NETWORK PROGRAMMING
Unit 4 TCP Client-Server Example

Author

Prof.Prasad M.Sawant

Assistant Professor

Department Of Computer Science

Pratibha College Of Commerce And Computer Studies ,Chichwad Pune

Unit 4 TCP Client-Server Example

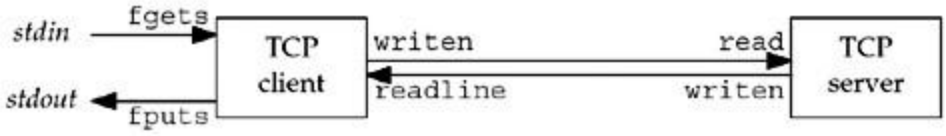
Outline of Unit

1. TCP Echo Server: main Function
2. TCP Echo Server: str_echo Function
3. TCP Echo Client: main Function
4. TCP Echo Client: str_cli Function
5. Normal Startup
6. Normal Termination
7. Connection Abort before accept Returns
8. Termination of Server Process,
9. SIGPIPE Signal
10. Crashing of Server Host,
11. Crashing and Rebooting of Server Host
12. Shutdown of Server Host

Introduction

TCP CLIENT - SERVER COMMUNICATION

Client – Server communication involves reading of text (character or text) from the client and writing the same into the server and server reading text and client writing the same. Following picture depicts the same.



Simple Echo client server

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.

Functions fgets() and fputs() are from standard I/O library. And writen() and readline() are function created by the W Richard Stevans (WRS)

Unit 4 TCP Client-Server Example

TCP Echo Server: `main` Function

The communication between client and server is understood by Echo client and Server. The Following code corresponds to Server side program.

```
 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 }
```

Line 1: It is the header created by the WRS which encapsulates a large number of header that are required for the functions that are referred.

Line 2 – 3: This the definition of the ***main()*** with command line arguments.

Line 5-8 : These are variable declarations of types that are used.

Line 9 : It is the system call to the **socket** function that returns a descriptor of type int.- in this case it is named as listenfd. The arguments are family type, stream type and protocol argument – normally 0)

Line 10: the function ***bzero()*** sets the address space to zero.

Line 11-12: Sets the internet socket address to wild card address and the server port to the number defined in **SERV_PORT** which is 9877 (specified by WRS). It is an intimation that the server is ready to accept a connection destined for any local interface in case the system is multi homed.

Line 14 : ***bind ()*** function binds the address specified by the address structure to the socket.

Line 15: The socket is converted into listening socket by the call to the listen()function

Line 17-18: The server blocks in the call to accept, waiting for a client connection to complete.

Line 19 – 24: For each client, ***fork()*** spawns a child and the child handles the new client. The child closes the listening socket and the parent closes the connected socket The child then calls ***str_echo ()*** to handle the client

TCP Echo Server : str_echo 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.

Line 6: MAXLINE is specified as constant of 4096 characters

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 recept of client's FIN causes the child's readline to return 0. This causes the str_echo function to return which terminates the child.

TCP Echo Client : Main Function

```
 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 }
```

Line 9 – 13: A TCP socket is created and an Internal 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) from the header.

Line 13: The function inet_pton () converts the argument received at the command line from presentation to numeric value and stores the same at the address specified as the third arguments.

Line 14 –1 5: Connection function establishes the connection with the server. The function str_cli () than handles the client processing.

TCP Echo Client: `str_cli` Function

```
 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 }
```

Line 6-7 : ***fgets*** reads a line of text and ***writen*** sends the line to the server.

Line 8 – 10: readline reads the line echoed back from the server and fputs writes it to the standard output.

Unit 4 TCP Client-Server Example

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 `linux`*.

```
linux % tcpserv01 &
[1] 17870
```

We first start the server in the background on the host linux.

linux % tcpserv01 &

[1] 17870

When the server starts, it calls socket, bind, listen, and accept, blocking in the call to accept.

Before starting the client, we run the netstat program to verify the state of the server's listening socket.

linux % netstat -a

```
linux % netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address       Foreign Address      State
tcp        0      0 *:9877              *:*                  LISTEN
```

netstat: 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`,

* It is host that on which u log

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 (the loopback address).

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 while waiting for a line to be sent from the client.
3. The server parent, on the other hand, calls `accept` again, and blocks while waiting for the next client connection.

Normal Termination

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

```
linux % 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

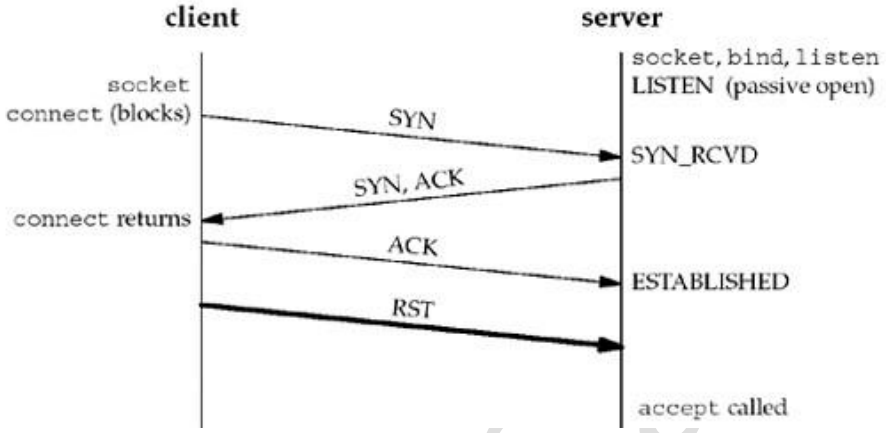
```
linux % netstat -a | grep 9877
tcp        0      0 *:9877                  *:*                     LISTEN
tcp        0      0 localhost:42758         localhost:9877          TIME_WAIT
```

The client's side of the connection (since the local port is 42758) enters the TIME_WAIT state, 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. Doing this also removes the heading line.)

Connection Abort before `accept` Returns

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

Receiving an RST for an ESTABLISHED connection before `accept` is called



Here, 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. Sometime 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 and have it call socket and connect. As soon as connect returns, set the SO_LINGER socket option to generate the RST and terminate.

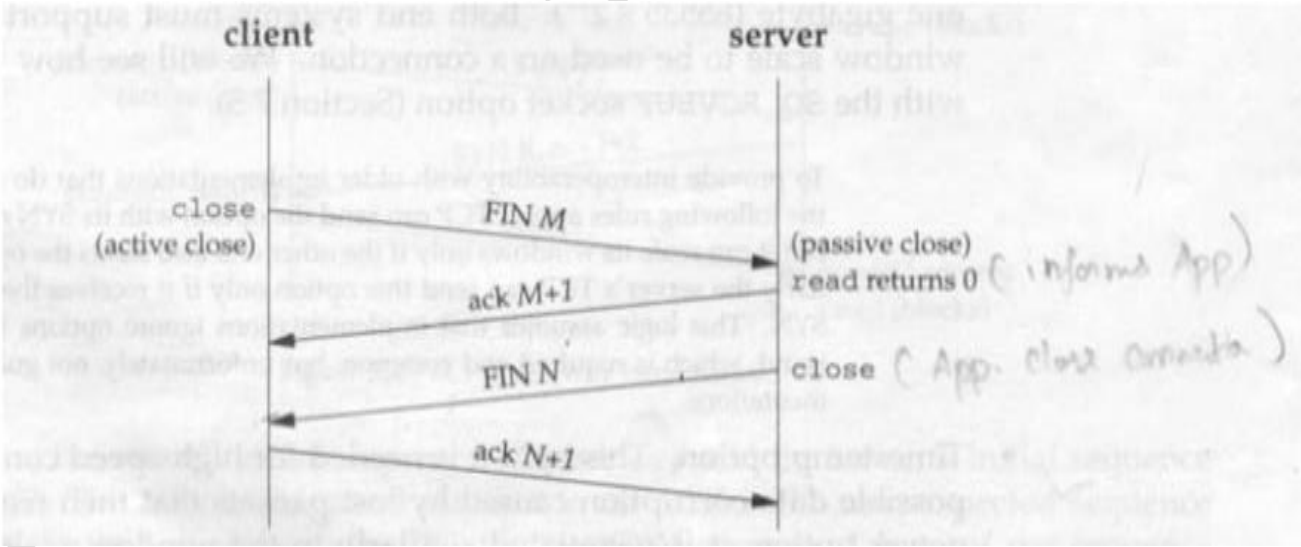
Unfortunately, what happens to the aborted connection is implementation-dependent Most SVR4 implementations, however, return an error to the process as the return from accept, and the error depends on the implementation. These SVR4 implementations return an errno of EPROTO ("protocol error"), but POSIX specifies that the return must be ECONNABORTED ("software caused connection abort") instead. The reason for the POSIX 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.

TCP connection Termination:

It takes **four segment** to terminate a TCP connection as shown below

1. One application calls ***close,*** and we say that this end performs the active close. This end's TCP sends 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 FIN is passed to the application as an end of file(after any data that may already be queued for the application to receive) and the receiver will not receive any further data from the sender.
3. When the received application closes its socket, the TCP sends FIN.
4. The TCP on the system that receives the FIN acknowledges the FIN.



Termination of Server Process

We will 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.

The following steps take place

1. We start the server and client and type one line to the client to verify that all is okay. 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
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 shows the state of the sockets.

`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`.

To see what happens with `SIGPIPE`, we modify our client as shown in Figure

`str_cli` that calls `writen` twice.

Unit 4 TCP Client-Server Example

```
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 }
```

7–9 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 one 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 Linux host, we get:

```
linux % tcpclill 127.0.0.1
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
Broken pipe                 this is printed by the shell
```

Crashing of Server Host

This scenario will test 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 goes 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 the operator .
2. We type a line of input to the client, it is written by writen 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 (*Tcpdump* prints out a description of the contents of packets on a network interface that match the boolean *expression*), we will see the client TCP continually retransmit the data segment, trying to receive ACK from the server. Berkley derived implementations transmit the date segments 12 times, waiting around 9 minutes before giving up. When the client finally gives up, 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 determine that the server was unreachable and responded with ICMP destination unreachable message, then error is either EHOSTUNREACH or ENETUNREACH.

Crashing and Rebooting of Server Host

In this scenario, we will establish a 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 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.

Shutdown of Server Host

When a Unix system is shutdown, the init process normally sends the SIGTERM signal to all processes (this signal can be caught), waits some fixed amount of time (often between 5 and 20 seconds), and then sends SIGKILL signal (which we cannot catch) to any process 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 SIGKILL signal. When the process terminates, all the open descriptors are closed, and we then follow the same sequence of steps discussed under ―termination of server process‖.
We need to select the select or poll function in the client to have the client detect the termination of the server process as soon it occurs.

Unit 4 TCP Client-Server Example

Exercise

Section A

1. Write and explain TCP Echo Server: main Function
2. Write and explain TCP Echo Server: str_echo Function
3. Write and explain TCP Echo Client: main Function
4. Write and explain TCP Echo Client: str_cli Function
5. Write a note on Connection abort before accept return
6. Write a note on SIGPIPE Signal

Section B

1. Explain Crashing of server Host
2. Explain Crashing and Rebooting of Server Host
3. Explain Shutdown of Server Host