Example: echo client and server

1. Client reads a line of text from standard input and sends to server.

2. Server sends line back to client.

3. Client receives line and displays.

fgets and fputs are from standard C I/O library

writen and readline are from § 3.9

(lets wait to describe them)
TCP echo Server:

Main function:

```c
#include "unp.h"

int main(int argc, char **argv)
{
    int listenfd, connfd;
    pid_t childpid;
    socklen_t clilen;
    struct sockaddr_in cliaddr, servaddr;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    // Further code goes here

    return 0;
}
```
Figure 5.2 TCP echo server.
TCP echo Server:

Main function:

Lines 9-15:
- Create a TCP socket.
- Fill in internet socket address structure with “Wildcard Address “INADDR_ANY” which is 0.
- We will accept any connection destined for us.
- Fill in server port with “SERV_PORT”= 9877 defined in unp.h
  Chosen based on Fig 2.10

Lines 17-18:
- Server waits until a client connection is complete.

Lines 19-24:
- For each client that connects “fork” spawns a child process. Child process handles the new client.
- Child closes listening socket but serves the connected socket.
- In line 24 parent closes connected socket so it can listen for another connection.
- Child calls function “str_echo” passing the connfd descriptor.
TCP echo SERVER Cont.

str_echo Function

```c
#include "unp.h"

void
str_echo(int sockfd)
{
    ssize_t n;
    char line[MAXLINE];

    for (;;) {
        if ( (n = Readline(sockfd, line, MAXLINE)) == 0)
            return;      /* connection closed by other end */

        Write(sockfd, line, n);
    }
```
TCP echo SERVER Cont.

str_echo Function

- This function reads the lines from client and echos back to client.

Lines 7-11:
- The function readline reads the next line from the socket.
- The function writen sends the line back to client.
- When the client closes the connection, receipt of FIN at child causes readline to return a 0. This causes str_echo to return which terminates the child.

Readline Function

[ Fig 3.17 ]

- This calls C library read function once for every byte of data.
```c
#include "unp.h"

ssize_t
readline(int fd, void *vptr, size_t maxlen)
{
    ssize_t n, rc;
    char c, *ptr;

    ptr = vptr;
    for (n = 1; n < maxlen; n++) {
        again:
            if ( (rc = read(fd, &c, 1)) == 1 ) {
                *ptr++ = c;
                if (c == '\n')
                    break; /* newline is stored, like fgets() */
            } else if (rc == 0) {
                if (n == 1)
                    return (0); /* EOF, no data read */
                else
                    break; /* EOF, some data was read */
            } else {
                if (errno == EINTR)
                    goto again;
                return (-1); /* error, errno set by read() */
            }
    }

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

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

Now Figure 3.17
TCP echo Client: Main Function:

```c
#include "unp.h"

int main(int argc, char **argv)
{
    int sockfd;
    struct sockaddr_in servaddr;

    if (argc != 2)
        err_quit("usage: tcpcli <IPaddress> ");

    sockfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(SERV_PORT);
    Inet_ppton(AF_INET, argv[1], &servaddr.sin_addr);

    Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));

    str_cli(stdin, sockfd);   /* do it all */
    exit(0);
}
```

Figure 5.4 TCP echo client.
#include "unp.h"  

void
str_cli(FILE *fp, int sockfd)
{
    char sendline[MAXLINE], recvline[MAXLINE];

    while (Fgets(sendline, MAXLINE, fp) != NULL) {
        Write(sockfd, sendline, strlen(sendline));

        if (Readline(sockfd, recvline, MAXLINE) == 0)
            err_quit("str_cli: server terminated prematurely");

        Fputs(recvline, stdout);
    }
}

Figure 5.5 str_cli function: client processing loop.
TCP echo Client:

Main Function:

Lines 9-13:
- TCP socket is created and Internet socket address structure is filled with IP address and port number.
- Servers IP address is taken from the command line argument argv[1]
- Servers port comes from unp.h “SERV_PORT” = 9877.

Lines 14-15:
- Connect establishes connection with server.
- Function str_cli reads a line of text from standard input, writes to server, reads back servers echo of line and outputs echoed line to standard output.

str_cli function:

Lines 6-7:
- fgets reads a line of text.
- Writen sends the line to the server.
str_cli function: Cont.

Lines 8-10:
  • Readline reads the line echoed back from the server.
  • fputs writes to standard output.

Lines 11-12:
  • Loop terminates when fgets returns a null pointer from end of file or an error

NOTE: fgets is a wrappers function!

Written Function:

[ Fig. 3.16 ]

This writes “n” bytes to a socket
```c
#include "unp.h"

ssize_t writen(int fd, const void *vptr, size_t n);
{
    size_t nleft;
    ssize_t nwritten;
    const char *ptr;

    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
            if (errno == EINTR)
                nwritten = 0; /* and call write() again */
            else
                return (-1); /* error */
        }
        nleft -= nwritten;
        ptr += nwritten;
    }
    return (n);
}
```

Figure 3.15 writen function: write n bytes to a descriptor.
Now Let's Back Up And Talk About “fork”

Section 4.7

Fork Function

- Process makes a copy of itself so one copy can handle one operation and another copy another operation.

Fork is called **ONCE** but returns **TWICE**.

- Returns once in calling process (this is the parent) with process ID of new process.
- Returns second time in child with value Ø so that return value tells you are parent or child.

[Fig. 5.2 again]

- When connection established accept returns and then the server calls `fork`. Child process serves client ( `str_echo(connfd)` )
- Parent closes connected socket since child now handles this client.
Now Back To Running Example

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).

\texttt{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. [Figure 5.4]

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. [Figure 5.2]

3. The server parent, on the other hand, calls `accept` again, and blocks, waiting for the next client connection. [Figure 5.2]

We have three processes, and all three are asleep (blocked): client, server parent, and server child.
```
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 21134 wait Ss p2 0:03.50 -ksh (ksh)
21149 21134 - R+ p2 0:00.05 ps -l
```
5.7 Normal Termination

At this point the connection is established and whatever we type to the client is echoed back.

```
bsdix $ tcpcli01 127.0.0.1
hello, world
hello, world
good bye
good bye
^D
```

we showed this line earlier
we now type this
and the line is echoed

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

```
bsdix $ netstat -a | grep 9877
tcp 0 0 localhost.1052 localhost.9877 TIME_WAIT
```
```
tcp 0 0 *.9877 **.* LISTEN
```
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 (Figure 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.

```
bsdix% 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  (tcperv01)
  21167  p1  R+   0:00.10   ps
```

The STAT of the child is now Z (for zombie).
Figure 2.4 TCP state transition diagram.
Figure 2.5 Packet exchange for TCP connection.
• Now we need to clean up Zombie Processes. They take up space in the Kernel.

• With lots of magic the solution is to add line 17 in figure 5.12 (see book sections 5.9 & 5.10 if you want details)

• Now when terminate only have the one (original) tcperv0l still running all Zombies disappear
Figure 5.11 Final (correct) version of sig_chld function that calls waitpid.

Figure 5.12 adds line 17 (and one other variable definition) to Fig. 5.2 TCP echo server
Data Format:

Passing Text Strings Between Client And Server

- So far our example server does not really look at the data from the client
- Modify server to look for two integers separated by a space, server returns sum of the two integers.
- All we change is the str_echo function.
- The function sscanf converts two arguments from text strings to long integers.

- Function snprintf converts to a text string.

The only thing we need to change is the str_echo function as shown in Fig 5.17
```c
#include "unp.h"

void str_echo(int sockfd)
{
  long arg1, arg2;
  ssize_t n;
  char line[MAXLINE];

  for (; ; ) {
    if ((n = Readline(sockfd, line, MAXLINE)) == 0)
      return; /* connection closed by other end */

    if (sscanf(line, "%ld%ld", &arg1, &arg2) == 2)
      snprintf(line, sizeof(line), "%ld\n", arg1 + arg2);
    else
      snprintf(line, sizeof(line), "input error\n");

    n = strlen(line);
    Write(sockfd, line, n);
  }
}
```

Figure 5.17 str_echo function that adds two numbers.
To Run This Example:

. / tcpserv08 &
. / tcpcli08 127.0.0.1
1  2
3
4  6
10
1
input error
^ d
kill  <> ( for . / tcpserv08)

This works with no problems.
What Happens When We Pass Numbers (Binary) Between Computers?

Using the same client and server main functions but change both the called functions to use binary not ASCII.

```c
struct args {
    long arg1;
    long arg2;
};

struct result {
    long sum;
};
```

*Figure 5.18 sum.h header.*
```c
#include "unp.h"
#include "sum.h"

void
str_cli(FILE *fp, int sockfd)
{
    char sendline[MAXLINE];
    struct args args;
    struct result result;

    while (Fgets(sendline, MAXLINE, fp) != NULL) {
        if (sscanf(sendline, "%ld%ld", &args.arg1, &args.arg2) != 2) {
            printf("invalid input: %s", sendline);
            continue;
        }
        Writen(sockfd, &args, sizeof(args));

        if (Readn(sockfd, &result, sizeof(result)) == 0)
            err_quit("str_cli: server terminated prematurely");

        printf("%ld\n", result.sum);
    }
}
```

**Figure 5.19** *str_cli* function that sends two binary integers to server.
#include "unp.h"
#include "sum.h"

void str_echo(int sockfd)
{
    ssize_t n;
    struct args args;
    struct result result;

    for (;;) {
        if ( (n = Readn(sockfd, &args, sizeof(args))) == 0)
            return;            /* connection closed by other end */

        result.sum = args.arg1 + args.arg2;
        Writen(sockfd, &result, sizeof(result));
    }
}
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 % tcpc1109 206.62.226.33
11 22
33
-11 -44
-55

we type this
and this is the server's reply

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 bsd4) it does not work.

bsd4 % tcpc1109 206.62.226.33
1 2
3
-22 -77
-16777314

we type this
and it works
then we type this
and it does not work
Two Common Solutions:
1. Pass all the numeric data as text strings as done in Figure 5.17 str_echo function that adds two numbers.
2. Explicitly define the binary formats (number of bits, big or little endian)