not specified, we call our ga_aistruct function twice to create two addrinfo structures: one with a socket type of SOCK_STREAM and another with a socket type of SOCK_DGRAM. But if the caller specifies a nonzero socket type, our ga_aistruct function is called only once, creating one addrinfo structure with that socket type.

Return canonical name

21-27 If the AI_CANONNAME flag was specified by the caller, we call uname to obtain the system name and return the nodename member (Section 9.7) as the canonical name.

We explain the aihead and aipnext pointers with the ga_aistruct function, which we describe next.

Our ga_aistruct function was called from Figures 11.24 and 11.25 to create an addrinfo structure for an IPv4 or IPv6 address, and from Figure 11.33 to create an addrinfo structure for a Unix domain socket. We show the first part of the function in Figure 11.34.

libgai/ga_aistruct.c

```
 5 int
 6 ga_aistruct(struct addrinfo ***paipnext, const struct addrinfo *hintsp,
 7               const void *addr, int family)
 8 {
 9     struct addrinfo *ai;

10     if ( (ai = calloc(1, sizeof(struct addrinfo))) == NULL)
11                 return (EAI_MEMORY);
12     ai->ai_next = NULL;
13     ai->ai_canonname = NULL;
14     **paipnext = ai;
15     *paipnext = &ai->ai_next;

16     if ( (ai->ai_socktype = hintsp->ai_socktype) == 0)
17         ai->ai_flags |= AI_CLONE;

18     ai->ai_protocol = hintsp->ai_protocol;
```

libgai/ga_aistruct.c

Figure 11.34 ga_aistruct function: first part.

Allocate addrinfo structure and add to linked list

10–15 An addrinfo structure is allocated and added to the linked list being built. Two pointers are used to build the linked list: aihead and aipnext. Both were allocated and initialized in Figure 11.20 for an IPv4 or an IPv6 socket, or in Figure 11.33 for a Unix domain socket. aihead is initialized to a null pointer and aipnext is initialized to point to aihead. We show this in Figure 11.35.

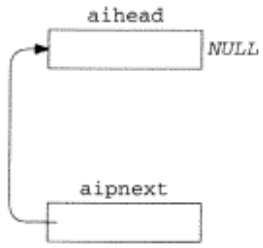

Figure 11.35 Initialization of linked list pointers.

aihead always points to the first addrinfo structure on the linked list (therefore its datatype is struct addrinfo *). aipnext normally points to the ai_next member of the last structure on the linked list (therefore its datatype is struct addrinfo **). We use the qualifier "normally" with regard to aipnext because upon initialization it really points to aihead, but after the first structure is allocated and placed onto the list, it always points to the ai_next member.

Returning to our `ga_aistruct` function, after a new structure is allocated the two statements

```
**paipnext = ai;
*paipnext = &ai->ai_next;
```

are executed. The first statement sets the `ai_next` pointer of the last structure on the list (or `aihead` if this new structure is the first on the list) to point to the newly allocated structure, and the second statement sets `aipnext` to point to the `ai_next` member of the newly allocated structure. The extra level of indirection is needed in both statements because the address of `aipnext` is an argument to the function (see Exercise 11.4). When the first structure is added to the list, we have the data structures shown in Figure 11.36.

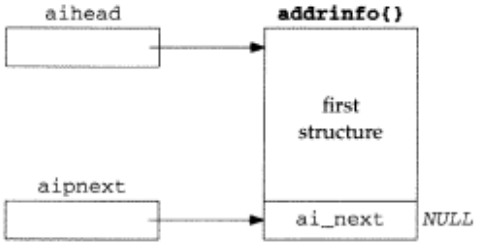

Figure 11.36 Linked list after first structure added.

When our `ga_aistruct` function is called the next time to allocate a second structure and add it to the list, we have the data structures shown in Figure 11.37.

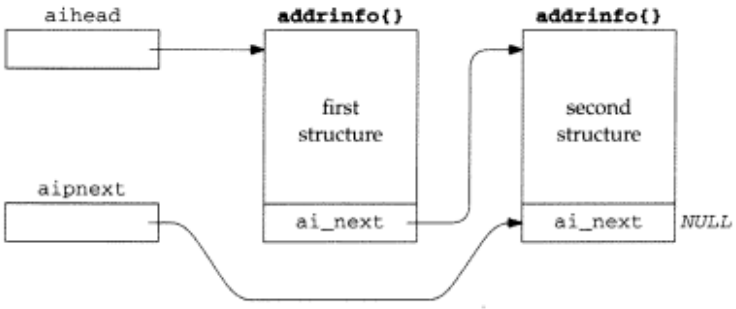

Figure 11.37 Linked list after second structure added.

Set socket type

16-17 The `ai_socktype` member is set to the socket type provided by the caller and if this is 0, the `AI_CLONE` flag is set.

Figure 11.38 shows the second part of the function: a `switch` with one `case` per address family to allocate a socket address structure and initialize it.

libgai/ga_aistruct.c

```
19      switch ((ai->ai_family = family)) {
20      case AF_INET:{
21              struct sockaddr_in *sinptr;
22                  /* allocate sockaddr_in{} and fill in all but port */
23              if ( (sinptr = calloc(1, sizeof(struct sockaddr_in))) == NULL)
24                          return (EAI_MEMORY);
25 #ifdef  HAVE_SOCKADDR_SA_LEN
26              sinptr->sin_len = sizeof(struct sockaddr_in);
27 #endif
28              sinptr->sin_family = AF_INET;
29              memcpy(&sinptr->sin_addr, addr, sizeof(struct in_addr));
30              ai->ai_addr = (struct sockaddr *) sinptr;
31              ai->ai_addrlen = sizeof(struct sockaddr_in);
32              break;
33          }
34      case AF_INET6:{
35              struct sockaddr_in6 *sin6ptr;
36                  /* allocate sockaddr_in6{} and fill in all but port */
37              if ( (sin6ptr = calloc(1, sizeof(struct sockaddr_in6))) == NULL)
38                          return (EAI_MEMORY);
39 #ifdef  HAVE_SOCKADDR_SA_LEN
40              sin6ptr->sin6_len = sizeof(struct sockaddr_in6);
41 #endif
42              sin6ptr->sin6_family = AF_INET6;
43              memcpy(&sin6ptr->sin6_addr, addr, sizeof(struct in6_addr));
44              ai->ai_addr = (struct sockaddr *) sin6ptr;
45              ai->ai_addrlen = sizeof(struct sockaddr_in6);
46              break;
47          }
48      case AF_LOCAL:{
49              struct sockaddr_un *unp;
50                  /* allocate sockaddr_un{} and fill in */
51              if (strlen(addr) >= sizeof(unp->sun_path))
52                  return(EAI_SERVICE);
53              if ( (unp = calloc(1, sizeof(struct sockaddr_un))) == NULL)
54                  return(EAI_MEMORY);
55              unp->sun_family = AF_LOCAL;
56              strcpy(unp->sun_path, addr);
57 #ifdef  HAVE_SOCKADDR_SA_LEN
58              unp->sun_len = SUN_LEN(unp);
59 #endif
60              ai->ai_addr = (struct sockaddr *) unp;
61              ai->ai_addrlen = sizeof(struct sockaddr_un);
62              if (hintsp->ai_flags & AI_PASSIVE)
63                  unlink(unp->sun_path);  /* OK if this fails */
64              break;
65          }
66      }
67      return (0);
68  }
```

libgai/ga_aistruct.c

Figure 11.38 ga_aistruct function: second part.

Allocate IPv4 socket address structure and initialize

20–33 A `sockaddr_in` structure is allocated and the `ai_addr` pointer in the `addrinfo` structure is set to point to it. The IP address, address family, and length members of the socket address structure are all initialized. The port number is not initialized until `ga_serv` is called, which in turn calls `ga_port`.

Allocate IPv6 socket address structure and initialize

34–47 A `sockaddr_in6` structure is allocated and initialized, similar to the IPv4 case.

Allocate Unix domain socket address structure and initialize

48–65 A `sockaddr_un` structure is allocated and initialized. The address is a pathname and if the `AI_PASSIVE` flag was specified by the caller, we try to `unlink` the pathname to prevent an error return when the caller calls `bind`. But it is not an error if the `unlink` fails.

Our `ga_echeck` function, which we show in Figure 11.39, was called from Figure 11.20 to perform some initial error checking on the caller's arguments.

libgai/ga_echeck.c

```
 5 int
 6 ga_echeck(const char *hostname, const char *servname,
 7           int flags, int family, int socktype, int protocol)
 8 {
 9     if (flags & ~(AI_PASSIVE | AI_CANONNAME))
10         return (EAI_BADFLAGS);  /* unknown flag bits */

11     if (hostname == NULL || hostname[0] == '\0') {
12         if (servname == NULL || servname[0] == '\0')
13             return (EAI_NONAME);    /* host or service must be specified */
14     }
15     switch (family) {
16     case AF_UNSPEC:
17         break;
18     case AF_INET:
19         if (socktype != 0 &&
20             (socktype != SOCK_STREAM &&
21              socktype != SOCK_DGRAM &&
22              socktype != SOCK_RAW))
23             return (EAI_SOCKTYPE);  /* invalid socket type */
24         break;
25     case AF_INET6:
26         if (socktype != 0 &&
27             (socktype != SOCK_STREAM &&
28              socktype != SOCK_DGRAM &&
29              socktype != SOCK_RAW))
30             return (EAI_SOCKTYPE);  /* invalid socket type */
31         break;
32     case AF_LOCAL:
33         if (socktype != 0 &&
34             (socktype != SOCK_STREAM &&
35              socktype != SOCK_DGRAM))
36             return (EAI_SOCKTYPE);  /* invalid socket type */
37         break;
```

```
38      default:
39          return (EAI_FAMILY);     /* unknown protocol family */
40      }
41      return (0);
42  }
```
libgai/ga_echeck.c

Figure 11.39 ga_echeck function.

9-14 The flags are verified and either a hostname or a service name must be specified.

15-41 Depending on the address family, only certain types of sockets are supported, and this verifies the socket type.

We do not check the caller's ai_protocol hint, if any, as few applications specify this value (which becomes the third argument to socket). Should an invalid combination be specified, such as a socket type of SOCK_DGRAM and a protocol of IPPROTO_TCP, the protocol hint is returned to the caller in Figure 11.34 and if the caller uses this value in a call to socket, an error of EPROTONOSUPPORT will be returned.

We have finished with the getaddrinfo function, and all the internal functions that it calls. Figure 11.40 shows the freeaddrinfo function, which releases all the memory in the linked list. We called this function from Figure 11.26 if an error occurred, and the user also calls it to release a linked list of structures.

libgai/freeaddrinfo.c

```
 1 #include    "gai_hdr.h"

 2 void
 3 freeaddrinfo(struct addrinfo *aihead)
 4 {
 5     struct addrinfo *ai, *ainext;

 6     for (ai = aihead; ai != NULL; ai = ainext) {
 7         if (ai->ai_addr != NULL)
 8             free(ai->ai_addr);  /* socket address structure */

 9         if (ai->ai_canonname != NULL)
10             free(ai->ai_canonname);

11         ainext = ai->ai_next;   /* can't fetch ai_next after free() */
12         free(ai);               /* the addrinfo{} itself */
13     }
14 }
```
libgai/freeaddrinfo.c

Figure 11.40 freeaddrinfo function: first part.

6-13 The linked list of addrinfo structures is traversed. If a socket address structure has been allocated, it is freed. If a canonical name string has been allocated, it is freed. Finally, the addrinfo structure itself is freed. We must be careful to save the contents of the structure's ai_next pointer before freeing the structure, as we cannot reference the structure after free returns.

Figure 11.41 shows our implementation of the getnameinfo function. It consists of a switch statement with one case per address family.

libgai/getnameinfo.c

```
 2 int
 3 getnameinfo(const struct sockaddr *sa, socklen_t salen,
 4             char *host, size_t hostlen,
 5             char *serv, size_t servlen, int flags)
 6 {
 7     switch (sa->sa_family) {
 8     case AF_INET:{
 9             struct sockaddr_in *sain = (struct sockaddr_in *) sa;
10             return (gn_ipv46(host, hostlen, serv, servlen,
11                              &sain->sin_addr, sizeof(struct in_addr),
12                              AF_INET, sain->sin_port, flags));
13         }
14     case AF_INET6:{
15             struct sockaddr_in6 *sain = (struct sockaddr_in6 *) sa;
16             return (gn_ipv46(host, hostlen, serv, servlen,
17                              &sain->sin6_addr, sizeof(struct in6_addr),
18                              AF_INET6, sain->sin6_port, flags));
19         }
20     case AF_LOCAL:{
21             struct sockaddr_un *un = (struct sockaddr_un *) sa;
22             if (hostlen > 0)
23                 snprintf(host, hostlen, "%s", "/local");
24             if (servlen > 0)
25                 snprintf(serv, servlen, "%s", un->sun_path);
26             return (0);
27         }
28     default:
29         return (1);
30     }
31 }
```

libgai/getnameinfo.c

Figure 11.41 `getnameinfo` function.

Handle IPv4 and IPv6 socket address structures

8-19 We call our `gn_ipv46` function (shown next) to handle IPv4 and IPv6 socket address structures.

Handle Unix domain socket address structures

20-27 For a Unix domain socket address structure we return `/local` as the hostname and the pathname that is bound to the socket as the service name. If no pathname is bound to the socket, then the returned service name will be a null string.

We return the hostname and service name using snprintf instead of strncpy. If we used the latter we could write

```
strncpy(host, "/local", hostlen);
```

While this guarantees that we do not overflow the caller's buffer, if hostlen is less than or equal to 6, then the caller's buffer will not be null terminated. But we are writing a library routine and we should always return a null-terminated string if that is what the caller expects. This could cause problems for the caller at a later time in the program. Therefore we should always write

```
strncpy(host, "/local", hostlen-1);
host[hostlen-1] = '\0';
```

which guarantees that we do not overwrite the caller's buffer and that the result is null terminated. We use snprintf instead of these two statements, since it will not overflow the destination and it guarantees that the destination is null terminated. An alternate design would be to define our own library function that calls strncpy and null terminates the result, but calling the existing snprintf seems simpler.

Figure 11.42 is our gn_ipv46 function, which handles IPv4 and IPv6 socket address structures for getnameinfo.

Return hostname

12-23 If the NI_NUMERICHOST flag is specified, we call inet_ntop to return the presentation format of the IP address; otherwise gethostbyaddr searches for the hostname corresponding to the IP address. If gethostbyaddr succeeds and the NI_NOFQDN (no fully qualified domain name) flag is specified, the hostname is terminated at the first period in the name.

Handle failure of gethostbyaddr

24-29 If gethostbyaddr fails (which, unfortunately is all too common given the number of misconfigured DNS servers on the Internet; see Section 14.8 of TCPv3) and the NI_NAMEREQD flag was specified, an error is returned. Otherwise the address string corresponding to the IP address is formed by inet_ntop.

Return service string

32-42 If the NI_NUMERICSERV flag is specified, just the decimal port number is returned. Otherwise getservbyport is called. The final argument is a null pointer unless the NI_DGRAM flag is specified. If getservbyport fails, the decimal port number is returned instead.

libgai/gn_ipv46.c

```
 5 int
 6 gn_ipv46(char *host, size_t hostlen, char *serv, size_t servlen,
 7           void *aptr, size_t alen, int family, int port, int flags)
 8 {
 9     char   *ptr;
10     struct hostent *hptr;
11     struct servent *sptr;

12     if (hostlen > 0) {
13         if (flags & NI_NUMERICHOST) {
14             if (inet_ntop(family, aptr, host, hostlen) == NULL)
15                 return (1);
16         } else {
17             hptr = gethostbyaddr(aptr, alen, family);
18             if (hptr != NULL && hptr->h_name != NULL) {
19                 if (flags & NI_NOFQDN) {
20                     if ( (ptr = strchr(hptr->h_name, '.')) != NULL)
21                         *ptr = 0;   /* overwrite first dot */
22                 }
23                 snprintf(host, hostlen, "%s", hptr->h_name);
24             } else {
25                 if (flags & NI_NAMEREQD)
26                     return (1);
27                 if (inet_ntop(family, aptr, host, hostlen) == NULL)
28                     return (1);
29             }
30         }
31     }
32     if (servlen > 0) {
33         if (flags & NI_NUMERICSERV) {
34             snprintf(serv, servlen, "%d", ntohs(port));
35         } else {
36             sptr = getservbyport(port, (flags & NI_DGRAM) ? "udp" : NULL);
37             if (sptr != NULL && sptr->s_name != NULL)
38                 snprintf(serv, servlen, "%s", sptr->s_name);
39             else
40                 snprintf(serv, servlen, "%d", ntohs(port));
41         }
42     }
43     return (0);
44 }
```

libgai/gn_ipv46.c

Figure 11.42 `gn_ipv46` function: handle IPv4 and IPv6 socket address structures.

11.17 Summary

`getaddrinfo` is a useful function that lets us write protocol-independent code. But calling it directly takes a few steps, and there are still repetitive details that must be handled for different scenarios: go through all the returned structures, ignore error returns from `socket`, set the `SO_REUSEADDR` socket option for TCP servers, and the like. We simplify all these details with our five functions `tcp_connect`, `tcp_listen`,

`udp_client`, `udp_connect`, and `udp_server`. We showed the use of these functions in writing protocol-independent versions of our TCP and UDP daytime clients and daytime servers.

`gethostbyname` and `gethostbyaddr` are also examples of functions that are not normally reentrant. The two functions share a static result structure to which both return a pointer. We encounter this problem of reentrancy again with threads in Chapter 23 and discuss ways around the problem. We discussed the `_r` versions of these two functions that some vendors provide, which is one solution, but it requires a change in all the applications that call the functions.

Exercises

11.1 In Figure 11.8 the caller must pass a pointer to an integer to obtain the size of the protocol address. If the caller does not do this (i.e., passes a null pointer as the final argument), how can the caller still obtain the actual size of the protocol's addresses?

11.2 Modify Figure 11.10 to call `getnameinfo` instead of `sock_ntop`. What flags should you pass to `getnameinfo`?

11.3 In Section 7.5 we discussed port stealing with the `SO_REUSEADDR` socket option. To see how this works, build the protocol-independent UDP daytime server in Figure 11.15. Start one instance of the server in one window, binding the wildcard address and some port of your choosing. Start a client in another window and verify that this server is handling the client (note the `printf` in the server). Then start another instance of the server in another window, this time binding one of the host's unicast addresses and the same port as the first server. What problem do you immediately encounter? Fix this problem and restart this second server. Start a client, send a datagram, and verify that the second server has stolen the port from the first server. If possible, start the second server again, from a different login account from the first server, to see if the stealing still succeeds, because some vendors will not allow the second bind unless the user ID is the same as that of the process that has already bound the port.

11.4 When discussing Figure 11.34 we noted that the address of `aipnext` is an argument to the `ga_aistruct` function, necessitating an extra level of indirection when referencing the variable. Why do we not make `aipnext` a global variable, instead of passing its address as an argument?

11.5 In our discussion of Unix domain at the end of Section 11.5 we mentioned that none of the IANA service names begin with a slash. Do any of these service names contain a slash?

11.6 At the end of Section 2.10 we showed two `telnet` examples: to the `daytime` server and to the `echo` server . Knowing that a client goes through the two steps `gethostbyname` and `connect`, which lines output by the client indicate which steps?

11.7 `gethostbyaddr` can take a long time (up to 80 seconds) to return an error if a hostname cannot be found for an IP address. Write a new function named `getnameinfo_timeo` that takes an additional integer argument specifying the maximum number of seconds to wait for a reply. If the timer expires and the `NI_NAMEREQD` flag is not specified, just call `inet_ntop` and return an address string.

12

Daemon Processes and `inetd` Superserver

12.1 Introduction

A *daemon* is a process that runs in the background and is independent of control from all terminals. Unix systems typically have many processes that are daemons (on the order or 20 to 50), running in the background, performing different administrative tasks.

The reason for wanting independence from all terminals is in case the daemon is started from a terminal (as opposed to starting from an initialization script). We want to be able to use that terminal for other tasks at a later time. For example, if we start the daemon from a terminal, log off the terminal, and someone else logs in on the terminal, we do not want any daemon error messages appearing during the next user's terminal session. Similarly, signals generated from terminal keys (e.g., the interrupt signal) must not affect any daemons that were started from that terminal earlier. While it is easy to run our server in the background (by ending the shell command line with an ampersand), we should have our program put itself in the background automatically and we also need to make it independent of any terminal.

There are numerous ways to start a daemon:

1. During system startup many daemons are started by the system initialization scripts. These scripts are often in the directory `/etc` or in a directory whose name begins with `/etc/rc`, but their location and contents are implementation dependent. Daemons started by these scripts begin with superuser privileges.

   A few network servers are often started from these scripts: the `inetd` superserver (our next item), a Web server, and a mail server (often `sendmail`). The `syslogd` daemon that we describe in Section 12.2 is normally started by one of these scripts.

2. Many network servers are started by the inetd superserver, which we describe later in this chapter. inetd itself is started from one of the scripts in step 1. inetd listens for network requests (Telnet, FTP, etc.) and when a request arrives, it invokes the actual server (Telnet server, FTP server, etc.).
3. The execution of programs on a regular basis is performed by the cron daemon, and programs that it invokes run as daemons. The cron daemon itself is started in step 1 during system startup.
4. The execution of a program at one time in the future is specified by the at command. The cron daemon normally initiates these programs when their time arrives, so these programs run as daemons.
5. Daemons can be started from user terminals, either in the foreground or in the background. This is often done when testing a daemon, or restarting a daemon that was terminated for some reason.

Since a daemon does not have a controlling terminal, it needs some way to output messages when something happens, either normal informational messages, or emergency messages that need to be handled by an administrator. The syslog function is the standard way to output these messages, and it sends the messages to the syslogd daemon.

12.2 syslogd Daemon

Unix systems normally start a daemon named syslogd from one of the system initializations scripts, and it runs as long as the system is up. Berkeley-derived implementations of syslogd perform the following actions upon startup:

1. The configuration file, normally /etc/syslog.conf, is read, specifying what to do with each type of log message that the daemon can receive. These messages can be appended to a file (a special case of which is the file /dev/console, which writes the message to the console), written to a specific user (if that user is logged in), or forwarded to the syslogd daemon on another host.
2. A Unix domain socket is created and bound to the pathname /var/run/log (/dev/log on some systems).
3. A UDP socket is created and bound to port 514 (the syslog service).
4. The pathname /dev/klog is opened. Any error messages from within the kernel appear as input on this device.

The syslogd daemon then runs in an infinite loop that calls select, waiting for any one of its three descriptors (from steps 2, 3, and 4) to be readable, reads the log message, and does what the configuration file says to do with that message. If the daemon receives the SIGHUP signal, it rereads its configuration file.

We could send log messages to the `syslogd` daemon from our daemons by creating a Unix domain datagram socket and sending our messages to the pathname that the daemon has bound, but an easier interface is the `syslog` function that we describe in the next section. Alternately we could create a UDP socket and send our log messages to the loopback address and port 514.

Newer implementations disable the creation of the UDP socket unless specified by the administrator, as allowing anyone to send UDP datagrams to this port (possibly filling its socket receive buffer) might prevent legitimate log messages from being received.

Differences exist between the various implementations of `syslogd`. For example, Unix domain sockets are used by Berkeley-derived implementations but System V implementations use a streams log driver. Different Berkeley-derived implementations use different pathnames for the Unix domain socket. We can ignore all these details if we use the `syslog` function.

12.3 `syslog` Function

Since a daemon does not have a controlling terminal, it cannot just `fprintf` to `stderr`. The common technique for logging messages from a daemon is to call the `syslog` function.

```
#include <syslog.h>

void syslog(int priority, const char *message, ... );
```

Although this function was originally developed for BSD systems, it is provided by most Unix vendors today. Posix says nothing about `syslog` but it is required by Unix 98.

The *priority* argument is a combination of a *level* and a *facility*, which we show in Figures 12.1 and 12.2. The *message* is like a format string to `printf`, with the addition of a `%m` specification, which is replaced with the error message corresponding to the current value of `errno`. A newline can appear at the end of the *message* but is not mandatory.

Log messages have a *level* between 0 and 7, which we show in Figure 12.1. These are ordered values. If no *level* is specified by the sender, `LOG_NOTICE` is the default.

level	Value	Description
LOG_EMERG	0	system is unusable (highest priority)
LOG_ALERT	1	action must be taken immediately
LOG_CRIT	2	critical conditions
LOG_ERR	3	error conditions
LOG_WARNING	4	warning conditions
LOG_NOTICE	5	normal but significant condition (default)
LOG_INFO	6	informational
LOG_DEBUG	7	debug-level messages (lowest priority)

Figure 12.1 *level* of log messages.

Log messages also contain a *facility* to identify the type of process sending the message. We show the different values in Figure 12.2. If no *facility* is specified, LOG_USER is the default.

facility	Description
LOG_AUTH	security/authorization messages
LOG_AUTHPRIV	security/authorization messages (private)
LOG_CRON	cron daemon
LOG_DAEMON	system daemons
LOG_FTP	FTP daemon
LOG_KERN	kernel messages
LOG_LOCAL0	local use
LOG_LOCAL1	local use
LOG_LOCAL2	local use
LOG_LOCAL3	local use
LOG_LOCAL4	local use
LOG_LOCAL5	local use
LOG_LOCAL6	local use
LOG_LOCAL7	local use
LOG_LPR	line printer system
LOG_MAIL	mail system
LOG_NEWS	network news system
LOG_SYSLOG	messages generated internally by syslogd
LOG_USER	random user-level messages (default)
LOG_UUCP	UUCP system

Figure 12.2 *facility* of log messages.

For example, the following call could be issued by a daemon when a call to the rename function unexpectedly fails:

```
syslog(LOG_INFO|LOG_LOCAL2, "rename(%s, %s): %m", file1, file2);
```

The purpose of the *facility* and *level* is to allow all messages from a given facility to be handled the same in the /etc/syslog.conf file, or to allow all messages of a given level to be handled the same. For example, the configuration file could contain the lines

```
kern.*              /dev/console
local7.debug        /var/log/cisco.log
```

to specify that all kernel messages get logged to the console, and all debug messages from the local7 facility get appended to the file /var/log/cisco.log.

When the application calls syslog the first time, it creates a Unix domain datagram socket and then calls connect to the well-known pathname of the socket created by the syslogd daemon (e.g., /var/run/log). This socket remains open until the process terminates. Alternately, the process can call openlog and closelog.

```
#include <syslog.h>

void openlog(const char *ident, int options, int facility);

void closelog(void);
```

`openlog` can be called before the first call to `syslog` and `closelog` can be called when the application is finished sending log messages.

ident is a string that will be prepended to each log message by `syslog`. Often this is the program name.

The *options* argument is formed as the logical OR of one or more of the constants in Figure 12.3.

options	Description
`LOG_CONS`	log to console if cannot send to `syslogd` daemon
`LOG_NDELAY`	do not delay open, create socket now
`LOG_PERROR`	log to standard error as well as sending to `syslogd` daemon
`LOG_PID`	log the process ID with each message

Figure 12.3 *options* for `openlog`.

Normally the Unix domain socket is not created when `openlog` is called. Instead it will be opened upon the first call to `syslog`. The `LOG_NDELAY` option causes the socket to be created when `openlog` is called.

The *facility* argument to `openlog` specifies a default facility for any subsequent calls to `syslog` that do not specify a facility. Some daemons call `openlog` and specify the facility (which normally does not change for a given daemon) and then specify only the *level* in each call to `syslog` (since the *level* can change depending on the error).

Log messages can also be generated by the `logger` command. This can be used from within shell scripts, for example, to send messages to `syslogd`.

12.4 `daemon_init` Function

Figure 12.4 shows a function named `daemon_init` that we can call (normally from a server) to daemonize the process.

`fork`

10-11 We first call `fork` and then the parent terminates and the child continues. If the process was started as a shell command in the foreground, when the parent terminates the shell thinks the command is done. This automatically runs the child process in the background. Also, the child inherits the process group ID from the parent but gets its own process ID. This guarantees that the child is not a process group leader, which is required for the next call to `setsid`.

`setsid`

12-13 `setsid` is a Posix.1 function that creates a new session. (Chapter 9 of APUE talks about process relationships and sessions in detail.) The process becomes the session leader of the new session, becomes the process group leader of a new process group, and has no controlling terminal.

Ignore `SIGHUP` and `fork` again

14-16 We ignore the `SIGHUP` signal and call `fork` again. When this function returns, the parent is really the first child, and it terminates, leaving the second child running. The

daemon_init.c

```
 1 #include    "unp.h"
 2 #include    <syslog.h>

 3 #define MAXFD   64

 4 extern int daemon_proc;          /* defined in error.c */

 5 void
 6 daemon_init(const char *pname, int facility)
 7 {
 8     int     i;
 9     pid_t   pid;

10     if ( (pid = Fork()) != 0)
11         exit(0);                 /* parent terminates */

12         /* 1st child continues */
13     setsid();                    /* become session leader */

14     Signal(SIGHUP, SIG_IGN);
15     if ( (pid = Fork()) != 0)
16         exit(0);                 /* 1st child terminates */

17         /* 2nd child continues */
18     daemon_proc = 1;             /* for our err_XXX() functions */

19     chdir("/");                  /* change working directory */

20     umask(0);                    /* clear our file mode creation mask */

21     for (i = 0; i < MAXFD; i++)
22         close(i);

23     openlog(pname, LOG_PID, facility);
24 }
```

daemon_init.c

Figure 12.4 daemon_init function: daemonize the process.

purpose of this second fork is to guarantee that the daemon cannot automatically acquire a controlling terminal should it open a terminal device in the future. Under SVR4, when a session leader without a controlling terminal opens a terminal device (that is not currently some other session's controlling terminal), the terminal becomes the controlling terminal of the session leader. But by calling fork a second time we guarantee that the second child is no longer a session leader, so it cannot acquire a controlling terminal. We must ignore SIGHUP because when the session leader terminates (the first child), all processes in the session (our second child) are sent the SIGHUP signal.

Set flag for error functions

17-18 We set the global daemon_proc nonzero. This external is defined by our err_XXX functions (Section D.4) and when its value is nonzero, this tells them to call

`syslog` instead of doing an `fprintf` to standard error. This saves us from having to go through all our code and call one of our error functions if the server is not being run as a daemon (i.e., when we are testing the server) but call `syslog` if it is being run as a daemon.

Change working directory and clear file mode creation mask

19–20 We change the working directory to the root directory, although some daemons might have a reason to change to some other directory. For example, a printer daemon might change to the printer's spool directory, where it does all its work. Should the daemon ever generate a `core` file, that file is generated in the current working directory. Another reason to change the working directory is that the daemon could have been started in any filesystem, and if it remains there, that filesystem cannot be unmounted. The file mode creation mask is reset to 0 so that if the daemon creates its own files, permission bits in the inherited file mode creation mask do not affect the permission bits of the new files.

Close any open descriptors

21–22 We close any open descriptors that are inherited from the process that executed the daemon (normally a shell). The problem is determining the highest descriptor in use: there is no Unix function that provides this value. There are ways to determine the maximum number of descriptors that the process can open, but even this gets complicated (see p. 43 of APUE) because the limit can be infinite. Our solution is to close the first 64 descriptors, even though most of these are probably not open.

Some daemons `open /dev/null` for reading and writing and duplicate the descriptor to standard input, standard output, and standard error. This guarantees that these common descriptors are open, and a read from any of these descriptors returns 0 (end-of-file) and the kernel just discards anything written to any of these three descriptors. The reason for opening these descriptors is so that any library function called by the daemon that assumes it can read from standard input, or write to either standard output or standard error, will not fail. Alternately, some daemons open a log file that they will write to while running and duplicate its descriptor to standard output and standard error.

Use `syslogd` for errors

23 `openlog` is called. The first argument is from the caller and is normally the name of the program (e.g., `argv[0]`). We specify that the process ID should be added to each log message. The *facility* is also specified by the caller, as one of the values from Figure 12.2, or 0 if the default of `LOG_USER` is OK.

We note that since a daemon runs without a controlling terminal it should never receive the `SIGHUP` signal from the kernel. Therefore many daemons use this signal as a notification from the administrator that the daemon's configuration file has changed, and the daemon should reread the file. Two other signals that a daemon should never receive are `SIGINT` and `SIGWINCH`, and these can also be used to notify a daemon of some change.

Example: Daytime Server as a Daemon

Figure 12.5 is a modification of our protocol-independent daytime server from Figure 11.10 that calls our daemon_init function to run as a daemon.

inetd/daytimetcpsrv2.c

```
 1 #include    "unp.h"
 2 #include    <time.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     listenfd, connfd;
 7     socklen_t addrlen, len;
 8     struct sockaddr *cliaddr;
 9     char    buff[MAXLINE];
10     time_t  ticks;

11     daemon_init(argv[0], 0);

12     if (argc == 2)
13         listenfd = Tcp_listen(NULL, argv[1], &addrlen);
14     else if (argc == 3)
15         listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
16     else
17         err_quit("usage: daytimetcpsrv2 [ <host> ] <service or port>");

18     cliaddr = Malloc(addrlen);

19     for ( ; ; ) {
20         len = addrlen;
21         connfd = Accept(listenfd, cliaddr, &len);
22         err_msg("connection from %s", Sock_ntop(cliaddr, len));

23         ticks = time(NULL);
24         snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
25         Write(connfd, buff, strlen(buff));

26         Close(connfd);
27     }
28 }
```

inetd/daytimetcpsrv2.c

Figure 12.5 Protocol-independent daytime server that runs as a daemon.

There are only two changes: we call our daemon_init function as soon as the program starts, and we call our err_msg function, instead of printf, to print the client's IP address and port. Indeed, if we want our programs to be able to run as a daemon, we must avoid calling the printf and fprintf functions and use our err_msg function instead.

If we run this program on our host solaris and then check the /var/adm/messages file (where we send all LOG_USER messages) after connecting from our host bsdi, we have:

```
Jun  4 15:15:33 solaris.kohala.com daytimetcpsrv2[14882]:
connection from ::ffff:206.62.226.35.3356
```

(We have wrapped the one long line.) The date, time, and FQDN are prefixed automatically by the `syslogd` daemon.

12.5 `inetd` Daemon

On a typical Unix system there could be many servers in existence, just waiting for a client request to arrive. Examples are FTP, Telnet, Rlogin, TFTP, and so on. With systems before 4.3BSD, each of these services had a process associated with it. This process was started at boot time from the file `/etc/rc`, and each process did nearly identical startup tasks: create a socket, `bind` the server's well-known port to the socket, wait for a connection (if TCP) or a datagram (if UDP), and then `fork`. The child process serviced the client and the parent waited for the next client request. There are two problems with this model.

1. All these daemons contained nearly identical startup code, first with respect to socket creation, and also with respect to becoming a daemon process (similar to our `daemon_init` function).
2. Each daemon took a slot in the process table, but each daemon was asleep most of the time.

The 4.3BSD release simplified this by providing an Internet *superserver*: the `inetd` daemon. This daemon can be used by servers that use either TCP or UDP. It does not handle other protocols, such as Unix domain sockets. This daemon fixes the two problems just mentioned.

1. It simplifies writing daemon processes, since most of the startup details are handled by `inetd`. This obviates the need for each server to call our `daemon_init` function.
2. It allows a single process (`inetd`) to be waiting for incoming client requests for multiple services, instead of one process for each service. This reduces the total number of processes in the system.

The `inetd` process establishes itself as a daemon using the techniques that we described with our `daemon_init` function. It then reads and processes its configuration file, typically `/etc/inetd.conf`. This file specifies the services that the superserver is to handle, and what to do when a service request arrives. Each line contains the fields shown in Figure 12.6. Some sample lines are:

```
ftp     stream  tcp  nowait  root    /usr/bin/ftpd     ftpd -l
telnet  stream  tcp  nowait  root    /usr/bin/telnetd  telnetd
login   stream  tcp  nowait  root    /usr/bin/rlogind  rlogind -s
tftp    dgram   udp  wait    nobody  /usr/bin/tftpd    tftpd -s /tftpboot
```

The actual name of the server is always passed as the first argument to a program when it is execed.

Field	Description
service-name	must be in /etc/services
socket-type	stream (TCP) or dgram (UDP)
protocol	must be in /etc/protocols: either tcp or udp
wait-flag	normally nowait for TCP or wait for UDP
login-name	from /etc/passwd: typically root
server-program	full pathname to exec
server-program-arguments	arguments for exec

Figure 12.6 Fields in inetd.conf file.

This figure and the sample lines are just examples. Most vendors have added their own features to inetd. Examples are the ability to handle remote procedure call (RPC) servers, in addition to TCP and UDP servers, and the ability to handle protocols other than TCP and UDP. Also, the pathname to exec and the command-line arguments to the server obviously depend on the implementation.

The interaction of IPv6 with /etc/inetd.conf depends on the vendor. Some use a *protocol* of tcp6 or udp6 to indicate that an IPv6 socket should be created for the server.

A picture of what the inetd daemon does is shown in Figure 12.7.

1. On startup it reads the /etc/inetd.conf file and creates a socket of the appropriate type (stream or datagram) for all the services specified in the file. The maximum number of servers that inetd can handle depends on the maximum number of descriptors that inetd can create. Each new socket is added to a descriptor set that will be used in a call to select.
2. bind is called for the socket, specifying the well-known port for the server and the wildcard IP address. This TCP or UDP port number is obtained by calling getservbyname with the *service-name* and the *protocol* fields from the configuration file as arguments.
3. For TCP sockets, listen is called, so that incoming connection requests are accepted. This step is not done for datagram sockets.
4. After all the sockets are created, select is called to wait for any of the sockets to become readable. Recall from Section 6.3 that a listening TCP socket becomes readable when a new connection is ready to be accepted and a UDP socket becomes readable when a datagram arrives. inetd spends most of its time blocked in this call to select, waiting for a socket to be readable.
5. When select returns that a socket is readable, if the socket is a TCP socket, accept is called to accept the new connection.
6. The inetd daemon forks and the child process handles the service request. This is similar to a standard concurrent server (Section 4.8).

   The child closes all descriptors other than the socket descriptor that it is handling: the new connected socket returned by accept for a TCP server, or the original UDP socket.

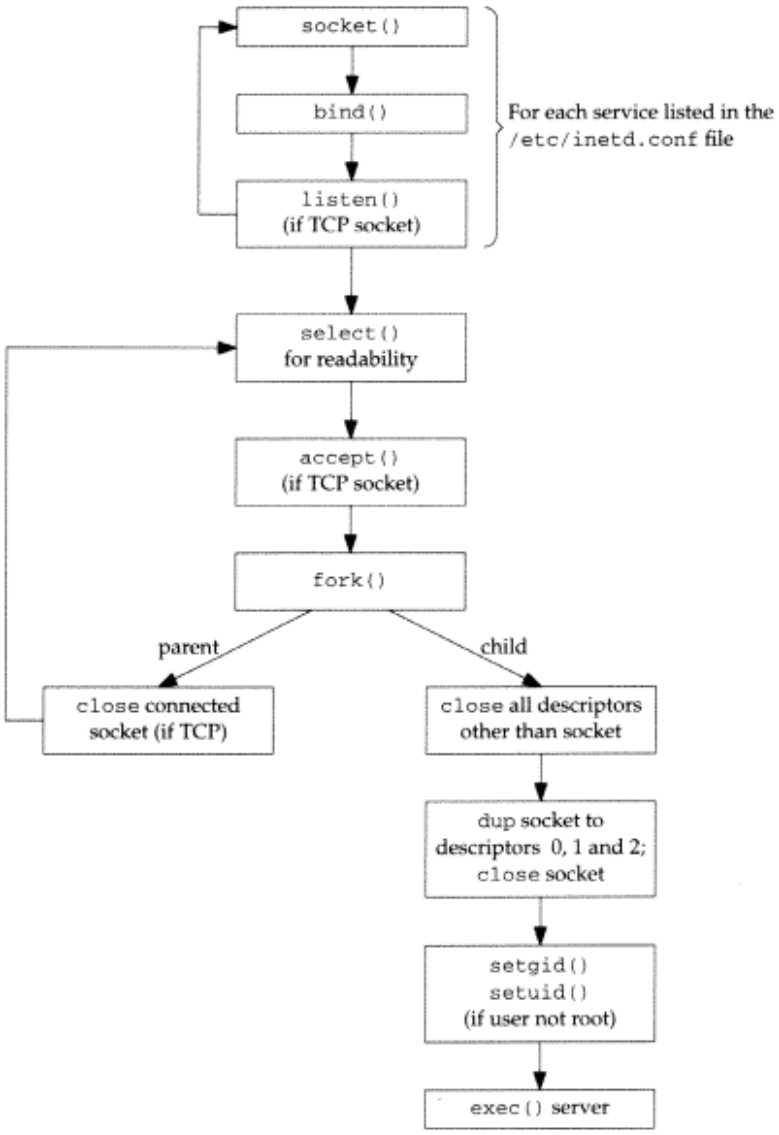

Figure 12.7 Steps performed by `inetd`.

The child calls `dup2` three times, duplicating the socket onto descriptors 0, 1, and 2 (standard input, standard output, and standard error). The original socket descriptor is then closed. By doing this, the only descriptors that are open in the child are 0, 1, and 2. If the child reads from standard input, it is reading from the socket and anything it writes to standard output or standard error is written to the socket.

The child calls getpwnam to get the password file entry for the *login-name* specified in the configuration file. If this field is not root, then the child becomes the specified user by executing the setgid and setuid function calls. (Since the inetd process is executing with a user ID of 0, the child process inherits this user ID across the fork, so it is able to become any user that it chooses.)

The child process now does an exec to execute the appropriate *server-program* to handle the request, passing the arguments specified in the configuration file.

7. If the socket is a stream socket, the parent process must close the connected socket (like our standard concurrent server). The parent calls select again, waiting for the next socket to become readable.

If we look in more detail at the descriptor handling that is taking place, Figure 12.8 shows the descriptors in inetd when a new connection request arrives from an FTP client.

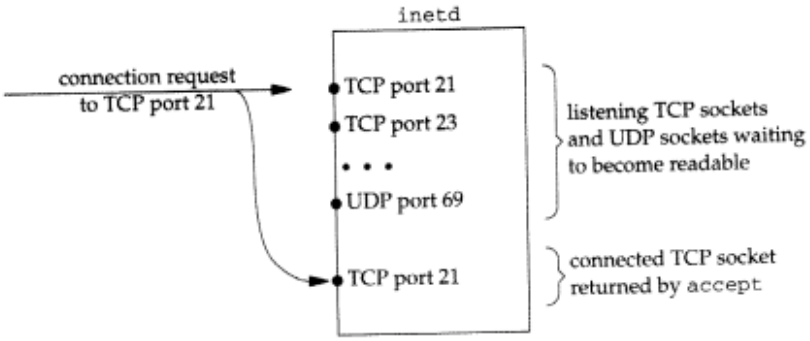

Figure 12.8 inetd descriptors when connection request arrives for TCP port 21.

The connection request is directed to TCP port 21, but a new connected socket is created by accept.

Figure 12.9 shows the descriptors in the child, after the call to fork, after the child has closed all the descriptors other than the connected socket.

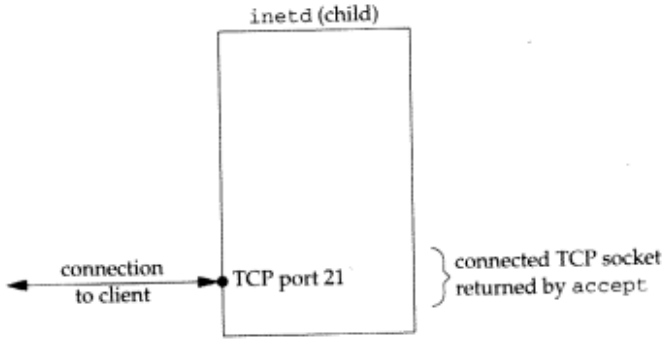

Figure 12.9 inetd descriptors in child.

The next step is for the child to duplicate the connected socket to descriptors 0, 1, and 2 and then close the connected socket. This gives us the descriptors shown in Figure 12.10.

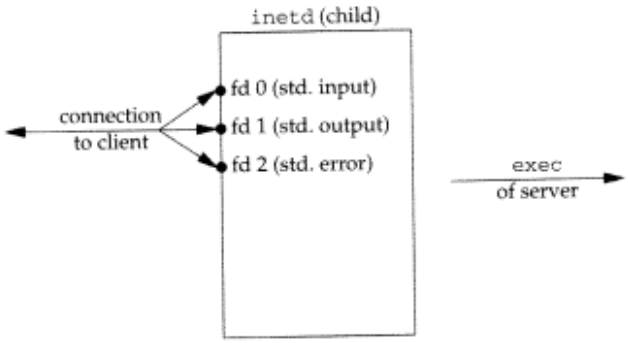

Figure 12.10 inetd descriptors after dup2.

The child then calls exec, and recall from Section 4.7 that all descriptors normally remain open across an exec, so the real server that is execed uses any of the descriptors 0, 1, or 2 to communicate with the client. These should be the only descriptors open in the server.

The scenario we have described handles the case where the configuration file specifies nowait for the server. This is typical for all TCP services and it means that inetd need not wait for its child to terminate before accepting another connection for that service. If another connection request arrives for the same service, it is returned to the parent process as soon as it calls select again. Steps 4, 5, and 6 listed earlier are executed again, and another child process handles this new request.

Specifying the wait flag for a datagram service changes the steps done by the parent process. This flag says that inetd must wait for its child to terminate before selecting on this UDP socket again. The following changes occur:

1. When fork returns in the parent, the parent saves the process ID of the child. This allows the parent to know when this specific child process terminates, by looking at the value returned by waitpid.
2. The parent disables the socket from future selects by using the FD_CLR macro to turn off the bit in its descriptor set. This means that the child process takes over the socket until it terminates.
3. When the child terminates, the parent is notified by a SIGCHLD signal, and the parent's signal handler obtains the process ID of the terminating child. It reenables select for the corresponding socket by turning on the bit in its descriptor set for this socket.

The reason that a datagram server must take over the socket until it terminates, preventing inetd from selecting on that socket for readability (awaiting another client datagram) is because there is only one socket for a datagram server, unlike a TCP server

that has a listening socket and one connected socket per client. If inetd did not turn off readability for the datagram socket, and say the parent (inetd) executed before the child, then the datagram from the client would still be in the socket receive buffer, causing select to return readable again, causing inetd to fork another (unneeded) child. inetd must ignore the datagram socket until it knows that the child has read the datagram from the socket receive queue. The way that inetd knows when that child is finished with the socket is by receiving SIGCHLD, indicating that the child has terminated. We show an example of this in Section 20.7.

The five standard Internet services that we described in Figure 2.13 are handled internally by inetd. (See Exercise 12.2.)

Since inetd is the process that calls accept for a TCP server, the actual server that is invoked by inetd normally calls getpeername to obtain the IP address and port number of the client. Recall Figure 4.18 where we showed that after a fork and an exec (which is what inetd does) the only way for the actual server to obtain the identify of the client is to call getpeername.

inetd is normally not used for high-volume servers, notably mail and Web servers. sendmail, for example, is normally run as a standard concurrent server, as we described in Section 4.8. In this mode the process control cost for each client connection is just a fork, while the cost for a TCP server invoked by inetd is a fork and an exec. Web servers use a variety of techniques to minimize the process control overhead for each client connection, as we discuss in Chapter 27.

12.6 daemon_inetd Function

Figure 12.11 shows a function named daemon_inetd that we can call from a server that we know is invoked by inetd.

daemon_inetd.c

```
1 #include    "unp.h"
2 #include    <syslog.h>

3 extern int daemon_proc;          /* defined in error.c */

4 void
5 daemon_inetd(const char *pname, int facility)
6 {
7     daemon_proc = 1;             /* for our err_XXX() functions */
8     openlog(pname, LOG_PID, facility);
9 }
```

daemon_inetd.c

Figure 12.11 daemon_inetd function: daemonize process run by inetd.

This function is trivial, compared to daemon_init, because all of the daemonization steps are performed by inetd when it starts. All that we do is set the daemon_proc flag for our error functions (Figure D.4) and call openlog, with the same arguments as the call in Figure 12.4.

Example: Daytime Server as a Daemon Invoked by `inetd`

Figure 12.12 is a modification of our daytime server from Figure 12.5 that can be invoked by `inetd`.

inetd/daytimetcpsrv3.c

```
 1 #include    "unp.h"
 2 #include    <time.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     socklen_t len;
 7     struct sockaddr *cliaddr;
 8     char    buff[MAXLINE];
 9     time_t  ticks;

10     daemon_inetd(argv[0], 0);

11     cliaddr = Malloc(MAXSOCKADDR);
12     len = MAXSOCKADDR;
13     Getpeername(0, cliaddr, &len);
14     err_msg("connection from %s", Sock_ntop(cliaddr, len));

15     ticks = time(NULL);
16     snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
17     Write(0, buff, strlen(buff));

18     Close(0);                    /* close TCP connection */
19     exit(0);
20 }
```

inetd/daytimetcpsrv3.c

Figure 12.12 Protocol-independent daytime server that can be invoked by `inetd`.

There are two major changes in this program. First, all the socket creation code is gone: the calls to `tcp_listen` and to `accept`. Those steps are done by `inetd` and we reference the TCP connection using descriptor 0 (standard input). Second, the infinite `for` loop is gone, because we are invoked once per client connection. After servicing this client we terminate.

Call `getpeername`

11-14 Since we do not call `tcp_listen`, we do not know the size of the socket address structure that it returns, and since we do not call `accept`, we do not know the client's protocol address. Therefore we allocate a buffer for the socket address structure using our `MAXSOCKADDR` constant and call `getpeername` with descriptor 0 as the first argument.

To run this example on our BSD/OS system, we first assign the service a name and port, adding the following line to `/etc/services`:

```
mydaytime       9999/tcp
```

We then add a line to `/etc/inetd.conf`:

```
mydaytime  stream  tcp  nowait  rstevens
      /usr/home/rstevens/daytimetcpsrv3  daytimetcpsrv3
```

(We have wrapped the long line.) We place the executable in the specified file and send the SIGHUP signal to inetd, telling it to reread its configuration file. The next step is to execute netstat to verify that a listening socket has been created on TCP port 9999:

```
bsdi % netstat -na | grep 9999
tcp        0    0      *.9999              *.*                    LISTEN
```

We then invoke the server from another host:

```
alpha % telnet bsdi 9999
Trying 206.62.226.35...
Connected to bsdi.
Escape character is '^]'.
Thu Jun  5 11:13:50 1997
Connection closed by foreign host.
```

The /var/log/messages file (where we have directed the LOG_USER facility messages to be logged in our /etc/syslog.conf file) contains the entry

```
Jun  5 11:13:50 bsdi daytimetcpsrv3[28724]: connection from 206.62.226.42.1042
```

12.7 Summary

Daemons are processes that run in the background independent of control from all terminals. Many network servers run as daemons. All output from a daemon is normally sent to the syslogd daemon by calling the syslog function. The administrator then has complete control over what happens to these messages, based on the daemon that sent the message and the severity of the message.

To start an arbitrary program and have it run as a daemon requires a few steps: call fork to run in the background, call setsid to create a new Posix.1 session and become the session leader, fork again to avoid obtaining a new controlling terminal, change the working directory and the file mode creation mask, and close all unneeded files. Our daemon_init function handles all these details.

Many Unix servers are started by the inetd daemon. It handles all of the required daemonization steps and when the actual server is started, the socket is open on standard input, standard output, and standard error. This lets us omit the calls to socket, bind, listen, and accept, since all these steps are handled by inetd.

Exercises

12.1 What happens in Figure 12.5 if we wait to call daemon_init until the command-line arguments have been processed, so that the call to err_quit appears before the program becomes a daemon?

12.2 For the five services handled internally by `inetd` (Figure 2.13), considering the TCP version and the UDP version of each service, which of the 10 servers do you think are implemented with a call to `fork`, and which do not require a `fork`?

12.3 What happens if we create a UDP socket, bind port 7 to the socket (the standard `echo` server in Figure 2.13), and send a UDP datagram to a `chargen` server?

12.4 The Solaris 2.x manual page for `inetd` describes a `-t` flag that causes `inetd` to call `syslog` (with a facility of `LOG_DAEMON` and a level of `LOG_NOTICE`) to log the client's IP address and port for any TCP service that `inetd` handles. How does `inetd` obtain this information?

This manual page also says that `inetd` cannot do this for a UDP service. Why?

Is there a way around this limitation for UDP services?

13

Advanced I/O Functions

13.1 Introduction

This chapter covers a variety of functions and techniques that we lump into the category of "advanced I/O." First is setting a timeout on an I/O operation, which can be done in three different ways.

Next are three more variations on the `read` and `write` functions: `recv` and `send`, which allow a fourth argument that contains flags from the process to the kernel, `readv` and `writev`, which lets us specify a vector of buffers to input into or output from, and `recvmsg` and `sendmsg`, which combine all the features from the other I/O functions along with the new capability of receiving and sending ancillary data.

We also consider how to determine how much data is in the socket receive buffer and how to use the C standard I/O library with sockets. We finish the chapter with a brief look at T/TCP, TCP for Transactions, which can avoid the three-way handshake.

13.2 Socket Timeouts

There are three ways to place a timeout on an I/O operation involving a socket.

1. Call `alarm`, which generates the `SIGALRM` signal when the specified time has expired. This involves signal handling, which can differ from one implementation to the next, and it may interfere with other existing calls to `alarm` in the process.
2. Block waiting for I/O in `select`, which has a time limit built in, instead of blocking in a call to `read` or `write`.

3. Use the newer SO_RCVTIMEO and SO_SNDTIMEO socket options. The problem with this approach is that not all implementations support these two socket options.

All three techniques work with input and output operations (e.g., read, write, and the other variations such as recvfrom and sendto) but we would also like a technique that we can use with connect, since a TCP connect can take a long time to time out (typically 75 seconds). select can be used to place a timeout on connect only when the socket is in a nonblocking mode (which we show in Section 15.3), and the two socket options do not work with connect. We also note that the first two techniques work with any descriptor while the third technique works only with socket descriptors.

We now show examples of all three techniques.

connect with a Timeout Using SIGALRM

Figure 13.1 shows our function connect_timeo that calls connect with an upper limit specified by the caller. The first three arguments are the three required by connect and the fourth argument is the number of seconds to wait.

lib/connect_timeo.c

```
 1 #include    "unp.h"

 2 static void connect_alarm(int);

 3 int
 4 connect_timeo(int sockfd, const SA *saptr, socklen_t salen, int nsec)
 5 {
 6     Sigfunc *sigfunc;
 7     int     n;

 8     sigfunc = Signal(SIGALRM, connect_alarm);
 9     if (alarm(nsec) != 0)
10         err_msg("connect_timeo: alarm was already set");

11     if ( (n = connect(sockfd, saptr, salen)) < 0) {
12         close(sockfd);
13         if (errno == EINTR)
14             errno = ETIMEDOUT;
15     }
16     alarm(0);                   /* turn off the alarm */
17     Signal(SIGALRM, sigfunc);   /* restore previous signal handler */

18     return (n);
19 }

20 static void
21 connect_alarm(int signo)
22 {
23     return;                     /* just interrupt the connect() */
24 }
```

lib/connect_timeo.c

Figure 13.1 connect with a timeout.

Establish signal handler

8 A signal handler is established for SIGALRM. The current signal handler (if any) is saved, so we can restore it at the end of the function.

Set alarm

9-10 The alarm clock for the process is set to the number of seconds specified by the caller. The return value from alarm is the number of seconds currently remaining in the alarm clock for the process (if one has already been set by the process) or 0 (if there is no current alarm). In the former case we print a warning message, since we are wiping out that previously set alarm. (See Exercise 13.2.)

Call connect

11-15 connect is called and if the function is interrupted (EINTR), we set the errno value to ETIMEDOUT instead. The socket is closed to prevent the three-way handshake from continuing.

Turn off alarm and restore any previous signal handler

16-18 The alarm is turned off by setting it to 0 and the previous signal handler (if any) is restored.

Handle SIGALRM

20-24 The signal handler just returns, assuming this return will interrupt the pending connect, causing connect to return an error of EINTR. Recall our signal function (Figure 5.6) that does not set the SA_RESTART flag when the signal being caught is SIGALRM.

One point to make with this example is that we can always reduce the timeout period for a connect using this technique, but we cannot extend the kernel's existing timeout. That is, on a Berkeley-derived kernel the timeout for a connect is normally 75 seconds. We can specify a smaller value for our function, say 10, but if we specify a larger value, say 80, the connect itself will still time out after 75 seconds.

Another point with this example is that we use the interruptibility of the system call (connect) to return before the kernel's time limit expires. This is fine when we perform the system call and can handle the EINTR error return. But in Section 26.6 we encounter a library function that performs the system call, and the library function reissues the system call when EINTR is returned. We can still use SIGALRM in this scenario, but we will see in Figure 26.10 that we also have to use sigsetjmp and siglongjmp to get around the library's ignoring of EINTR.

recvfrom with a Timeout Using SIGALRM

Figure 13.2 is a redo of our dg_cli function from Figure 8.8, but with a call to alarm to interrupt the recvfrom if a reply is not received within 5 seconds.

Handle timeout from recvfrom

8-22 We establish a signal handler for SIGALRM and then call alarm for a 5-second timeout before each call to recvfrom. If recvfrom is interrupted by our signal handler, we

advio/dgclitimeo3.c

```
 1 #include    "unp.h"
 2 static void sig_alrm(int);
 3 void
 4 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 5 {
 6     int     n;
 7     char    sendline[MAXLINE], recvline[MAXLINE + 1];
 8     Signal(SIGALRM, sig_alrm);
 9     while (Fgets(sendline, MAXLINE, fp) != NULL) {
10         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
11         alarm(5);
12         if ( (n = recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL)) < 0) {
13             if (errno == EINTR)
14                 fprintf(stderr, "socket timeout\n");
15             else
16                 err_sys("recvfrom error");
17         } else {
18             alarm(0);
19             recvline[n] = 0;    /* null terminate */
20             Fputs(recvline, stdout);
21         }
22     }
23 }
24 static void
25 sig_alrm(int signo)
26 {
27     return;                     /* just interrupt the recvfrom() */
28 }
```

advio/dgclitimeo3.c

Figure 13.2 `dg_cli` function with `alarm` to timeout `recvfrom`.

print a message and continue. If a line is read from the server, we turn off the pending `alarm` and print the reply.

`SIGALRM` signal handler

24–28 Our signal handler just returns, to interrupt the blocked `recvfrom`.

This example works correctly because we are reading only one reply each time we establish an `alarm`. In Section 18.4 we use the same technique but since we are reading multiple replies for a given `alarm`, a race condition exists that we must handle.

`recvfrom` with a Timeout Using `select`

We demonstrate the second technique for setting a timeout (using `select`) in Figure 13.3. It shows our function named `readable_timeo` that waits up to a specified number of seconds for a descriptor to become readable.

lib/readable_timeo.c

```
 1 #include    "unp.h"

 2 int
 3 readable_timeo(int fd, int sec)
 4 {
 5     fd_set  rset;
 6     struct timeval tv;

 7     FD_ZERO(&rset);
 8     FD_SET(fd, &rset);

 9     tv.tv_sec = sec;
10     tv.tv_usec = 0;

11     return (select(fd + 1, &rset, NULL, NULL, &tv));
12         /* > 0 if descriptor is readable */
13 }
```

lib/readable_timeo.c

Figure 13.3 `readable_timeo` function: wait for a descriptor to become readable.

Prepare arguments for `select`

7–10 The bit corresponding to the descriptor is turned on in the read descriptor set. A `timeval` structure is set to the number of seconds that the caller wants to wait.

Block in `select`

11–12 `select` then waits for the descriptor to become readable, or for the timeout to expire. The return value of this function is the return value of `select`: –1 on an error, 0 if a timeout occurs, or a positive value specifying the number of ready descriptors.

This function does not perform the read operation; it just waits for the descriptor to be ready for reading. Therefore this function can be used with any type of socket: TCP or UDP.

It is trivial to create a similar function named `writable_timeo` that waits for a descriptor to become writable.

We use this function in Figure 13.4, which is a redo of our `dg_cli` function from Figure 8.8. This new version calls `recvfrom` only when our `readable_timeo` function returns a positive value.

We do not call `recvfrom` until the function `readable_timeo` tells us that the descriptor is readable. This guarantees that `recvfrom` will not block.

`recvfrom` with a Timeout Using the `SO_RCVTIMEO` Socket Option

Our final example demonstrates the `SO_RCVTIMEO` socket option. We set this option once for a descriptor, specifying the timeout value, and this timeout then applies to all read operations on that descriptor. The nice thing about this method is that we set the option only once, compared to the previous two methods which required doing something before every operation on which we want to place a time limit. But this socket option applies only to read operations, and the similar option `SO_SNDTIMEO` applies

advio/dgclitimeo1.c

```
 1 #include    "unp.h"

 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5     int     n;
 6     char    sendline[MAXLINE], recvline[MAXLINE + 1];

 7     while (Fgets(sendline, MAXLINE, fp) != NULL) {

 8         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

 9         if (Readable_timeo(sockfd, 5) == 0) {
10             fprintf(stderr, "socket timeout\n");
11         } else {
12             n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
13             recvline[n] = 0;    /* null terminate */
14             Fputs(recvline, stdout);
15         }
16     }
17 }
```

advio/dgclitimeo1.c

Figure 13.4 `dg_cli` function that calls `readable_timeo` to set a timeout.

only to write operations: neither socket option can be used to set a timeout for a `connect`.

Figure 13.5 shows another version of our `dg_cli` function that uses the `SO_RCVTIMEO` socket option.

Set socket option

8-10 The fourth argument to `setsockopt` is a pointer to a `timeval` structure that is filled in with the desired timeout.

Test for timeout

15-17 If the I/O operation times out, the function (`recvfrom` in this case) returns `EWOULDBLOCK`.

13.3 `recv` and `send` Functions

These two functions are similar to the standard `read` and `write` functions, but one additional argument is required.

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

ssize_t recv(int sockfd, void *buff, size_t nbytes, int flags);

ssize_t send(int sockfd, const void *buff, size_t nbytes, int flags);
```

Both return: number of bytes read or written if OK, –1 on error

advio/dgclitimeo2.c

```
 1 #include    "unp.h"
 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5     int     n;
 6     char    sendline[MAXLINE], recvline[MAXLINE + 1];
 7     struct timeval tv;
 8     tv.tv_sec = 5;
 9     tv.tv_usec = 0;
10     Setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));
11     while (Fgets(sendline, MAXLINE, fp) != NULL) {
12         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
13         n = recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
14         if (n < 0) {
15             if (errno == EWOULDBLOCK) {
16                 fprintf(stderr, "socket timeout\n");
17                 continue;
18             } else
19                 err_sys("recvfrom error");
20         }
21         recvline[n] = 0;        /* null terminate */
22         Fputs(recvline, stdout);
23     }
24 }
```

advio/dgclitimeo2.c

Figure 13.5 `dg_cli` function that uses the `SO_RCVTIMEO` socket option to set a timeout.

The first three arguments to `recv` and `send` are the same as the first three arguments to `read` and `write`. The *flags* argument is either 0, or is formed by logically OR'ing one or more of the constants shown in Figure 13.6.

flags	Description	`recv`	`send`
`MSG_DONTROUTE`	bypass routing table lookup		•
`MSG_DONTWAIT`	only this operation is nonblocking	•	•
`MSG_OOB`	send or receive out-of-band data	•	•
`MSG_PEEK`	peek at incoming message	•	
`MSG_WAITALL`	wait for all the data	•	

Figure 13.6 *flags* for I/O functions.

`MSG_DONTROUTE` This flag tells the kernel that the destination is on a locally attached network and not to perform a lookup of the routing table. We provided additional information on this feature with the `SO_DONTROUTE` socket option (Section 7.5). This feature can

be enabled for a single output operation with the MSG_DONTROUTE flag, or enabled for all output operations for a given socket using the socket option.

MSG_DONTWAIT This flag specifies nonblocking for a single I/O operation, without having to turn on the nonblocking flag for the socket, perform the I/O operation, and then turn off the nonblocking flag. We describe nonblocking I/O in Chapter 15 along with turning the nonblocking flag on and off for all I/O operations on a socket.

This flag is new with Net/3 and might not be supported on all systems.

MSG_OOB With send, this flag specifies that out-of-band data is being sent. With TCP only 1 byte should be sent as out-of-band data, as we describe in Chapter 21. With recv, this flag specifies that out-of-band data is to be read instead of normal data.

MSG_PEEK This flag lets us look at the data that is available to be read, without having the system discard the data after the recv or recvfrom returns. We talk more about this in Section 13.7.

MSG_WAITALL This flag was introduced with 4.3BSD Reno. It tells the kernel not to return from a read operation until the requested number of bytes have been read. If the system supports this flag, we can then omit the readn function (Figure 3.14) and replace it with the macro

```
#define  readn(fd, ptr, n)  recv(fd, ptr, n, MSG_WAITALL)
```

Even if we specify MSG_WAITALL, the function can still return fewer than the requested number of bytes if (a) a signal is caught, (b) the connection is terminated, or (c) an error is pending for the socket.

There are additional flags used by other protocols, but not TCP/IP. For example, the OSI transport layer is record based (not a byte stream such as TCP) and supports the MSG_EOR flag for output operations to specify the end of a logical record. T/TCP (TCP for transactions, described in Section 13.9) supports a new MSG_EOF flag to combine an output operation with the sending of a FIN.

There is a fundamental design problem with the *flags* argument: it is passed by value; it is not a value–result argument. Therefore it can be used only to pass flags from the process to the kernel. The kernel cannot pass back flags to the process. This is not a problem with TCP/IP, because it is rare to need to pass flags back to the process from the kernel. But when the OSI protocols were added to 4.3BSD Reno, the need arose to return MSG_EOR to the process with an input operation. The decision was made with

4.3BSD Reno to leave the arguments to the commonly used input functions (recv and recvfrom) as is and change the msghdr structure that is used with recvmsg and sendmsg. We will see in Section 13.5 that an integer msg_flags member was added to this structure, and since the structure is passed by reference, the kernel can modify these flags on return. This also means that if a process needs to have the flags updated by the kernel, the process must call recvmsg instead of either recv or recvfrom.

13.4 readv and writev Functions

These two functions are similar to read and write, but readv and writev let us read into or write from one or more buffers with a single function call. These operations are called *scatter read* (since the input data is scattered into multiple application buffers) and *gather write* (since multiple buffers are gathered for a single output operation).

```
#include <sys/uio.h>

ssize_t readv(int filedes, const struct iovec *iov, int iovcnt);

ssize_t writev(int filedes, const struct iovec *iov, int iovcnt);
```

Both return: number of bytes read or written, –1 on error

The second argument to both functions is a pointer to an array of iovec structures, which is defined by including the <sys/uio.h> header:

```
struct iovec {
  void   *iov_base;   /* starting address of buffer */
  size_t  iov_len;    /* size of buffer */
};
```

The readv and writev functions have not yet been standardized by Posix. But the iovec structure is also used with the recvmsg and sendmsg functions (Section 13.5) and this structure is standardized by Posix.1g. The datatypes shown for the members of the iovec structure are those specified by Posix.1g. You may encounter implementations that define iov_base to be a char *, and iov_len to be an int.

There is some limit to the number of elements in the array of iovec structures that an implementation allows. 4.4BSD, for example, allows up to 1024 while Solaris 2.5 has a limit of 16. Posix.1g requires that the constant IOV_MAX be defined by including the <sys/uio.h> header and that its value be at least 16.

The readv and writev functions can be used with any descriptor, not just sockets. Also, writev is an atomic operation. For a record-based protocol such as UDP, one call to writev generates a single UDP datagram.

We mentioned one use of writev with the TCP_NODELAY socket option in Section 7.9. We said that a write of 4 bytes followed by a write of 396 bytes could invoke the Nagle algorithm and a preferred solution is to call writev for the two buffers.

13.5 recvmsg and sendmsg Functions

These two functions are the most general of all the I/O functions. Indeed, we could replace all calls to `read`, `readv`, `recv`, and `recvfrom` with calls to `recvmsg`. Similarly all calls to the various output functions could be replaced with calls to `sendmsg`.

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

ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);

ssize_t sendmsg(int sockfd, struct msghdr *msg, int flags);
```

Both return: number of bytes read or written if OK, –1 on error

Both functions package most of the arguments into a `msghdr` structure:

```
struct msghdr {
  void         *msg_name;       /* protocol address */
  socklen_t     msg_namelen;    /* size of protocol address */
  struct iovec *msg_iov;        /* scatter/gather array */
  size_t        msg_iovlen;     /* # elements in msg_iov */
  void         *msg_control;    /* ancillary data; must be aligned for
                                   a cmsghdr structure */
  socklen_t     msg_controllen; /* length of ancillary data */
  int           msg_flags;      /* flags returned by recvmsg() */
};
```

The `msghdr` structure that we show originated with 4.3BSD Reno and is the one specified in Posix.1g. Some systems (Solaris 2.5) still use an older `msghdr` structure that originated with 4.2BSD. This older structure does not have the `msg_flags` member and the `msg_control` and `msg_controllen` members are named `msg_accrights` and `msg_accrightslen`. The only form of ancillary data supported by these older systems is the passing of file descriptors (called access rights). 4.3BSD Reno added more forms of ancillary data when the OSI protocols were added and therefore generalized the structure member names.

The `msg_name` and `msg_namelen` members are used when the socket is not connected (e.g., an unconnected UDP socket). They are similar to the fifth and sixth arguments to `recvfrom` and `sendto`: `msg_name` points to a socket address structure in which the caller stores the destination's protocol address for `sendmsg`, or in which `recvmsg` stores the sender's protocol address. If a protocol address does not need to be specified (e.g., a TCP socket or a connected UDP socket), `msg_name` should be set to a null pointer. `msg_namelen` is a value for `sendmsg` but a value–result for `recvmsg`.

The `msg_iov` and `msg_iovlen` members specify the array of input or output buffers (the array of `iovec` structures), similar to the second and third arguments for `readv` or `writev`.

The `msg_control` and `msg_controllen` members specify the location and size of the optional ancillary data. `msg_controllen` is a value–result argument for `recvmsg`. We describe ancillary data in Section 13.6.

With `recvmsg` and `sendmsg` we must distinguish between two flag variables: the *flags* argument, which is passed by value, and the `msg_flags` member of the `msghdr`

structure, which is passed by reference (since the address of the structure is passed to the function).

- The msg_flags member is used only by recvmsg. When recvmsg is called, the *flags* argument is copied into the msg_flags member (p. 502 of TCPv2) and this value is used by the kernel to drive its receive processing. This value is then updated based on the result of recvmsg.
- The msg_flags member is ignored by sendmsg because this function uses the *flags* argument to drive its output processing. This means if we want to set the MSG_DONTWAIT flag in a call to sendmsg, we set the *flags* argument to this value; setting the msg_flags member to this value has no effect.

Figure 13.7 summarizes which flags are examined by the kernel for both the input and output functions, and which of the msg_flags might be returned by recvmsg. There is no column for sendmsg msg_flags because, as we mentioned, it is not used.

Flag	Examined by: send *flags* sendto *flags* sendmsg *flags*	Examined by: recv *flags* recvfrom *flags* recvmsg *flags*	Returned by: recvmsg msg_flags
MSG_DONTROUTE	•		
MSG_DONTWAIT	•	•	
MSG_PEEK		•	
MSG_WAITALL		•	
MSG_EOR	•		•
MSG_OOB	•	•	•
MSG_BCAST			•
MSG_MCAST			•
MSG_TRUNC			•
MSG_CTRUNC			•

Figure 13.7 Summary of input and output flags by various I/O functions.

The first four flags are only examined and never returned, the next two are both examined and returned, and the last four are only returned. The following comments apply to the six flags that are returned by recvmsg:

MSG_BCAST — This flag is new with BSD/OS and is returned if the datagram was received as a link-layer broadcast or with a destination IP address that is a broadcast address. This flag is a better way of determining if a UDP datagram was sent to a broadcast address, compared to the IP_RECVDSTADDR socket option.

MSG_MCAST — This flag is new with BSD/OS and is returned if the datagram was received as a link-layer multicast.

MSG_TRUNC	This flag is returned if the datagram was truncated: the kernel has more data to return than the process has allocated room for (the sum of all the iov_len members). We discuss this more in Section 20.3.
MSG_CTRUNC	This flag is returned if the ancillary data was truncated: the kernel has more ancillary data to return than the process has allocated room for (msg_controllen).
MSG_EOR	This flag is cleared if the returned data does not end a logical record, or the flag is turned on if the returned data ends a logical record. TCP does *not* use this flag, since it is a byte-stream protocol.
MSG_OOB	This flag is *never* returned for TCP out-of-band data. This flag is returned by other protocol suites (e.g., the OSI protocols).

Implementations might return some of the input *flags* in the msg_flags member, so we should examine only those flag values that we are interested in (e.g., the last six in Figure 13.7).

Figure 13.8 shows a msghdr structure and the various information that it points to. We assume in this figure that the process is about to call recvmsg for a UDP socket.

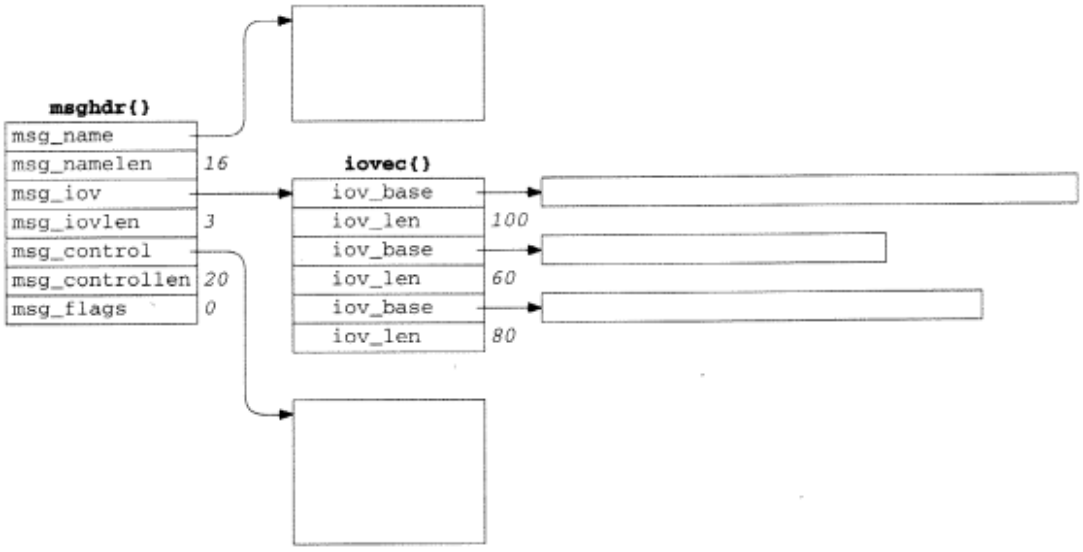

Figure 13.8 Data structures when recvmsg is called for a UDP socket.

Sixteen bytes are allocated for the protocol address and 20 bytes are allocated for the ancillary data. An array of three iovec structures is initialized: the first specifies a 100-byte buffer, the second a 60-byte buffer, and the third an 80-byte buffer. We also

assume that the IP_RECVDSTADDR socket option has been set for the socket, to receive the destination IP address from the UDP datagram.

We then assume that a 170-byte UDP datagram arrives from 198.69.10.2, port 2000, destined for our UDP socket with a destination IP address of 206.62.226.35. Figure 13.9 shows all the information in the msghdr structure when recvmsg returns.

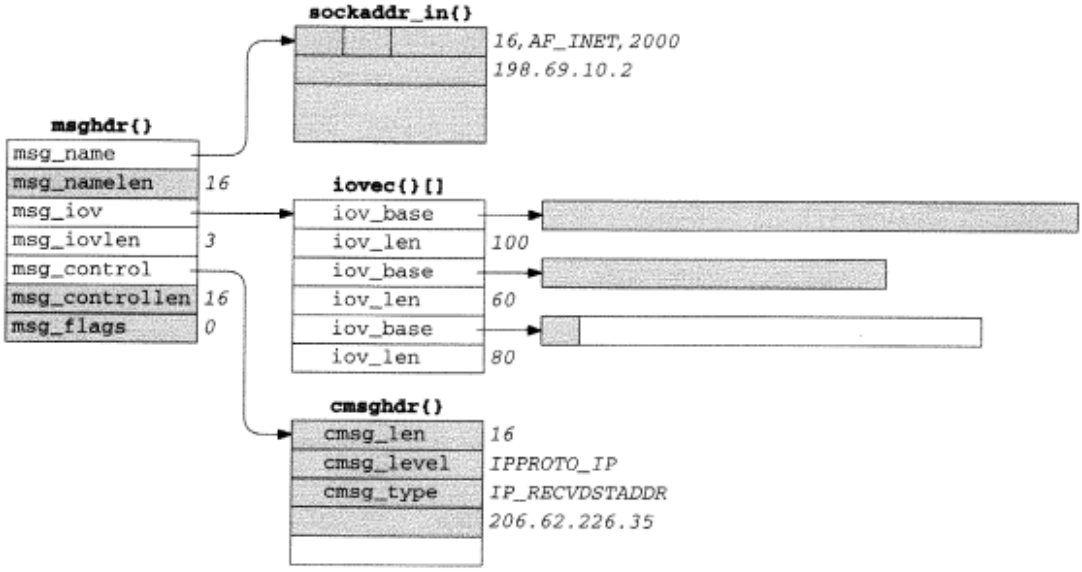

Figure 13.9 Update of Figure 13.8 when recvmsg returns.

The shaded fields are modified by recvmsg. The following items have changed from Figure 13.8 to Figure 13.9:

- The buffer pointed to by msg_name has been filled in as an Internet socket address structure containing the source IP address and source UDP port from the received datagram.
- msg_namelen, a value–result argument, is updated with the amount of data stored in msg_name. Nothing changes since its value before the call was 16 and its value when recvmsg returns is also 16.
- The first 100 bytes of data are stored in the first buffer, the next 60 bytes are stored in the second buffer, and the final 10 bytes are stored in the third buffer. The last 70 bytes of the final buffer are not modified. The return value of the recvmsg function is the size of the datagram, 170.
- The buffer pointed to by msg_control is filled in as a cmsghdr structure. (We say more about ancillary data in Section 13.6 and more about this particular socket option in Section 20.2.) The cmsg_len is 16, cmsg_level is IPPROTO_IP, cmsg_type is IP_RECVDSTADDR, and the next 4 bytes contain the destination IP address from the received UDP datagram. The final 4 bytes of the 20-byte buffer that we supplied to hold the ancillary data are not modified.

- The `msg_controllen` member is updated with the actual amount of ancillary data that was stored; it is also a value–result argument and its result on return is 16.
- The `msg_flags` member is updated by `recvmsg` but there are no flags to return to the process.

Figure 13.10 summarizes the differences between the five groups of I/O functions that we have described.

Function	Any descriptor	Only socket descriptor	Single read/write buffer	Scatter/gather read/write	Optional flags	Optional peer address	Optional control information
`read, write`	•		•				
`readv, writev`	•			•			
`recv, send`		•	•		•		
`recvfrom, sendto`		•	•		•	•	
`recvmsg, sendmsg`		•		•	•	•	•

Figure 13.10 Comparison of the five groups of I/O functions.

13.6 Ancillary Data

Ancillary data can be sent and received using the `msg_control` and `msg_controllen` members of the `msghdr` structure with the `sendmsg` and `recvmsg` functions. Another term for ancillary data is *control information*. In this section we describe the concept and show the structure and macros that are used to build and process ancillary data, but we save the code examples for later chapters that describe the actual uses of ancillary data.

Figure 13.11 is a summary of the various uses of ancillary data that we cover in this text.

Protocol	`cmsg_level`	`cmsg_type`	Description
IPv4	`IPPROTO_IP`	`IP_RECVDSTADDR`	receive destination address with UDP datagram
		`IP_RECVIF`	receive interface index with UDP datagram
IPv6	`IPPROTO_IPV6`	`IPV6_DSTOPTS`	specify/receive destination options
		`IPV6_HOPLIMIT`	specify/receive hop limit
		`IPV6_HOPOPTS`	specify/receive hop-by-hop options
		`IPV6_NEXTHOP`	specify next-hop address
		`IPV6_PKTINFO`	specify/receive packet information
		`IPV6_RTHDR`	specify/receive routing header
Unix domain	`SOL_SOCKET`	`SCM_RIGHTS`	send/receive descriptors
		`SCM_CREDS`	send/receive user credentials

Figure 13.11 Summary of uses for ancillary data.

The OSI protocol suite also uses ancillary data for various purposes that we do not discuss in this text.

Ancillary data consists of one or more *ancillary data objects*, each one beginning with a `cmsghdr` structure, defined by including `<sys/socket.h>`:

```
struct cmsghdr {
  socklen_t  cmsg_len;   /* length in bytes, including this structure */
  int        cmsg_level; /* originating protocol */
  int        cmsg_type;  /* protocol-specific type */
      /* followed by unsigned char cmsg_data[] */
};
```

We have already seen this structure in Figure 13.9, when used with the `IP_RECVDSTADDR` socket option to return the destination IP address of a received UDP datagram. The ancillary data pointed to by `msg_control` must be suitably aligned for a `cmsghdr` structure . We show one way to do this in Figure 14.11.

Figure 13.12 shows an example of two ancillary data objects in the control buffer.

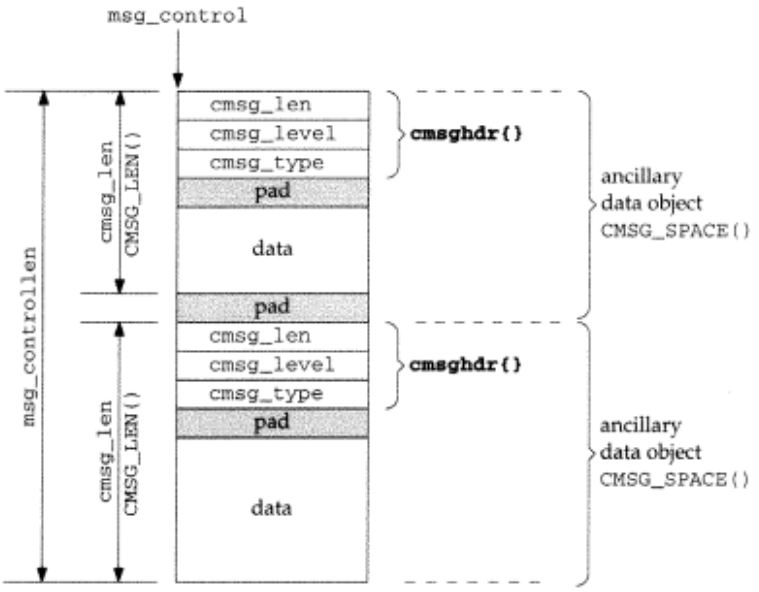

Figure 13.12 Ancillary data containing two ancillary data objects.

`msg_control` points to the first ancillary data object and the total length of the ancillary data is specified by `msg_controllen`. Each object is preceded by a `cmsghdr` structure that describes the object. There can be padding between the `cmsg_type` member and the actual data, and there can also be padding at the end of the data, before the next ancillary data object. The five `CMSG_`*xxx* macros that we describe shortly account for this possible padding.

Not all implementations support multiple ancillary data objects in the control buffer.

Figure 13.13 shows the format of the cmsghdr structure when used with a Unix domain socket for descriptor passing (Section 14.7) or credential passing (Section 14.8).

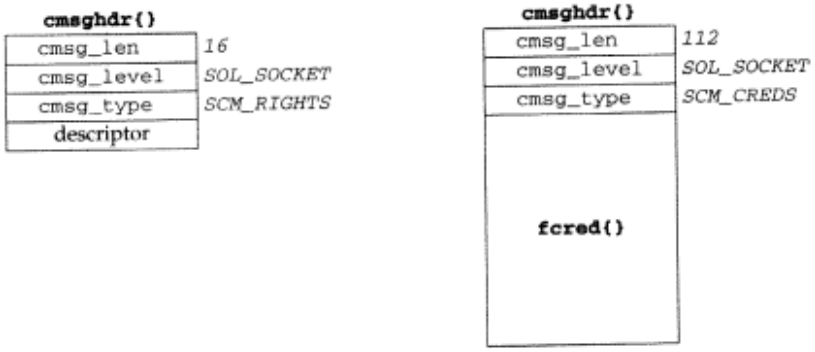

Figure 13.13 cmsghdr structure when used with Unix domain sockets.

In this figure we assume each of the three members of the cmsghdr structure occupies 4 bytes and there is no padding between the cmsghdr structure and the actual data. When descriptors are passed, the contents of the cmsg_data array are the actual descriptor values. In this figure we show only one descriptor being passed, but in general more than one can be passed (in which case the cmsg_len value will be 12 plus 4 times the number of descriptors, assuming each descriptor occupies 4 bytes).

Since the ancillary data returned by recvmsg can contain any number of ancillary data objects and to hide the possible padding from the application, the following five macros are defined by including the <sys/socket.h> header to simplify the processing of the ancillary data.

```
#include <sys/socket.h>
#include <sys/param.h>   /* for ALIGN macro on many implementations */

struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *mhdrptr);
```

Returns: pointer to first cmsghdr structure or NULL if no ancillary data

```
struct cmsghdr *CMSG_NXTHDR(struct msghdr *mhdrptr, struct cmsghdr *cmsgptr);
```

Returns: pointer to next cmsghdr structure or NULL if no more ancillary data objects

```
unsigned char *CMSG_DATA(struct cmsghdr *cmsgptr);
```

Returns: pointer to first byte of data associated with cmsghdr structure

```
unsigned int CMSG_LEN(unsigned int length);
```

Returns: value to store in cmsg_len given the amount of data

```
unsigned int CMSG_SPACE(unsigned int length);
```

Returns: total size of an ancillary data object given the amount of data

Posix.1g defines the first three macros, [Stevens and Thomas 1997] define the last two.

These macros would be used in the following pseudocode:

```
struct msghdr   msg;
struct cmsghdr  *cmsgptr;

/* fill in msg structure */

/* call recvmsg() */

for (cmsgptr = CMSG_FIRSTHDR(&msg); cmsgptr != NULL;
     cmsgptr = CMSG_NXTHDR(&msg, cmsgptr)) {
    if (cmsgptr->cmsg_level == ... &&
        cmsgptr->cmsg_type == ... ) {
        u_char  *ptr;

        ptr = CMSG_DATA(cmsgptr);
        /* process data pointed to by ptr */
    }
}
```

`CMSG_FIRSTHDR` returns a pointer to the first ancillary data object, or a null pointer if there is no ancillary data in the `msghdr` structure (either `msg_control` is a null pointer, or `cmsg_len` is less than the size of a `cmsghdr` structure). `CMSG_NXTHDR` returns a null pointer when there is not another ancillary data object in the control buffer.

Many existing implementations of `CMSG_FIRSTHDR` never look at `msg_controllen` and just return the value of `cmsg_control`. In Figure 20.2 we test the value of `msg_controllen` before calling this macro.

The difference between `CMSG_LEN` and `CMSG_SPACE` is that the former does not account for any padding following the data portion of the ancillary data object and is therefore the value to store in `cmsg_len`, while the latter accounts for the padding at the end and is therefore the value to use if dynamically allocating space for the ancillary data object.

13.7 How Much Data Is Queued?

There are times when we want to see how much data is queued to be read on a socket, without reading the data. Three techniques are available.

1. If the goal is not to block in the kernel because we have something else to do when nothing is ready to be read, nonblocking I/O can be used. We describe this in Chapter 15.
2. If we want to examine the data but still leave it on the receive queue for some other part of our process to read, we can use the `MSG_PEEK` flag (Figure 13.6). If we want to do this, but we are not sure that something is ready to be read, we can combine this flag with a nonblocking socket or combine this flag with the `MSG_DONTWAIT` flag.

Be aware that the amount of data on the receive queue can change between two successive calls to `recv` for a stream socket. For example, assume we call `recv` for a TCP socket specifying a buffer length of 1024 along with the `MSG_PEEK` flag, and the return value is 100. If we then call `recv` again, it is possible for more than 100 bytes to be returned (assuming we specify a buffer length greater than 100), because more data can be received by TCP between our two calls.

In the case of a UDP socket with a datagram on the receive queue, if we call `recvfrom` specifying `MSG_PEEK`, followed by another call without specifying `MSG_PEEK`, the return values from both calls (the datagram size, its contents, and the sender's address) will be the same, even if additional datagrams are added to the socket receive buffer between the two calls. (We are assuming, of course, that some other process is not sharing the same descriptor and reading from this socket at the same time.)

3. Some implementations support the `FIONREAD` command of `ioctl`. The third argument to `ioctl` is a pointer to an integer, and the value returned in that integer is the current number of bytes on the socket's receive queue (p. 553 of TCPv2). This value is the total number of bytes queued, which for a UDP socket includes all queued datagrams. Also be aware that the count returned for a UDP socket by Berkeley-derived implementations includes the space required for the socket address structure containing the sender's IP address and port for each datagram (16 bytes for IPv4; 24 bytes for IPv6).

13.8 Sockets and Standard I/O

In all our examples so far we have used what is sometimes called *Unix I/O*, the `read` and `write` functions and their variants (`recv`, `send`, etc.). These functions work with *descriptors* and are normally implemented as system calls within the Unix kernel.

Another method of performing I/O is the *standard I/O library*. It is specified by the ANSI C standard and is intended to be portable to non-Unix systems that support ANSI C. The standard I/O library handles some of the details that we must worry about ourselves when using the Unix I/O functions, such as automatically buffering the input and output streams. Unfortunately its handling of a stream's buffering can present a new set of problems that we must worry about. Chapter 5 of APUE covers the standard I/O library in detail and [Plauger 1992] presents and discusses a complete implementation of the standard I/O library.

> The term *stream* is used with the standard I/O library, as in "we open an input stream" or "we flush the output stream." Do not confuse this with the System V streams subsystem, which we discuss in Chapter 33.

The standard I/O library can be used with sockets, but there are a few items to consider.

- A standard I/O stream can be created from any descriptor by calling the `fdopen` function. Similarly, given a standard I/O stream, we can obtain the

corresponding descriptor by calling `fileno`. Our first encounter with `fileno` was in Figure 6.9 when we wanted to call `select` on a standard I/O stream. `select` works only with descriptors, so we had to obtain the descriptor for the standard I/O stream.

- TCP and UDP sockets are full-duplex. Standard I/O streams can also be full-duplex: we just open the stream with a type of `r+`, which means read–write. But on such a stream an output function cannot be followed by an input function without an intervening call to `fflush`, `fseek`, `fsetpos`, or `rewind`. Similarly, an input function cannot be followed by an output function without an intervening call to `fseek`, `fsetpos`, or `rewind`, unless the input function encounters an end-of-file. The problem with these latter three functions is that they all call `lseek`, which fails on a socket.
- The easiest way to handle this read–write problem is to open two standard I/O streams for a given socket: one for reading, and one for writing.

Example: `str_echo` Function Using Standard I/O

We now redo our TCP echo server (Figure 5.3) to use standard I/O instead of `readline` and `writen`. Figure 13.14 is a version of our `str_echo` function that uses standard I/O. (This version has a problem that we describe shortly.)

advio/str_echo_stdio02.c

```
 1 #include    "unp.h"

 2 void
 3 str_echo(int sockfd)
 4 {
 5     char    line[MAXLINE];
 6     FILE   *fpin, *fpout;

 7     fpin = Fdopen(sockfd, "r");
 8     fpout = Fdopen(sockfd, "w");

 9     for ( ; ; ) {
10         if (Fgets(line, MAXLINE, fpin) == NULL)
11             return;              /* connection closed by other end */

12         Fputs(line, fpout);
13     }
14 }
```

advio/str_echo_stdio02.c

Figure 13.14 `str_echo` function recoded to use standard I/O.

Convert descriptor into input stream and output stream

7-13 Two standard I/O streams are created by `fdopen`: one for input and one for output. The calls to `readline` and `writen` are replaced with calls to `fgets` and `fputs`.

If we run our server with this version of `str_echo` and then run our client, we see the following.

```
solaris % tcpcli02 206.62.226.33
hello, world                    we type this line, but nothing echoed
and hi                          and this one, still no echo
hello??                         and this one, still no echo
^D                              and our end-of-file character
hello, world                    and then the three echoed lines are output
and hi
hello??
```

There is a buffering problem here because nothing is echoed by the server until we enter our end-of-file character. The following steps are taking place:

- We type the first line of input and it is sent to the server.
- The server reads the line with `fgets` and echoes it with `fputs`.
- But the server's standard I/O stream is *fully buffered* by the standard I/O library. This means the library copies the echoed line into its standard I/O buffer for this stream but does not write the buffer to the descriptor, because the buffer is not full.
- We type the second line of input and it is sent to the server.
- The server reads the line with `fgets` and echoes it with `fputs`.
- Again, the server's standard I/O library just copies the line into its buffer but does not write the buffer because it still is not full.
- The same scenario happens with the third line of input that we enter.
- We type our end-of-file character, and our `str_cli` function (Figure 6.13) calls `shutdown`, sending a FIN to the server.
- The server TCP receives the FIN, which `fgets` reads, causing `fgets` to return a null pointer.
- The `str_echo` function returns to the server `main` function (Figure 5.12) and the child terminates by calling `exit`.
- The C library function `exit` calls the standard I/O cleanup function (pp. 162–164 of APUE) and the output buffer that was partially filled by our calls to `fputs` is now output.
- The server child process then terminates, causing its connected socket to be closed, sending a FIN to the client, completing the TCP four-packet termination sequence.
- The three echoed lines are received by our `str_cli` function and output.
- `str_cli` then receives an end-of-file on its socket, and the client terminates.

The problem here is the buffering performed automatically by the standard I/O library on the server. There are three types of buffering performed by the standard I/O library.

1. *Fully buffered* means that I/O takes place only when the buffer is full, the process explicitly calls `fflush`, or the process terminates by calling `exit`. A common size for the standard I/O buffer is 8192 bytes.

2. *Line buffered* means that I/O takes place when a newline is encountered, when the process calls `fflush`, or when the process terminates by calling `exit`.
3. *Unbuffered* means that I/O takes place each time a standard I/O output function is called.

Most Unix implementations of the standard I/O library use the following rules:

- Standard error is always unbuffered.
- Standard input and standard output are fully buffered, unless they refer to a terminal device, in which case they are line buffered.
- All other streams are fully buffered unless they refer to a terminal device, in which case they are line buffered.

Since a socket is not a terminal device, the problem seen with our `str_echo` function in Figure 13.14 is that the output stream (`fpout`) is fully buffered. There are two solutions: we can force the output stream to be line buffered by calling `setvbuf`, or we can force each echoed line to be output by calling `fflush` after each call to `fputs`. Applying either of these changes corrects the behavior of our `str_echo` function.

> Another solution is to avoid the standard I/O library altogether and use the `sfio` library. It is described in [Korn and Vo 1991] and the source code is publicly available.
>
> Be aware that some implementations of the standard I/O library still have a problem with descriptors greater than 255. This can be a problem with network servers that handle lots of descriptors. Check the definition of the `FILE` structure in your `<stdio.h>` header to see what type of variable holds the descriptor.

13.9 T/TCP: TCP for Transactions

T/TCP is a slight modification to TCP that can avoid the three-way handshake between hosts that have communicated with each other recently. T/TCP is described in detail in TCPv3, RFC 1379 [Braden 1992b], and RFC 1644 [Braden 1994].

> The most widespread implementation of T/TCP is in FreeBSD.

T/TCP can combine the SYN, FIN, and data into a single segment, assuming the size of the data is less than the MSS. We show this in Figure 13.15. The first segment is the client's SYN, FIN, and data, generated by one call to `sendto`. This combines the functionality of `connect`, `write`, and `shutdown`. The server does the normal steps of `socket`, `bind`, `listen`, and `accept`, the latter returning when the client's segment arrives. The server sends its reply with `send` and closes the socket. This causes the server's SYN, FIN, and reply to be sent to the client. If we compare this to Figure 2.5, we see that not only are fewer segments required in the network (three for T/TCP, 10 for TCP, and two for UDP), but the time that it takes for the client to initiate the connection, send a request, and read the reply has been decreased by one RTT.

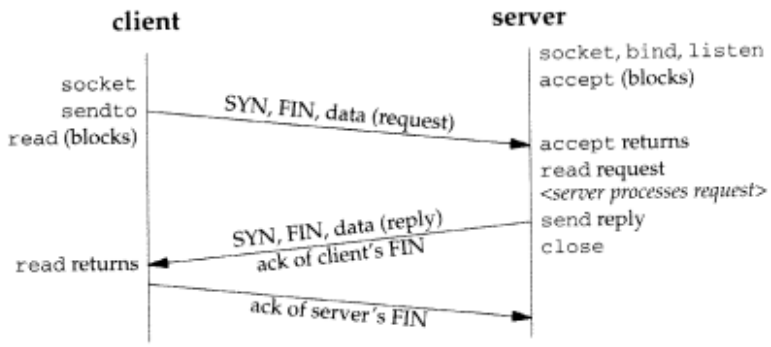

Figure 13.15 Time line of minimal T/TCP transaction.

The benefit of T/TCP is that all the reliability of TCP is retained (sequence numbers, timeouts, retransmissions, and the like), unlike a UDP solution, which pushes the reliability into the application. T/TCP also maintains TCP's slow start and congestion avoidance, features that are often missing from UDP applications.

> We are ignoring some details here, all of which are covered in TCPv3. For example, the first time this client talks to this server the three-way handshake is required. But this can be avoided in the future as long as some cached information on both ends does not expire, and as long as neither end crashes and reboots. The three segments shown form the minimum request–reply exchange. Additional segments are required if either the request or the reply do not fit into one segment. The term "transaction" means a client request and the server's reply. Common examples are a DNS request and the server's reply, and an HTTP request and the server's reply. The term is not being used to refer to a two-phase commit protocol.

A few changes are made to the sockets API to handle T/TCP. We note that on a system providing T/TCP no changes need be made to any TCP applications, unless the features of T/TCP are desired. All existing TCP applications continue to work using the sockets API that we have already described.

- A client calls `sendto` to combine the connection establishment with the sending of data. This replaces the separate calls to `connect` and `write`. The server's protocol address is now passed to `sendto`, instead of `connect`.
- A new output flag, `MSG_EOF` is provided (Figure 13.6) to indicate that no more data will be sent on this socket. This allows us to combine an output operation (`send` or `sendto`) with a `shutdown`. Specifying this flag with a `sendto` that also specifies the server's address is the way that a segment is sent containing a SYN, FIN, and data. Also note in Figure 13.15 that the server sends its reply using `send` and not `write`. The reason is to specify `MSG_EOF` to send the FIN with the reply. (Do not confuse this new flag with the existing `MSG_EOR` flag, which indicates the end-of-record for record-oriented protocols.)
- A new socket option, `TCP_NOPUSH` is defined with a *level* of `IPPROTO_TCP`. When this option is enabled, it prevents TCP from sending a segment just to

empty the socket send buffer. Clients should set this option when sending a request with a single `sendto`, if the request exceeds the MSS, as it may reduce the number of segments sent. Pages 47–49 of TCPv3 talk about this new socket option in more detail.

- A client that wants to establish a connection with a server and send a request using T/TCP should call `socket`, `setsockopt` (to enable the `TCP_NOPUSH` option), and `sendto` (specifying `MSG_EOF` if only one request is to be sent). If `setsockopt` fails with an error of `ENOPROTOOPT` or if `sendto` fails with an error of `ENOTCONN`, then the host does not support T/TCP. In this case the client just calls `connect` and `write`, possibly followed by `shutdown` (if only one request is to be sent).
- The only change required by a server is to call `send` with the `MSG_EOF` flag to send a reply, instead of `write`, if the server wants to send a FIN with the reply.
- Compile-time tests for T/TCP can use `#ifdef MSG_EOF`.

Appendix B of TCPv3 contains example T/TCP client and server code.

13.10 Summary

There are three ways to set a time limit on a socket operation:

- use the `alarm` function and the `SIGALRM` signal,
- use the time limit that is provided by `select`, and
- use the newer `SO_RCVTIMEO` and `SO_SNDTIMEO` socket options.

The first is easy to use but involves signal handling and as we see in Section 18.5 can lead to race conditions. Using `select` means that we block in this function, with its provided time limit, instead of blocking in a call to `read`, `write`, or `connect`. The third alternative, the new socket options, is also easy to use but not provided by all implementations.

`recvmsg` and `sendmsg` are the most general of the five groups of I/O functions that are provided. They combine the ability to specify an `MSG_`*xxx* flag (from `recv` and `send`), with the ability to return or specify the peer's protocol address (from `recvfrom` and `sendto`), with the ability to use multiple buffers (from `readv` and `writev`), along with two new features: returning flags to the application and receiving or sending ancillary data.

We describe 10 different forms of ancillary data in the text, six of which are new with IPv6. Ancillary data consists of one or more ancillary data objects, each object preceded by a `cmsghdr` structure specifying its length, protocol level, and type of data. Five functions beginning with `CMSG_` are used to build and parse ancillary data.

Sockets can be used with the C standard I/O library, but doing this adds another level of buffering to that already being performed by TCP. Indeed, a lack of understanding of the buffering performed by the standard I/O library is the most common problem with the library. Since a socket is not a terminal device, the common solution to this potential problem is to set the standard I/O stream unbuffered.

T/TCP is a simple enhancement to TCP that can avoid the three-way handshake, allowing a faster response by a server to a client's query, if that client and server have communicated recently. From a programming perspective a client takes advantage of T/TCP by calling `sendto` instead of the normal sequence of `connect`, `write`, and `shutdown`.

Exercises

13.1 What happens in Figure 13.1 when we reset the signal handler, if the process has not established a handler for `SIGALRM`?

13.2 In Figure 13.1 we print a warning if the process already has an `alarm` timer set. Modify the function to reset this `alarm` for the process after the `connect`, before the function returns.

13.3 Modify Figure 11.7 as follows: before calling `read`, call `recv` specifying `MSG_PEEK`. When this returns, call `ioctl` with a command of `FIONREAD` and print the number of bytes queued on the socket's receive buffer. Then call `read` to actually read the data.

13.4 What happens to the data in a standard I/O buffer that has not yet been output if the process falls off the end of the `main` function instead of calling `exit`?

13.5 Apply each of the two changes described following Figure 13.14 and verify that each one corrects the buffering problem.

14

Unix Domain Protocols

14.1 Introduction

The Unix domain protocols are not an actual protocol suite, but a way of performing client–server communication on a single host using the same API that is used for clients and servers on different hosts: sockets or XTI. The Unix domain protocols are an alternative to the IPC methods described in Volume 2 of this series, when the client and server are on the same host. Details on the actual implementation of Unix domain sockets in a Berkeley-derived kernel are provided in Part 3 of TCPv3.

Two types of sockets are provided in the Unix domain: stream sockets (similar to TCP) and datagram sockets (similar to UDP). Even though a raw socket is also provided, its semantics have never been documented, it is not used by any program that the author is aware of, and it is not defined by Posix.1g.

Unix domain sockets are used for three reasons.

1. On Berkeley-derived implementations Unix domain sockets are often twice as fast as a TCP socket, when both peers are on the same host (pp. 223–224 of TCPv3). One application takes advantage of this: the X Window System. When an X11 client starts and opens a connection to the X11 server, the client checks the value of the `DISPLAY` environment variable, which specifies the server's hostname, window, and screen. If the server is on the same host as the client, the client opens a Unix domain stream connection to the server; otherwise the client opens a TCP connection to the server.
2. Unix domain sockets are used when passing descriptors between processes on the same host. We provide a complete example of this in Section 14.7.

3. Newer implementations of Unix domain sockets provide the client's credentials (user ID and group IDs) to the server, which can provide additional security checking. We describe this in Section 14.8.

The protocol addresses used to identify clients and servers in the Unix domain are pathnames within the normal filesystem. Recall that IPv4 uses a combination of 32-bit addresses and 16-bit port numbers for its protocol addresses, and IPv6 uses a combination of 128-bit addresses and 16-bit port numbers for its protocol addresses. These pathnames are not normal Unix files: we cannot read from or write to these files except from a program that has associated the pathname with a Unix domain socket.

14.2 Unix Domain Socket Address Structure

Figure 14.1 shows the Unix domain socket address structure, defined by including the `<sys/un.h>` header.

```
struct sockaddr_un {
  uint8_t       sun_len;
  sa_family_t   sun_family;     /* AF_LOCAL */
  char          sun_path[104];  /* null-terminated pathname */
};
```

Figure 14.1 Unix domain socket address structure: `sockaddr_un`.

Earlier BSD releases defined the size of the `sun_path` array as 108 bytes, not the 104 that we show in this figure. Posix.1g requires only that its size be at least 100 bytes. The reason for these limits is an implementation artifact dating back to 4.2BSD requiring that this structure fit in a 128-byte mbuf (a kernel memory buffer).

The pathname stored in the `sun_path` array must be null terminated. The macro `SUN_LEN` is provided and it takes a pointer to a `sockaddr_un` structure and returns the length of the structure, including the number of nonnull bytes in the pathname. The unspecified address is indicated by a null string as the pathname, that is, a structure with `sun_path[0]` equal to 0. This is the Unix domain equivalent of the IPv4 `INADDR_ANY` constant and the IPv6 `IN6ADDR_ANY_INIT` constant.

Posix.1g renames the Unix domain protocols as "local IPC," to remove the dependence on the Unix operating system. The historical constant `AF_UNIX` becomes `AF_LOCAL`. Nevertheless, we still use the term "Unix domain" as that has become its de facto name, regardless of the underlying operating system. Also, even with Posix.1g attempting to make these operating system independent, the socket address structure still retains the _un suffix!

Example: `bind` of Unix Domain Socket

The program in Figure 14.2 creates a Unix domain socket, `bind`s a pathname to it, and then calls `getsockname` and prints the bound pathname.

unixdomain/unixbind.c

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int     sockfd;
6     socklen_t len;
7     struct sockaddr_un addr1, addr2;
8     if (argc != 2)
9         err_quit("usage: unixbind <pathname>");
10    sockfd = Socket(AF_LOCAL, SOCK_STREAM, 0);
11    unlink(argv[1]);            /* OK if this fails */
12    bzero(&addr1, sizeof(addr1));
13    addr1.sun_family = AF_LOCAL;
14    strncpy(addr1.sun_path, argv[1], sizeof(addr1.sun_path) - 1);
15    Bind(sockfd, (SA *) &addr1, SUN_LEN(&addr1));
16    len = sizeof(addr2);
17    Getsockname(sockfd, (SA *) &addr2, &len);
18    printf("bound name = %s, returned len = %d\n", addr2.sun_path, len);
19    exit(0);
20 }
```

unixdomain/unixbind.c

Figure 14.2 `bind` of a pathname to a Unix domain socket.

Remove pathname first

11 The pathname to `bind` to the socket is the command-line argument. But the `bind` will fail if the pathname already exists in the filesystem. Therefore we call `unlink` to delete the pathname, in case it already exists. If it does not exist, `unlink` returns an error, which we ignore.

`bind` and then `getsockname`

12–18 We copy the command-line argument using `strncpy`, to avoid overflowing the structure if the pathname is too long. Since we initialize the structure to 0 and then subtract one from the size of the `sun_path` array, we know the pathname is null terminated. `bind` is called and we use the macro `SUN_LEN` to calculate the length argument for the function. We then call `getsockname` to fetch the name that was just bound and print the result.

If we run this program under BSD/OS we have the following results:

```
bsdi % umask                                    first print our umask value
0002                                            shells print this value in octal
bsdi % unixbind /tmp/foo.bar
bound name = /tmp/foo.bar, returned len = 14
bsdi % unixbind /tmp/foo.bar                    run it again
bound name = /tmp/foo.bar, returned len = 14
bsdi % ls -l /tmp/foo.bar
srwxrwxrwx  1 rstevens  wheel  0 May 20 11:02 /tmp/foo.bar
bsdi % ls -lF /tmp/foo.bar
srwxrwxrwx  1 rstevens  wheel  0 May 20 11:02 /tmp/foo.bar=
```

We first print our `umask` value, because Posix.1g specifies that the file access permissions of the resulting pathname be modified by this value. Our value of 2 turns off the other-write bit (sometimes called world-write). We then run the program and see that the length returned by `getsockname` is 14: 1 byte for the `sun_len` member, 1 byte for the `sun_family` member, and 12 bytes for the actual pathname (excluding the terminating null byte). This is an example of a value–result argument whose result when the function returns differs from its value when the function was called. We can output the pathname using the `%s` format of `printf` because the pathname is null terminated in the `sun_path` member. We then run the program again, to verify that calling `unlink` removes the pathname.

We run `ls -l` to see the file permissions and the file type. Under 4.4BSD the file type is a socket, which is printed as `s`. We also notice that all nine permission bits are on, as 4.4BSD does not modify this default with our `umask` value. Finally we run `ls` again, with the `-F` option, which causes 4.4BSD to append an equals sign to the pathname.

> Posix.2 knows nothing about sockets and only specifies that the `-F` option print a slash for a directory, an asterisk for an executable file, and a vertical bar for a FIFO.

We now run the same program under Solaris 2.5.

```
solaris % umask
02
solaris % unixbind /tmp/foo.bar
bound name = /tmp/foo.bar, returned len = 110
solaris % unixbind /tmp/foo.bar
bound name = /tmp/foo.bar, returned len = 110
solaris % ls -lF /tmp/foo.bar
p---------   1 rstevens other1          0 May 20 11:36 /tmp/foo.bar|
```

The first difference is that the length returned by `getsockname` is 110, the total size of the Solaris `sockaddr_un` structure. This is OK, since the pathname is null terminated in the `sun_path` member. We also notice that the default file permissions are 0: all read, write, and execute permissions are turned off. Since all the permission bits are off, we cannot tell whether our `umask` value was used or not. Finally we notice that `ls -l` indicates that the pathname is a FIFO, which is how all Unix domain sockets appear under SVR4, and the `-F` option prints the vertical bar for a FIFO.

14.3 `socketpair` Function

The `socketpair` function creates two sockets that are then connected together. This function applies to only Unix domain sockets.

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

int socketpair(int family, int type, int protocol, int sockfd[2]);
```

Returns: nonzero if OK, −1 on error

The *family* must be `AF_LOCAL` and the *protocol* must be 0. The *type*, however, can be either `SOCK_STREAM` or `SOCK_DGRAM`. The two socket descriptors that are created are returned as *sockfd[0]* and *sockfd[1]*.

> This function is similar to the Unix `pipe` function: two descriptors are returned, and each descriptor is connected to the other. Indeed, Berkeley-derived implementations implement `pipe` by performing the same internal operations as `socketpair` (pp. 253–254 of TCPv3).

The two created sockets are unnamed; that is, there is no implicit `bind` involved.

The result of `socketpair` with a *type* of `SOCK_STREAM` is called a *stream pipe*. It is similar to a regular Unix pipe (created by the `pipe` function), but a stream pipe is *full-duplex*; that is, both descriptors can be read and written. We show a picture of a stream pipe created by `socketpair` in Figure 14.7.

> Posix.1 does not require full-duplex pipes. On SVR4 `pipe` returns two full-duplex descriptors, while Berkeley-derived kernels traditionally return two half-duplex descriptors (Figure 17.31 of TCPv3).

14.4 Socket Functions

There are several differences and restrictions in the socket functions when using Unix domain sockets. We list the Posix.1g requirements when applicable, and note that not all implementations are currently at this level.

1. The default file access permissions for a pathname created by `bind` should be 0777 (read, write, and execute by user, group, and other), modified by the current `umask` value.
2. The pathname associated with a Unix domain socket should be an absolute pathname, not a relative pathname. The reason to avoid the latter is that its resolution depends on the current working directory of the caller. That is, if the server binds a relative pathname, then the client must be in the same directory as the server (or must know this directory) for the client's call to either `connect` or `sendto` to succeed.

   > Posix.1g says that binding a relative pathname to a Unix domain socket gives unpredictable results.

3. The pathname specified in a call to `connect` must be a pathname that is currently bound to an open Unix domain socket of the same type (stream or datagram). Errors occur if (a) the pathname exists but is not a socket, (b) the pathname exists and is a socket but no open socket descriptor is associated with the pathname, or (c) the pathname exists and is an open socket but is of the wrong type (that is, a Unix domain stream socket cannot connect to a pathname associated with a Unix domain datagram socket, and vice versa).
4. The permission testing associated with the `connect` of a Unix domain socket is the same as if `open` had been called for write-only access to the pathname.

5. Unix domain stream sockets are similar to TCP sockets: they provide a byte stream interface to the process with no record boundaries.
6. If a call to `connect` for a Unix domain stream socket finds that the listening socket's queue is full (Section 4.5), `ECONNREFUSED` is returned immediately. This differs from TCP: the TCP listener ignores an arriving SYN if the socket's queue is full, and the TCP connector retries by sending the SYN several times.
7. Unix domain datagram sockets are similar to UDP sockets: they provide an unreliable datagram service that preserves record boundaries.
8. Unlike UDP sockets, sending a datagram on an unbound Unix domain datagram socket does not bind a pathname to the socket. (Recall that sending a UDP datagram on an unbound UDP socket causes an ephemeral port to be bound to the socket.) This means the receiver of the datagram will be unable to send a reply unless the sender has bound a pathname to its socket. Similarly, unlike TCP and UDP, calling `connect` for a Unix domain datagram socket does not bind a pathname to the socket.

14.5 Unix Domain Stream Client–Server

We now recode our TCP echo client–server from Chapter 5 to use Unix domain sockets. Figure 14.3 shows the server, which is a modification of Figure 5.12 to use the Unix domain stream protocol instead of TCP.

8 The datatype of the two socket address structure is now `sockaddr_un`.

10 The first argument to `socket` is `AF_LOCAL` to create a Unix domain stream socket.

11–15 The constant `UNIXSTR_PATH` is defined in `unp.h` to be `/tmp/unix.str`. We first `unlink` the pathname, in case it exists from an earlier run of the server, and then initialize the socket address structure before calling `bind`. An error from `unlink` is OK.

Notice that this call to `bind` differs from the call in Figure 14.2. Here we specify the size of the socket address structure (the third argument) as the total size of the `sockaddr_un` structure, not just the number of bytes occupied by the pathname. Both lengths are valid, since the pathname must be null terminated.

The remainder of the function is the same as Figure 5.12. The same `str_echo` function is used (Figure 5.3).

Figure 14.4 (p. 380) is the Unix domain stream protocol echo client. It is a modification of Figure 5.4.

6 The socket address structure to contain the server's address is now a `sockaddr_un` structure.

7 The first argument to `socket` is `AF_LOCAL`.

8–10 The code to fill in the socket address structure is identical to the code shown for the server: initialize the structure to 0, set the family to `AF_LOCAL`, and copy the pathname into the `sun_path` member.

12 The function `str_cli` is the same as earlier (Figure 6.13 was the last version that we developed).

unixdomain/unixstrserv01.c

```
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     listenfd, connfd;
 6     pid_t   childpid;
 7     socklen_t clilen;
 8     struct sockaddr_un cliaddr, servaddr;
 9     void    sig_chld(int);

10     listenfd = Socket(AF_LOCAL, SOCK_STREAM, 0);

11     unlink(UNIXSTR_PATH);
12     bzero(&servaddr, sizeof(servaddr));
13     servaddr.sun_family = AF_LOCAL;
14     strcpy(servaddr.sun_path, UNIXSTR_PATH);

15     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

16     Listen(listenfd, LISTENQ);

17     Signal(SIGCHLD, sig_chld);

18     for ( ; ; ) {
19         clilen = sizeof(cliaddr);
20         if ( (connfd = accept(listenfd, (SA *) &cliaddr, &clilen)) < 0) {
21             if (errno == EINTR)
22                 continue;       /* back to for() */
23             else
24                 err_sys("accept error");
25         }
26         if ( (childpid = Fork()) == 0) {     /* child process */
27             Close(listenfd);    /* close listening socket */
28             str_echo(connfd);   /* process the request */
29             exit(0);
30         }
31         Close(connfd);          /* parent closes connected socket */
32     }
33 }
```

unixdomain/unixstrserv01.c

Figure 14.3 Unix domain stream protocol echo server.

14.6 Unix Domain Datagram Client–Server

We now recode our UDP client–server from Sections 8.3 and 8.5 to use Unix domain datagram sockets. Figure 14.5 shows the server, which is a modification of Figure 8.3.

unixdomain/unixstrcli01.c

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int     sockfd;
6     struct sockaddr_un servaddr;
7     sockfd = Socket(AF_LOCAL, SOCK_STREAM, 0);
8     bzero(&servaddr, sizeof(servaddr));
9     servaddr.sun_family = AF_LOCAL;
10    strcpy(servaddr.sun_path, UNIXSTR_PATH);
11    Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
12    str_cli(stdin, sockfd);     /* do it all */
13    exit(0);
14 }
```

unixdomain/unixstrcli01.c

Figure 14.4 Unix domain stream protocol echo client.

unixdomain/unixdgserv01.c

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int     sockfd;
6     struct sockaddr_un servaddr, cliaddr;
7     sockfd = Socket(AF_LOCAL, SOCK_DGRAM, 0);
8     unlink(UNIXDG_PATH);
9     bzero(&servaddr, sizeof(servaddr));
10    servaddr.sun_family = AF_LOCAL;
11    strcpy(servaddr.sun_path, UNIXDG_PATH);
12    Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));
13    dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
14 }
```

unixdomain/unixdgserv01.c

Figure 14.5 Unix domain datagram protocol echo server.

6 The datatype of the two socket address structures is now `sockaddr_un`.

7 The first argument to `socket` is `AF_LOCAL` to create a Unix domain datagram socket.

8–12 The constant `UNIXDG_PATH` is defined in `unp.h` to be `/tmp/unix.dg`. We first `unlink` the pathname, in case it exists from an earlier run of the server, and then initialize the socket address structure before calling `bind`. An error from `unlink` is OK.

13 The same `dg_echo` function is used (Figure 8.4).

Figure 14.6 is the Unix domain datagram protocol echo client. It is a modification of Figure 8.7.

unixdomain/unixdgcli01.c

```
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd;
 6     struct sockaddr_un cliaddr, servaddr;

 7     sockfd = Socket(AF_LOCAL, SOCK_DGRAM, 0);

 8     bzero(&cliaddr, sizeof(cliaddr));   /* bind an address for us */
 9     cliaddr.sun_family = AF_LOCAL;
10     strcpy(cliaddr.sun_path, tmpnam(NULL));

11     Bind(sockfd, (SA *) &cliaddr, sizeof(cliaddr));

12     bzero(&servaddr, sizeof(servaddr));     /* fill in server's address */
13     servaddr.sun_family = AF_LOCAL;
14     strcpy(servaddr.sun_path, UNIXDG_PATH);

15     dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr));

16     exit(0);
17 }
```

unixdomain/unixdgcli01.c

Figure 14.6 Unix domain datagram protocol echo client.

6 The socket address structure to contain the server's address is now a `sockaddr_un` structure. We also allocate one of these structures to contain the client's address, which we talk about shortly.

7 The first argument to `socket` is `AF_LOCAL`.

8–11 Unlike our UDP client, when using the Unix domain datagram protocol we must explicitly `bind` a pathname to our socket, so that the server has a pathname to which it can send its reply. We call `tmpnam` to assign a unique pathname that we then `bind` to our socket. Recall from Section 14.4 that sending a datagram on an unbound Unix domain datagram socket does not implicitly bind a pathname to the socket. Therefore if we omit this step, the server's call to `recvfrom` in the `dg_echo` function returns a null pathname, which then causes an error when the server calls `sendto`.

12–14 The code to fill in the socket address structure with the server's well-known pathname is identical to the code shown earlier for the server.

15 The function `dg_cli` is the same as earlier (Figure 8.8).

14.7 Passing Descriptors

When we think of passing an open descriptor from one process to another, we normally think of either

1. a child sharing all the open descriptors with the parent after a call to `fork`, or
2. all descriptors normally remaining open when `exec` is called.

In the first example the process opens a descriptor, calls `fork`, and then the parent closes the descriptor, letting the child handle the descriptor. This passes an open descriptor from the parent to the child. But we would also like the ability for the child to open a descriptor and pass it back to the parent.

Current Unix systems provide a way to pass any open descriptor from one process to any other process. That is, there is no need for the processes to be related, such as a parent and its child. The technique requires us to first establish a Unix domain socket between the two processes and then use `sendmsg` to send a special message across the Unix domain socket. This message is handled specially by the kernel, passing the open descriptor from the sender to the receiver.

> The black magic performed by the 4.4BSD kernel in passing an open descriptor across a Unix domain socket is described in detail in Chapter 18 of TCPv3.
>
> SVR4 uses a different technique within the kernel to pass an open descriptor, the `I_SENDFD` and `I_RECVFD` `ioctl` commands, described in Section 15.5.1 of APUE. But the process can still access this kernel feature using a Unix domain socket. In this text we describe the use of Unix domain sockets to pass open descriptors, since this is the most portable programming technique: it works under both Berkeley-derived kernels and SVR4, whereas using the `I_SENDFD` and `I_RECVFD` `ioctl`s works only under SVR4.
>
> The 4.4BSD technique allows multiple descriptors to be passed with a single `sendmsg`, whereas the SVR4 technique passes only a single descriptor at a time. All our examples pass one descriptor at a time.

The steps involved in passing a descriptor between two processes are then:

1. Create a Unix domain socket, either a stream socket or a datagram socket.

   If the goal is to `fork` a child, have the child open the descriptor and pass the descriptor back to the parent, the parent can call `socketpair` to create a stream pipe that can be used to exchange the descriptor.

   If the processes are unrelated, then the server must create a Unix domain stream socket, `bind` a pathname to it, allowing the client to `connect` to that socket. The client can then send a request to the server to open some descriptor, and the server can pass back the descriptor across the Unix domain socket. Alternately, a Unix domain datagram socket can also be used between the client and server, but there is little advantage in doing this, and the possibility exists for a datagram to be discarded. We will use a stream socket between the client and server in our example later in this section.

2. One process opens a descriptor by calling any of the Unix functions that returns a descriptor: `open`, `pipe`, `mkfifo`, `socket`, or `accept`, for example. *Any* type of descriptor can be passed from one process to another, which is why we call the technique "descriptor passing" and not "file descriptor passing."

3. The sending process builds a `msghdr` structure (Section 13.5) containing the descriptor to be passed. Posix.1g specifies that the descriptor be sent as ancillary data (the `msg_control` member of the `msghdr` structure, Section 13.6), but older implementations use the `msg_accrights` member. The sending process calls `sendmsg` to send the descriptor across the Unix domain socket from step 1. At this point we say that the descriptor is "in flight." Even if the sending process closes the descriptor, after calling `sendmsg`, but before the receiving process calls `recvmsg` (in the next step), the descriptor remains open for the receiving process. Sending a descriptor increments the descriptor's reference count by one.
4. The receiving process calls `recvmsg` to receive the descriptor on the Unix domain socket from step 1. It is normal for the descriptor number in the receiving process to differ from the descriptor number in the sending process, but that is OK. Passing a descriptor is not passing a descriptor number, but involves creating a new descriptor in the receiving process that refers to the same file table entry within the kernel as the descriptor that was sent by the sending process.

The client and server must have some application protocol so that the receiver of the descriptor knows when to expect it. If the receiver calls `recvmsg` without allocating room to receive the descriptor, and a descriptor was passed and is ready to be read, the descriptor that was being passed is closed (p. 518 of TCPv2). Also, the `MSG_PEEK` flag should be avoided with `recvmsg` if a descriptor is expected, as the result is unpredictable.

Descriptor Passing Example

We now provide an example of descriptor passing. We will write a program named `mycat` that takes a pathname as a command-line argument, opens the file, and copies it to standard output. But instead of calling the normal Unix `open` function, we call our own function named `my_open`. This function creates a stream pipe and calls `fork` and `exec` to initiate another program that opens the desired file. This program must then pass the open descriptor back to the parent across the stream pipe.

Figure 14.7 shows the first step: our `mycat` program after creating a stream pipe by calling `socketpair`. We designate the two descriptors returned by `socketpair` as `[0]` and `[1]`.

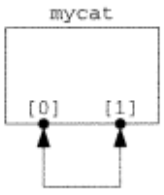

Figure 14.7 `mycat` program after creating stream pipe using `socketpair`.

The process then calls `fork` and the child calls `exec` to execute the `openfile` program. The parent closes the `[1]` descriptor and the child closes the `[0]` descriptor.

(There is no difference in either end of the stream pipe. The child could close [1] and the parent could close [0].) This gives us the arrangement shown in Figure 14.8.

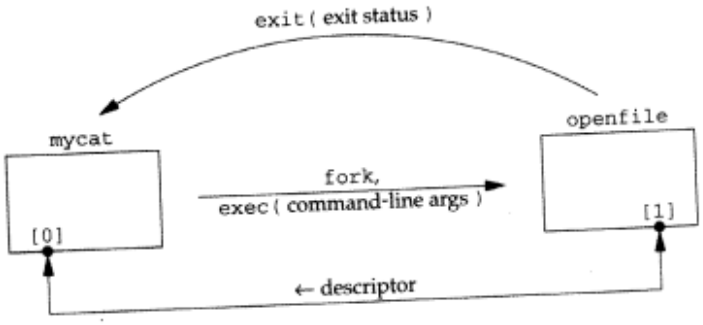

Figure 14.8 mycat program after invoking openfile program.

The parent must pass three pieces of information to the openfile program: (1) the pathname of the file to open, (2) the open mode (read-only, read–write, or write-only), and (3) the descriptor number corresponding to its end of the stream pipe (what we show as [1]). We choose to pass these three items as command-line arguments in the call to exec. An alternative method is to send these three items as data across the stream pipe. The openfile program sends back the open descriptor across the stream pipe and terminates. The exit status of the program tells the parent whether the file could be opened, and if not, what type of error occurred.

The advantage in executing another program to open the file is that the program could be set-user-ID to root, allowing it to open files that we normally do not have permission to open. This program could extend the concept of normal Unix permissions (user, group, and other) to any form of access checking that it desires.

We begin with the mycat program, shown in Figure 14.9.

unixdomain/mycat.c

```
1 #include    "unp.h"
2 int     my_open(const char *, int);
3 int
4 main(int argc, char **argv)
5 {
6     int     fd, n;
7     char    buff[BUFFSIZE];
8     if (argc != 2)
9         err_quit("usage: mycat <pathname>");
10    if ( (fd = my_open(argv[1], O_RDONLY)) < 0)
11        err_sys("cannot open %s", argv[1]);
12    while ( (n = Read(fd, buff, BUFFSIZE)) > 0)
13        Write(STDOUT_FILENO, buff, n);
```

```
14      exit(0);
15  }
```
unixdomain/mycat.c

Figure 14.9 `mycat` program: copy a file to standard output.

If we replace the call to `my_open` with a call to `open`, this simple program just copies a file to standard output.

The function `my_open`, shown in Figure 14.10 is intended to look like the normal Unix `open` function to its caller. It takes two arguments, a pathname and an open mode (such as `O_RDONLY` to mean read-only), opens the file, and returns a descriptor.

unixdomain/myopen.c

```
 1 #include    "unp.h"

 2 int
 3 my_open(const char *pathname, int mode)
 4 {
 5     int     fd, sockfd[2], status;
 6     pid_t   childpid;
 7     char    c, argsockfd[10], argmode[10];

 8     Socketpair(AF_LOCAL, SOCK_STREAM, 0, sockfd);

 9     if ( (childpid = Fork()) == 0) {      /* child process */
10         Close(sockfd[0]);
11         snprintf(argsockfd, sizeof(argsockfd), "%d", sockfd[1]);
12         snprintf(argmode, sizeof(argmode), "%d", mode);
13         execl("./openfile", "openfile", argsockfd, pathname, argmode,
14               (char *) NULL);
15         err_sys("execl error");
16     }
17     /* parent process - wait for the child to terminate */
18     Close(sockfd[1]);           /* close the end we don't use */

19     Waitpid(childpid, &status, 0);
20     if (WIFEXITED(status) == 0)
21         err_quit("child did not terminate");
22     if ( (status = WEXITSTATUS(status)) == 0)
23         Read_fd(sockfd[0], &c, 1, &fd);
24     else {
25         errno = status;         /* set errno value from child's status */
26         fd = -1;
27     }

28     Close(sockfd[0]);
29     return (fd);
30 }
```
unixdomain/myopen.c

Figure 14.10 `my_open` function: open a file and return a descriptor.

Create stream pipe

8 `socketpair` creates a stream pipe. Two descriptors are returned: `sockfd[0]` and `sockfd[1]`. This is the state that we show in Figure 14.7.

fork and exec

9-16 `fork` is called and the child then closes one end of the stream pipe. The descriptor number of the other end of the stream pipe is formatted into the `argsockfd` array and the open mode is formatted into the `argmode` array. We call `snprintf` because the arguments to `exec` must be character strings. The `openfile` program is executed. The `execl` function should not return unless it encounters an error. On success the `main` function of the `openfile` program starts executing.

Parent waits for child

17-22 The parent closes the other end of the stream pipe and calls `waitpid` to wait for the child to terminate. The termination status of the child is returned in the variable `status` and we first verify that the program terminated normally (i.e., it was not terminated by a signal). The `WEXITSTATUS` macro then converts the termination status into the exit status, whose value will be between 0 and 255. We will see shortly that if the `openfile` program encounters an error opening the requested file, it terminates with the corresponding `errno` value as its exit status.

Receive descriptor

23 Our function `read_fd`, shown next, receives the descriptor on the stream pipe. In addition to the descriptor, we read 1 byte of data, but do nothing with the data.

> When sending and receiving a descriptor across a stream pipe, we always send at least 1 byte of data, even if the receiver does nothing with the data. Otherwise the receiver cannot tell whether a return value of 0 from `read_fd` means "no data (but possibly a descriptor)" or "end-of-file."

Figure 14.11 shows the `read_fd` function, which calls `recvmsg` to receive data and a descriptor on a Unix domain socket. The first three arguments to this function are the same as for the `read` function, with a fourth argument being a pointer to an integer that will contain the received descriptor on return.

9-26 This function must deal with two versions of `recvmsg`: those with the `msg_control` member and those with the `msg_accrights` member. Our `config.h` header (Figure D.2) defines the constant `HAVE_MSGHDR_MSG_CONTROL` if the `msg_control` version is supported.

Make certain `msg_control` is suitably aligned

10-13 The `msg_control` buffer must be suitably aligned for a `cmsghdr` structure. Simply allocating a `char` array is inadequate. Here we declare a `union` of a `cmsghdr` structure with the character array, which guarantees that the array is suitably aligned. Another technique is to call `malloc` but that would require freeing the memory before the function returns.

27-45 `recvmsg` is called. If ancillary data is returned, the format is as shown in Figure 13.13. We verify that the length, level, and type are correct, then fetch the newly created descriptor, and return it through the caller's `recvfd` pointer. `CMSG_DATA` returns the pointer to the `cmsg_data` member of the ancillary data object as an `unsigned char` pointer. We cast this to an `int` pointer and fetch the integer descriptor that is pointed to.

lib/read_fd.c

```
 1 #include    "unp.h"

 2 ssize_t
 3 read_fd(int fd, void *ptr, size_t nbytes, int *recvfd)
 4 {
 5     struct msghdr msg;
 6     struct iovec iov[1];
 7     ssize_t n;
 8     int     newfd;

 9 #ifdef  HAVE_MSGHDR_MSG_CONTROL
10     union {
11         struct cmsghdr cm;
12         char    control[CMSG_SPACE(sizeof(int))];
13     } control_un;
14     struct cmsghdr *cmptr;

15     msg.msg_control = control_un.control;
16     msg.msg_controllen = sizeof(control_un.control);
17 #else
18     msg.msg_accrights = (caddr_t) & newfd;
19     msg.msg_accrightslen = sizeof(int);
20 #endif

21     msg.msg_name = NULL;
22     msg.msg_namelen = 0;

23     iov[0].iov_base = ptr;
24     iov[0].iov_len = nbytes;
25     msg.msg_iov = iov;
26     msg.msg_iovlen = 1;

27     if ( (n = recvmsg(fd, &msg, 0)) <= 0)
28         return (n);

29 #ifdef  HAVE_MSGHDR_MSG_CONTROL
30     if ( (cmptr = CMSG_FIRSTHDR(&msg)) != NULL &&
31         cmptr->cmsg_len == CMSG_LEN(sizeof(int))) {
32         if (cmptr->cmsg_level != SOL_SOCKET)
33             err_quit("control level != SOL_SOCKET");
34         if (cmptr->cmsg_type != SCM_RIGHTS)
35             err_quit("control type != SCM_RIGHTS");
36         *recvfd = *((int *) CMSG_DATA(cmptr));
37     } else
38         *recvfd = -1;           /* descriptor was not passed */
39 #else
40     if (msg.msg_accrightslen == sizeof(int))
41         *recvfd = newfd;
42     else
43         *recvfd = -1;       /* descriptor was not passed */

44 #endif

45     return (n);
46 }
```

lib/read_fd.c

Figure 14.11 `read_fd` function: receive data and a descriptor.

If the older `msg_accrights` member is supported, the length should be the size of an integer and the newly created descriptor is returned through the caller's `recvfd` pointer.

Figure 14.12 shows the `openfile` program. It takes the three command-line arguments that must be passed and calls the normal `open` function.

unixdomain/openfile.c

```
1 #include    "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int     fd;
6     ssize_t n;

7     if (argc != 4)
8         err_quit("openfile <sockfd#> <filename> <mode>");

9     if ( (fd = open(argv[2], atoi(argv[3]))) < 0)
10        exit((errno > 0) ? errno : 255);

11    if ( (n = write_fd(atoi(argv[1]), "", 1, fd)) < 0)
12        exit((errno > 0) ? errno : 255);

13    exit(0);
14 }
```

unixdomain/openfile.c

Figure 14.12 `openfile` function: open a file and pass back the descriptor.

Command-line arguments

7-12 Since two of the three command-line arguments were formatted into character strings by `my_open`, two are converted back into integers using `atoi`.

open the file

9-10 The file is opened by calling `open`. If an error is encountered, the `errno` value corresponding to the `open` error is returned as the exit status of the process.

Pass back descriptor

11-12 The descriptor is passed back by `write_fd`, which we show next. This process then terminates, but recall that earlier in the chapter we said that it is OK for the sending process to close the descriptor that was passed (which happens when we call `exit`) because the kernel knows that the descriptor is in flight, and keeps it open for the receiving process.

> The exit status must be between 0 and 255. The highest `errno` value is around 150. An alternate technique that doesn't require the `errno` values to be less than 256 would be to pass back an error indication as normal data in the call to `sendmsg`.

Figure 14.13 shows the final function, `write_fd`, which calls `sendmsg` to send a descriptor (and optional data, which we do not use) across a Unix domain socket.

lib/write_fd.c

```
 1 #include    "unp.h"

 2 ssize_t
 3 write_fd(int fd, void *ptr, size_t nbytes, int sendfd)
 4 {
 5     struct msghdr msg;
 6     struct iovec iov[1];

 7 #ifdef  HAVE_MSGHDR_MSG_CONTROL
 8     union {
 9         struct cmsghdr cm;
10         char    control[CMSG_SPACE(sizeof(int))];
11     } control_un;
12     struct cmsghdr *cmptr;

13     msg.msg_control = control_un.control;
14     msg.msg_controllen = sizeof(control_un.control);

15     cmptr = CMSG_FIRSTHDR(&msg);
16     cmptr->cmsg_len = CMSG_LEN(sizeof(int));
17     cmptr->cmsg_level = SOL_SOCKET;
18     cmptr->cmsg_type = SCM_RIGHTS;
19     *((int *) CMSG_DATA(cmptr)) = sendfd;
20 #else
21     msg.msg_accrights = (caddr_t) & sendfd;
22     msg.msg_accrightslen = sizeof(int);
23 #endif

24     msg.msg_name = NULL;
25     msg.msg_namelen = 0;

26     iov[0].iov_base = ptr;
27     iov[0].iov_len = nbytes;
28     msg.msg_iov = iov;
29     msg.msg_iovlen = 1;

30     return (sendmsg(fd, &msg, 0));
31 }
```

lib/write_fd.c

Figure 14.13 `write_fd` function: pass a descriptor by calling `sendmsg`.

As with `read_fd`, this function must deal with either ancillary data or the older access rights. In either case the `msghdr` structure is initialized and then `sendmsg` is called.

We show an example of descriptor passing in Section 25.7 that involves unrelated processes and an example in Section 27.9 that involves related processes. We will use the `read_fd` and `write_fd` functions that we just described.

14.8 Receiving Sender Credentials

In Figure 13.13 we showed another type of data that can be passed along a Unix domain socket as ancillary data: user credentials via the `fcred` structure, which is defined by including the `<sys/ucred.h>` header.

```
struct fcred {
  uid_t  fc_ruid;              /* real user ID */
  gid_t  fc_rgid;              /* real group ID */
  char   fc_login[MAXLOGNAME]; /* setlogin() name */
  uid_t  fc_uid;               /* effective user ID */
  short  fc_ngroups;           /* number of groups */
  gid_t  fc_groups[NGROUPS];   /* supplementary group IDs */
};
#define  fc_gid  fc_groups[0]  /* effective group ID */
```

Normally `MAXLOGNAME` is 16, and `NGROUPS` is also 16. `fc_ngroups` is always at least 1, with the first element of the array the effective group ID.

> This structure is actually defined as a `ucred` structure within the `fcred` structure, with `#defined` names to make it appear as a single structure. We show the "logical" definition of the structure.
>
> This feature is new with BSD/OS 2.1. We describe it, even though it is not widespread, because it is an important, yet simple, addition to the Unix domain protocols. When a client and server communicate using these protocols, the server often needs a way to know exactly who the client is, to validate that the client has permission to ask for the service being requested.

This information is always available on a Unix domain socket, subject to the following conditions:

- The credentials are sent as ancillary data when data is sent on the Unix domain socket, but only if the receiver of the data has enabled the `LOCAL_CREDS` socket option. The *level* for this option (Section 7.2) is 0.
- On a datagram socket, the credentials accompany every datagram. On a stream socket, the credentials are sent only once, the first time data is sent.
- Credentials cannot be sent along with a descriptor. That is, only one of the two types of ancillary data can be sent with a given message.

  > This is a limitation of the BSD/OS implementation. In general, we should be able to pass multiple types of ancillary data in a single call to `sendmsg`, since each ancillary data object has its own type and length (Figure 13.12).

- Users are not able to forge credentials. That is, when ancillary data is sent on a Unix domain socket, the kernel verifies that the ancillary data is not of level `SOL_SOCKET` and not of type `SCM_CREDS`. If the sender tries to forge its own credentials, the ancillary data is discarded by the kernel.

One point that we did not mention in the previous section is that when a descriptor is received under SVR4 (using `ioctl` with the `I_RECVFD` command), the kernel also passes the sender's credentials to the receiving process: a `strrecvfd` structure containing the newly created descriptor, the effective user ID, and the effective group ID. This is a form of credential passing that takes place every time a descriptor is passed. Additionally, when client–server connections are created under SVR4 using the `connld` streams module (similar to the creation of a new socket by `accept` on a Unix domain socket), the new descriptor is passed along with a `strrecvfd` structure containing the client's credentials. Under SVR4 we are not able to access these credentials using Unix domain sockets. (Section 15.5.1 of APUE details the use of the SVR4 `connld` streams module.) If a Berkeley-derived implementation does not support the new credential passing that we describe in this section, there is no guaranteed way for a Unix domain server to obtain the client's credentials. Section 15.5.2 of APUE details one work-around to obtain this information, but user credentials should always be supplied by the kernel.

Example

As an example of credential passing, we modify our Unix domain stream server to ask for the client's credentials. Figure 14.14 shows a new function, `read_cred`, that is similar to `read` but also returns an `fcred` structure containing the sender's credentials.

4–5 The first three arguments are identical to `read`, with the fourth argument being a pointer to an `fcred` structure that will be filled in. The format of the returned ancillary data is shown in Figure 13.13.

26–36 If credentials were returned, the length, level, and type of the ancillary data are verified, and the resulting structure copied back to the caller. If no credentials were returned, we set the structure to 0. Since the number of groups (`fc_ngroups`) is always one or more, its value of 0 indicates to the caller that no credentials were returned by the kernel.

The `main` function for our echo server, Figure 14.3, is unchanged. Figure 14.15 (p. 393) shows the new version of the `str_echo` function, modified from Figure 5.3. This function is called by the child after the parent has accepted a new client connection and called `fork`.

12 The `LOCAL_CREDS` socket option is enabled for the connected socket.

13–14 Our function `read_cred` is called the first time. We specify a length of 0 as we do not want any data; we want only the ancillary data.

17–27 If credentials were returned, they are printed.

28–32 The remainder of the loop is unchanged. This code reads lines from the client and writes them back to the client.

Our client from Figure 14.4 is unchanged.

unixdomain/readcred.c

```
 1 #include    "unp.h"
 2 #include    <sys/param.h>
 3 #include    <sys/ucred.h>

 4 ssize_t
 5 read_cred(int fd, void *ptr, size_t nbytes, struct fcred *fcredptr)
 6 {
 7     struct msghdr msg;
 8     struct iovec iov[1];
 9     ssize_t n;

10     union {
11         struct cmsghdr cm;
12         char    control[CMSG_SPACE(sizeof(struct fcred))];
13     } control_un;
14     struct cmsghdr *cmptr;

15     msg.msg_control = control_un.control;
16     msg.msg_controllen = sizeof(control_un.control);

17     msg.msg_name = NULL;
18     msg.msg_namelen = 0;

19     iov[0].iov_base = ptr;
20     iov[0].iov_len = nbytes;
21     msg.msg_iov = iov;
22     msg.msg_iovlen = 1;

23     if ( (n = recvmsg(fd, &msg, 0)) < 0)
24         return (n);
25     if (fcredptr) {
26         if (msg.msg_controllen > sizeof(struct cmsghdr)) {
27             cmptr = CMSG_FIRSTHDR(&msg);

28             if (cmptr->cmsg_len != CMSG_LEN(sizeof(struct fcred)))
29                 err_quit("control length = %d", cmptr->cmsg_len);
30             if (cmptr->cmsg_level != SOL_SOCKET)
31                 err_quit("control level != SOL_SOCKET");
32             if (cmptr->cmsg_type != SCM_CREDS)
33                 err_quit("control type != SCM_CREDS");
34             memcpy(fcredptr, CMSG_DATA(cmptr), sizeof(struct fcred));
35         } else
36             bzero(fcredptr, sizeof(struct fcred)); /* none returned */
37     }

38     return (n);
39 }
```

unixdomain/readcred.c

Figure 14.14 `read_cred` function: read and return sender's credentials.

unixdomain/strecho.c

```
 1 #include    "unp.h"
 2 #include    <sys/param.h>
 3 #include    <sys/ucred.h>

 4 ssize_t read_cred(int, void *, size_t, struct fcred *);

 5 void
 6 str_echo(int sockfd)
 7 {
 8     ssize_t n;
 9     const int on = 1;
10     char    line[MAXLINE];
11     struct fcred cred;

12     Setsockopt(sockfd, 0, LOCAL_CREDS, &on, sizeof(on));

13     if ( (n = read_cred(sockfd, NULL, 0, &cred)) < 0)
14         err_sys("read_cred error");
15     if (cred.fc_ngroups == 0)
16         printf("(no credentials returned)\n");
17     else {
18         printf("real user ID = %d\n", cred.fc_ruid);
19         printf("real group ID = %d\n", cred.fc_rgid);
20         printf("login name = %-*s\n", MAXLOGNAME, cred.fc_login);
21         printf("effective user ID = %d\n", cred.fc_uid);
22         printf("effective group ID = %d\n", cred.fc_gid);
23         printf("%d supplementary groups:", cred.fc_ngroups - 1);
24         for (n = 1; n < cred.fc_ngroups; n++)   /* [0] is the egid */
25             printf(" %d", cred.fc_groups[n]);
26         printf("\n");
27     }

28     for ( ; ; ) {
29         if ( (n = Readline(sockfd, line, MAXLINE)) == 0)
30             return;             /* connection closed by other end */

31         Writen(sockfd, line, n);
32     }
33 }
```

unixdomain/strecho.c

Figure 14.15 `str_echo` function that asks for client's credentials.

If we run the server in one window, and the client in another, here is the output from the server after running the client one time.

```
bsdi % unixstrserv02
real user ID = 482
real group ID = 52
login name = rstevens
effective user ID = 482
effective group ID = 52
7 supplementary groups: 20 0 1 2 3 5 7
```

This information is output only after the client has sent the first line for the server to echo, because we noted earlier that the credentials are sent by the kernel on a stream socket the first time data is sent on the socket (not when the connection is established). This differs from the SVR4 technique mentioned earlier, where the credentials are sent with the descriptor when the newly created descriptor is returned (which would be the equivalent of `accept` returning for the Unix domain socket).

14.9 Summary

Unix domain sockets are an alternative to IPC when the client and server are on the same host. The advantage in using Unix domain sockets over some form of IPC is that the API is nearly identical to a networked client–server. The advantage in using Unix domain sockets over TCP, when the client and server are on the same host, is the increased performance of Unix domain sockets over TCP on many implementations.

We modified our TCP and UDP echo client and echo servers to use the Unix domain protocols and the only major difference was having to `bind` a pathname to the UDP client's socket, so that the UDP server has somewhere to send the replies. Our code in Sections 14.5 and 14.6 manipulated the Unix domain socket address structures directly, but a better approach is to use the `tcp_XXX` and `udp_XXX` functions from Chapter 11, since our implementation of `getaddrinfo` supports Unix domain sockets.

Descriptor passing is a powerful technique between clients and servers on the same host and it takes place across a Unix domain socket. We showed an example in Section 14.7 that passed a descriptor from a child back to the parent. In Section 25.7 we show an example in which the client and server are unrelated, and in Section 27.9 we show another example that passes a descriptor from a parent to a child.

Exercises

14.1 What happens if a Unix domain server calls `unlink` after calling `bind`?

14.2 What happens if a Unix domain server does not `unlink` its well-known pathname when it terminates, and a client tries to `connect` to the server sometime after the server terminates?

14.3 Compile Figures 11.12 and 11.15, our protocol-independent UDP daytime client and server, and run them specifying a Unix domain socket (Section 11.6). What happens? How can you fix this?

14.4 Start with Figure 11.7 and modify it to call `sleep(5)` after the peer's protocol address is printed, and to also print the number of bytes returned by `read` each time `read` returns a positive value. Start with Figure 11.10 and modify it to call `write` for each byte of the result that is sent to the client. (We discussed similar modifications in the solution to Exercise 1.5.) Run the client and server on the same host using TCP. How many bytes are `read` by the client?

Run the client and server on the same host using a Unix domain socket. Does anything change?

Now call `send` instead of `write` in the server and specify the `MSG_EOR` flag. (You need a Berkeley-derived implementation to finish this exercise.) Run the client and server on the same host using a Unix domain socket. Does anything change?

14.5 Write a program to determine the values shown in Figure 4.10. One approach is to create a stream pipe and then `fork` into a parent and child. The parent enters a `for` loop, incrementing the backlog from 0 through 14. Each time through the loop the parent first writes the value of the backlog to the stream pipe. The child reads this value, creates a listening socket bound to the loopback address, and sets the backlog to that value. The child then writes to the stream pipe, just to tell the parent that it is ready. The parent then tries to establish as many connections as possible, but with an `alarm` set to 2 seconds, because the call to `connect` that hits the backlog limit will block, resending the SYN. The child never calls `accept`, to let the kernel queue the connections from the parent. When the parent's `alarm` expires, it knows from the loop counter which `connect` hit the backlog limit. The parent then closes its sockets and writes the next new backlog value to the stream pipe for the child. When the child reads this next value it closes its listening socket and creates a new listening socket, starting the procedure again.

14.6 Verify that omitting the call to `bind` in Figure 14.6 causes an error in the server.

15

Nonblocking I/O

15.1 Introduction

By default, sockets are blocking. This means that when we issue a socket call that cannot be completed immediately, our process is put to sleep, waiting for the condition to be true. We can divide the socket calls that may block into four categories.

1. Input operations: the `read`, `readv`, `recv`, `recvfrom`, and `recvmsg` functions. If we call one of these input functions for a blocking TCP socket (the default), and there is no data available in the socket receive buffer, we are put to sleep until some data arrives. Since TCP is a byte stream, we will be awakened when "some" data arrives: it could be a single byte of data, or it could be a full TCP segment of data. If we want to wait until some fixed amount of data is available, we can call our own function `readn` (Figure 3.14) or specify the `MSG_WAITALL` flag (Figure 13.6).

   Since UDP is a datagram protocol, if the socket receive buffer is empty for a blocking UDP socket, we are put to sleep until a UDP datagram arrives.

   With a nonblocking socket, if the input operation cannot be satisfied (at least 1 byte of data for a TCP socket or a complete datagram for a UDP socket), return is made immediately with an error of `EWOULDBLOCK`.

2. Output operations: the `write`, `writev`, `send`, `sendto`, and `sendmsg` functions. For a TCP socket we said in Section 2.9 that the kernel copies data from the application's buffer into the socket send buffer. If there is no room in the socket send buffer for a blocking socket, the process is put to sleep until there is room.

With a nonblocking TCP socket, if there is no room at all in the socket send buffer, return is made immediately with an error of EWOULDBLOCK. If there is some room in the socket send buffer, the return value will be the number of bytes that the kernel was able to copy into the buffer. (This is called a *short count*.)

We also said in Section 2.9 that there is no actual UDP socket send buffer. The kernel just copies the application data and moves it down the stack, prepending the UDP and IP headers. Therefore an output operation on a blocking UDP socket (the default) should never block.

3. Accepting incoming connections: the accept function. If accept is called for a blocking socket and a new connection is not available, the process is put to sleep.

   If accept is called for a nonblocking socket and a new connection is not available, the error EWOULDBLOCK is returned instead.

4. Initiating outgoing connections: the connect function for TCP. (Recall that connect can be used with UDP but it does not cause a "real" connection to be established; it just causes the kernel to store the peer's IP address and port number.) We showed in Section 2.5 that the establishment of a TCP connection involves a three-way handshake and the connect function does not return until the client receives the ACK of its SYN. This means that a TCP connect always blocks the calling process for at least the round-trip time (RTT) to the server.

   If connect is called for a nonblocking TCP socket and the connection cannot be established immediately, the connection establishment is initiated (e.g., the first packet of TCP's three-way handshake is sent) but the error EINPROGRESS is returned. Notice that this error differs from the error returned in the previous three scenarios. Also notice that some connections can be established immediately, normally when the server is on the same host as the client, so even with a nonblocking connect we must be prepared for connect to return OK. We show an example of a nonblocking connect in Section 15.3.

> Traditionally, System V has returned the error EAGAIN for a nonblocking I/O operation that cannot be satisfied while Berkeley-derived implementations have returned the error EWOULDBLOCK. To confuse things even more, Posix.1 specifies that EAGAIN is used while Posix.1g specifies that EWOULDBLOCK is used. Fortunately, most current systems (including SVR4 and 4.4BSD) define these two error codes to be the same (check your system's <sys/errno.h> header), so it doesn't matter which one we use. In this text we use EWOULDBLOCK, as specified by Posix.1g.

Section 6.2 summarized the different models available for I/O and compares nonblocking I/O to other models. In this chapter we provide examples of all four types of operations, and develop a new type of client, similar to a Web client, that initiates multiple TCP connections at the same time, using a nonblocking connect.

15.2 Nonblocking Reads and Writes: str_cli Function (Revisited)

We return once again to our str_cli function, which we have discussed in Sections 5.5 and 6.4. The latter version, which uses select, still uses blocking I/O. For example, if a line is available on standard input, we read it with fgets and then send it to the server with writen. But the call to writen can block, if the socket send buffer is full. While we are blocked in the call to writen, data could be available for reading from the socket receive buffer. Similarly, if a line of input is available from the socket we can block in the subsequent call to fputs, if standard output is slower than the network. Our goal in this section is to develop a version of this function that uses nonblocking I/O. This prevents us from blocking while we could be doing something productive.

Unfortunately the addition of nonblocking I/O complicates the function's buffer management noticeably, so we will present the function in pieces. We also change our handling of standard input and standard output to use read and write directly, instead of using standard I/O. This avoids using standard I/O with nonblocking descriptors, a recipe for disaster.

We maintain two buffers: to contains data going from standard input to the server, and fr contains data arriving from the server going to standard output. Figure 15.1 shows the arrangement of the to buffer and the pointers into the buffer.

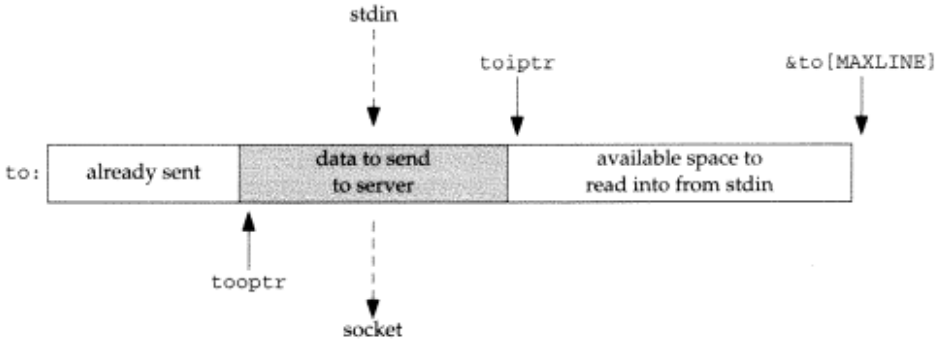

Figure 15.1 Buffer containing data from standard input going to the socket.

The pointer toiptr points to the next byte into which data can be read from standard input. tooptr points to the next byte that must be written to the socket. There are toiptr minus tooptr bytes to be written to the socket. The number of bytes that can be read from standard input is &to[MAXLINE] minus toiptr. As soon as tooptr reaches toiptr, both pointers are reset to the beginning of the buffer.

Figure 15.2 shows the corresponding arrangement of the fr buffer.

Figure 15.3 shows the first part of the function.

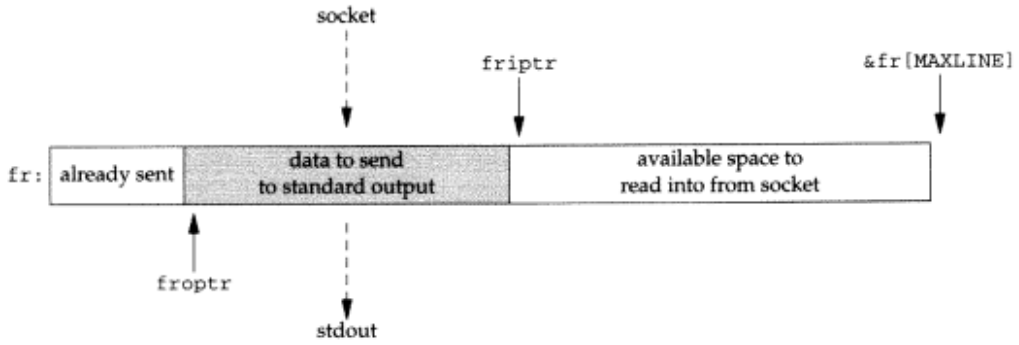

Figure 15.2 Buffer containing data from the socket going to standard output.

nonblock/strclinonb.c

```
 1 #include    "unp.h"

 2 void
 3 str_cli(FILE *fp, int sockfd)
 4 {
 5     int     maxfdp1, val, stdineof;
 6     ssize_t n, nwritten;
 7     fd_set  rset, wset;
 8     char    to[MAXLINE], fr[MAXLINE];
 9     char   *toiptr, *tooptr, *friptr, *froptr;

10     val = Fcntl(sockfd, F_GETFL, 0);
11     Fcntl(sockfd, F_SETFL, val | O_NONBLOCK);

12     val = Fcntl(STDIN_FILENO, F_GETFL, 0);
13     Fcntl(STDIN_FILENO, F_SETFL, val | O_NONBLOCK);

14     val = Fcntl(STDOUT_FILENO, F_GETFL, 0);
15     Fcntl(STDOUT_FILENO, F_SETFL, val | O_NONBLOCK);

16     toiptr = tooptr = to;       /* initialize buffer pointers */
17     friptr = froptr = fr;
18     stdineof = 0;

19     maxfdp1 = max(max(STDIN_FILENO, STDOUT_FILENO), sockfd) + 1;
20     for ( ; ; ) {
21         FD_ZERO(&rset);
22         FD_ZERO(&wset);
23         if (stdineof == 0 && toiptr < &to[MAXLINE])
24             FD_SET(STDIN_FILENO, &rset);    /* read from stdin */
25         if (friptr < &fr[MAXLINE])
26             FD_SET(sockfd, &rset);  /* read from socket */
27         if (tooptr != toiptr)
28             FD_SET(sockfd, &wset);  /* data to write to socket */
29         if (froptr != friptr)
30             FD_SET(STDOUT_FILENO, &wset);   /* data to write to stdout */

31         Select(maxfdp1, &rset, &wset, NULL, NULL);
```

nonblock/strclinonb.c

Figure 15.3 `str_cli` function: first part, initialize and call `select`.

Set descriptors nonblocking

10–15 All three descriptors are set nonblocking using `fcntl`: the socket to and from the server, standard input, and standard output.

Initialize buffer pointers

16–19 The pointers into the two buffers are initialized and the maximum descriptor plus one is calculated that will be used as the first argument for `select`.

Main loop: prepare to call `select`

20 As with the previous version of this function, Figure 6.13, the main loop of the function is a call to `select` followed by individual tests of the various conditions that we are interested in.

Specify descriptors that we are interested in

21–30 Both descriptor sets are set to 0 and then up to 2 bits are turned on in each set. If we have not yet read an end-of-file on standard input, and there is room for at least 1 byte of data in the `to` buffer, the bit corresponding to standard input is turned on in the read set. If there is room for at least 1 byte of data in the `fr` buffer, the bit corresponding to the socket is turned on in the read set. If there is data to write to the socket in the `to` buffer, the bit corresponding to the socket is turned on in the write set. Finally, if there is data in the `fr` buffer to send to standard output, the bit corresponding to standard output is turned on in the write set.

Call `select`

31 `select` is called, waiting for any one of the four possible conditions to be true. We do not specify a timeout for this function.

The next part of the function is shown in Figure 15.4. This code contains the first two tests (of four) that are made after `select` returns.

`read` from standard input

32–33 If standard input is readable, we call `read`. The third argument is the amount of available space in the `to` buffer.

Handle nonblocking error

34–35 If an error occurs and it is `EWOULDBLOCK`, nothing happens. Normally this condition "should not happen," that is, `select` telling us that the descriptor is readable and `read` returning `EWOULDBLOCK`, but we handle it nevertheless.

`read` returns end-of-file

36–40 If `read` returns 0 we are finished with the standard input. Our flag `stdineof` is set. If there is no more data in the `to` buffer to send (`tooptr` equals `toiptr`), `shutdown` sends a FIN to the server. If there is still data in the `to` buffer to send, the FIN cannot be sent until the buffer is written to the socket.

We output a line to standard error noting the end-of-file, along with the current time, and we show how we use this output after describing this function. Similar calls to `fprintf` are throughout this function.

nonblock/strclinonb.c

```
32          if (FD_ISSET(STDIN_FILENO, &rset)) {
33              if ( (n = read(STDIN_FILENO, toiptr, &to[MAXLINE] - toiptr)) < 0) {
34                  if (errno != EWOULDBLOCK)
35                      err_sys("read error on stdin");
36              } else if (n == 0) {
37                  fprintf(stderr, "%s: EOF on stdin\n", gf_time());
38                  stdineof = 1;   /* all done with stdin */
39                  if (tooptr == toiptr)
40                      Shutdown(sockfd, SHUT_WR);  /* send FIN */
41              } else {
42                  fprintf(stderr, "%s: read %d bytes from stdin\n", gf_time(), n);
43                  toiptr += n;    /* # just read */
44                  FD_SET(sockfd, &wset);  /* try and write to socket below */
45              }
46          }
47          if (FD_ISSET(sockfd, &rset)) {
48              if ( (n = read(sockfd, friptr, &fr[MAXLINE] - friptr)) < 0) {
49                  if (errno != EWOULDBLOCK)
50                      err_sys("read error on socket");
51              } else if (n == 0) {
52                  fprintf(stderr, "%s: EOF on socket\n", gf_time());
53                  if (stdineof)
54                      return;     /* normal termination */
55                  else
56                      err_quit("str_cli: server terminated prematurely");
57              } else {
58                  fprintf(stderr, "%s: read %d bytes from socket\n",
59                          gf_time(), n);
60                  friptr += n;    /* # just read */
61                  FD_SET(STDOUT_FILENO, &wset);   /* try and write below */
62              }
63          }
```

nonblock/strclinonb.c

Figure 15.4 `str_cli` function: second part, read from standard input or socket.

`read` returns data

41-45 When `read` returns data, we increment `toiptr` accordingly. We also turn on the bit corresponding to the socket in the write set, to cause the test for this bit to be true later in the loop, causing a `write` to be attempted to the socket.

This is one of the hard design decisions when writing code. We have a few alternatives here. Instead of setting the bit in the write set, we could do nothing, in which case `select` will test for writability of the socket the next time it is called. But this requires another loop around and another call to `select` when we already know that we have data to write to the socket. Another choice is to duplicate the code that writes to the socket here, but this seems wasteful and a potential source for an error (in case there is a bug in that piece of duplicated code, and we fix it in one location but not the other). Lastly, we could create a function that writes to the socket and call that function instead of duplicating the code, but that function needs to share

three of the local variables with str_cli, which would necessitate making these variables global. The choice made is the author's (biased) view about which alternative is best.

read from socket

47-63 These lines of code are similar to the if statement we just described when standard input is readable. If read returns EWOULDBLOCK, nothing happens. If we encounter an end-of-file from the server, this is OK if we have already encountered an end-of-file on the standard input, but it is not expected otherwise. If read returns some data, friptr is incremented and the bit for standard output is turned on in the write descriptor set, to try to write the data in the next part of the function.

Figure 15.5 shows the final portion of the function.

nonblock/strclinonb.c

```
64          if (FD_ISSET(STDOUT_FILENO, &wset) && ((n = friptr - froptr) > 0)) {
65              if ( (nwritten = write(STDOUT_FILENO, froptr, n)) < 0) {
66                  if (errno != EWOULDBLOCK)
67                      err_sys("write error to stdout");
68              } else {
69                  fprintf(stderr, "%s: wrote %d bytes to stdout\n",
70                          gf_time(), nwritten);
71                  froptr += nwritten;     /* # just written */
72                  if (froptr == friptr)
73                      froptr = friptr = fr;   /* back to beginning of buffer */
74              }
75          }
76          if (FD_ISSET(sockfd, &wset) && ((n = toiptr - tooptr) > 0)) {
77              if ( (nwritten = write(sockfd, tooptr, n)) < 0) {
78                  if (errno != EWOULDBLOCK)
79                      err_sys("write error to socket");
80              } else {
81                  fprintf(stderr, "%s: wrote %d bytes to socket\n",
82                          gf_time(), nwritten);
83                  tooptr += nwritten;     /* # just written */
84                  if (tooptr == toiptr) {
85                      toiptr = tooptr = to;   /* back to beginning of buffer */
86                      if (stdineof)
87                          Shutdown(sockfd, SHUT_WR);  /* send FIN */
88                  }
89              }
90          }
91      }
92  }
```

nonblock/strclinonb.c

Figure 15.5 str_cli function: third part, write to standard output or socket.

write to standard output

64-67 If standard output is writable and the number of bytes to write is greater than 0, write is called. If EWOULDBLOCK is returned, nothing happens. Notice that this condition is entirely possible, because the code at the end of the previous part of this function

turns on the bit for standard output in the write set, without knowing whether the `write` will succeed or not.

`write` OK

68-74 If the `write` is successful, `froptr` is incremented by the number of bytes written. If the output pointer has caught up with the input pointer, both pointers are reset to point to the beginning of the buffer.

`write` to socket

76-90 This section of code is similar to the code we just described for writing to the standard output. The one difference is that when the output pointer catches up with the input pointer, not only are both pointers reset to the beginning of the buffer, but if we have encountered an end-of-file on standard input, the FIN can now be sent to the server.

We now examine the operation of this function and the overlapping of the nonblocking I/O. Figure 15.6 shows our `gf_time` function that is called from our `str_cli` function.

lib/gf_time.c

```
 1 #include    "unp.h"
 2 #include    <time.h>

 3 char *
 4 gf_time(void)
 5 {
 6     struct timeval tv;
 7     static char str[30];
 8     char   *ptr;

 9     if (gettimeofday(&tv, NULL) < 0)
10         err_sys("gettimeofday error");

11     ptr = ctime(&tv.tv_sec);
12     strcpy(str, &ptr[11]);
13     /* Fri Sep 13 00:00:00 1986\n\0 */
14     /* 0123456789012345678901234 5  */
15     snprintf(str + 8, sizeof(str) - 8, ".%06ld", tv.tv_usec);

16     return (str);
17 }
```

lib/gf_time.c

Figure 15.6 `gf_time` function: return pointer to time string.

This function returns a string containing the current time, including microseconds, in the format

```
12:34:56.123456
```

This is intentionally in the same format as the timestamps output by `tcpdump`. Also notice that all the calls to `fprintf` in our `str_cli` function write to standard error, allowing us to separate standard output (the lines echoed by the server) from our diagnostic output. We can then run our client and `tcpdump` and take this diagnostic output

along with the tcpdump output and sort the two outputs together, ordered by the time. This lets us see what happens in our program and correlate it with the corresponding TCP action.

For example, we first run tcpdump on our host solaris, capturing only TCP segments to or from port 7 (the echo server), saving the output in the file named tcpd.

```
solaris % tcpdump -w tcpd tcp and port 7
```

We then run our TCP client on this host, specifying the server on the host bsdi.

```
solaris % tcpcli02 206.62.226.35 < 2000.lines > out 2> diag
```

Standard input is the file 2000.lines, the same file we used with Figure 6.13. Standard output is sent to the file out, and standard error is sent to the file diag. On completion we run

```
solaris % diff 2000.lines out
```

to verify that the echoed lines are identical to the input lines. Finally we terminate tcpdump with our interrupt key and then print the tcpdump records, sorting these records with the diagnostic output from the client. Figure 15.7 shows the first part of this result.

```
solaris % tcpdump -r tcpd -N | sort diag -
10:18:34.486392 solaris.33621 > bsdi.echo: S 1802738644:1802738644(0)
                                           win 8760 <mss 1460>
10:18:34.488278 bsdi.echo > solaris.33621: S 3212986316:3212986316(0)
                                           ack 1802738645 win 8760 <mss 1460>
10:18:34.488490 solaris.33621 > bsdi.echo: . ack 1 win 8760

10:18:34.491482: read 4096 bytes from stdin
10:18:34.518663 solaris.33621 > bsdi.echo: P 1:1461(1460) ack 1 win 8760
10:18:34.519016: wrote 4096 bytes to socket
10:18:34.528529 bsdi.echo > solaris.33621: P 1:1461(1460) ack 1461 win 8760
10:18:34.528785 solaris.33621 > bsdi.echo: . 1461:2921(1460) ack 1461 win 8760
10:18:34.528900 solaris.33621 > bsdi.echo: P 2921:4097(1176) ack 1461 win 8760
10:18:34.528958 solaris.33621 > bsdi.echo: . ack 1461 win 8760
10:18:34.536193 bsdi.echo > solaris.33621: . 1461:2921(1460) ack 4097 win 8760
10:18:34.536697 bsdi.echo > solaris.33621: P 2921:3509(588) ack 4097 win 8760
10:18:34.544636: read 4096 bytes from stdin
10:18:34.568505: read 3508 bytes from socket
10:18:34.580373 solaris.33621 > bsdi.echo: . ack 3509 win 8760
10:18:34.582244 bsdi.echo > solaris.33621: P 3509:4097(588) ack 4097 win 8760
10:18:34.593354: wrote 3508 bytes to stdout
10:18:34.617272 solaris.33621 > bsdi.echo: P 4097:5557(1460) ack 4097 win 8760
10:18:34.617610 solaris.33621 > bsdi.echo: P 5557:7017(1460) ack 4097 win 8760
10:18:34.617908 solaris.33621 > bsdi.echo: P 7017:8193(1176) ack 4097 win 8760
10:18:34.618062: wrote 4096 bytes to socket
10:18:34.623310 bsdi.echo > solaris.33621: . ack 8193 win 8760
10:18:34.626129 bsdi.echo > solaris.33621: . 4097:5557(1460) ack 8193 win 8760
10:18:34.626339 solaris.33621 > bsdi.echo: . ack 5557 win 8760
10:18:34.626611 bsdi.echo > solaris.33621: P 5557:6145(588) ack 8193 win 8760
10:18:34.628396 bsdi.echo > solaris.33621: . 6145:7605(1460) ack 8193 win 8760
10:18:34.643524: read 4096 bytes from stdin
10:18:34.667305: read 2636 bytes from socket
```

```
10:18:34.670324 solaris.33621 > bsdi.echo: . ack 7605 win 8760
10:18:34.672221 bsdi.echo > solaris.33621: P 7605:8193(588) ack 8193 win 8760
10:18:34.691039: wrote 2636 bytes to stdout
```

Figure 15.7 Sorted output from `tcpdump` and diagnostic output.

We have wrapped the long lines containing the SYNs and we have also removed the `(DF)` notations from the Solaris segments, denoting that it sets the don't-fragment bit (path MTU discovery). The `-N` option to `tcpdump` prints only the host portion (`solaris`) of the fully qualified domain name (`solaris.kohala.com`).

Using this output we can draw a time line of what's happening. We show this in Figure 15.8 with time increasing down the page.

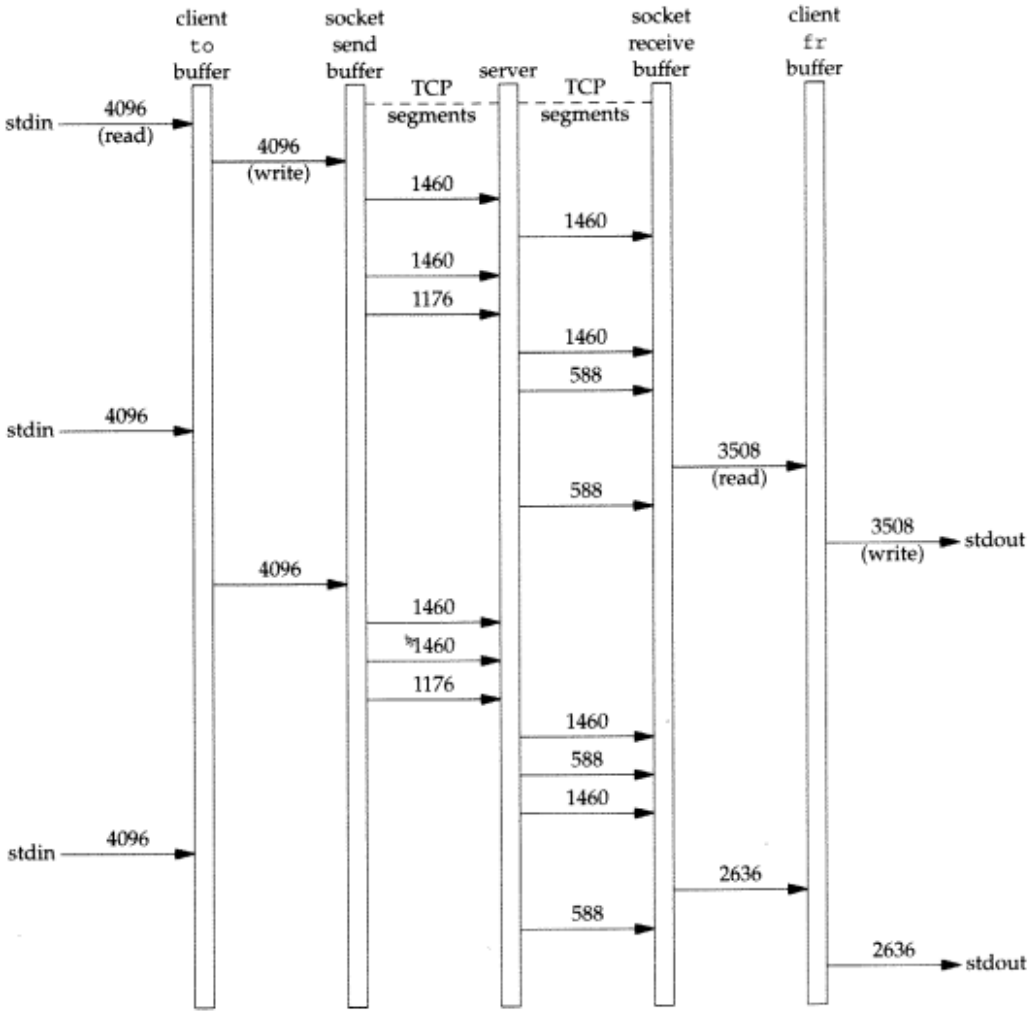

Figure 15.8 Time line of nonblocking example.

In this figure we do not show the ACK segments. Also realize that when the program outputs "wrote N bytes to stdout," the `write` has returned, possibly causing TCP to send one or more segments of data.

What we can see from this time line are the dynamics of a client–server exchange. Using nonblocking I/O lets the program take advantage of these dynamics, reading or writing when the operation can take place. We let the kernel tell us when an I/O operation can occur by using the `select` function.

We can time our nonblocking version using the same 2000-line file and the same server (a 175-ms RTT from the client) as in Section 6.7. The clock time is now 6.9 seconds, compared to 12.3 seconds for the version in Section 6.7. Therefore nonblocking I/O reduces the overall time for this example that sends a file to the server.

A Simpler Version of `str_cli`

The nonblocking version of `str_cli` that we just showed is nontrivial: about 135 lines of code, compared to 40 lines for the version using `select` with blocking I/O in Figure 6.13, and 20 lines for our original stop-and-wait version (Figure 5.5). We know that doubling the size of the code from 20 to 40 lines was worth the effort, because the speed increased by almost a factor of 30 in a batch mode and using `select` with blocking descriptors was not overly complicated. But is it worth the effort to code an application using nonblocking I/O, given the complexity of the resulting code? The answer is no. Whenever we find the need to use nonblocking I/O, it will usually be simpler to split the application into either processes (using `fork`) or threads (Chapter 23).

Figure 15.10 is yet another version of our `str_cli` function, with the function dividing itself into two processes using `fork`.

The function immediately calls `fork` to split into a parent and child. The child copies lines from the server to standard output and the parent copies lines from standard input to the server, as shown in Figure 15.9.

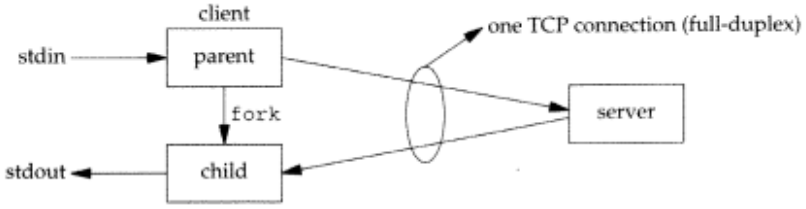

Figure 15.9 `str_cli` function using two processes.

We explicitly note that the TCP connection is full-duplex and that the parent and child are sharing the same socket descriptor: the parent writes to the socket and the child reads from the socket. There is only one socket, one socket receive buffer, and one socket send buffer, but this socket is referenced by two descriptors: one in the parent and one in the child.

We again need to worry about the termination sequence. Normal termination occurs when the end-of-file on standard input is encountered. The parent reads this

nonblock/strclifork.c

```
1 #include    "unp.h"

2 void
3 str_cli(FILE *fp, int sockfd)
4 {
5     pid_t   pid;
6     char    sendline[MAXLINE], recvline[MAXLINE];

7     if ( (pid = Fork()) == 0) {  /* child: server -> stdout */
8         while (Readline(sockfd, recvline, MAXLINE) > 0)
9             Fputs(recvline, stdout);

10        kill(getppid(), SIGTERM);   /* in case parent still running */
11        exit(0);
12    }
13    /* parent: stdin -> server */
14    while (Fgets(sendline, MAXLINE, fp) != NULL)
15        Writen(sockfd, sendline, strlen(sendline));

16    Shutdown(sockfd, SHUT_WR);  /* EOF on stdin, send FIN */
17    pause();
18    return;
19 }
```

nonblock/strclifork.c

Figure 15.10 Version of `str_cli` function that uses `fork`.

end-of-file and calls `shutdown` to send a FIN. (The parent cannot call `close`. See Exercise 15.1.) But when this happens, the child needs to continue copying from the server to the standard output, until it reads an end-of-file on the socket.

It is also possible for the server process to terminate prematurely (Section 5.12), and if this occurs, the child will read an end-of-file on the socket. If this happens the child must tell the parent to stop copying from the standard input to the socket (see Exercise 15.2). In Figure 15.10 the child sends the `SIGTERM` signal to the parent, in case the parent is still running (see Exercise 15.3). Another way to handle this would be for the child to terminate, and have the parent catch `SIGCHLD`, if the parent is still running.

The parent calls `pause` when it has finished copying, which puts it to sleep until a signal is caught. Even though our parent does not catch any signals, this puts the parent to sleep until it receives the `SIGTERM` signal from the child. The default action of this signal is to terminate the process, which is fine for this example. The reason we make the parent wait for the child is to measure an accurate clock time for this version of `str_cli`. Normally the child finishes after the parent, but since we measure the clock time using the shell's `time` command, the measurement ends when the parent terminates.

Notice the simplicity of this version, compared to the nonblocking I/O version shown earlier in this section. Our nonblocking version managed four different I/O streams at the same time, and since all four were nonblocking, we had to concern ourself with partial reads and writes for all four streams. But in the `fork` version, each

process handles only two I/O streams, copying from one to the other. There is no need for nonblocking I/O because if there is no data to read from the input stream, there is nothing to write to the corresponding output stream.

Timing of `str_cli`

We have now shown four different versions of the `str_cli` function. We summarize the clock time required for these versions, along with a version using threads (Figure 23.2), when copying 2000 lines from a Solaris 2.5 client to a server with an RTT of 175 ms:

- 354.0 sec, stop-and-wait (Figure 5.5),
- 12.3 sec, `select` and blocking I/O (Figure 6.13),
- 6.9 sec, nonblocking I/O (Figure 15.3),
- 8.7 sec, `fork` (Figure 15.10), and
- 8.5 sec, threaded version (Figure 23.2).

Our nonblocking I/O version is almost twice as fast as our version using blocking I/O with `select`. Our simple version using `fork` is slower than our nonblocking I/O version. Nevertheless, given the complexity of the nonblocking I/O code, versus the `fork` code, we recommend the simple approach.

15.3 Nonblocking `connect`

When a TCP socket is set nonblocking and then `connect` is called, `connect` returns immediately with an error of `EINPROGRESS` but the TCP three-way handshake continues. We then check for either a successful or unsuccessful completion of the connection establishment using `select`. There are three uses for a nonblocking `connect`.

1. We can overlap other processing with the three-way handshake. A `connect` takes one round-trip time to complete (Section 2.5) and this can be anywhere from a few milliseconds on a LAN to hundreds of milliseconds or a few seconds on a WAN. There might be other processing we wish to perform during this time.
2. We can establish multiple connections at the same time using this technique. This has become popular with Web browsers, and we show an example of this in Section 15.5.
3. Since we wait for the connection establishment to complete using `select`, we can specify a time limit for `select`, allowing us to shorten the timeout for the `connect`. Many implementations have a timeout for `connect` that is between 75 seconds and several minutes. There are times when an application wants a shorter timeout, and using a nonblocking `connect` is one way to accomplish this. Section 13.2 talks about other ways to place timeouts on socket operations.

As simple as the nonblocking `connect` sounds, there are details that we must handle.

- Even though the socket is nonblocking, if the server to which we are connecting is on the same host, the connection establishment normally takes place immediately when we call `connect`. We must handle this scenario.
- Berkeley-derived implementations (and Posix.1g) have the following two rules regarding `select` and nonblocking `connect`s: (1) when the connection completes successfully, the descriptor becomes writable (p. 531 of TCPv2), and (2) when the connection establishment encounters an error, the descriptor becomes both readable and writable (p. 530 of TCPv2).

  These two rules regarding `select` fall out from our rules in Section 6.3 about the conditions that make a descriptor ready. A TCP socket is writable if there is available space in the send buffer (which will always be the case for a connecting socket, since we have not yet written anything to the socket) *and* the socket is connected (which occurs only when the three-way handshake completes). A pending error causes a socket to be both readable and writable.

There are many portability problems with nonblocking `connect`s that we mention in the examples that follow.

15.4 Nonblocking `connect`: Daytime Client

Figure 15.11 shows our function `connect_nonb`, which performs a nonblocking `connect`. We replace the call to `connect` in Figure 1.5 with

```
if (connect_nonb(sockfd, (SA *) &servaddr, sizeof(servaddr), 0) < 0)
    err_sys("connect error");
```

The first three arguments are the normal arguments to `connect` and the fourth argument is the number of seconds to wait for the connection to complete. A value of 0 implies no timeout on the `select`; hence the kernel will use its normal TCP connection establishment timeout.

Set socket nonblocking

9–10 We call `fcntl` to set the socket nonblocking.

11–14 We initiate the nonblocking `connect`. The error we expect is `EINPROGRESS`, indicating that the connection has started but is not yet complete (p. 466 of TCPv2). Any other error is returned to the caller

Overlap processing with connection establishment

15 At this point we can do whatever we want while we wait for the connection to complete.

Check for immediate completion

16–17 If the nonblocking `connect` returned 0, the connection is complete. As we have said, this can occur when the server is on the same host as the client.

lib/connect_nonb.c

```
1 #include    "unp.h"

2 int
3 connect_nonb(int sockfd, const SA *saptr, socklen_t salen, int nsec)
4 {
5     int     flags, n, error;
6     socklen_t len;
7     fd_set  rset, wset;
8     struct timeval tval;

9     flags = Fcntl(sockfd, F_GETFL, 0);
10    Fcntl(sockfd, F_SETFL, flags | O_NONBLOCK);

11    error = 0;
12    if ( (n = connect(sockfd, saptr, salen)) < 0)
13        if (errno != EINPROGRESS)
14            return (-1);

15    /* Do whatever we want while the connect is taking place. */

16    if (n == 0)
17        goto done;              /* connect completed immediately */

18    FD_ZERO(&rset);
19    FD_SET(sockfd, &rset);
20    wset = rset;
21    tval.tv_sec = nsec;
22    tval.tv_usec = 0;

23    if ( (n = Select(sockfd + 1, &rset, &wset, NULL,
24                     nsec ? &tval : NULL)) == 0) {
25        close(sockfd);          /* timeout */
26        errno = ETIMEDOUT;
27        return (-1);
28    }
29    if (FD_ISSET(sockfd, &rset) || FD_ISSET(sockfd, &wset)) {
30        len = sizeof(error);
31        if (getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &error, &len) < 0)
32            return (-1);        /* Solaris pending error */
33    } else
34        err_quit("select error: sockfd not set");

35  done:
36    Fcntl(sockfd, F_SETFL, flags);  /* restore file status flags */

37    if (error) {
38        close(sockfd);          /* just in case */
39        errno = error;
40        return (-1);
41    }
42    return (0);
43 }
```

lib/connect_nonb.c

Figure 15.11 Issue a nonblocking connect.

Call `select`

18-24 We call `select` and wait for the socket to be ready for either reading or writing. We zero out `rset`, turn on the bit corresponding to `sockfd` in this descriptor set, and then copy `rset` into `wset`. This assignment is probably a structure assignment, since descriptor sets are normally represented as structures. We also initialize the `timeval` structure and then call `select`. If the caller specifies a fourth argument of 0 (use the default timeout) we must specify a null pointer as the final argument to `select` and not a `timeval` structure with a value of 0 (which means do not wait at all).

Handle timeouts

25-28 If `select` returns 0, the timer expired, and we return `ETIMEDOUT` to the caller. We also close the socket, to prevent the three-way handshake from proceeding any farther.

Check for readability or writability

29-34 If the descriptor is readable or writable, we call `getsockopt` to fetch the socket's pending error (`SO_ERROR`). If the connection completed successfully, this value will be 0. If the connection encountered an error, this value is the `errno` value corresponding to the connection error (e.g., `ECONNREFUSED`, `ETIMEDOUT`, etc.). We also encounter our first portability problem. If an error occurred, Berkeley-derived implementations of `getsockopt` return 0 with the pending error returned in our variable `error`. But Solaris causes `getsockopt` itself to return −1 with `errno` set to the pending error. Our code handles both scenarios.

Turn off nonblocking and return

36-42 We restore the file status flags and return. If our `error` variable is nonzero from `getsockopt`, that value is stored in `errno` and the function returns −1.

As we said earlier, there are portability problems with various socket implementations and nonblocking connects. First, it is possible for the connection to complete and for data to arrive from the peer before `select` is called. In this case the socket will be both readable and writable on success, the same as if the connection had failed. Our code in Figure 15.11 handles this scenario by calling `getsockopt` and checking the pending error for the socket.

Next is how to determine whether the connection completed successfully or not, if we cannot assume that writability is the only way success is returned. Various solutions have been posted to Usenet. These would replace our call to `getsockopt` in Figure 15.11.

1. Call `getpeername` instead of `getsockopt`. If this fails with `ENOTCONN`, the connection failed and we must then call `getsockopt` with `SO_ERROR` to fetch the pending error for the socket.
2. Call `read` with a length of 0. If the `read` fails, the `connect` failed and the `errno` from `read` indicates the reason for the connection failure. If the connection succeeded, `read` should return 0.
3. Call `connect` again. It should fail and if the error is `EISCONN`, the socket is already connected and the first connection succeeded.

Unfortunately nonblocking connects are one of the most nonportable areas of network programming. Be prepared for portability problems, especially with older implementations. A simpler technique is to create a thread (Chapter 23) to handle the connection.

Interrupted connect

What happens if our call to connect on a normal blocking socket is interrupted, say, by a caught signal, before TCP's three-way handshake completes? Assuming the connect is not automatically restarted, it returns EINTR. But we cannot call connect again to wait for the connection to complete. Doing so will return EADDRINUSE.

What we must do in this scenario is call select, just as we have done in this section for a nonblocking connect. Then select returns when the connection completes successfully (making the socket writable) or when the connection fails (making the socket readable and writable).

> Posix.1g explicitly specifies what a call to the XTI function t_connect does when interrupted by a caught signal but says nothing about how to handle this with connect. What we just described corresponds to the handling of this scenario by Berkeley-derived kernels.

15.5 Nonblocking connect: Web Client

A real-world example of nonblocking connects started with the Netscape Web client (Section 13.4 of TCPv3). The client establishes an HTTP connection with a Web server and fetches a home page. On that page are often numerous references to other Web pages. Instead of fetching these other pages serially, one at a time, the client can fetch more than one at the same time, using nonblocking connects. Figure 15.12 shows an example of establishing multiple connections in parallel. The leftmost scenario shows all three connections performed serially. We assume that the first connection takes 10 units of time, the second 15, and the third 4, for a total of 29 units of time.

In the middle scenario we perform two connections in parallel. At time 0 the first two connections are started, and when the first of these finishes, we start the third. The total time is almost halved, from 29 to 15, but realize that this is the ideal case. If the parallel connections are sharing a common link (say the client is behind a dialup modem link to the Internet) each can compete against each other for the limited resources, and all the individual connection times might get longer. For example, the time of 10 might be 15, the time of 15 might be 20, and the time of 4 might be 6. Nevertheless, the total time would be 21, still shorter than the serial scenario.

In the third scenario we perform three connections in parallel, and we again assume that there is no interference between the three connections (the ideal case). But the total time is the same (15 units) as the second scenario given the example times that we choose.

When dealing with Web clients, the first connection is done by itself, followed by multiple connections for the references found in the data from that first connection. We show this in Figure 15.13.

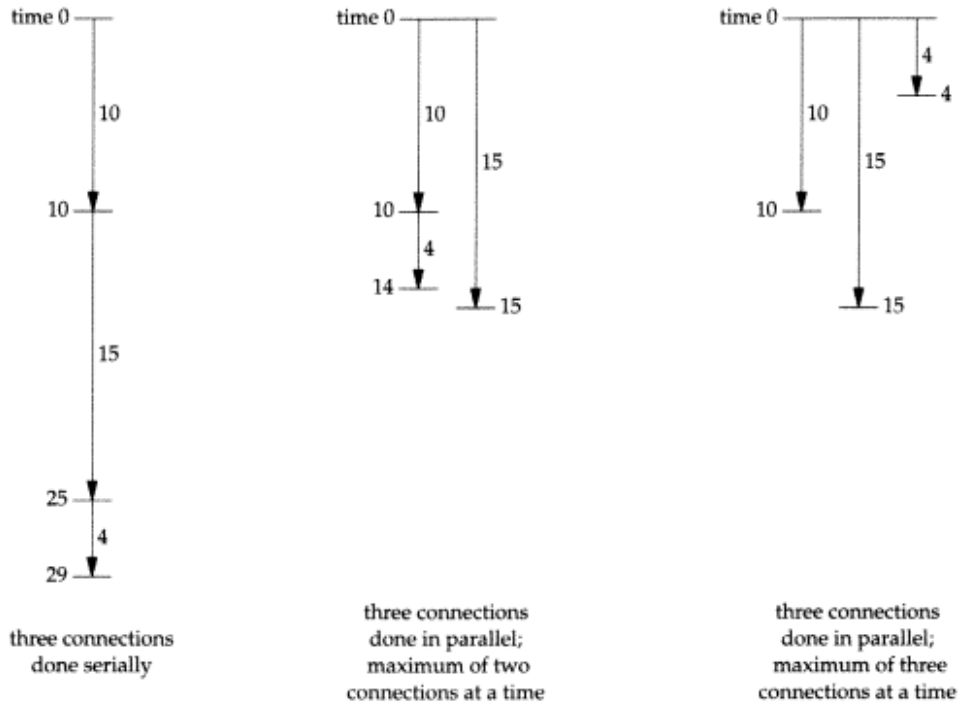

Figure 15.12 Establishing multiple connections in parallel.

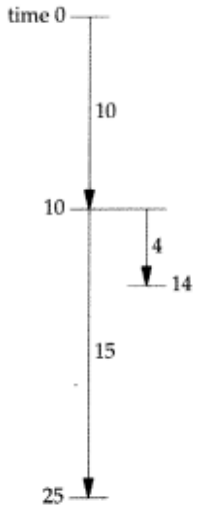

Figure 15.13 Complete first connection, then multiple connections in parallel.

As even further optimization, the client can start parsing the data that is returned for the first connection, before the first connection completes, and initiate additional connections as soon as it knows that additional connections are needed.

Since we are doing multiple nonblocking connects at the same time, we cannot use our connect_nonb function from Figure 15.11, because it does not return until the connection is established. Instead we must keep track of multiple connections ourself.

Our program will read up to 20 files from a Web server. We specify as command-line arguments the maximum number of parallel connections, the server's hostname, and then each of the filenames to fetch from the server. A typical execution of our program is:

```
solaris % web  3  www.foobar.com  /  image1.gif  image2.gif \
image3.gif  image4.gif  image5.gif \
image6.gif  image7.gif
```

The command-line arguments specify three simultaneous connections, the server's hostname, the filename for the home page (/ the server's root page), and seven files to then read (which in this example are all GIF images). These seven files would normally be referenced on the home page, and a Web client would read the home page and parse the HTML to obtain these filenames. We do not want to complicate this example with HTML parsing, so we just specify the filenames on the command line.

This is a larger example so we will show it in pieces. Figure 15.14 is our web.h header that each file includes.

nonblock/web.h

```
 1 #include    "unp.h"

 2 #define MAXFILES    20
 3 #define SERV        "80"        /* port number or service name */

 4 struct file {
 5     char   *f_name;             /* filename */
 6     char   *f_host;             /* hostname or IPv4/IPv6 address */
 7     int     f_fd;               /* descriptor */
 8     int     f_flags;            /* F_xxx below */
 9 } file[MAXFILES];

10 #define F_CONNECTING    1       /* connect() in progress */
11 #define F_READING       2       /* connect() complete; now reading */
12 #define F_DONE          4       /* all done */

13 #define GET_CMD     "GET %s HTTP/1.0\r\n\r\n"

14             /* globals */
15 int     nconn, nfiles, nlefttoconn, nlefttoread, maxfd;
16 fd_set  rset, wset;

17             /* function prototypes */
18 void    home_page(const char *, const char *);
19 void    start_connect(struct file *);
20 void    write_get_cmd(struct file *);
```

nonblock/web.h

Figure 15.14 web.h header.

Define `file` structure

2-13 The program reads up to `MAXFILES` files from the Web server. We maintain a `file` structure with information about each file: its name (copied from the command-line argument), the hostname or IP address of the server to read the file from, the socket descriptor being used for the file, and a set of flags to specify what we are doing with this file (connecting, reading, or done).

Define globals and function prototypes

14-20 We define the global variables and function prototypes for our functions that we describe shortly.

Figure 15.15 shows the first part of the `main` program.

nonblock/web.c

```
 1 #include    "web.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     i, fd, n, maxnconn, flags, error;
 6     char    buf[MAXLINE];
 7     fd_set  rs, ws;

 8     if (argc < 5)
 9         err_quit("usage: web <#conns> <hostname> <homepage> <file1> ...");
10     maxnconn = atoi(argv[1]);

11     nfiles = min(argc - 4, MAXFILES);
12     for (i = 0; i < nfiles; i++) {
13         file[i].f_name = argv[i + 4];
14         file[i].f_host = argv[2];
15         file[i].f_flags = 0;
16     }
17     printf("nfiles = %d\n", nfiles);

18     home_page(argv[2], argv[3]);

19     FD_ZERO(&rset);
20     FD_ZERO(&wset);
21     maxfd = -1;
22     nlefttoread = nlefttoconn = nfiles;
23     nconn = 0;
```

nonblock/web.c

Figure 15.15 First part of simultaneous `connect`: globals and start of `main`.

Process command-line arguments

11-17 The `file` structures are filled in with the relevant information from the command-line arguments.

Read home page

18 The function `home_page`, which we show next, creates a TCP connection, sends a command to the server, and then reads the home page. This is the first connection, which is done by itself, before we start establishing multiple connections in parallel.

Initialize globals

19–23 Two descriptor sets, one for reading and one for writing, are initialized. maxfd is the maximum descriptor for select (which we initialize to –1, since descriptors are nonnegative), nlefttoread is the number of files remaining to be read (when this reaches 0 we are finished), nlefttoconn is the number of files that still need a TCP connection, and nconn is the number of connections currently open (which can never exceed the first command-line argument).

Figure 15.16 shows the home_page function that is called once when the main function begins.

nonblock/home_page.c

```
 1 #include    "web.h"

 2 void
 3 home_page(const char *host, const char *fname)
 4 {
 5     int     fd, n;
 6     char    line[MAXLINE];

 7     fd = Tcp_connect(host, SERV);   /* blocking connect() */

 8     n = snprintf(line, sizeof(line), GET_CMD, fname);
 9     Writen(fd, line, n);

10     for ( ; ; ) {
11         if ( (n = Read(fd, line, MAXLINE)) == 0)
12             break;              /* server closed connection */

13         printf("read %d bytes of home page\n", n);
14         /* do whatever with data */
15     }
16     printf("end-of-file on home page\n");
17     Close(fd);
18 }
```

nonblock/home_page.c

Figure 15.16 home_page function.

Establish connection with server

7 Our tcp_connect establishes a connection with the server.

Send HTTP command to server; read reply

8–17 An HTTP GET command is issued for the home page (often named /). The reply is read (we do not do anything with the reply) and the connection is closed.

The next function, start_connect shown in Figure 15.17, initiates a nonblocking connect.

Create socket, set nonblocking

7–13 We call our host_serv function (Figure 11.5) to look up and convert the hostname and service name, returning a pointer to an array of addrinfo structures. We use only the first structure. A TCP socket is created and the socket is set nonblocking.

nonblock/start_connect.c

```
1 #include    "web.h"

2 void
3 start_connect(struct file *fptr)
4 {
5     int     fd, flags, n;
6     struct addrinfo *ai;

7     ai = Host_serv(fptr->f_host, SERV, 0, SOCK_STREAM);

8     fd = Socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
9     fptr->f_fd = fd;
10    printf("start_connect for %s, fd %d\n", fptr->f_name, fd);

11        /* Set socket nonblocking */
12    flags = Fcntl(fd, F_GETFL, 0);
13    Fcntl(fd, F_SETFL, flags | O_NONBLOCK);

14        /* Initiate nonblocking connect to the server. */
15    if ( (n = connect(fd, ai->ai_addr, ai->ai_addrlen)) < 0) {
16        if (errno != EINPROGRESS)
17            err_sys("nonblocking connect error");
18        fptr->f_flags = F_CONNECTING;
19        FD_SET(fd, &rset);          /* select for reading and writing */
20        FD_SET(fd, &wset);
21        if (fd > maxfd)
22            maxfd = fd;

23    } else if (n >= 0)              /* connect is already done */
24        write_get_cmd(fptr);        /* write() the GET command */
25 }
```

nonblock/start_connect.c

Figure 15.17 Initiate nonblocking `connect`.

Initiate nonblocking `connect`

14–22 The nonblocking `connect` is initiated and the file's flag is set to `F_CONNECTING`. The socket descriptor is turned on in both the read set and the write set, since `select` will wait for either condition as an indication that the connection has finished. We also update `maxfd`, if necessary.

Handle connection complete

23–24 If `connect` returns success, the connection is already complete and the function `write_get_cmd` (shown next) sends a command to the server.

We set the socket nonblocking for the `connect` but never reset it to its default blocking mode. This is OK because we write only a small amount of data to the socket (the `GET` command in the next function) and we assume that this command is much smaller than the socket send buffer. Even if `write` returns a short count because of the nonblocking flag, our `writen` function handles this. Leaving the socket nonblocking has no effect on the subsequent `read`s that are performed because we always call `select` to wait for the socket to become readable.

Figure 15.18 shows the function write_get_cmd, which sends an HTTP GET command to the server.

nonblock/write_get_cmd.c

```
 1 #include    "web.h"

 2 void
 3 write_get_cmd(struct file *fptr)
 4 {
 5     int     n;
 6     char    line[MAXLINE];

 7     n = snprintf(line, sizeof(line), GET_CMD, fptr->f_name);
 8     Writen(fptr->f_fd, line, n);
 9     printf("wrote %d bytes for %s\n", n, fptr->f_name);

10     fptr->f_flags = F_READING;  /* clears F_CONNECTING */

11     FD_SET(fptr->f_fd, &rset);  /* will read server's reply */
12     if (fptr->f_fd > maxfd)
13         maxfd = fptr->f_fd;
14 }
```

nonblock/write_get_cmd.c

Figure 15.18 Send an HTTP GET command to the server.

Build command and send it

7–9 The command is built and written to the socket.

Set flags

10–13 The file's F_READING flag is set, which also clears the F_CONNECTING flag (if set). This indicates to the main loop that this descriptor is ready for input. The descriptor is also turned on in the read set and maxfd updated, if necessary.

We now return to the main function in Figure 15.19, picking up from where we left off in Figure 15.15. This is the main loop of the program: as long as there are more files to process (nlefttoread is greater than 0), start another connection if possible, and then select on all active descriptors, handling both nonblocking connection completions and the arrival of data.

Can we initiate another connection?

24–35 If we are not at the specified limit of simultaneous connections, and there are additional connections to establish, find a file that we have not yet processed (indicated by a f_flags of 0), and call start_connect to initiate the connection. The number of active connections is incremented (nconn) and the number of connections remaining to be established is decremented (nlefttoconn).

select: wait for something to happen

36–37 select waits for either readability or writability. Descriptors that have a nonblocking connect in progress will be enabled in both sets, while descriptors with a completed connection that are waiting for data from the server will be enabled in just the read set.

nonblock/web.c

```
24      while (nlefttoread > 0) {
25          while (nconn < maxnconn && nlefttoconn > 0) {
26                  /* find a file to read */
27              for (i = 0; i < nfiles; i++)
28                  if (file[i].f_flags == 0)
29                      break;
30              if (i == nfiles)
31                  err_quit("nlefttoconn = %d but nothing found", nlefttoconn);
32              start_connect(&file[i]);
33              nconn++;
34              nlefttoconn--;
35          }
36          rs = rset;
37          ws = wset;
38          n = Select(maxfd + 1, &rs, &ws, NULL, NULL);
39          for (i = 0; i < nfiles; i++) {
40              flags = file[i].f_flags;
41              if (flags == 0 || flags & F_DONE)
42                  continue;
43              fd = file[i].f_fd;
44              if (flags & F_CONNECTING &&
45                  (FD_ISSET(fd, &rs) || FD_ISSET(fd, &ws))) {
46                  n = sizeof(error);
47                  if (getsockopt(fd, SOL_SOCKET, SO_ERROR, &error, &n) < 0 ||
48                      error != 0) {
49                      err_ret("nonblocking connect failed for %s",
50                              file[i].f_name);
51                  }
52                      /* connection established */
53                  printf("connection established for %s\n", file[i].f_name);
54                  FD_CLR(fd, &wset);  /* no more writeability test */
55                  write_get_cmd(&file[i]);    /* write() the GET command */
56              } else if (flags & F_READING && FD_ISSET(fd, &rs)) {
57                  if ( (n = Read(fd, buf, sizeof(buf))) == 0) {
58                      printf("end-of-file on %s\n", file[i].f_name);
59                      Close(fd);
60                      file[i].f_flags = F_DONE;   /* clears F_READING */
61                      FD_CLR(fd, &rset);
62                      nconn--;
63                      nlefttoread--;
64                  } else {
65                      printf("read %d bytes from %s\n", n, file[i].f_name);
66                  }
67              }
68          }
69      }
70      exit(0);
71  }
```

nonblock/web.c

Figure 15.19 Main loop of `main` function.

Handle all ready descriptors

39–55 We now process each element in the array of `file` structures to determine which descriptors need processing. If the `F_CONNECTING` flag is set and the descriptor is on in either the read set or the write set, the nonblocking `connect` is finished. As we described with Figure 15.11, we call `getsockopt` to fetch the pending error for the socket. If this value is 0, the connection completed successfully. In that case we turn off the descriptor in the write set and call `write_get_cmd` to send the HTTP request to the server.

See if descriptor has data

56–67 If the `F_READING` flag is set and the descriptor is ready for reading, we call `read`. If the connection was closed by the other end, we close the socket, set the `F_DONE` flag, turn off the descriptor in the read set, and decrement the number of active connections and the total number of connections to be processed.

There are two optimizations that we do not perform in this example (to avoid complicating it even more). First, we could terminate the `for` loop in Figure 15.19 when we have processed the number of descriptors that `select` said were ready. Next, we could decrease the value of `maxfd` when possible, to save `select` from examining descriptor bits that are no longer set. Since the number of descriptors that this code deals with at any one time is probably less than 10, and not in the thousands, it is doubtful that either of these optimizations is worth the additional complications.

Performance of Simultaneous Connections

What is the performance gain in establishing multiple connections at the same time? Figure 15.20 shows the clock time required to fetch a Web server's home page, followed by nine image files from that server. The RTT to the server is about 150 ms. The home page size was 4017 bytes and the average size of the nine image files was 1621 bytes. TCP's segment size was 512 bytes. We also include in this figure, for comparison, values for a version of this program that we develop in Section 23.9 using threads.

# Simultaneous connections	Clock time (seconds), nonblocking	Clock time (seconds), threads
1	6.0	6.3
2	4.1	4.2
3	3.0	3.1
4	2.8	3.0
5	2.5	2.7
6	2.4	2.5
7	2.3	2.3
8	2.2	2.3
9	2.0	2.2

Figure 15.20 Clock time for various numbers of simultaneous connections.

Most of the improvement is obtained with three simultaneous connections (the clock time is halved) and the performance increase is much less with four or more simultaneous connections.

> We provide this example using simultaneous connects because it is a nice example using nonblocking I/O and one whose performance impact can be measured. It is also a feature used by a popular Web application, the Netscape browser. There are pitfalls in this technique if there is any congestion in the network. Chapter 21 of TCPv1 describes TCP's slow start and congestion avoidance algorithms in detail. When multiple connections are established from a client to a server, there is no communication between the connections at the TCP layer. That is, if one connection encounters a packet loss, the other connections to the same server are not notified, and it is highly probable that the other connections will soon encounter packet loss unless they slow down. These additional connections are sending more packets into an already congested network. This technique also increases the load at any given time on the server.

15.6 Nonblocking `accept`

We stated in Chapter 6 that a listening socket is returned as readable by `select` when a completed connection is ready to be `accept`ed. Therefore, if we are using `select` to wait for incoming connections, we should not need to set the listening socket nonblocking, because if `select` tells us that the connection is ready, `accept` should not block.

Unfortunately there is a timing problem that can trip us up here [Gierth 1996]. To see this problem we modify our TCP echo client (Figure 5.4) to establish the connection and then send an RST to the server. Figure 15.21 shows this new version.

Set `SO_LINGER` socket option

16-19 Once the connection is established we set the `SO_LINGER` socket option, setting the `l_onoff` flag to 1 and the `l_linger` time to 0. As stated in Section 7.5, this causes an RST to be sent on a TCP socket when the connection is closed. We then `close` the socket.

Next, we modify our TCP server from Figures 6.21 and 6.22 to pause after `select` returns that the listening socket is readable, but before calling `accept`. In the following code from the beginning of Figure 6.22 the two lines preceded by a plus sign are new.

```
            if (FD_ISSET(listenfd, &rset)) {    /* new client connection */
+               printf("listening socket readable\n");
+               sleep(5);
                clilen = sizeof(cliaddr);
                connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
```

What we are simulating here is a busy server that cannot call `accept` as soon as `select` returns that the listening socket is readable. Normally this slowness on the part of the server is not a problem (indeed this is why a queue of completed connections is maintained), but when combined with the RST from the client, after the connection is established, we can have a problem.

In Section 5.11 we noted that when the client aborts the connection before the server calls `accept`, Berkeley-derived implementations do not return the aborted connection

nonblock/tcpcli03.c

```
 1 #include    "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd;
 6     struct linger ling;
 7     struct sockaddr_in servaddr;
 8     if (argc != 2)
 9         err_quit("usage: tcpcli <IPaddress>");
10     sockfd = Socket(AF_INET, SOCK_STREAM, 0);
11     bzero(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_port = htons(SERV_PORT);
14     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
15     Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
16     ling.l_onoff = 1;           /* cause RST to be sent on close() */
17     ling.l_linger = 0;
18     Setsockopt(sockfd, SOL_SOCKET, SO_LINGER, &ling, sizeof(ling));
19     Close(sockfd);
20     exit(0);
21 }
```

nonblock/tcpcli03.c

Figure 15.21 TCP echo client that creates connection and sends an RST.

to the server, while other implementations should return `ECONNABORTED` but often return `EPROTO` instead. Consider a Berkeley-derived implementation.

- The client establishes the connection and then aborts it as in Figure 15.21.
- `select` returns readable to the server process, but it takes the server a short time to call `accept`.
- Between the server's return from `select` and its calling `accept`, the RST is received from the client.
- The completed connection is removed from the queue and we assume that no other completed connections exist.
- The server calls `accept`, but since there are no completed connections, it blocks.

The server will remain blocked in the call to `accept` until some other client establishes a connection. But in the meantime, assuming a server like Figure 6.22, the server is blocked in the call to `accept` and will not handle any other ready descriptors.

This problem is somewhat similar to the denial of service attack described in Section 6.8, but with this new bug the server breaks out of the blocked `accept` as soon as another client establishes a connection.

The fix for this problem is to

1. always set a listening socket nonblocking if we use `select` to tell us when a connection is ready to be `accept`ed, and
2. ignore the following errors on the subsequent call to `accept`: `EWOULDBLOCK` (for Berkeley-derived implementations, when the client aborts the connection), `ECONNABORTED` (for Posix.1g implementations, when the client aborts the connection), `EPROTO` (for SVR4 implementations, when the client aborts the connection), and `EINTR` (if signals are being caught).

15.7 Summary

Our example of nonblocking reads and writes in Section 15.2 took our `str_cli` echo client and modified it to use nonblocking I/O on the TCP connection to the server. `select` is normally used with nonblocking I/O, to determine when a descriptor is readable or writable. This version of our client is the fastest version that we show, although the code modifications are nontrivial. We then showed that it is simpler to divide the client into two pieces using `fork`, and we employ the same technique using threads in Figure 23.2.

Nonblocking `connect`s let us do other processing while TCP's three-way handshake takes place, instead of being blocked in the call to `connect`. Unfortunately these are also nonportable, with different implementations having different ways of indicating that the connection completed OK or encountered an error. We used nonblocking connects to develop a new client, which is similar to a Web client, that opens multiple TCP connections at the same time to reduce the clock time required to fetch numerous files from a server. Initiating multiple connections like this can reduce the clock time but is also "network unfriendly," with regard to TCP's congestion avoidance.

Exercises

15.1 In our discussion of Figure 15.10 we mentioned that the parent must call `shutdown`, not `close`. Why?

15.2 What happens in Figure 15.10 if the server process terminates prematurely, the child receives the end-of-file and terminates, but the child does not notify the parent?

15.3 What happens in Figure 15.10 if the parent dies unexpectedly before the child, and the child then reads an end-of-file on the socket?

15.4 What happens in Figure 15.11 if we remove the two lines

```
if (n == 0)
    goto done;          /* connect completed immediately */
```

15.5 In Section 15.3 we said that it is possible for data to arrive for a socket before `connect` returns. How can this happen?

16

ioctl Operations

16.1 Introduction

The `ioctl` function has traditionally been the system interface used for everything that didn't fit into some other nicely defined category. Posix is getting rid of `ioctl`, by creating specific wrapper functions to replace `ioctl`s whose functionality is being standardized by Posix. For example, the Unix terminal interface was traditionally accessed using `ioctl` but Posix.1 created 12 new functions for terminals: `tcgetattr` to get the terminal attributes, `tcflush` to flush pending input or output, and so on. In a similar vein, Posix.1g is replacing one `ioctl`: the new `sockatmark` function (Section 21.3) replaces the `SIOCATMARK` `ioctl`. Nevertheless numerous `ioctl`s remain for implementation-dependent features related to network programming: obtaining the interface information, and accessing the routing table and the ARP cache, for example.

This chapter provides an overview of the `ioctl` requests related to network programming, but many of these are implementation dependent. Additionally, newer Berkeley-derived implementations use sockets in the `AF_ROUTE` domain (routing sockets) to accomplish many of these operations. We cover routing sockets in Chapter 17.

A common use of `ioctl` by network programs (typically servers) is to obtain information on all the host's interfaces when the program starts: the interface addresses, whether the interface supports broadcasting, whether the interface supports multicasting, and so on. We develop our own function to return this information and provide an implementation using `ioctl` in this chapter, and another implementation using routing sockets in Chapter 17.

16.2 ioctl Function

This function affects an open file, referenced by the *fd* argument.

```
#include <unistd.h>

int ioctl(int fd, int request, ... /* void *arg */ );
```

Returns: 0 if OK, –1 on error

The third argument is always a pointer, but the type of pointer depends on the *request*.

> 4.4BSD defines the second argument to be an `unsigned long` instead of an `int`, but that is not a problem, since header files define the constants that are used for this argument.
>
> Some implementations specify the third argument as a `void *` pointer, instead of the ANSI C ellipsis notation.
>
> There is no standard for the header to include to define the function prototype for `ioctl`, since it is not standardized by Posix. Many systems define it in `<unistd.h>`, as we show, but traditional BSD systems define it in `<sys/ioctl.h>`.

We can divide the *request*s related to networking into six categories.

- socket operations
- file operations
- interface operations
- ARP cache operations
- routing table operations
- streams system (Chapter 33)

Recall from Figure 7.15 that not only do some of the `ioctl` operations overlap some of the `fcntl` operations (e.g., setting a socket nonblocking), but there are also some operations that can be specified more than one way using `ioctl` (e.g., setting the process group ownership of a socket).

Figure 16.1 lists the *request*s, along with the datatype of what the *arg* address must point to. The following sections describe these requests in more detail.

16.3 Socket Operations

There are three `ioctl` requests explicitly for sockets (pp. 551–553 of TCPv2). All three require that the third argument to `ioctl` be a pointer to an integer.

`SIOCATMARK` Return through the integer pointed to by the third argument a nonzero value if the socket's read pointer is currently at the out-of-band mark, or a zero value if the read pointer is not at the out-of-band mark. We describe out-of-band data in more detail in Chapter 21. Posix.1g replaces this request with the `sockatmark` function and we show an implementation of this new function using `ioctl` in Section 21.3.

Category	*request*	Description	Datatype
socket	SIOCATMARK	at out-of-band mark ?	int
	SIOCSPGRP	set process ID or process group ID of socket	int
	SIOCGPGRP	get process ID or process group ID of socket	int
file	FIONBIO	set/clear nonblocking flag	int
	FIOASYNC	set/clear asynchronous I/O flag	int
	FIONREAD	get # bytes in receive buffer	int
	FIOSETOWN	set process ID or process group ID of file	int
	FIOGETOWN	get process ID or process group ID of file	int
interface	SIOCGIFCONF	get list of all interfaces	struct ifconf
	SIOCSIFADDR	set interface address	struct ifreq
	SIOCGIFADDR	get interface address	struct ifreq
	SIOCSIFFLAGS	set interface flags	struct ifreq
	SIOCGIFFLAGS	get interface flags	struct ifreq
	SIOCSIFDSTADDR	set point-to-point address	struct ifreq
	SIOCGIFDSTADDR	get point-to-point address	struct ifreq
	SIOCGIFBRDADDR	get broadcast address	struct ifreq
	SIOCSIFBRDADDR	set broadcast address	struct ifreq
	SIOCGIFNETMASK	get subnet mask	struct ifreq
	SIOCSIFNETMASK	set subnet mask	struct ifreq
	SIOCGIFMETRIC	get interface metric	struct ifreq
	SIOCSIFMETRIC	set interface metric	struct ifreq
	SIOC*xxx*	(many more; implementation dependent)	
ARP	SIOCSARP	create/modify ARP entry	struct arpreq
	SIOCGARP	get ARP entry	struct arpreq
	SIOCDARP	delete ARP entry	struct arpreq
routing	SIOCADDRT	add route	struct rtentry
	SIOCDELRT	delete route	struct rtentry
streams	I_*xxx*	(see Section 33.5)	

Figure 16.1 Summary of networking `ioctl` requests.

`SIOCGPGRP` Return through the integer pointed to by the third argument either the process ID or the process group ID that is set to receive the `SIGIO` or `SIGURG` signal for this socket. This request is identical to an `fcntl` of `F_GETOWN` and we note in Figure 7.15 that Posix.1g standardizes the `fcntl`.

`SIOCSPGRP` Set either the process ID or the process group ID to receive the `SIGIO` or `SIGURG` signal for this socket from the integer pointed to by the third argument. This request is identical to an `fcntl` of `F_SETOWN` and we note in Figure 7.15 that Posix.1g standardizes the `fcntl`.

16.4 File Operations

The next group of requests begin with `FIO` and may apply to certain types of files, in addition to sockets. We cover only the requests that apply to sockets (p. 553 of TCPv2).

The following five requests all require that the third argument to ioctl point to an integer.

FIONBIO	The nonblocking flag for the socket is cleared or turned on, depending whether the third argument to ioctl points to a zero or nonzero value, respectively. This request accomplishes the same effect as the O_NONBLOCK file status flag that can be set and cleared with the F_SETFL command to the fcntl function.
FIOASYNC	The flag that governs the receipt of asynchronous I/O signals (SIGIO) for the socket is cleared or turned on, depending whether the third argument to ioctl points to a zero or nonzero value, respectively. This flag accomplishes the same effect as the O_ASYNC file status flag that can be set and cleared with the F_SETFL command to the fcntl function.
FIONREAD	Return in the integer pointed to by the third argument to ioctl the number of bytes currently in the socket receive buffer. This feature also works for files, pipes, and terminals. We said more about this request in Section 13.7.
FIOSETOWN	Equivalent to SIOCSPGRP for a socket.
FIOGETOWN	Equivalent to SIOCGPGRP for a socket.

16.5 Interface Configuration

One of the first steps employed by many programs that deal with the network interfaces on a system is to obtain from the kernel all the interfaces configured on the system. This is done with the SIOCGIFCONF request, which uses the ifconf structure, which in turn uses the ifreq structure, both of which are shown in Figure 16.2.

Before calling ioctl we allocate a buffer and an ifconf structure and then initialize the latter. We show a picture of this in Figure 16.3 (p. 430), assuming our buffer size is 1024 bytes. The third argument to ioctl is a pointer to our ifconf structure.

If we assume that the kernel returns two ifreq structures, we could have the arrangement shown in Figure 16.4 (p. 430) when the ioctl returns. The shaded regions have been modified by ioctl. The buffer has been filled in with the two structures and the ifc_len member of the ifconf structure has been updated to reflect the amount of information stored in the buffer. We assume in this figure that each ifreq structure occupies 32 bytes.

A pointer to an ifreq structure is also used as an argument to the remaining interface ioctls shown in Figure 16.1, which we describe in Section 16.7. Notice that each ifreq structure contains a union and there are numerous #defines to hide the fact that these fields are members of a union. All the references to the individual members are made using the defined names. Be aware that some systems have added many implementation-dependent members to the ifr_ifru union.

<net/if.h>

```
struct ifconf {
    int  ifc_len;                   /* size of buffer, value-result */
    union {
        caddr_t ifcu_buf;           /* input from user -> kernel */
        struct  ifreq *ifcu_req;    /* return from kernel -> user */
    } ifc_ifcu;
};
#define  ifc_buf  ifc_ifcu.ifcu_buf /* buffer address */
#define  ifc_req  ifc_ifcu.ifcu_req /* array of structures returned */

#define  IFNAMSIZ    16

struct ifreq {
    char    ifr_name[IFNAMSIZ];     /* interface name, e.g., "le0" */
    union {
        struct  sockaddr ifru_addr;
        struct  sockaddr ifru_dstaddr;
        struct  sockaddr ifru_broadaddr;
        short   ifru_flags;
        int     ifru_metric;
        caddr_t ifru_data;
    } ifr_ifru;
};
#define  ifr_addr       ifr_ifru.ifru_addr      /* address */
#define  ifr_dstaddr    ifr_ifru.ifru_dstaddr   /* other end of p-to-p link */
#define  ifr_broadaddr  ifr_ifru.ifru_broadaddr /* broadcast address */
#define  ifr_flags      ifr_ifru.ifru_flags     /* flags */
#define  ifr_metric     ifr_ifru.ifru_metric    /* metric */
#define  ifr_data       ifr_ifru.ifru_data      /* for use by interface */
```

<net/if.h>

Figure 16.2 `ifconf` and `ifreq` structures used with various interface `ioctl` requests.

16.6 `get_ifi_info` Function

Since many programs need to know all the interfaces on a system, we will develop a function of our own named `get_ifi_info` that returns a linked list of structures, one for each interface that is currently "up." In this section we will implement this function using the `SIOCGIFCONF ioctl` and in Chapter 17 we will develop a version using routing sockets.

BSD/OS provides a function named `getifaddrs` with similar functionality.

Searching the entire BSD/OS 2.1 source tree shows that 12 programs issue the `SIOCGIFCONF ioctl` to determine the interfaces that are present.

We first define the `ifi_info` structure in a new header named `unpifi.h`, shown in Figure 16.5.

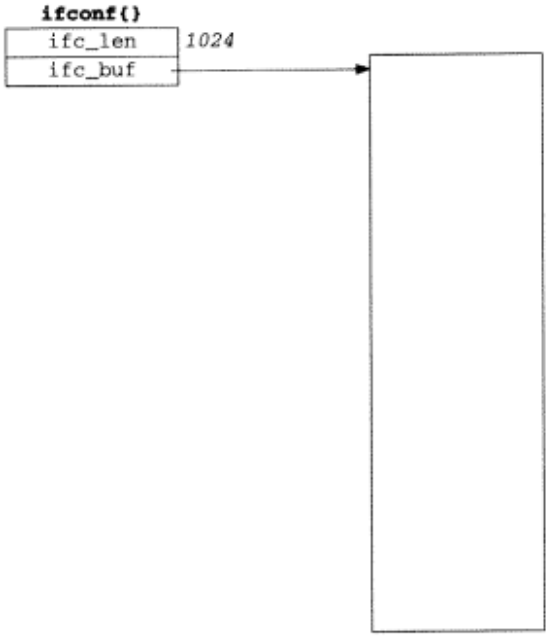

Figure 16.3 Initialization of `ifconf` structure before `SIOCGIFCONF`.

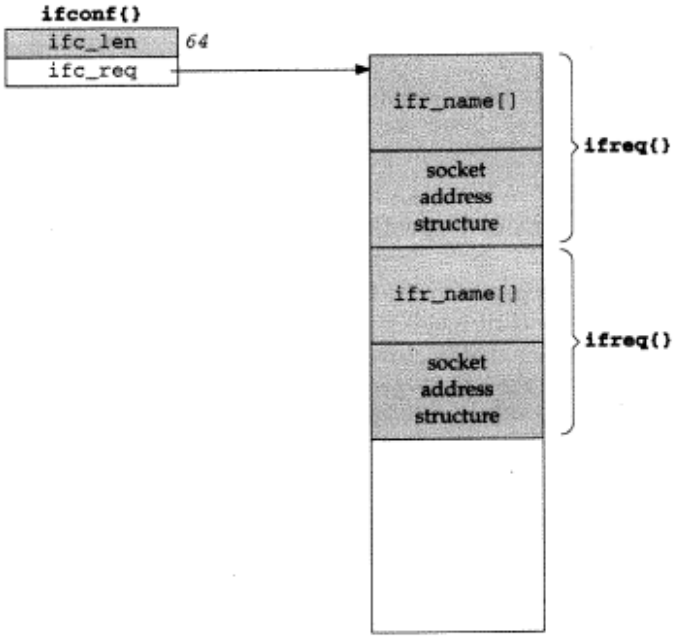

Figure 16.4 Values returned by `SIOCGIFCONF`.

ioctl/unpifi.h

```
 1 /* Our own header for the programs that need interface configuration info.
 2    Include this file, instead of "unp.h". */

 3 #ifndef __unp_ifi_h
 4 #define __unp_ifi_h

 5 #include    "unp.h"
 6 #include    <net/if.h>

 7 #define IFI_NAME    16          /* same as IFNAMSIZ in <net/if.h> */
 8 #define IFI_HADDR    8          /* allow for 64-bit EUI-64 in future */

 9 struct ifi_info {
10     char    ifi_name[IFI_NAME]; /* interface name, null terminated */
11     u_char  ifi_haddr[IFI_HADDR];   /* hardware address */
12     u_short ifi_hlen;           /* #bytes in hardware address: 0, 6, 8 */
13     short   ifi_flags;          /* IFF_xxx constants from <net/if.h> */
14     short   ifi_myflags;        /* our own IFI_xxx flags */
15     struct sockaddr *ifi_addr;  /* primary address */
16     struct sockaddr *ifi_brdaddr;   /* broadcast address */
17     struct sockaddr *ifi_dstaddr;   /* destination address */
18     struct ifi_info *ifi_next;  /* next of these structures */
19 };

20 #define IFI_ALIAS   1           /* ifi_addr is an alias */

21                     /* function prototypes */
22 struct ifi_info *get_ifi_info(int, int);
23 struct ifi_info *Get_ifi_info(int, int);
24 void    free_ifi_info(struct ifi_info *);

25 #endif  /* __unp_ifi_h */
```

ioctl/unpifi.h

Figure 16.5 unpifi.h header.

9-19 A linked list of these structures is returned by our function, each structure's ifi_next member pointing to the next one. We return in this structure just the information that a typical application is probably interested in: the interface name, the hardware address (e.g., an Ethernet address), the interface flags (to let the application determine if the interface supports broadcasting or multicasting, or is a point-to-point interface), and the interface address, the broadcast address, and the destination address for a point-to-point link. All of the memory used to hold the ifi_info structures, along with the socket address structures contained within, are obtained dynamically. Therefore we also provide a free_ifi_info function to free all this memory.

> Most hardware addresses today are 48-bit MAC addresses (e.g., Ethernet, token ring, etc.). But there is a trend toward 64-bit identifiers, called *EUI-64* [IEEE 1997b]. IPv6 addresses contain an EUI-64 value in the low-order 64 bits (Section A.5), and there is a simple way to encapsulate a 48-bit MAC address within a 64-bit EUI. We therefore allocate enough room in our ifi_info structure for a 64-bit identifier and also store the length of the hardware address.

Before showing the implementation of our get_ifi_info function, we show a simple program that calls this function and then outputs all the information. This

program is a miniature version of the `ifconfig` program and is shown in Figure 16.6.

ioctl/prifinfo.c

```
 1 #include    "unpifi.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     struct ifi_info *ifi, *ifihead;
 6     struct sockaddr *sa;
 7     u_char *ptr;
 8     int     i, family, doaliases;
 9     if (argc != 3)
10         err_quit("usage: prifinfo <inet4|inet6> <doaliases>");
11     if (strcmp(argv[1], "inet4") == 0)
12         family = AF_INET;
13 #ifdef  IPV6
14     else if (strcmp(argv[1], "inet6") == 0)
15         family = AF_INET6;
16 #endif
17     else
18         err_quit("invalid <address-family>");
19     doaliases = atoi(argv[2]);
20     for (ifihead = ifi = Get_ifi_info(family, doaliases);
21          ifi != NULL; ifi = ifi->ifi_next) {
22         printf("%s: <", ifi->ifi_name);
23         if (ifi->ifi_flags & IFF_UP)            printf("UP ");
24         if (ifi->ifi_flags & IFF_BROADCAST)     printf("BCAST ");
25         if (ifi->ifi_flags & IFF_MULTICAST)     printf("MCAST ");
26         if (ifi->ifi_flags & IFF_LOOPBACK)      printf("LOOP ");
27         if (ifi->ifi_flags & IFF_POINTOPOINT)   printf("P2P ");
28         printf(">\n");
29         if ( (i = ifi->ifi_hlen) > 0) {
30             ptr = ifi->ifi_haddr;
31             do {
32                 printf("%s%x", (i == ifi->ifi_hlen) ? "  " : ":", *ptr++);
33             } while (--i > 0);
34             printf("\n");
35         }
36         if ( (sa = ifi->ifi_addr) != NULL)
37             printf("  IP addr: %s\n",
38                    Sock_ntop_host(sa, sizeof(*sa)));
39         if ( (sa = ifi->ifi_brdaddr) != NULL)
40             printf("  broadcast addr: %s\n",
41                    Sock_ntop_host(sa, sizeof(*sa)));
42         if ( (sa = ifi->ifi_dstaddr) != NULL)
43             printf("  destination addr: %s\n",
44                    Sock_ntop_host(sa, sizeof(*sa)));
45     }
46     free_ifi_info(ifihead);
47     exit(0);
48 }
```

ioctl/prifinfo.c

Figure 16.6 `prifinfo` program that calls our `get_ifi_info` function.

20-45 The program is a `for` loop that calls `get_ifi_info` once and then steps through all the `ifi_info` structures that are returned.

22-35 The interface name and flags are all printed. If the length of the hardware address is greater than 0, it is printed as hexadecimal numbers. (Our `get_ifi_info` function returns an `ifi_hlen` of 0 if it is not available.)

36-44 The three IP addresses are printed, if returned.

If we run this program on our host `solaris` (Figure 1.16) we have the following output:

```
solaris % prifinfo inet4 0
lo0: <UP MCAST LOOP >
  IP addr: 127.0.0.1
le0: <UP BCAST MCAST >
  IP addr: 206.62.226.33
  broadcast addr: 206.62.226.63
```

The first command-line argument of `inet4` specifies IPv4 addresses, and the second argument of 0 specifies that no address aliases are to be returned (we described IP address aliases in Section A.4). Note that under Solaris the hardware address of the Ethernet interface is not available.

If we add three alias addresses to the Ethernet interface (`le0`), with host IDs of 44, 45, and 46, and if we change the second command-line argument to 1, we have

```
solaris % prifinfo inet4 1
lo0: <UP MCAST LOOP >
  IP addr: 127.0.0.1
le0: <UP BCAST MCAST >
  IP addr: 206.62.226.33              primary IP address
  broadcast addr: 206.62.226.63
le0:1: <UP BCAST MCAST >
  IP addr: 206.62.226.44              first alias
  broadcast addr: 206.62.226.63
le0:2: <UP BCAST MCAST >
  IP addr: 206.62.226.45              second alias
  broadcast addr: 206.62.226.63
le0:3: <UP BCAST MCAST >
  IP addr: 206.62.226.46              third alias
  broadcast addr: 206.62.226.63
```

If we run the same program under BSD/OS, using the implementation of `get_ifi_info` from Figure 17.16 (which can easily obtain the hardware address), we have:

```
bsdi % prifinfo inet4 1
we0: <UP BCAST MCAST >
  0:0:c0:6f:2d:40
  IP addr: 206.62.226.66
  broadcast addr: 206.62.226.95
ef0: <UP BCAST MCAST >
  0:20:af:9c:ee:95
  IP addr: 206.62.226.35              primary IP address
  broadcast addr: 206.62.226.63
```

```
ef0: <UP BCAST MCAST >
  0:20:af:9c:ee:95
  IP addr: 206.62.226.50              alias
  broadcast addr: 206.62.226.63
lo0: <UP MCAST LOOP >
  IP addr: 127.0.0.1
```

For this example we directed the program to print the aliases and we see that one alias is defined for the second Ethernet interface (`ef0`) with a host ID of 50.

> This output depends on how the interface alias addresses are established. In this example the alias address is assigned to the Ethernet interface `ef0`, and this was the common technique with BSD/OS 2.1. But with BSD/OS 3.0 the recommended technique is to assign the alias addresses to the loopback interface `lo0`.

We now show our implementation of `get_ifi_info` that uses the `SIOCGIFCONF` `ioctl`. Figure 16.7 shows the first part of the function, which obtains the interface configuration from the kernel.

lib/get_ifi_info.c

```
 1 #include    "unpifi.h"
 2 struct ifi_info *
 3 get_ifi_info(int family, int doaliases)
 4 {
 5     struct ifi_info *ifi, *ifihead, **ifipnext;
 6     int     sockfd, len, lastlen, flags, myflags;
 7     char   *ptr, *buf, lastname[IFNAMSIZ], *cptr;
 8     struct ifconf ifc;
 9     struct ifreq *ifr, ifrcopy;
10     struct sockaddr_in *sinptr;
11     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
12     lastlen = 0;
13     len = 100 * sizeof(struct ifreq);   /* initial buffer size guess */
14     for ( ; ; ) {
15         buf = Malloc(len);
16         ifc.ifc_len = len;
17         ifc.ifc_buf = buf;
18         if (ioctl(sockfd, SIOCGIFCONF, &ifc) < 0) {
19             if (errno != EINVAL || lastlen != 0)
20                 err_sys("ioctl error");
21         } else {
22             if (ifc.ifc_len == lastlen)
23                 break;          /* success, len has not changed */
24             lastlen = ifc.ifc_len;
25         }
26         len += 10 * sizeof(struct ifreq);   /* increment */
27         free(buf);
28     }
29     ifihead = NULL;
30     ifipnext = &ifihead;
31     lastname[0] = 0;
```

lib/get_ifi_info.c

Figure 16.7 Issue `SIOCGIFCONF` request to obtain interface configuration.

Create an Internet socket

11 We create a UDP socket that will be used with the `ioctl`s. Either a TCP or a UDP socket can be used (p. 163 of TCPv2).

Issue `SIOCGIFCONF` request in a loop

12-28 A fundamental problem with the `SIOCGIFCONF` request is that some implementations do not return an error if the buffer is not large enough to hold the result. Instead, the result is truncated and success is returned (a return value of 0 from `ioctl`). This means the only way we know that our buffer is large enough is to issue the request, save the return length, issue the request again with a larger buffer, and compare the length with the saved value. Only if the two lengths are the same is our buffer large enough.

> Berkeley-derived implementations do not return an error if the buffer is too small (pp. 118–119 of TCPv2); the result is just truncated to fit the available buffer. Solaris 2.5, on the other hand, returns `EINVAL` if the returned length would be greater than or equal to the buffer length. But we cannot assume success if the returned length is less than the buffer size, because Berkeley-derived implementations can return less than the buffer size if another structure does not fit.
>
> Some implementations provide a `SIOCGIFNUM` request that returns the number of interfaces. This allows the application to then allocate a buffer of sufficient size before issuing the `SIOCGIFCONF` request, but this new request is not widespread.
>
> Allocating a fixed-sized buffer for the result from the `SIOCGIFCONF` request has become a problem with the growth of the Web, because large Web servers are allocating many alias addresses to a single interface. Solaris 2.5, for example, had a limit of 256 aliases per interface, but this limit increases to 8192 with 2.6. Sites with large numbers of aliases discovered that programs with fixed-size buffers for the interface information started failing. Even though Solaris returns an error if the buffer is too small, these programs allocate their fixed-size buffer, issue the `ioctl`, but then die if an error was returned.

12-15 We dynamically allocate a buffer, starting with room for 100 `ifreq` structures. We also keep track of the length returned by the last `SIOCGIFCONF` request in `lastlen` and initialize this to 0.

19-20 If an error of `EINVAL` is returned by `ioctl`, and we have not yet had a successful return (i.e., `lastlen` is still 0), we have not yet allocated a buffer large enough and continue through the loop.

22-23 If `ioctl` returns OK, then if the returned length equals `lastlen`, the length has not changed (our buffer is large enough) and we `break` out of the loop since we have all the information.

26-27 Each time around the loop we increase the buffer size to hold 10 more `ifreq` structures.

Initialize linked list pointers

29-31 Since we will be returning a pointer to the head of a linked list of `ifi_info` structures, we use the two variables `ifihead` and `ifipnext` to hold pointers to the list as we build it. This is the same technique that we described with Figure 11.34.

The next part of our `get_ifi_info` function, the beginning of the main loop, is shown in Figure 16.8.

lib/get_ifi_info.c

```
32      for (ptr = buf; ptr < buf + ifc.ifc_len;) {
33          ifr = (struct ifreq *) ptr;

34  #ifdef  HAVE_SOCKADDR_SA_LEN
35          len = max(sizeof(struct sockaddr), ifr->ifr_addr.sa_len);
36  #else
37          switch (ifr->ifr_addr.sa_family) {
38  #ifdef  IPV6
39          case AF_INET6:
40              len = sizeof(struct sockaddr_in6);
41              break;
42  #endif
43          case AF_INET:
44          default:
45              len = sizeof(struct sockaddr);
46              break;
47          }
48  #endif  /* HAVE_SOCKADDR_SA_LEN */
49          ptr += sizeof(ifr->ifr_name) + len;     /* for next one in buffer */

50          if (ifr->ifr_addr.sa_family != family)
51              continue;           /* ignore if not desired address family */

52          myflags = 0;
53          if ( (cptr = strchr(ifr->ifr_name, ':')) != NULL)
54              *cptr = 0;          /* replace colon will null */
55          if (strncmp(lastname, ifr->ifr_name, IFNAMSIZ) == 0) {
56              if (doaliases == 0)
57                  continue;       /* already processed this interface */
58              myflags = IFI_ALIAS;
59          }
60          memcpy(lastname, ifr->ifr_name, IFNAMSIZ);

61          ifrcopy = *ifr;
62          Ioctl(sockfd, SIOCGIFFLAGS, &ifrcopy);
63          flags = ifrcopy.ifr_flags;
64          if ((flags & IFF_UP) == 0)
65              continue;           /* ignore if interface not up */

66          ifi = Calloc(1, sizeof(struct ifi_info));
67          *ifipnext = ifi;        /* prev points to this new one */
68          ifipnext = &ifi->ifi_next;  /* pointer to next one goes here */

69          ifi->ifi_flags = flags; /* IFF_xxx values */
70          ifi->ifi_myflags = myflags;     /* IFI_xxx values */
71          memcpy(ifi->ifi_name, ifr->ifr_name, IFI_NAME);
72          ifi->ifi_name[IFI_NAME - 1] = '\0';
```

lib/get_ifi_info.c

Figure 16.8 Process interface configuration.

Step to next socket address structure

32-49 As we loop through all the `ifreq` structures, `ifr` points to each structure and we then increment `ptr` to point to the next one. But we must deal with newer systems that provide a length field for socket address structures, and older systems that do not provide this length. Even though the declaration in Figure 16.2 declares the socket address structure contained within the `ifreq` structure as a generic socket address structure, on newer systems this can be any type of socket address structure. Indeed, on 4.4BSD a datalink socket address structure is also returned for each interface (p. 118 of TCPv2). Therefore if the length member is supported, we must use its value to update our pointer to the next socket address structure. Otherwise we use a length based on the address family, using the size of the generic socket address structure (16 bytes) as the default.

> On systems that support IPv6, there is no standard as to whether or not the `SIOCGIFCONF` request returns IPv6 addresses. We put in a `case` for IPv6, for newer systems, just in case. The problem is that the `union` in the `ifreq` structure defines the returned addresses as generic 16-byte `sockaddr` structures, which are adequate for 16-byte IPv4 `sockaddr_in` structures, but too small for 24-byte IPv6 `sockaddr_in6` structures. If IPv6 addresses were returned, it would probably break existing code that assumes a fixed-size `sockaddr` structure in each `ifreq` structure.

50-51 We ignore any addresses from families other than those desired by the caller.

Handle aliases

52-60 We must detect any aliases that may exist for the interface, that is, additional addresses that have been assigned to the interface. Note from our examples following Figure 16.6 that under Solaris the interface name for an alias contains a colon, while under 4.4BSD the interface name does not change for an alias. To handle both cases we save the last interface name in `lastname` and only compare up to a colon, if present. If a colon is not present, we still ignore this interface if the name is equivalent to the last interface that we processed.

Fetch interface flags

61-65 We issue an `ioctl` of `SIOCGIFFLAGS` (Section 16.5) to fetch the interface flags. The third argument to `ioctl` is a pointer to an `ifreq` structure that must contain the name of the interface for which we want the flags. We make a copy of the `ifreq` structure before issuing the `ioctl`, because if we didn't, this request would overwrite the IP address of the interface, since both are members of the same `union` in Figure 16.2. If the interface is not up, we ignore it.

Allocate and initialize `ifi_info` structure

66-72 At this point we know that we will return this interface to the caller. We allocate memory for our `ifi_info` structure and add it to the end of the linked list that we are building. We copy the interface flags and name into the structure. We make certain that the interface name is null terminated, and since `calloc` initializes the allocated region to all zero bits, we know that `ifi_hlen` is initialized to 0 and that `ifi_next` is initialized to a null pointer.

Figure 16.9 contains the last part of our function.

ioctl/get_ifi_info.c

```
 73          switch (ifr->ifr_addr.sa_family) {
 74          case AF_INET:
 75              sinptr = (struct sockaddr_in *) &ifr->ifr_addr;
 76              if (ifi->ifi_addr == NULL) {
 77                  ifi->ifi_addr = Calloc(1, sizeof(struct sockaddr_in));
 78                  memcpy(ifi->ifi_addr, sinptr, sizeof(struct sockaddr_in));

 79 #ifdef  SIOCGIFBRDADDR
 80                  if (flags & IFF_BROADCAST) {
 81                      Ioctl(sockfd, SIOCGIFBRDADDR, &ifrcopy);
 82                      sinptr = (struct sockaddr_in *) &ifrcopy.ifr_broadaddr;
 83                      ifi->ifi_brdaddr = Calloc(1, sizeof(struct sockaddr_in));
 84                      memcpy(ifi->ifi_brdaddr, sinptr, sizeof(struct sockaddr_in));
 85                  }
 86 #endif

 87 #ifdef  SIOCGIFDSTADDR
 88                  if (flags & IFF_POINTOPOINT) {
 89                      Ioctl(sockfd, SIOCGIFDSTADDR, &ifrcopy);
 90                      sinptr = (struct sockaddr_in *) &ifrcopy.ifr_dstaddr;
 91                      ifi->ifi_dstaddr = Calloc(1, sizeof(struct sockaddr_in));
 92                      memcpy(ifi->ifi_dstaddr, sinptr, sizeof(struct sockaddr_in));
 93                  }
 94 #endif
 95              }
 96              break;

 97          default:
 98              break;
 99          }
100      }
101      free(buf);
102      return (ifihead);           /* pointer to first structure in linked list */
103  }
```

ioctl/get_ifi_info.c

Figure 16.9 Fetch and return interface addresses.

73–78 We copy the IP address that was returned from our original `SIOCGIFCONF` request in the structure we are building.

79–96 If the interface supports broadcasting, we fetch the broadcast address with an `ioctl` of `SIOCGIFBRDADDR`. We allocate memory for the socket address structure containing this address and add it to the `ifi_info` structure that we are building. Similarly, if the interface is a point-to-point interface, the `SIOCGIFDSTADDR` returns the IP address of the other end of the link.

> There is not a `case` for `AF_INET6` because as we mentioned earlier, it is not known whether IPv6 implementations will return IPv6 addresses with the `SIOCGIFCONF` request or not.

Figure 16.10 shows the `free_ifi_info` function, which takes a pointer that was returned by `get_ifi_info` and frees all the dynamic memory.

ioctl/get_ifi_info.c

```
104 void
105 free_ifi_info(struct ifi_info *ifihead)
106 {
107     struct ifi_info *ifi, *ifinext;

108     for (ifi = ifihead; ifi != NULL; ifi = ifinext) {
109         if (ifi->ifi_addr != NULL)
110             free(ifi->ifi_addr);
111         if (ifi->ifi_brdaddr != NULL)
112             free(ifi->ifi_brdaddr);
113         if (ifi->ifi_dstaddr != NULL)
114             free(ifi->ifi_dstaddr);
115         ifinext = ifi->ifi_next;    /* can't fetch ifi_next after free() */
116         free(ifi);                  /* the ifi_info{} itself */
117     }
118 }
```

ioctl/get_ifi_info.c

Figure 16.10 `free_ifi_info` function: free dynamic memory allocated by `get_ifi_info`.

16.7 Interface Operations

As we showed in the previous section, the `SIOCGIFCONF` request returns the name and a socket address structure for each interface that is configured. There are a multitude of other requests that we can then issue to set or get all the other characteristics of the interface. The *get* version of these requests (`SIOCGxxx`) is often issued by the `netstat` program, and the *set* version (`SIOCSxxx`) is often issued by the `ifconfig` program. Any user can get the interface information, while it takes superuser privileges to set the information.

These requests take or return an `ifreq` structure whose address is specified as the third argument to `ioctl`. The interface is always identified by its name: `le0`, `lo0`, `ppp0`, or whatever in the `ifr_name` member.

Many of these requests use a socket address structure to specify or return an IP address or address mask with the application. For IPv4, the address or mask is contained in the `sin_addr` member of an Internet socket address structure.

`SIOCGIFADDR`	Return the unicast address in the `ifr_addr` member.
`SIOCSIFADDR`	Sets the interface address from the `ifr_addr` member. The initialization function for the interface is also called.
`SIOCGIFFLAGS`	Return the interface flags in the `ifr_flags` member. The names of the various flags are `IFF_xxx` and are defined by including the `<net/if.h>` header. The flags indicate, for example, if the interface is up (`IFF_UP`), if the interface is a point-to-point interface (`IFF_POINTOPOINT`), if the interface supports broadcasting (`IFF_BROADCAST`), and so on.
`SIOCSIFFLAGS`	Set the interface flags from the `ifr_flags` member.

SIOCGIFDSTADDR Return the point-to-point address in the ifr_dstaddr member.

SIOCSIFDSTADDR Set the point-to-point address from the ifr_dstaddr member.

SIOCGIFBRDADDR Return the broadcast address in the ifr_broadaddr member. The application must first fetch the interface flags and then issue the correct request: SIOCGIFBRDADDR for a broadcast interface or SIOCGIFDSTADDR for a point-to-point interface.

SIOCSIFBRDADDR Set the broadcast address from the ifr_broadaddr member.

SIOCGIFNETMASK Return the subnet mask in the ifr_addr member.

SIOCSIFNETMASK Set the subnet mask from the ifr_addr member.

SIOCGIFMETRIC Return the interface metric in the ifr_metric member. The interface metric is maintained by the kernel for each interface but is used by the routing daemon routed. The interface metric is added to the hop count (to make an interface less favorable).

SIOCSIFMETRIC Set the interface routing metric from the ifr_metric member.

In this section we have described the generic interface requests. Many implementations have added additional requests.

16.8 ARP Cache Operations

The ARP cache is also manipulated with the ioctl function. These requests use an arpreq structure, shown in Figure 16.11 and defined by including the <net/if_arp.h> header.

<net/if_arp.h>

```
struct arpreq {
    struct  sockaddr  arp_pa;      /* protocol address */
    struct  sockaddr  arp_ha;      /* hardware address */
    int               arp_flags;   /* flags */
};

#define  ATF_INUSE       0x01  /* entry in use */
#define  ATF_COM         0x02  /* completed entry (hardware addr valid) */
#define  ATF_PERM        0x04  /* permanent entry */
#define  ATF_PUBL        0x08  /* published entry (respond for other host) */
```

<net/if_arp.h>

Figure 16.11 arpreq structure used with ioctl requests for ARP cache.

The third argument to ioctl must point to one of these structures. The following three *request*s are supported:

SIOCSARP Add a new entry to the ARP cache or modify an existing entry. arp_pa is an Internet socket address structure containing the IP address and

`arp_ha` is a generic socket address structure with `sa_family` set to `AF_UNSPEC` and `sa_data` containing the hardware address (e.g., the 6-byte Ethernet address). The two flags `ATF_PERM` and `ATF_PUBL` can be specified by the application. The other two flags, `ATF_INUSE` and `ATF_COM`, are set by the kernel.

`SIOCDARP` Delete an entry from the ARP cache. The caller specifies the Internet address for the entry to be deleted.

`SIOCGARP` Get an entry from the ARP cache. The caller specifies the Internet address and the corresponding Ethernet address is returned along with the flags.

Only the superuser can add or delete an entry. These three requests are normally issued by the `arp` program.

These ARP-related `ioctl` requests are not supported on some newer systems, which use routing sockets for these ARP operations.

Notice that there is no way with `ioctl` to list all the entries in the ARP cache. Most versions of the `arp` command, when invoked with the `-a` flag (list all entries in the ARP cache), read the kernel's memory (`/dev/kmem`) to obtain the current contents of the ARP cache. We will see an easier (and better) way to do this using `sysctl` in Section 17.4.

Example: Print Hardware Addresses of Host

We now use our `my_addrs` function from Figure 9.7 to return all of a host's IP addresses, followed by an `ioctl` of `SIOCGARP` for each IP address, to obtain and print the hardware addresses. We show our program in Figure 16.12.

Get list of addresses and loop through each one

12-13 We call `my_addrs` to obtain the host's IP addresses and then loop through each address.

Print IP address

14-17 We print the IP address using `inet_ntop` and then `switch` based on the address family returned by `my_addrs`. We handle only IPv4 addresses, as vendors will probably not support IPv6 addresses with the `SIOCGARP` request.

Issue `ioctl` and print hardware address

18-26 We fill in the `arp_pa` structure as an IPv4 socket address structure containing the IPv4 address. `ioctl` is called and the resulting hardware address is printed.

Running this program on our `solaris` host gives:

```
solaris % prmac
206.62.226.33: 8:0:20:78:e3:e3
```

ioctl/prmac.c

```
 1 #include    "unp.h"
 2 #include    <net/if_arp.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     family, sockfd;
 7     char    str[INET6_ADDRSTRLEN];
 8     char  **pptr;
 9     unsigned char *ptr;
10     struct arpreq arpreq;
11     struct sockaddr_in *sin;

12     pptr = my_addrs(&family);
13     for ( ; *pptr != NULL; pptr++) {
14         printf("%s: ", Inet_ntop(family, *pptr, str, sizeof(str)));
15         switch (family) {
16         case AF_INET:
17             sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

18             sin = (struct sockaddr_in *) &arpreq.arp_pa;
19             bzero(sin, sizeof(struct sockaddr_in));
20             sin->sin_family = AF_INET;
21             memcpy(&sin->sin_addr, *pptr, sizeof(struct in_addr));

22             Ioctl(sockfd, SIOCGARP, &arpreq);

23             ptr = &arpreq.arp_ha.sa_data[0];
24             printf("%x:%x:%x:%x:%x:%x\n", *ptr, *(ptr + 1),
25                    *(ptr + 2), *(ptr + 3), *(ptr + 4), *(ptr + 5));
26             break;

27         default:
28             err_quit("unsupported address family: %d", family);
29         }
30     }
31     exit(0);
32 }
```

ioctl/prmac.c

Figure 16.12 Print a host's hardware addresses.

16.9 Routing Table Operations

Two `ioctl` requests are provided to operate on the routing table. These two requests require that the third argument to `ioctl` be a pointer to an `rtentry` structure, which is defined by including the <net/route.h> header. These requests are normally issued by the `route` program. Only the superuser can issue these requests.

`SIOCADDRT` Add an entry to the routing table.

`SIOCDELRT` Delete an entry from the routing table.

There is no way with `ioctl` to list all the entries in the routing table. This operation is usually performed by the `netstat` program when invoked with the `-r` flag. This program obtains the routing table by reading the kernel's memory (`/dev/kmem`). As with listing the ARP cache, we will see an easier (and better) way to do this using `sysctl` in Section 17.4.

16.10 Summary

The `ioctl` commands that are used in network programs can be divided into six categories:

- socket operations (are we at the out-of-band mark?),
- file operations (set or clear the nonblocking flag),
- interface operations (return interface list, obtain broadcast address),
- ARP table operations (create, modify, get, delete),
- routing table operations (add or delete), and
- streams system (Chapter 33).

We will use the socket and file operations and obtaining the interface list is such a common operation that we developed our own function to do this. We will use this function numerous times in the remainder of the text. Only a few specialized programs use the `ioctl`s with the ARP cache and the routing table.

Exercises

16.1 In Section 16.7 we said that the broadcast address returned by the `SIOCGIFBRDADDR` request is returned in the `ifr_broadaddr` member. But on p. 173 of TCPv2 notice that it is returned in the `ifr_dstaddr` member. Does this matter?

16.2 Modify the `get_ifi_info` program to issue its first `SIOCGIFCONF` request for one `ifreq` structure and then increment the length each time around the loop by the size of one of these structures. Then put some statements in the loop to print the buffer size each time the request is issued, whether or not `ioctl` returns an error, and upon success print the returned buffer length. Run the `prifinfo` program and see how your system handles this request when the buffer size is too small. Also print the address family for any returned structures whose address family is not the desired value to see what other structures are returned by your system.

16.3 Modify the `get_ifi_info` function to return information about an alias address if the additional address is on a different subnet from the previous address for this interface. That is, our version in Section 16.6 ignored the aliases 206.62.226.44 through 206.62.226.46, which is OK since they are on the same subnet as the primary address for the interface, 206.62.226.33. But if, in this example, an alias is on a different subnet, say 192.3.4.5, return an `ifi_info` structure with the information about the additional address.

16.4 If your system supports the `SIOCGIFNUM` `ioctl`, then modify Figure 16.7 to issue this request and use the return value as the initial buffer size guess.

17

Routing Sockets

17.1 Introduction

Traditionally the Unix routing table within the kernel has been accessed using `ioctl` commands. In Section 16.9 we described the two commands that are provided: `SIOCADDRT` and `SIOCDELRT`, to add or delete a route. We also mentioned that no command exists to dump the entire routing table, and instead programs such as `netstat` read the kernel memory to obtain the contents of the routing table. One additional piece to this hodgepodge is that routing daemons such as `gated` need to monitor ICMP redirect messages that are received by the kernel, and they often do this by creating a raw ICMP socket (Chapter 25) and listening on this socket to all received ICMP messages.

4.3BSD Reno cleaned up the interface to the kernel's routing subsystem by creating the `AF_ROUTE` domain. The only type of socket supported in the route domain is a raw socket. Three types of operations are supported on a routing socket.

1. A process can send a message to the kernel by writing to a routing socket. For example, this is how routes are added and deleted.
2. A process can read a message from the kernel on a routing socket. This is how the kernel notifies a process that an ICMP redirect has been received and processed.

   Some operations involve both steps: for example, the process sends a message to the kernel on a routing socket asking for all the information on a given route, and the process reads back the response from the kernel on the routing socket.
3. A process can use the `sysctl` function (Section 17.4) to either dump the routing table or to list all the configured interfaces.

The first two operations require superuser privileges, while the last operation can be performed by any process.

Technically, the third operation is not performed using a routing socket but invokes the generic `sysctl` function. But we will see that one of the input parameters is the address family, which is `AF_ROUTE` for the operations we describe in this chapter, and the information returned is in the same format as the information returned by the kernel on a routing socket. Indeed, the `sysctl` processing for the `AF_ROUTE` family is part of the routing socket code in a 4.4BSD kernel (pp. 632–643 of TCPv2).

The `sysctl` utility appeared in 4.4BSD. Unfortunately not all implementations that support routing sockets provide `sysctl`. For example, AIX 4.2, Digital Unix 4.0, and Solaris 2.6 all support routing sockets, but none supports `sysctl`.

17.2 Datalink Socket Address Structure

We will encounter the datalink socket address structures as return values contained in some of the messages returned on a routing socket. Figure 17.1 shows the definition of the structure, defined by including `<net/if_dl.h>`.

```
struct sockaddr_dl {
  uint8_t      sdl_len;
  sa_family_t  sdl_family;   /* AF_LINK */
  uint16_t     sdl_index;    /* system assigned index, if > 0 */
  uint8_t      sdl_type;     /* IFT_ETHER, etc. from <net/if_types.h> */
  uint8_t      sdl_nlen;     /* name length, starting in sdl_data[0] */
  uint8_t      sdl_alen;     /* link-layer address length */
  uint8_t      sdl_slen;     /* link-layer selector length */
  char         sdl_data[12]; /* minimum work area, can be larger;
                                contains i/f name and link-layer address */
};
```

Figure 17.1 Datalink socket address structure.

Each interface has a unique positive index, and we will see this returned by the `if_nametoindex` and `if_nameindex` functions later in this chapter, along with the IPv6 multicasting socket options in Chapter 19.

The `sdl_data` member contains both the name and link-layer address (e.g., the 48-bit MAC address for an Ethernet interface). The name begins at `sdl_data[0]` and is not null terminated. The link-layer address begins `sdl_nlen` bytes after the name. This header defines the following macro to return the pointer to the link-layer address:

```
#define LLADDR(s)   ((caddr_t)((s)->sdl_data + (s)->sdl_nlen))
```

These socket address structures are variable length (p. 89 of TCPv2). If the link-layer address and name exceed 12 bytes, the structure will be larger than 20 bytes. The size is normally rounded up to the next multiple of 4 bytes, on 32-bit systems. We will also see in Figure 20.3 that when one of these structures is returned by the `IP_RECVIF` socket option, all three lengths are 0 and there is no `sdl_data` member at all.

17.3 Reading and Writing

After a process creates a routing socket, it can send commands to the kernel by writing to the socket and read information from the kernel by reading from the socket. There are 12 different routing commands, 5 of which can be issued by the process. These are defined by including the <net/route.h> header and are shown in Figure 17.2.

Message type	To kernel?	From kernel?	Description	Structure type
RTM_ADD	•	•	add route	rt_msghdr
RTM_CHANGE	•	•	change gateway, metrics, or flags	rt_msghdr
RTM_DELADDR		•	address being removed from interface	ifa_msghdr
RTM_DELETE	•	•	delete route	rt_msghdr
RTM_GET	•	•	report metrics and other route information	rt_msghdr
RTM_IFINFO		•	interface going up, down, etc.	if_msghdr
RTM_LOCK	•	•	lock specified metrics	rt_msghdr
RTM_LOSING		•	kernel suspects route is failing	rt_msghdr
RTM_MISS		•	lookup failed on this address	rt_msghdr
RTM_NEWADDR		•	address being added to interface	ifa_msghdr
RTM_REDIRECT		•	kernel told to use different route	rt_msghdr
RTM_RESOLVE		•	request to resolve destination to link-layer address	rt_msghdr

Figure 17.2 Types of messages exchanged across a routing socket.

Three different structures are exchanged across a routing socket, as shown in the final column of this figure: rt_msghdr, if_msghdr, and ifa_msghdr, which we show in Figure 17.3.

```
struct rt_msghdr {     /* from <net/route.h> */
  u_short  rtm_msglen;   /* to skip over non-understood messages */
  u_char   rtm_version;  /* future binary compatibility */
  u_char   rtm_type;     /* message type */

  u_short  rtm_index;    /* index for associated ifp */
  int      rtm_flags;    /* flags, incl. kern & message, e.g., DONE */
  int      rtm_addrs;    /* bitmask identifying sockaddrs in msg */
  pid_t    rtm_pid;      /* identify sender */
  int      rtm_seq;      /* for sender to identify action */
  int      rtm_errno;    /* why failed */
  int      rtm_use;      /* from rtentry */
  u_long   rtm_inits;    /* which metrics we are initializing */
  struct rt_metrics  rtm_rmx; /* metrics themselves */
};

struct if_msghdr {     /* from <net/if.h> */
  u_short  ifm_msglen;   /* to skip over non-understood messages */
  u_char   ifm_version;  /* future binary compatibility */
  u_char   ifm_type;     /* message type */

  int      ifm_addrs;    /* like rtm_addrs */
  int      ifm_flags;    /* value of if_flags */
  u_short  ifm_index;    /* index for associated ifp */
  struct if_data  ifm_data;/* statistics and other data about if */
};
```

```
struct ifa_msghdr {   /* from <net/if.h> */
  u_short  ifam_msglen;  /* to skip over non-understood messages */
  u_char   ifam_version; /* future binary compatibility */
  u_char   ifam_type;    /* message type */

  int      ifam_addrs;   /* like rtm_addrs */
  int      ifam_flags;   /* value of ifa_flags */
  u_short  ifam_index;   /* index for associated ifp */
  int      ifam_metric;  /* value of ifa_metric */
};
```

Figure 17.3 The three structures returned with routing messages.

The first three members of each structure are the same: length, version, and type of message. The type is one of the constants from the first column in Figure 17.2. The length member allows an application to skip over message types that it does not understand.

The members `rtm_addrs`, `ifm_addrs`, and `ifam_addrs` are bitmasks specifying which of eight possible socket address structures follow the message. Figure 17.4 shows the constants and values for this bitmask, which are defined by including the `<net/route.h>` header.

Bitmask		Array index		Socket address structure containing
Constant	Value	Constant	Value	
RTA_DST	0x01	*RTAX_DST*	0	destination address
RTA_GATEWAY	0x02	*RTAX_GATEWAY*	1	gateway address
RTA_NETMASK	0x04	*RTAX_NETMASK*	2	network mask
RTA_GENMASK	0x08	*RTAX_GENMASK*	3	cloning mask
RTA_IFP	0x10	*RTAX_IFP*	4	interface name
RTA_IFA	0x20	*RTAX_IFA*	5	interface address
RTA_AUTHOR	0x40	*RTAX_AUTHOR*	6	author of redirect
RTA_BRD	0x80	*RTAX_BRD*	7	broadcast or point-to-point destination address
		RTAX_MAX	8	max #elements

Figure 17.4 Constants used to refer to socket address structures in routing messages.

When multiple socket address structures are present, they are always in the order shown in the table.

Example: Fetch and Print a Routing Table Entry

We now show an example using routing sockets. Our program takes a command-line argument consisting of an IPv4 dotted-decimal address and sends an `RTM_GET` message to the kernel for this address. The kernel looks up the address in its IPv4 routing table and returns an `RTM_GET` message with information about the routing table entry. For example, if we execute

```
bsdi # getrt 4.5.6.7
dest: 0.0.0.0
gateway: 206.62.226.62
netmask: 0.0.0.0
```

on our host `bsdi`, we see that this destination address uses the default route (which is stored in the routing table with a destination IP address of 0.0.0.0 and a mask of 0.0.0.0). The next-hop router is our router `gw` (recall Figure 1.16). If we execute

```
bsdi # getrt 206.62.226.32
dest: 206.62.226.32
gateway: AF_LINK, index=2
netmask: 255.255.255.224
```

specifying the main Ethernet as the destination, the destination is the network itself. The gateway is now the outgoing interface, returned as a `sockaddr_dl` structure with an interface index of 2.

Before showing the source code we show what we write to the routing socket in Figure 17.5 along with what is returned by the kernel.

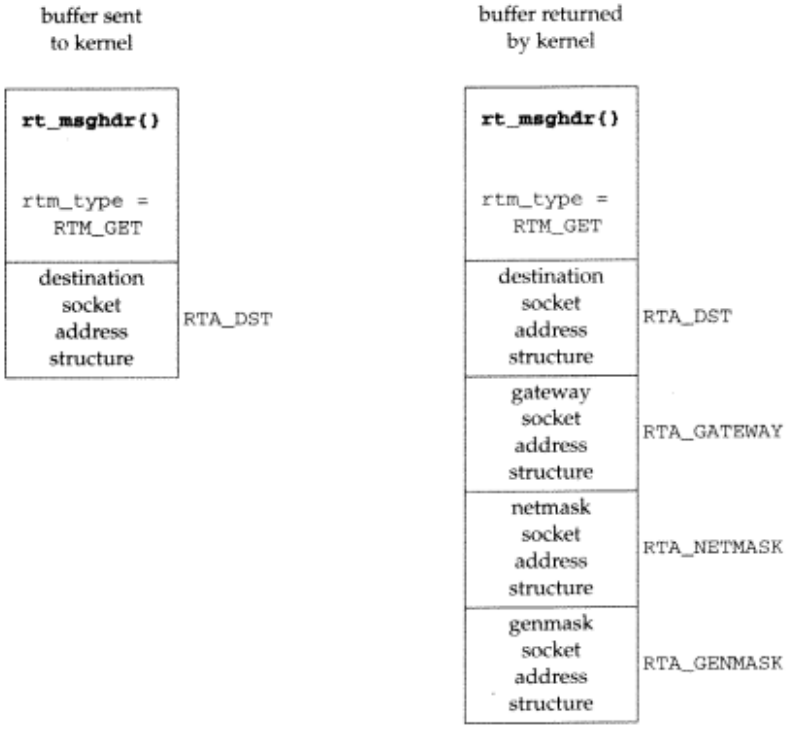

Figure 17.5 Data exchanged with kernel across routing socket for `RTM_GET` command.

We build a buffer containing an `rt_msghdr` structure followed by a socket address structure containing the destination address for the kernel to look up. The `rtm_type` is `RTM_GET` and the `rtm_addrs` is `RTA_DST` (recall Figure 17.4), indicating that the only socket address structure following the `rt_msghdr` structure is one containing the destination address. This command can be used with any protocol family (that provides a routing table) because the family of the address to look up is contained in the socket address structure.

After sending the message to the kernel we `read` back the reply, and it has the format shown in the right of Figure 17.5: an `rt_msghdr` structure followed by up to four socket address structures. Which of the four socket address structures get returned depends on the routing table entry and we are told which of the four by the value in the `rtm_addrs` member of the returned `rt_msghdr` structure. The family of each socket address structure is contained in the `sa_family` member, and as we saw in our examples earlier, one time the returned gateway was an IPv4 socket address structure and the next time it was a datalink socket address structure.

Figure 17.6 shows the first part of our program.

route/getrt.c

```
 1 #include    "unproute.h"
 2 #define BUFLEN  (sizeof(struct rt_msghdr) + 512)
 3                     /* sizeof(struct sockaddr_in6) * 8 = 192 */
 4 #define SEQ     9999
 5 int
 6 main(int argc, char **argv)
 7 {
 8     int     sockfd;
 9     char   *buf;
10     pid_t   pid;
11     ssize_t n;
12     struct rt_msghdr *rtm;
13     struct sockaddr *sa, *rti_info[RTAX_MAX];
14     struct sockaddr_in *sin;
15     if (argc != 2)
16         err_quit("usage: getrt <IPaddress>");
17     sockfd = Socket(AF_ROUTE, SOCK_RAW, 0);     /* need superuser privileges */
18     buf = Calloc(1, BUFLEN);    /* and initialized to 0 */
19     rtm = (struct rt_msghdr *) buf;
20     rtm->rtm_msglen = sizeof(struct rt_msghdr) + sizeof(struct sockaddr_in);
21     rtm->rtm_version = RTM_VERSION;
22     rtm->rtm_type = RTM_GET;
23     rtm->rtm_addrs = RTA_DST;
24     rtm->rtm_pid = pid = getpid();
25     rtm->rtm_seq = SEQ;
26     sin = (struct sockaddr_in *) (rtm + 1);
27     sin->sin_len = sizeof(struct sockaddr_in);
28     sin->sin_family = AF_INET;
29     Inet_pton(AF_INET, argv[1], &sin->sin_addr);
30     Write(sockfd, rtm, rtm->rtm_msglen);
31     do {
32         n = Read(sockfd, rtm, BUFLEN);
33     } while (rtm->rtm_type != RTM_GET || rtm->rtm_seq != SEQ ||
34              rtm->rtm_pid != pid);
```

route/getrt.c

Figure 17.6 First half of program to issue `RTM_GET` command on routing socket.

1-3 Our `unproute.h` header includes some files that are needed and then includes our `unp.h` file. The constant `BUFLEN` is the size of the buffer that we allocate to hold our message to the kernel, along with the kernel's reply. We need room for one `rt_msghdr` structure and possibly eight socket address structures (the maximum number that are ever returned on a routing socket). Since an IPv6 socket address structure is 24 bytes in size, the value of 512 is more than adequate.

Create routing socket

17 We create a raw socket in the `AF_ROUTE` domain, and as we said earlier, this requires superuser privileges. A buffer is allocated and initialized to 0.

Fill in `rt_msghdr` structure

18-25 We fill in the structure with our request. We store our process ID and a sequence number of our choosing in the structure. We will compare these values in the responses that we read, looking for the correct reply.

Fill in Internet socket address structure with destination

26-29 Following the `rt_msghdr` structure, we build a `sockaddr_in` structure containing the destination IPv4 address for the kernel to look up in its routing table. All we set are the address length, the address family, and the address.

`write` message to kernel and `read` reply

30-34 We `write` the message to the kernel and `read` back the reply. Since other processes may have routing sockets open, and since the kernel passes a copy of all routing messages to all routing sockets, we must check the message type, sequence number, and process ID to verify that the message received is the one we are waiting for.

The last half of this program is shown in Figure 17.7. This half processes the reply.

route/getrt.c

```
35      rtm = (struct rt_msghdr *) buf;
36      sa = (struct sockaddr *) (rtm + 1);
37      get_rtaddrs(rtm->rtm_addrs, sa, rti_info);
38      if ( (sa = rti_info[RTAX_DST]) != NULL)
39          printf("dest: %s\n", Sock_ntop_host(sa, sa->sa_len));

40      if ( (sa = rti_info[RTAX_GATEWAY]) != NULL)
41          printf("gateway: %s\n", Sock_ntop_host(sa, sa->sa_len));

42      if ( (sa = rti_info[RTAX_NETMASK]) != NULL)
43          printf("netmask: %s\n", Sock_masktop(sa, sa->sa_len));

44      if ( (sa = rti_info[RTAX_GENMASK]) != NULL)
45          printf("genmask: %s\n", Sock_masktop(sa, sa->sa_len));

46      exit(0);
47  }
```

route/getrt.c

Figure 17.7 Last half of program to issue `RTM_GET` command on routing socket.

34-35 `rtm` points to the `rt_msghdr` structure and `sa` points to the first socket address structure that follows.

36 `rtm_addrs` is a bitmask of which of the eight possible socket address structure follow the `rt_msghdr` structure. Our `get_rtaddrs` function (which we show next) takes this mask, and the pointer to the first socket address structure (`sa`), and fills in the array `rti_info` with pointers to the corresponding socket address structures. Assuming that all four socket address structures shown in Figure 17.5 are returned by the kernel, the resulting `rti_info` array will be as shown in Figure 17.8.

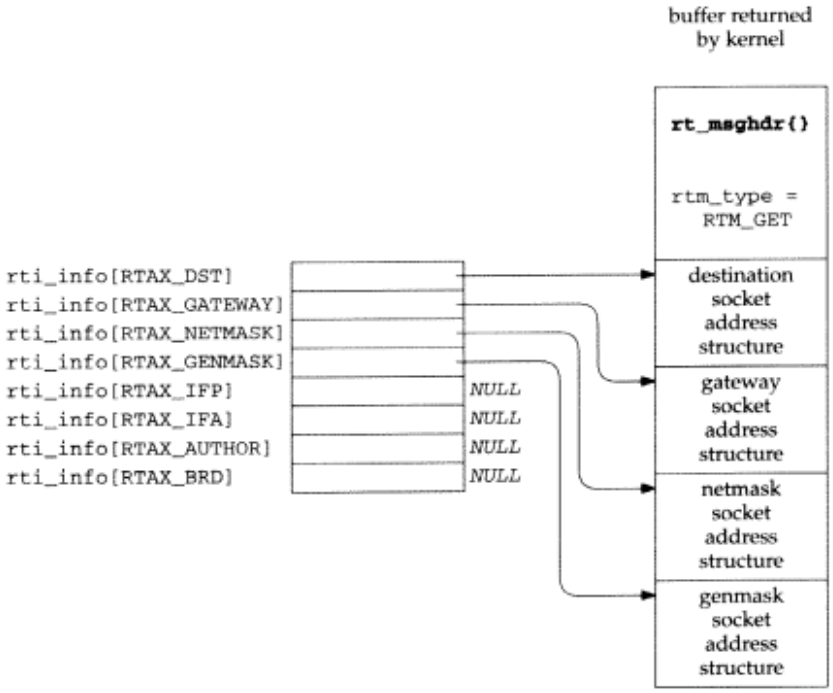

Figure 17.8 `rti_info` structure filled in by our `get_rtaddrs` function.

Our program then goes through the `rti_info` array, doing what it wants with all the nonnull pointers in the array.

37-44 Each of the four possible addresses are printed, if present. We call our `sock_ntop_host` function to print the destination address and the gateway address, but call our `sock_masktop` to print the two masks. We show this new function shortly.

Figure 17.9 shows our `get_rtaddrs` function that we called from Figure 17.7.

Loop through eight possible pointers

17-23 `RTAX_MAX` is 8 in Figure 17.4, the maximum number of socket address structures that are returned in a routing message from the kernel. The loop in this function looks at each of the eight `RTA_xxx` bitmask constants from Figure 17.4 that can be set in the `rtm_addrs`, `ifm_addrs`, or `ifam_addrs` members of the structures in Figure 17.3. If the bit is set, the corresponding element in the `rti_info` array is set to the pointer to the socket address structure; otherwise the array element is set to a null pointer.

libroute/get_rtaddrs.c

```
1 #include    "unproute.h"

2 /*
3  * Round up 'a' to next multiple of 'size'
4  */
5 #define ROUNDUP(a, size) (((a) & ((size)-1)) ? (1 + ((a) | ((size)-1))) : (a))

6 /*
7  * Step to next socket address structure;
8  * if sa_len is 0, assume it is sizeof(u_long).
9  */
10 #define NEXT_SA(ap) ap = (SA *) \
11     ((caddr_t) ap + (ap->sa_len ? ROUNDUP(ap->sa_len, sizeof (u_long)) : \
12                                   sizeof(u_long)))

13 void
14 get_rtaddrs(int addrs, SA *sa, SA **rti_info)
15 {
16     int     i;

17     for (i = 0; i < RTAX_MAX; i++) {
18         if (addrs & (1 << i)) {
19             rti_info[i] = sa;
20             NEXT_SA(sa);
21         } else
22             rti_info[i] = NULL;
23     }
24 }
```

libroute/get_rtaddrs.c

Figure 17.9 Build array of pointers to socket address structures in routing message.

Step to next socket address structure

2-12 The socket address structures are variable length, but this code assumes that each has an `sa_len` field specifying its length. There are two complications that must be handled. First, the two masks, the network mask and the cloning mask, can be returned in a socket address structure with an `sa_len` of 0, but this really occupies the size of an `unsigned long`. (Chapter 19 of TCPv2 discusses the cloning feature of the 4.4BSD routing table.) This value represents a mask of all zero bits, which we printed as 0.0.0.0 for the network mask of the default route in our earlier example. Second, each socket address structure can be padded at the end so that the next one begins on a specific boundary, which in this case is the size of an `unsigned long` (e.g., a 4-byte boundary for a 32-bit architecture). Although `sockaddr_in` structures occupy 16 bytes, which requires no padding, the masks often have padding at the end.

The last function that we must show for our example program is `sock_masktop` in Figure 17.10, which returns the presentation string for one of the two mask values that can be returned. Masks are stored in socket address structures. The `sa_family` member is undefined but they do contain an `sa_len` of 0, 5, 6, 7, or 8 for 32-bit IPv4 masks. When the length is greater than 0, the actual mask starts at the same offset from the beginning as does the IPv4 address in a `sockaddr_in` structure: 4 bytes from the

beginning of the structure (as shown in Figure 18.21, p. 577 of TCPv2), which is the `sa_data[2]` member of the generic socket address structure.

libroute/sock_masktop.c

```
1 #include    "unproute.h"

2 char *
3 sock_masktop(SA *sa, socklen_t salen)
4 {
5     static char str[INET6_ADDRSTRLEN];
6     unsigned char *ptr = &sa->sa_data[2];

7     if (sa->sa_len == 0)
8         return ("0.0.0.0");
9     else if (sa->sa_len == 5)
10        snprintf(str, sizeof(str), "%d.0.0.0", *ptr);
11    else if (sa->sa_len == 6)
12        snprintf(str, sizeof(str), "%d.%d.0.0", *ptr, *(ptr + 1));
13    else if (sa->sa_len == 7)
14        snprintf(str, sizeof(str), "%d.%d.%d.0", *ptr, *(ptr + 1), *(ptr + 2));
15    else if (sa->sa_len == 8)
16        snprintf(str, sizeof(str), "%d.%d.%d.%d",
17                 *ptr, *(ptr + 1), *(ptr + 2), *(ptr + 3));
18    else
19        snprintf(str, sizeof(str), "(unknown mask, len = %d, family = %d)",
20                 sa->sa_len, sa->sa_family);
21    return (str);
22 }
```

libroute/sock_masktop.c

Figure 17.10 Convert a mask value to its presentation format.

7-21 If the length is 0, the implied mask is 0.0.0.0. If the length is 5, only the first byte of the 32-bit mask is stored, with an implied value of 0 for the remaining 3 bytes. When the length is 8, all 4 bytes of the mask are stored.

In this example we want to read the kernel's reply, because the reply contains the information we are looking for. But in general the return value from our `write` to the routing socket tells us if the command succeeded or not. If that is all the information we need, we can call `shutdown` with a second argument of `SHUT_RD` to prevent a reply from being sent. For example, if we are deleting a route, a return of 0 from `write` means success, while an error return of `ESRCH` means the route could not be found (p. 608 of TCPv2). Similarly, an error return of `EEXIST` from `write` when adding a route means the entry already exists. In our example in Figure 17.6, if the routing table entry does not exist (say our host does not have a default route), then `write` returns an error of `ESRCH`.

17.4 `sysctl` Operations

Our main interest in routing sockets is the use of the `sysctl` function to examine both the routing table and the interface list. Whereas the creation of a routing socket (a raw

socket in the AF_ROUTE domain) requires superuser privileges, any process can examine the routing table and the interface list using sysctl.

```
#include <sys/param.h>
#include <sys/sysctl.h>

int sysctl(int *name, u_int namelen, void *oldp, size_t *oldlenp,
           void *newp, size_t newlen);
```

Returns: 0 if OK, –1 on error

This function uses names that look like SNMP (Simple Network Management Protocol) MIB names (Management Information Base). Chapter 25 of TCPv1 talks about SNMP and its MIB in detail. These names are hierarchical.

The *name* argument is an array of integers specifying the name, and *namelen* specifies the number of elements in the array. The first element in the array specifies which subsystem of the kernel the request is directed to. The second element specifies some part of that subsystem, and so on. Figure 17.11 shows the hierarchical arrangement with some of the constants used at the first three levels.

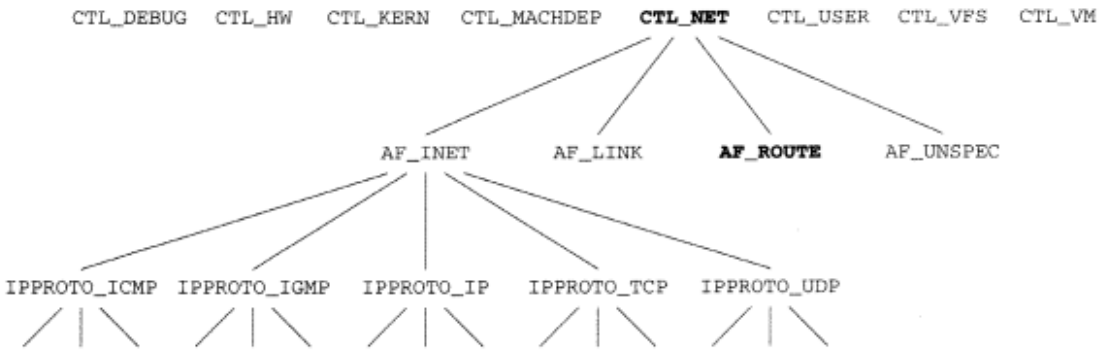

Figure 17.11 Hierarchical arrangement of sysctl names.

To fetch a value, *oldp* points to a buffer into which the kernel stores the value. *oldlenp* is a value–result argument: when the function is called the value pointed to by *oldlenp* specifies the size of this buffer, and on return the value contains the amount of data stored in the buffer by the kernel. If the buffer is not large enough, ENOMEM is returned. As a special case, *oldp* can be a null pointer and *oldlenp* a nonnull pointer, and the kernel determines how much data the call would have returned and returns this size through *oldlenp*.

To set a new value, *newp* points to a buffer of size *newlen*. If a new value is not being specified, *newp* should be a null pointer and *newlen* should be 0.

The sysctl manual page details all the various system information that can be obtained with this function: information on the filesystems, virtual memory, kernel limits, hardware, and so on. Our interest is in the networking subsystem, designated by the first element of the *name* array being set to CTL_NET. (The CTL_*xxx* constants are defined by including the <sys/sysctl.h> header.) The second element can then be

- AF_INET: get or set variables affecting the Internet protocols. The next level specifies the protocol using one of the IPPROTO_*xxx* constants. BSD/OS 3.0 provides about 30 variables at this level, controlling such features as whether the kernel should generate an ICMP redirect, whether TCP should use the RFC 1323 options, whether UDP checksums should be sent, and so on. We show an example of this use of sysctl at the end of this section.
- AF_LINK: get or set link-layer information, such as the number of PPP interfaces.
- AF_ROUTE: return information on either the routing table or the interface list. We describe this information shortly.
- AF_UNSPEC: get or set some socket layer variables, such as the maximum size of a socket send or receive buffer.

When the second element of the *name* array is AF_ROUTE, the third element (a protocol number) is always 0 (since there are not protocols within the AF_ROUTE family, as there are within the AF_INET family, for example), the fourth element is an address family, and the fifth and sixth levels specify what to do. We summarize this in Figure 17.12.

name[]	Return IPv4 routing table	Return IPv4 ARP cache	Return interface list
0	CTL_NET	CTL_NET	CTL_NET
1	AF_ROUTE	AF_ROUTE	AF_ROUTE
2	0	0	0
3	AF_INET	AF_INET	AF_INET
4	NET_RT_DUMP	NET_RT_FLAGS	NET_RT_IFLIST
5	0	RTF_LLINFO	0

Figure 17.12 sysctl information returned for route domain.

Three operations are supported, specified by *name*[4]. (The NET_RT_*xxx* constants are defined by including the <sys/socket.h> header.) The information returned by these three operations is returned through the *oldp* pointer in the call to sysctl. This buffer contains a variable number of RTM_*xxx* messages (Figure 17.2).

1. NET_RT_DUMP returns the routing table for the address family specified by *name*[3]. If this address family is 0, the routing tables for all address families are returned.

   The routing table is returned as a variable number of RTM_GET messages with each message followed by up to four socket address structures: the destination, gateway, network mask, and cloning mask of the routing table entry. We showed one of these messages on the right side of Figure 17.5 and our code in Figure 17.7 parsed one of these messages. All that changes with this sysctl operation is that one or more of these messages are returned by the kernel.

2. NET_RT_FLAGS returns the routing table for the address family specified by *name*[3] but only the routing table entries with an RTF_*xxx* flag value that

contains the flag specified by *name*[5]. All ARP cache entries in the routing table have the RTF_LLINFO flag bit set.

The information is returned in the same format as the previous item.

3. NET_RT_IFLIST returns information on all configured interfaces. If *name*[5] is nonzero, it is an interface index number, and only information on that interface is returned. (We say more about interface indexes in Section 17.6.) All the addresses assigned to each interface are also returned and if *name*[3] is nonzero, only addresses for that address family are returned.

   For each interface one RTM_IFINFO message is returned, followed by one RTM_NEWADDR message for each address assigned to the interface. The RTM_IFINFO message is followed by one datalink socket address structure and each RTM_NEWADDR message is followed by up to three socket address structures: the interface address, the network mask, and the broadcast address. We show a picture of these two messages in Figure 17.13.

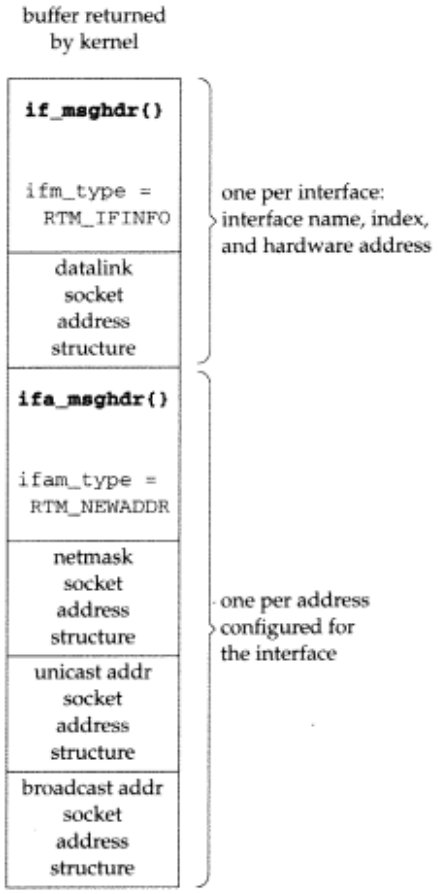

Figure 17.13 Information returned for sysctl, CTL_NET, NET_RT_IFLIST command.

As IPv6 support is added to 4.4BSD-derived kernels, support should be added for the *name*[1] member to be AF_INET6 (to set and fetch IPv6-specific variables), and for *name*[3] in Figure 17.12 to be AF_INET6 (to dump the IPv6 routing table, IPv6 neighbor cache, or to return IPv6 interface addresses).

Example: Determine If UDP Checksums Are Enabled

We now provide a simple example of sysctl with the Internet protocols to check whether UDP checksums are enabled. Some UDP applications (e.g., BIND) check whether UDP checksums are enabled when they start, and if not, try to enable them. Naturally it takes superuser privileges to enable a feature such as this, but all we do now is check whether the feature is enabled or not. Figure 17.14 is our program.

route/checkudpsum.c

```
 1 #include    "unproute.h"
 2 #include    <netinet/udp.h>
 3 #include    <netinet/ip_var.h>
 4 #include    <netinet/udp_var.h> /* for UDPCTL_xxx constants */

 5 int
 6 main(int argc, char **argv)
 7 {
 8     int     mib[4], val;
 9     size_t  len;

10     mib[0] = CTL_NET;
11     mib[1] = AF_INET;
12     mib[2] = IPPROTO_UDP;
13     mib[3] = UDPCTL_CHECKSUM;

14     len = sizeof(val);
15     Sysctl(mib, 4, &val, &len, NULL, 0);
16     printf("udp checksum flag: %d\n", val);

17     exit(0);
18 }
```

route/checkudpsum.c

Figure 17.14 Check whether UDP checksums are enabled.

Include system headers

2-4 We must include the <netinet/udp_var.h> header to obtain the definition of the UDP sysctl constants. The two other headers are required for this header.

Call sysctl

10-16 We allocate an integer array with four elements and store the constants that correspond to the hierarchy shown in Figure 17.11. Since we are only fetching a variable, and not storing into a variable, we specify a null pointer for the *newp* argument to sysctl and a value of 0 for the *newlen* argument. *oldp* points to an integer variable of ours into which the result is stored and *oldlenp* points to a value–result variable for the size of this integer. The flag that we print will be either 0 (disabled) or 1 (enabled).

17.5 get_ifi_info Function

We now return to the example from Section 16.6: returning all the interfaces that are up as a linked list of `ifi_info` structures (Figure 16.5). The `prifinfo` program remains the same (Figure 16.6) but we now show a version of the `get_ifi_info` function that uses `sysctl` instead of the `SIOCGIFCONF ioctl` that was used in Figure 16.7.

We first show the function `net_rt_iflist` in Figure 17.15. This function calls `sysctl` with the `NET_RT_IFLIST` command to return the interface list for a specified address family.

libroute/net_rt_iflist.c

```
 1 #include    "unproute.h"

 2 char *
 3 net_rt_iflist(int family, int flags, size_t *lenp)
 4 {
 5     int     mib[6];
 6     char   *buf;

 7     mib[0] = CTL_NET;
 8     mib[1] = AF_ROUTE;
 9     mib[2] = 0;
10     mib[3] = family;            /* only addresses of this family */
11     mib[4] = NET_RT_IFLIST;
12     mib[5] = flags;             /* interface index, or 0 */
13     if (sysctl(mib, 6, NULL, lenp, NULL, 0) < 0)
14         return (NULL);

15     if ( (buf = malloc(*lenp)) == NULL)
16         return (NULL);
17     if (sysctl(mib, 6, buf, lenp, NULL, 0) < 0) {
18         free(buf);
19         return (NULL);
20     }
21     return (buf);
22 }
```

libroute/net_rt_iflist.c

Figure 17.15 Call `sysctl` to return interface list.

7–14 The array `mib` is initialized as shown in Figure 17.12 to return the interface list and all configured addresses of the specified family. `sysctl` is then called twice. In the first call the third argument is null, which returns in the variable pointed to by `lenp` the buffer size required to hold all the interface information.

15–21 Space is then allocated for the buffer and `sysctl` is called again, this time with a nonnull third argument. This time the variable pointed to by `lenp` will contain upon return the amount of information stored in the buffer, and this variable is allocated by the caller. A pointer to the buffer is also returned to the caller.

> Since the size of the routing table or the number of interfaces can change between the two calls to `sysctl`, the value returned by the first call contains a 10% fudge factor (pp. 639–640 of TCPv2).

Figure 17.16 shows the first half of the `get_ifi_info` function.

route/get_ifi_info.c

```
 3 struct ifi_info *
 4 get_ifi_info(int family, int doaliases)
 5 {
 6     int     flags;
 7     char   *buf, *next, *lim;
 8     size_t  len;
 9     struct if_msghdr *ifm;
10     struct ifa_msghdr *ifam;
11     struct sockaddr *sa, *rti_info[RTAX_MAX];
12     struct sockaddr_dl *sdl;
13     struct ifi_info *ifi, *ifisave, *ifihead, **ifipnext;

14     buf = Net_rt_iflist(family, 0, &len);

15     ifihead = NULL;
16     ifipnext = &ifihead;

17     lim = buf + len;
18     for (next = buf; next < lim; next += ifm->ifm_msglen) {
19         ifm = (struct if_msghdr *) next;
20         if (ifm->ifm_type == RTM_IFINFO) {
21             if (((flags = ifm->ifm_flags) & IFF_UP) == 0)
22                 continue;       /* ignore if interface not up */

23             sa = (struct sockaddr *) (ifm + 1);
24             get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
25             if ( (sa = rti_info[RTAX_IFP]) != NULL) {
26                 ifi = Calloc(1, sizeof(struct ifi_info));
27                 *ifipnext = ifi;    /* prev points to this new one */
28                 ifipnext = &ifi->ifi_next;  /* ptr to next one goes here */

29                 ifi->ifi_flags = flags;
30                 if (sa->sa_family == AF_LINK) {
31                     sdl = (struct sockaddr_dl *) sa;
32                     if (sdl->sdl_nlen > 0)
33                         snprintf(ifi->ifi_name, IFI_NAME, "%*s",
34                                  sdl->sdl_nlen, &sdl->sdl_data[0]);
35                     else
36                         snprintf(ifi->ifi_name, IFI_NAME, "index %d",
37                                  sdl->sdl_index);

38                     if ( (ifi->ifi_hlen = sdl->sdl_alen) > 0)
39                         memcpy(ifi->ifi_haddr, LLADDR(sdl),
40                                min(IFI_HADDR, sdl->sdl_alen));
41                 }
42             }
```

route/get_ifi_info.c

Figure 17.16 `get_ifi_info` function, first half.

6-14 We declare the local variables and then call our net_rt_iflist function.

17-19 The for loop steps through each routing message in the buffer filled in by sysctl. We assume that the message is an if_msghdr structure and look at the ifm_type field. (Recall that the first three members of all three structures are identical so it doesn't matter which of the three structures we use to look at the type member.)

Check if interface is up

20-22 An RTM_IFINFO structure is returned for each interface. If the interface is not up, it is ignored.

Determine which socket address structures are present

23-24 sa points to the first socket address structure following the if_msghdr structure. Our get_rtaddrs function initializes the rti_info array, depending on which socket address structures are present.

Handle interface name

25-42 If the socket address structure with the interface name is present, an ifi_info structure is allocated and the interface flags are stored. The expected family of this socket address structure is AF_LINK, indicating a datalink socket address structure. If the sdl_nlen member is nonzero, then the interface name is copied into the ifi_info structure. Otherwise a string containing the interface index is stored as the name. If the sdl_alen member is nonzero, then the hardware address (e.g., the Ethernet address) is copied into the ifi_info structure and its length is also returned as ifi_hlen.

Figure 17.17 shows the second half of our get_ifi_info function, which returns the IP addresses for the interface.

Return IP addresses

43-63 An RTM_NEWADDR message is returned by sysctl for each address associated with the interface: the primary address and all aliases. If we have already filled in the IP address for this interface, then we are dealing with an alias. In that case, if the caller wants the alias address, we must allocate memory for another ifi_info structure, copy the fields that have been filled in, and then fill in the addresses that have been returned.

Return broadcast and destination addresses

64-73 If the interface supports broadcasting, the broadcast address is returned and if the interface is a point-to-point interface, the destination address is returned.

route/get_ifi_info.c

```
43          } else if (ifm->ifm_type == RTM_NEWADDR) {
44              if (ifi->ifi_addr) {    /* already have an IP addr for i/f */
45                  if (doaliases == 0)
46                      continue;
47
                        /* we have a new IP addr for existing interface */
48                  ifisave = ifi;
49                  ifi = Calloc(1, sizeof(struct ifi_info));
50                  *ifipnext = ifi;    /* prev points to this new one */
51                  ifipnext = &ifi->ifi_next;  /* ptr to next one goes here */
52                  ifi->ifi_flags = ifisave->ifi_flags;
53                  ifi->ifi_hlen = ifisave->ifi_hlen;
54                  memcpy(ifi->ifi_name, ifisave->ifi_name, IFI_NAME);
55                  memcpy(ifi->ifi_haddr, ifisave->ifi_haddr, IFI_HADDR);
56              }
57              ifam = (struct ifa_msghdr *) next;
58              sa = (struct sockaddr *) (ifam + 1);
59              get_rtaddrs(ifam->ifam_addrs, sa, rti_info);
60              if ( (sa = rti_info[RTAX_IFA]) != NULL) {
61                  ifi->ifi_addr = Calloc(1, sa->sa_len);
62                  memcpy(ifi->ifi_addr, sa, sa->sa_len);
63              }
64              if ((flags & IFF_BROADCAST) &&
65                  (sa = rti_info[RTAX_BRD]) != NULL) {
66                  ifi->ifi_brdaddr = Calloc(1, sa->sa_len);
67                  memcpy(ifi->ifi_brdaddr, sa, sa->sa_len);
68              }
69              if ((flags & IFF_POINTOPOINT) &&
70                  (sa = rti_info[RTAX_BRD]) != NULL) {
71                  ifi->ifi_dstaddr = Calloc(1, sa->sa_len);
72                  memcpy(ifi->ifi_dstaddr, sa, sa->sa_len);
73              }
74          } else
75              err_quit("unexpected message type %d", ifm->ifm_type);
76      }
77      /* "ifihead" points to the first structure in the linked list */
78      return (ifihead);           /* ptr to first structure in linked list */
79  }
```

route/get_ifi_info.c

Figure 17.17 `get_ifi_info` function, second half.

17.6 Interface Name and Index Functions

RFC 2133 [Gilligan et al. 1997] defines four functions that deal with interface names and indexes. These four functions are used with IPv6 multicasting, as we describe in Chapter 19. The basic concept is that each interface has a unique name and a unique positive index (0 is never used as an index).

```
#include <net/if.h>

unsigned int if_nametoindex(const char *ifname);

                                        Returns: positive interface index if OK, 0 on error

char *if_indextoname(unsigned int ifindex, char *ifname);

                                    Returns: pointer to interface name if OK, NULL on error

struct if_nameindex *if_nameindex(void);

                                        Returns: nonnull pointer if OK, NULL on error

void if_freenameindex(struct if_nameindex *ptr);
```

`if_nametoindex` returns the index of the interface whose name is *ifname*. `if_indextoname` returns a pointer to the interface name, given its *ifindex*. The *ifname* argument points to a buffer of size `IFNAMSIZ` (defined by including the `<net/if.h>` header; also shown in Figure 16.2) that the caller must allocate to hold the result, and this pointer is also the return value upon success.

`if_nameindex` returns a pointer to an array of `if_nameindex` structures:

```
struct if_nameindex {
  unsigned int   if_index;  /* 1, 2, ... */
  char          *if_name;   /* null terminated name: "le0", ... */
};
```

The final entry in this array contains a structure with an `if_index` of 0 and an `if_name` that is a null pointer. The memory for this array along with the names pointed to by the array members is dynamically obtained and is returned by calling `if_freenameindex`.

We now provide an implementation of these four functions using routing sockets.

`if_nametoindex` Function

Figure 17.18 shows the `if_nametoindex` function.

libroute/if_nametoindex.c

```
 1 #include    "unpifi.h"
 2 #include    "unproute.h"
 3 unsigned int
 4 if_nametoindex(const char *name)
 5 {
 6     unsigned int index;
 7     char   *buf, *next, *lim;
 8     size_t  len;
 9     struct if_msghdr *ifm;
10     struct sockaddr *sa, *rti_info[RTAX_MAX];
11     struct sockaddr_dl *sdl;
12     if ( (buf = net_rt_iflist(0, 0, &len)) == NULL)
13         return (0);
14     lim = buf + len;
15     for (next = buf; next < lim; next += ifm->ifm_msglen) {
16         ifm = (struct if_msghdr *) next;
17         if (ifm->ifm_type == RTM_IFINFO) {
18             sa = (struct sockaddr *) (ifm + 1);
19             get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
20             if ( (sa = rti_info[RTAX_IFP]) != NULL) {
21                 if (sa->sa_family == AF_LINK) {
22                     sdl = (struct sockaddr_dl *) sa;
23                     if (strncmp(&sdl->sdl_data[0], name, sdl->sdl_nlen) == 0) {
24                         index = sdl->sdl_index;     /* save before free() */
25                         free(buf);
26                         return (index);
27                     }
28                 }
29             }
30         }
31     }
32     free(buf);
33     return (0);                 /* no match for name */
34 }
```

libroute/if_nametoindex.c

Figure 17.18 Return an interface index given its name.

Get interface list

12-13 Our `net_rt_iflist` function returns the interface list.

Process only `RTM_IFINFO` messages

17-30 We process the messages in the buffer (Figure 17.13), looking only for the `RTM_IFINFO` messages. When we find one, we call our `get_rtaddrs` function to set up the pointers to the socket address structures and if an interface name structure is present (the `RTAX_IFP` element of the `rti_info` array), the interface name is compared to the argument.

`if_indextoname` Function

The next function, `if_indextoname`, is shown in Figure 17.19.

libroute/if_indextoname.c

```
 1 #include    "unpifi.h"
 2 #include    "unproute.h"

 3 char *
 4 if_indextoname(unsigned int index, char *name)
 5 {
 6     char   *buf, *next, *lim;
 7     size_t  len;
 8     struct if_msghdr *ifm;
 9     struct sockaddr *sa, *rti_info[RTAX_MAX];
10     struct sockaddr_dl *sdl;

11     if ( (buf = net_rt_iflist(0, index, &len)) == NULL)
12         return (NULL);

13     lim = buf + len;
14     for (next = buf; next < lim; next += ifm->ifm_msglen) {
15         ifm = (struct if_msghdr *) next;
16         if (ifm->ifm_type == RTM_IFINFO) {
17             sa = (struct sockaddr *) (ifm + 1);
18             get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
19             if ( (sa = rti_info[RTAX_IFP]) != NULL) {
20                 if (sa->sa_family == AF_LINK) {
21                     sdl = (struct sockaddr_dl *) sa;
22                     if (sdl->sdl_index == index) {
23                         strncpy(name, sdl->sdl_data, sdl->sdl_nlen);
24                         name[sdl->sdl_nlen] = 0;    /* null terminate */
25                         free(buf);
26                         return (name);
27                     }
28                 }
29             }
30         }
31     }
32     free(buf);
33     return (NULL);              /* no match for index */
34 }
```

libroute/if_indextoname.c

Figure 17.19 Return an interface name given its index.

This function is nearly identical to the previous function, but instead of looking for an interface name, we compare the interface index against the caller's argument. Also, the second argument to our `net_rt_iflist` function is the desired index, so the result should contain the information for only the desired interface. When a match is found, the interface name is returned and it is also null terminated.

if_nameindex Function

The next function, if_nameindex, returns an array of if_nameindex structures, containing all the interface names and indexes. It is shown in Figure 17.20.

libroute/if_nameindex.c

```
 1 #include    "unpifi.h"
 2 #include    "unproute.h"

 3 struct if_nameindex *
 4 if_nameindex(void)
 5 {
 6     char   *buf, *next, *lim;
 7     size_t  len;
 8     struct if_msghdr *ifm;
 9     struct sockaddr *sa, *rti_info[RTAX_MAX];
10     struct sockaddr_dl *sdl;
11     struct if_nameindex *result, *ifptr;
12     char   *namptr;

13     if ( (buf = net_rt_iflist(0, 0, &len)) == NULL)
14         return (NULL);

15     if ( (result = malloc(len)) == NULL)     /* overestimate */
16         return (NULL);
17     ifptr = result;
18     namptr = (char *) result + len;     /* names start at end of buffer */

19     lim = buf + len;
20     for (next = buf; next < lim; next += ifm->ifm_msglen) {
21         ifm = (struct if_msghdr *) next;
22         if (ifm->ifm_type == RTM_IFINFO) {
23             sa = (struct sockaddr *) (ifm + 1);
24             get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
25             if ( (sa = rti_info[RTAX_IFP]) != NULL) {
26                 if (sa->sa_family == AF_LINK) {
27                     sdl = (struct sockaddr_dl *) sa;
28                     namptr -= sdl->sdl_nlen + 1;
29                     strncpy(namptr, &sdl->sdl_data[0], sdl->sdl_nlen);
30                     namptr[sdl->sdl_nlen] = 0;  /* null terminate */
31                     ifptr->if_name = namptr;
32                     ifptr->if_index = sdl->sdl_index;
33                     ifptr++;
34                 }
35             }
36         }
37     }
38     ifptr->if_name = NULL;      /* mark end of array of structs */
39     ifptr->if_index = 0;
40     free(buf);
41     return (result);            /* caller can free() this when done */
42 }
```

libroute/if_nameindex.c

Figure 17.20 Return all the interface names and indexes.

Get interface list; allocate room for result

13-18 We call our `net_rt_iflist` function to return the interface list. We also use the returned size as the size of the buffer that we allocate to contain the array of `if_nameindex` structures that we return. This is an overestimate but is simpler than making two passes through the interface list: one to count the number of interfaces and the total sizes of the names, and another to fill in the information. We create the `if_nameindex` array at the beginning of this buffer and store the interface names starting at the end of the buffer.

Process only `RTM_IFINFO` messages

22-36 We process all the messages looking for the `RTM_IFINFO` messages, and the datalink socket address structures that follow. The interface name and index are stored in the array that we are building.

Terminate array

38-39 The final entry in the array has a null `if_name` and an index of 0.

`if_freenameindex` Function

The final function, shown in Figure 17.21, frees the memory that was allocated for the array of `if_nameindex` structures and the names contained therein.

libroute/if_nameindex.c

```
43 void
44 if_freenameindex(struct if_nameindex *ptr)
45 {
46     free(ptr);
47 }
```

libroute/if_nameindex.c

Figure 17.21 Free the memory allocated by `if_nameindex`.

This function is trivial because we stored both the array of structures and the names in the same buffer. If we had called `malloc` for each name, then to free the memory we would have to go through the entire array, `free` the memory for each name, and then free the array.

17.7 Summary

The last of the socket address structures that we encounter in this text are `sockaddr_dl` structures, the variable-length datalink socket address structures. Berkeley-derived kernels associate these with interfaces, returning the interface index, name, and hardware address in one of these structures.

Five types of messages can be written to a routing socket by a process and 12 different messages can be returned by the kernel asynchronously on a routing socket. We showed an example where the process asks the kernel for information on a routing table

entry, and the kernel responds with all the details. These kernel responses contain up to eight socket address structures and we have to parse this message to obtain each piece of information.

The `sysctl` function is a general way to fetch and store operating system parameters. The information that we are interested in with `sysctl` is

- dumping the interface list,
- dumping the routing table, and
- dumping the ARP cache.

The changes required by IPv6 to the sockets API include four functions to map between interface names and their indexes. Each interface is assigned a unique positive index. Berkeley-derived implementations already associate an index with each interface, so we are easily able to implement these functions using `sysctl`.

Exercises

17.1 What would you expect the `sdl_len` field of a datalink socket address structure to contain for a device named `eth10` whose link-layer address is a 64-bit IEEE EUI-64 address?

17.2 In Figure 17.6 disable the `SO_USELOOPBACK` socket option before calling `write`. What happens?