
Assignment No: 6 Socket Programming in C/C++ on Linux.

TCP Client , TCP Server ,UDP Client , UDP Server

---

a. TCP sockets

Server accepts operation and floating point numbers from the clients; performs arithmetic operations and sends the result back to client.

Objective of the Assignment: To understand Socket programming in C

Prerequisite: Students must have knowledge of socket programming

---

Theory :

What is Socket ?

The steps involved in establishing a 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

The steps involved in establishing a 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. There are a number of ways to do this, but the simplest is to use the `read()` and `write()` system calls.

Socket Types

When a socket is created, the program has to specify the *address domain* and the *socket type*.
Two processes can communicate with each other only if their sockets are of the same type and in the same domain.

TWO Types of Domain :

There are two widely used address domains, the ***unix domain***, in which two processes which share a common file system communicate, and the ***Internet domain***, in which two processes running on any two hosts on the Internet communicate. Each of these has its own address format.

AF_UNIX

Hide Copy Code

```
struct sockaddr_un
  {
    sa_family_t sun_family ;
    char sun_path[];
  };
```

Use struct sockaddr_un if you are using AF_UNIX on your domain. It is required to include <sys/un.h>

AF_INET

Hide Copy Code

```
struct sockaddr_in
  {
    short int  sin_family ;
    int        sin_port;
    struct in_addr sin_addr;
  };
```

Use struct sockaddr_in if you are using AF_INT on your domain.

Address of socket

The address of a socket in the Unix domain is a character string which is basically an entry in the file system.

The **address of a socket in the Internet domain consists of the Internet address of the host machine (every computer on the Internet has a unique 32 bit address, often referred to as its IP** address). In addition, each socket needs a port number on that host. Port numbers are 16 bit unsigned integers. The lower numbers are reserved in Unix for standard services. For example, the port number for the FTP server is 21. It is important that standard services be at the same port on all computers so that clients will know their addresses. However, port numbers above 2000 are generally available.

Two types of Socket

There are two widely used socket types, *stream sockets*, and *datagram sockets*. Stream sockets treat communications as a continuous stream of characters, while datagram sockets have to read entire messages at once. Each uses its own communciations protocol. Stream sockets use TCP (Transmission Control Protocol), which is a reliable, stream oriented protocol, and datagram sockets use UDP (Unix Datagram Protocol), which is unreliable and message oriented.

Domain	AF_UNIX - connect inside same machine AF_INET – connect with different machine
Type	SOCK_STREAM – TCP connection SOCK_DGRAM – UDP connection
Protocol	Define here when there is any additional protocol. Otherwise, define it as 0

Server Header code:

```
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/types.h>  /* for Socket data types */
#include <sys/socket.h> /* for socket(), connect(), send(), and recv() */
#include <netinet/in.h> /* for IP Socket data types */
#include <arpa/inet.h>  /* for sockaddr_in and inet_addr() */
#include <stdlib.h>     /* for atoi() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */
```

The server code uses a number of programming constructs, and so we will go through it line by line.

```
#include <netinet/in.h>
```

The header file `netinet/in.h` contains constants and structures needed for internet domain addresses.

Declaration of main( ):-

```
int main(int argc, char *argv[])
{
    int sockfd, newsockfd, portno, clilen, n;
```

- sockfd and newsockfd are file descriptors, i.e. array subscripts into the file descriptor table . These two variables store the values returned by the socket system call and the accept system call.
- portno stores the port number on which the server accepts connections.
- clilen stores the size of the address of the client. This is needed for the accept system call.
- In is the return value for the read() and write() calls; i.e. it contains the number of characters read or written.

```
    char buffer[256];
```

The server reads characters from the socket connection into this buffer.

Declaring Socket ( ) :-

At the beginning, a socket function needs to be declared to get the socket descriptor.

int socket(int domain, int type, int protocol)

Domain	AF_UNIX - connect inside same machine AF_INET – connect with different machine
Type	SOCK_STREAM – TCP connection SOCK_DGRAM – UDP connection
Protocol	Define here when there is any additional protocol. Otherwise, define it as 0

Structure declaration in socket programming :

struct sockaddr_in serverstruct;

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

serverstruct.sin_family=AF_UNIX;

serverstruct.sin_addr.s_addr=inet_addr("127.0.0.1");

serverstruct.sin_port=1025;

Assign address to socket: bind()

bind(sockfd, (struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)

error("ERROR on binding");

The `bind()` system call binds a socket to an address, in this case the address of the current host and port number on which the server will run.

It takes three arguments, the **socket file descriptor, the address to which is bound, and the size of the address to which it is bound. The second argument is a pointer** to a structure of type `sockaddr`, but what is passed in is a structure of type `sockaddr_in`, and so this must be cast to the correct type. This can fail for a number of reasons, the most obvious being that this socket is already in use on this machine

Using Listen( ):-

listen(sockfd,5);

The `listen` system call allows the process to listen on the socket for connections. The first argument is the socket file descriptor, and the second is the size of the backlog queue, i.e., the number of connections that can be waiting while the process is handling a particular connection. This should be set to 5, the maximum size permitted by most systems. If the first argument is a valid socket, this call cannot fail, and so the code doesn't check for errors.

Incomming command to accept ( ) :-

int client_len=sizeof(serverstruct);

session_id=accept(server_id,(struct sockaddr*)&serverstruct,&client_len);

The accept() system call causes the process to block until a client connects to the server. Thus, it wakes up the process when a connection from a client has been successfully established. It returns a new file descriptor, and all communication on this connection should be done using the new file descriptor. The **second argument is a reference pointer** to the address of the client on the other end of the connection, and **the third argument is the size of this structure**.

Ex : - The server gets a socket for an incoming client connection by calling accept()

int s= accept(sockid, &clientAddr, &addrLen);

- **s**: integer, the new socket (used for data-transfer)
- **sockid**: integer, the orig. socket (being listened on)
- **clientAdd**r: struct sockaddr, address of the active participant
  - filled in upon return
- **addrLen**: sizeof(clientAddr): value/result parameter
  - must be set appropriately before call
  - adjusted upon return
- **accept()**
  - is blocking: waits for connection before returning
  - dequeues the next connection on the queue for socket (sockid)

Exchanging data with stream socket:read ( ) and write ( ) :-

read(session_id,&n1,sizeof(n1));

the `read()` will block until there is something for it to read in the socket, i.e. after the client has executed a `write()`. It will read either the total number of characters in the socket or 255, whichever is less, and return the number of characters read.

TCP SERVER CODE

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>


int main()
{
 int n1,n2,res=0;
 char operator;
 int session_id;
 int server_id=socket(AF_UNIX,SOCK_STREAM,0);
 if(server_id<0)
 {
  printf("Error in getting socket\n");
  return 0;
 }


 struct sockaddr_in serverstruct,clientstruct;
 serverstruct.sin_family=AF_UNIX;
 serverstruct.sin_addr.s_addr=inet_addr("127.0.0.2");
 serverstruct.sin_port=1027;

 int                                          i=bind(server_id,(struct
sockaddr*)&serverstruct,sizeof(serverstruct));
 if(i<0)
 {
  printf("Error in bind\n");
  return 0;
 }


 i=listen(server_id,10);
 if(i<0)
 {
  printf("Error in listening\n");
  return 0;
 }
 int client_len=sizeof(serverstruct);
 while(1)
 {
```

```
  printf("Waiting for the client\n");
  session_id=accept(server_id,(struct
sockaddr*)&serverstruct,&client_len);
  read(session_id,&n1,sizeof(n1));
  read(session_id,&n2,sizeof(n2));
  read(session_id,&operator,sizeof(operator));


  switch(operator)
  {
   case '+':
    res=n1+n2;
    break;
   case '-':
    res=n1-n2;
    break;
   case '*':
    res=n1*n2;
    break;
   case '/':
    res=n1/n2;
    break;
   default:
    printf("invalid operation\n");
  }

  printf("From CLIENT:n1=%d\n",n1);
  printf("From CLIENT:n2=%d\n",n2);
  printf("From CLIENT:operator=%c\n",operator);


  int b=write(session_id,&res,sizeof(res));

  close(session_id);
 }
}
```

a. UDP sockets

Server accepts operation and floating point numbers from the clients; performs arithmetic operations and sends the result back to client.

Objective of the Assignment: To understand Socket System Calls in C

Prerequisite: Students must have knowledge of socket programming

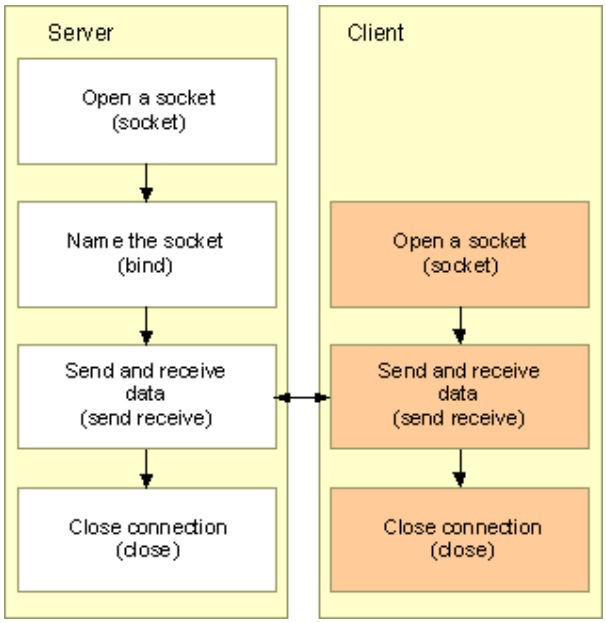
---

Theory :

UDP Protocol Connectionless communication behaves differently than connection-oriented communication, so the method for sending and receiving data is substantially different. First we'll discuss the receiver (or server, if you prefer) because the connectionless receiver requires little change when compared with the connection-oriented servers. After that we'll look at the sender.

In IP, connectionless communication is accomplished through UDP/IP. UDP doesn't guarantee reliable data transmission and is capable of sending data to multiple destinations and receiving it from multiple sources. For example, if a client sends data to a server, the

data is transmitted immediately regardless of whether the server is ready to receive it. If the server receives data from the client, it doesn't acknowledge the receipt. Data is transmitted using datagrams, which are discrete message packets. The following Figure shows a simplified UDP communication flow between server and 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().

In this section, we will describe the two new functions `recvfrom()` and `sendto()`.

The recvfrom() Function

This function is similar to the `read()` function, but three additional arguments are required. The `recvfrom()` function is defined as follows:

```
#include <sys/socket.h>

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 examples 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.

The sendto() Function

This function is similar to the send() function, but three additional arguments are required. The sendto() function is defined as follows:

```
#include <sys/socket.h>
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 examples 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.

Tips on Socket Structures

Socket address structures are an integral part of every network program. We allocate them, fill them in, and pass pointers to them to various socket functions. Sometimes we pass a pointer to one of these structures to a socket function and it fills in the contents.

We always pass these structures by reference (i.e., we pass a pointer to the structure, not the structure itself), and we always pass the size of the structure as another argument.

When a socket function fills in a structure, the length is also passed by reference, so that its value can be updated by the function. We call these value-result arguments.

Always, set the structure variables to NULL (i.e., '\0') by using memset() for bzero() functions, otherwise it may get unexpected junk values in your structure.

The bzero ( ) :-

bzero((char *) &serv_addr, sizeof(serv_addr));

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.

Conclusion : Hence we implemented UDP socket programming successfully.