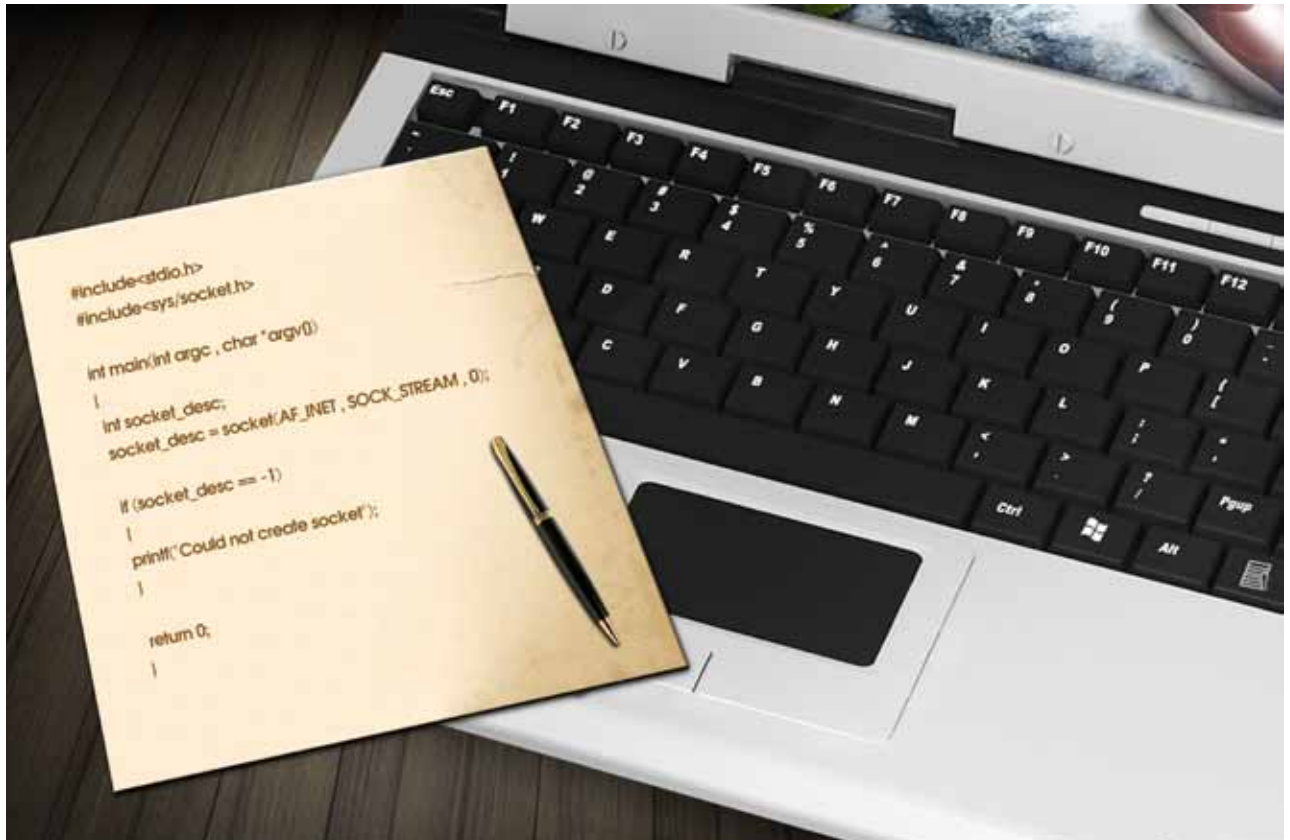


Programming Socket in C (TCP)

This article explores the basics of writing programs in C using BSD 'sockets' in the IP version 4 protocol. The article published in the January 2013 issue of *OSFY* covered UDP socket programming; here, we take a look at socket programs using TCP (Transmission Control Protocol).



Consider a network application—say, a file transfer program. It has a client and server. The client connects to the server to request files for download, or to transfer files to the server (upload). Such applications need to conform to some protocol; the most common is TCP/IP, which is segregated into layers like the application, transport, network and link layers. Sockets help us get the services of the transport layer (TCP, UDP or SCTP sockets). Using RAW sockets, you can directly get the services of the network layer.

With TCP, first a connection is established, then data is transferred, and then the connection is terminated. To establish a connection, three packets are exchanged between client and server, and after this 'handshake', data transfer takes place. In normal scenarios, one side initiates connection termination; and four packets are exchanged for a proper termination.

What does a socket program look like?

A typical socket program would have two source files, one for the server and one for the client. Our files are named

lfy_tcp_client.c and *lfy_tcp_server.c*. In this example, there is another file, *functions.c*, which contains some common utility functions for both client and server. Please download these from Github, https://github.com/mybodhizone/lfy_tcp_socket, after which they need to be compiled to make executables—say, *myclient* and *myserver*. Run these in two terminals, as shown below:

```
[localhost tcp]$ gcc lfy_tcp_server.c functions.c -o myserver
[localhost tcp]$ ./myserver
TCP Server: Waiting for new client connection
Server: Connection from client -127.0.0.1
Server: Received from client:hello
```

```
[localhost tcp]$ gcc lfy_tcp_client.c functions.c -o myclient
[localhost tcp]$ ./myclient 127.0.0.1 11710
Client - Trying to connect to server
Client - Connected Successfully to server
Enter Data For the server or press CTRL-D to exit
```

```
hello
Data Sent To Server
Received from server:
hello
Enter Data For the server or press CTRL-D to exit
```

If you have read the previous article in this series, you will know that my client and server are both running on the same Fedora 15 system in two terminal windows. The server listens on the 'port number' 11710 on the loop-back address 127.0.0.1. When the client requests some service from the server, the server processes it and waits for another client or another request from the same client. As before, the service simply returns the string received from the client.

Now let's take simple algorithms for the TCP client and server. First, the client:

```
Get the socket descriptor (using the socket function).
Update a socket address structure with the server's IP
address and port.
Complete TCP's three-way handshake by calling the connect
function.
Get data for sending through the socket and send it over the
socket.
Receive data from the socket (from the server). Here, the
process waits—that is, gets blocked.
Close the socket descriptor.
```

A simple TCP server algorithm:

```
Get the socket descriptor (using the socket function).
Update a socket address structure with the local address and
port number.
Associate (bind) the above address with the socket.
Call the "listen" function.
Repeat forever {
    Get blocked in "accept" call, till it returns a
connection descriptor.
    serve_client (connection descriptor) {
        Receive data from a client (Here the process waits
that is get blocked)
        Process Received data (Specific to the particular
problem)
        Send data to this client.
        If client closes connection,
        then
            close connection descriptor.
            break (continue waiting in accept)
        else
            continue with this function (server_client)
    }
}
```

Now, please refer to the downloaded source code. The *socket*

function returns a socket descriptor (IPv4, *SOCK_STREAM* (TCP)) in the variable *sd*. An address structure *struct sockaddr_in* (whose declaration is in the file */usr/include/netinet/in.h* in Linux) is needed; you can get it by including the header file *<netinet/in.h>*. In the client code, you need to populate this address structure (the *serveraddress* variable) with the address family, server IP address, and port number. Note that this needs to be populated in big-endian order, so use functions/macros like *htons* and *inet_addr*. Here, *htons(atoi(argv[2]))* transforms the string the user passes as the command-line argument into a big-endian short integer. Similarly, *inet_addr(argv[1])* transforms the IP address command-line argument into a 32-bit IPv4 address in big-endian order.

Next, call the *connect* function in the client:

```
ret = connect (sd, (struct sockaddr*)&serveraddress, sizeof
(serveraddress));
```

If successful, it means that TCP three-way handshake is completed and you get a connected socket, which can be used to send or receive data. Now you can use the 'read' and 'write' functions to read/write data. *An important point to be noted here is that UDP deals with entire datagrams, but TCP is a byte-oriented protocol.* This requires the receiver to understand how much data to read, but this information should come from the sender—lest the receiver reads sent data but still waits to receive more. I have sent the data length in bytes to the receiver before sending the data. The receiver first reads the data length, then iterates to read those many bytes. The code below tries to read 'n' (passed as argument) bytes of data from the socket descriptor 'sd' to storage (address of a variable or name of an array).

```
int myread (int sd, int n, void* buffer) {
    int ret = 0;
    int pointer = 0;
    int totnoofbytes = 0;
    while (1) {
        ret = read (sd, buffer + pointer, n -
totnoofbytes);
        if (0 > ret) {
            perror ("I am getting error in
read!!\n");
            return -1;
        }
        totnoofbytes = totnoofbytes + ret;
        pointer = pointer + ret;
        //totnoofbytes = 0 means, nothing more to
read
        if ((totnoofbytes == n) || (ret == 0)) {
            return totnoofbytes;
        }
    }
}
```

Similarly, while writing data, there is a function called 'mywrite'. These utility functions are in *functions.c*. The *bind* function associates a local address with the socket, and is normally used on the server side. Check out the following code:

```
serveraddress.sin_family = AF_INET;
serveraddress.sin_port = htons (MYPORT); //PORT NO
serveraddress.sin_addr.s_addr = htonl (INADDR_ANY); //IP ADDRESS
ret = bind(sd, (struct sockaddr*)&serveraddress, sizeof (serveraddress));
```

The server's port number and IP address (and address family) that are stored in the socket's address structure are associated with the socket descriptor using the *bind* function. The purpose of *htonl (INADDR_ANY)* is to use a 'wild-card' address—if the server has multiple IP addresses, it can receive data on any of these. The *listen* function is used to change the state of the socket, so it can now *accept* connections. It also specifies the queue size of pending connections, to be retrieved.


Note the server-side accept function; the first argument is the listening *socket* (descriptor-created using the *socket* call). The function blocks till a client connects; whenever it returns successfully, a new connection socket is returned, which can be used to communicate with a particular client. Also note the second and third arguments of *accept*—the address structure that will be populated with the client's IP address and port number, and the length, which will be updated accordingly:

```
length = sizeof (cliaddr);
connfd = accept (sd, (struct sockaddr*)&cliaddr, &length);
```

In my example, the server is an 'iterative' server, which means that it can serve, at most, one client at a time. You can try to make it a concurrent server by using the *fork* or *pthread_create* calls. Also note the use of the *inet_ntop* function:

```
printf ("Data Received from %s",
inet_ntop (AF_INET, &cliaddr.sin_addr,
clientname, sizeof (clientname)));
```

This converts 32-bit IPv4 addresses (in network byte order) into dotted decimal notation (presentation format), and stores it in the *clientname* array.

The server code runs infinitely; to terminate it, press *Ctrl-C*. (While you are writing your own code, it's better to write a signal handler for *Ctrl-C* and do any required cleanup; I've omitted this aspect to keep things simple.) 

References

- [1] Stevens, W. Richard and Rago, Stephen A. 'Advanced Programming in the Unix Environment', 2nd Edition, Pearson Education
- [2] Stevens W. Richard, Fenner Bill, Rudoff A.M., 'Unix Network Programming' (The sockets Networking API – Vol -1), Third Edition, Pearson Education
- [3] Useful Internet links are given below:
 - a. Internet Protocol Specification: <http://www.ietf.org/rfc/rfc791.txt>
 - b. User Datagram Protocol(UDP): <http://www.ietf.org/rfc/rfc768.txt>
 - c. Transmission Control Protocol: <http://tools.ietf.org/html/rfc793>
 - d. For general programming/data structures tutorials: <http://www.mybodhizone.com/>
 - e. For programming related questions and answers: <http://www.ask.mybodhizone.com/>

Acknowledgement

I would like to thank Tanmoy Bandyopadhyay for his help in reviewing this article.

By: Swati Mukerjee

The author has more than 12 years of experience in academics and corporate training. Her areas of interest are digital logic, computer architecture, computer networking, data structures, Linux, programming languages like C and C++, shell scripting and Python scripting. For any queries or feedback, she can be reached at swati.mukerjee@gmail.com.