Birla Institute of Technology & Science, Pilani, K. K. BIRLA Goa campus
Computer Networks
Second Semester 2017-2018
Lab 3
Manual on Socket Programming

Simple TCP Server (In C)

1. Header Files to be used :

1. **sys/socket.h**
2. **netinet/in.h**
3. **arpa/inet.h**

2. What is a socket ?

A **network socket** is an endpoint of an inter-process communication flow across a computer network. Today, most communication between computers is based on the Internet Protocol; therefore most network sockets are **Internet sockets**.

1. In this case – a socket will be used to facilitate the communication between a server and its clients. In other words, a socket can be used by two processes on different or same machines to communicate with one another

3. Transmission Control Protocol (TCP)

TCP provides a *connection oriented service*, since it is based on connections between clients and servers.

TCP provides reliability. When a TCP client send data to the server, it requires an acknowledgement in return. If an acknowledgement is not received, TCP automatically retransmit the data and waits for a longer period of time.

4. Logic behind creating a TCP server using socket programming

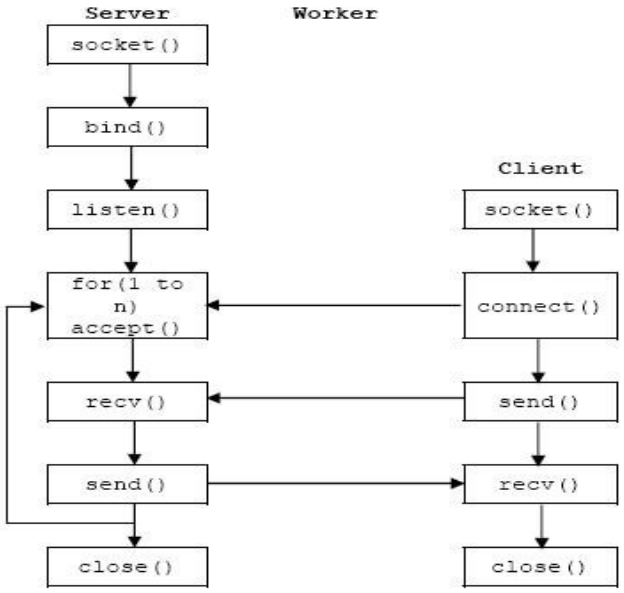

Fig1. TCP Client -server connection process

The steps involved in establishing a TCP socket on the *server* side are as follows:

1. Create a socket with the socket() system call
2. Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine.
3. Listen for connections with the listen() system call
4. Accept a connection with the accept() system call. This call typically blocks until a client connects with the server.
5. Send and receive data

1. socket() : This function is the constructor function for a socket. The *socket*() function shall create an unbound socket in a communications domain, and return a file descriptor that can be used in later function calls that operate on sockets. That file descriptor is used by the other functions as a socket.

Function Declaration : int socket(int *domain*, int *type*, int *protocol*);
example : sockfd = socket(AF_INET, SOCK_STREAM, 0)

Note : AF_INET refers to a family of Internet Addresses. Usually, it is used to define the domain for a socket as to what type of addresses the socket will cater to.

SOCK_STREAM refers to how the data will be transferred using the socket. In this case, it provides sequenced, reliable, bidirectional, connection-mode byte streams, and may provide a transmission mechanism for out-of-band data (TCP).

Note: There is one more scoket type called SOCK_DGRAM. It refers to datagram sockets which uses UDP (User Datagram Protocol), which is unreliable and message oriented.

Socket() will return a value less than 0 if there some error that occurred in getting the file descriptor.

2. Defining the address : **sys**/socket.h provides a type called sockaddr_in used to define an address to bind the socket to. It contains the following members :

Type	Name	Descr.
sa_family_t	sin_family	AF_INET.
in_port_t	sin_port	Port number.
struct in_addr	sin_addr	IP address.

struct sockaddr_in addr;

Note: A sockaddr_in is a structure containing an internet address. This structure is defined in <netinet/in.h>. Here is the definition:

```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
         char    sin_zero[8];
       }
```

bzero(&buffer, sizeof(buffer));

The function bzero() sets all values in a buffer to zero. It takes two arguments, the first is a pointer to the buffer and the second is the size of the buffer. Thus, this line initializes serv_addr to zeros.
addr.sin_family = AF_INET;

The variable addr is a structure of type struct sockaddr_in. This structure has four fields. The first field is short sin_family, which contains a code for the address family. It should always be set to the symbolic constant AF_INET.

addr.sin_port = htons(9999);

The second field of addr is unsigned short sin_port , which contain the port number. However, instead of simply copying the port number to this field, it is necessary to convert this to network byte order using the function htons() which converts a port number in host byte order to a port number in network byte order.

Note: htons stands for host to network short. Any time you send an integer greater than 1 byte in size over the network, you must first convert it to network byte order using htons

addr.sin_addr.s_addr = INADDR_ANY; //Refers to address available on the machine

The third field of sockaddr_in is a structure of type struct in_addr which contains only a single field unsigned long s_addr. This field contains the IP address of the host. For server code, this will always be the IP address of the machine on which the server is running, and there is a symbolic constant INADDR_ANY which gets this address.

3. bind() : This function is used to bind the socket to the address as defined above. Once this is done – the server using this socket will now be bound to the port and all communication by the server will be done through this port.

Example : bind(sockfd, (struct sockaddr*)&addr, sizeof(addr))

4. listen() : The *listen*() function shall mark a connection-mode socket, specified by the *socket* argument, as accepting connections. Once this is called – the server acts as a listener on the defined port.

Example : listen(sockfd, 20).

Note : The second argument is the backlog argument that tries to impose a limit on the number of queued incoming connections.

5. accept() : The accept() function waits for an incoming connection from a client. This function accepts the connection to the server socket, saves the client address into a predefined buffer. It then returns the value of the socket/file descriptor being used by the client.

Example :

int addrlen = sizeof(cliaddr);

clientfd = accept(sockfd, (struct sockaddr*)&cliaddr, &addrlen);

6. **recv() :** The recv() function gets the actual content of the client request or message. It returns the size of the message received.

Example : recv(clientfd, buffer, MAXBUF, 0) // the message from clientfd will be stored in the buffer with the maximum limit as MAXBUF.

7. send() : The server can respond with the send() function. It is usually called after the client has sent a message/request.

Example : send(clientfd, buffer, recv(clientfd, buffer, MAXBUF, 0), 0); // send a message contained in buffer to clientfd. It should be of the same size as the received message from client. So technically it is an echo server as recv puts the contents of the request in buffer.

5. Logic behind a TCP Client

The steps involved in establishing a TCP socket on the *client* side are as follows:

1. Create a socket with the `socket()` system call
2. Connect the socket to the address of the server using the `connect()` system call
3. Send and receive data.

A Client program is pretty similar to the server described above. Looking at the diagram above, you will find that it just has to create a file descriptor (socket) like any other socket program.
Then, instead of binding it to an address and listening on the same port – the program simply has to connect to the address and the port given here.

Then the client can use send() and recv() as described above.

Example of connect :

```
/*---Initialize server address/port struct---*/
   #define SERVER_ADDR     "127.0.0.1"
   struct sockaddr_in dest;
   bzero(&dest, sizeof(dest));
   dest.sin_family = AF_INET;
   dest.sin_port = htons(9999);
   if ( inet_aton(SERVER_ADDR, &dest.sin_addr.s_addr) == 0 ) //ensures the SERVER_ADDR given is correct and converts the SERVER_ADDR into a network byte to put in the struct sockaddr_in dest
   {
      perror(SERVER_ADDR);
      exit(errno);
   }

   /*---Connect to server---*/
   if ( connect(sockfd, (struct sockaddr*)&dest, sizeof(dest)) != 0 ) //connects the client socket sockfd to the destination address
   {
      perror("Connect ");
      exit(errno);
   }
```

6. User Datagram Protocol (UDP)

UDP is a simple transport-layer protocol which is also known as connectionless protocol as there is no guarantee that a UDP will reach the destination. The problem of UDP is its lack of reliability: if a datagram reaches its final destination but the checksum detects an error, or if the datagram is dropped in the network, it is not automatically retransmitted.

7. UDP client-server connection:

There are some fundamental differences between TCP and UDP sockets. UDP is a connection-less, unreliable, datagram protocol (TCP is instead connection-oriented, reliable and stream based). There are some instances when it makes to use UDP instead of TCP. Some popular applications built around UDP are DNS, NFS, SNMP and for example, some Skype services and streaming media.

Figure 2 shows the the interaction between a UDP client and server. First of all, the client does not establish a connection with the server. Instead, the client just sends a datagram to the server using the sendto function which requires the address of the destination as a parameter. Similarly, the server does not accept a connection from a client. Instead, the server just calls the recvfrom function, which waits until data arrives from some client. recvfrom returns the IP address of the client, along with the datagram, so the server can send a response to the client.

As shown in the Figure, the steps of establishing a UDP socket communication on the client side are as follows:

- Create a socket using the socket() function;
- Send and receive data by means of the recvfrom() and sendto() functions.

The steps of establishing a UDP socket communication on the server side are as follows:

- Create a socket with the socket() function;
- Bind the socket to an address using the bind() function;
- Send and receive data by means of recvfrom() and sendto().

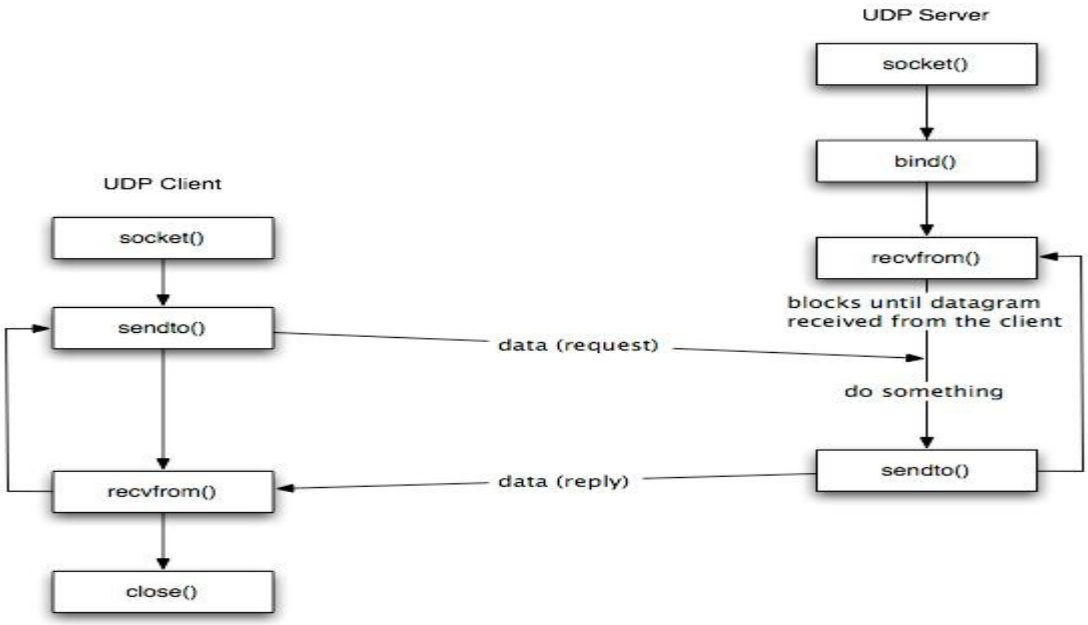

Fig2. UDP client -server connection process

In UDP client-server connection,we will describe the two new functions recvfrom() and sendto().

1. recvfrom() : This function is similar to the read() function, but three additional arguments are required.

Example: ssize_t recvfrom(int sockfd, void* buff, size_t nbytes, int flags, struct sockaddr* from, socklen_t *addrlen);

The first three arguments sockfd, buff, and nbytes, are identical to the first three arguments of read and write. sockfd is the socket descriptor, buff is the pointer to read into, and nbytes is number of bytes to read. In our example, we will set all the values of the flags argument to 0. The recvfrom function fills in the socket address structure pointed to by from with the protocol address of who sent the datagram. The number of bytes stored in the socket address structure is returned in the integer pointed by addrlen.

The function returns the number of bytes read if it succeeds, -1 on error.

2. sendto() : This function is similar to the send() function, but three additional arguments are required.

Example: ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags, const struct sockaddr *to, socklen_t addrlen);

The first three arguments sockfd, buff, and nbytes, are identical to the first three arguments of recv. sockfd is the socket descriptor, buff is the pointer to write from, and nbytes is number of bytes to write. In our example, we will set all the values of the flags argument to 0. The to argument is a socket address structure containing the protocol address (e.g., IP address and port number) of where the data is sent. addlen specified the size of this socket.

The function returns the number of bytes written if it succeeds, -1 on error.

NOTE:
1. Upon using executing the server code repeatedly, you may encounter the following error:
   socket--bind: Address already in use
This will occur when you don't close the socket properly and just terminate the program.
You can wait for some time to execute your code again or just change the port.

References:

1. Socket tutorials
http://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html
http://www.cs.dartmouth.edu/~campbell/cs60/socketprogramming.html