1.2 A Simple Daytime Client

Let's 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.

Figure 1.5 TCP daytime client.

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 }
```

This is the format that we will use for all the source code in the text. Each nonblank line is numbered. The text describing portions of the code notes the starting and ending line numbers in the left margin, as shown shortly. Sometimes a 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 a 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 will have the following output:

```
solaris % a.out 206.168.112.96                    our input
Mon May 26 20:58:40 2003                          the program's output
```

Whenever we display interactive input and output, we will show our typed input in **bold** and the computer output `like this`. *Comments are added on the right side in italics*. We will 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 (OS) as well.

There are many details to 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 will 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 American National Standards Institute (ANSI) C compiler (also referred to as an ISO C compiler).

Create 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 can 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 is required, given the precedence rules of C (the less-than operator has a higher precedence than assignment). As a matter of coding style, the authors always place a space between the two opening parentheses, as a visual indicator that the left-hand side of the comparison is also an assignment. (This style is copied from the Minix source code [Tanenbaum 1987].) 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 API that we are using is called the *sockets API*. In the preceding paragraph, we referred to a function named `socket` that is part of the sockets API. In the preceding paragraph, we also referred 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 will describe them in Section D.3.

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 2.18), 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 of TCPv3 made the mistake of swapping the second and third arguments to `memset` in 10 occurrences in the first printing. 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 acceptable 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 could 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 IPv6 (which we will 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 will 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. Everytime 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 will talk more about generic socket address structures when explaining Figure 3.3.

Read and display server's reply

19–25 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

```
Mon May 26 20 : 58 : 40 2003\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 will be 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 a value less than 0 (an error).

In this example, the end of the record is being denoted by the server closing the connection. This technique is also used by version 1.0 of the Hypertext Transfer Protocol (HTTP). Other techniques are available. For example, the Simple Mail Transfer Protocol (SMTP) marks the end of a record with the two-byte sequence of an ASCII carriage return followed by an ASCII linefeed. Sun Remote Procedure Call (RPC) and the Domain Name System (DNS) 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 an application wants to delineate the ends 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 will go into much more detail on all the points we just described.

[Team LiB]

PREVIOUS NEXT