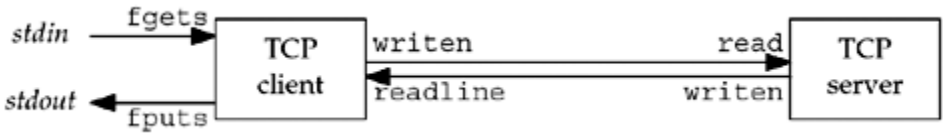
5.1 Introduction

We will 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 really one full-duplex TCP connection. The `fgets` and `fputs` functions are from the standard I/O library and the `writen` and `readline` functions were shown in Section 3.9.

While we will develop our own implementation of an echo server, most TCP/IP implementations provide such a server, using both TCP and UDP (Section 2.12). 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 their 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 must understand exactly what needs to be stored in the protocol-specific address structures. Second, we have not yet covered the library functions that can make this more portable. These functions will be covered in Chapter 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 diagrammed 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 2.10. 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 it 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.

Figure 5.2 TCP echo server (improved in Figure 5.12).

tcpdiserv/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 }
```

5.3 TCP Echo Server: `str_echo` Function

The function `str_echo`, shown in Figure 5.3, performs the server processing for each client: It reads data from the client and echoes it back to the client.

Read a buffer and echo the buffer

8–9 `read` reads data 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 `read` 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.

Figure 5.3 `str_echo` function: echoes data on a socket.

lib/str_echo.c

```
 1 #include    "unp.h"

 2 void
 3 str_echo(int sockfd)
 4 {
 5     ssize_t n;
 6     char    buf[MAXLINE];

 7   again:
 8     while ( (n = read(sockfd, buf, MAXLINE)) > 0)
 9         Writen(sockfd, buf, n);

10     if (n < 0 && errno == EINTR)
11         goto again;
12     else if (n < 0)
13         err_sys("str_echo: read error");
14 }
```

Figure 5.4 TCP echo client.

tcpcliserv/tcpli01.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 }
```

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) 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: It reads a line of text from standard input, writes it to the server, reads back the server's echo of the line, and outputs the echoed line to standard output.

Figure 5.5 `str_cli` function: client processing loop.

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 }
```

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 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 (EOF) 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.