UNIT 2 SOCKET INTERFACE

Structure **Page Nos.**

2.0 Introduction 23

2.1 Objectives 23

2.2 Elementary Socket System Calls 23

2.2.1 Socket System Call 24

2.2.2 Bind System Call 27

2.2.3 Connect System Call 28

2.2.4 Listen System Call 30

2.2.5 Accept System Call 31

2.3 Elementary Data Transfer Calls 34

2.4 Closing a Socket 38

2.5 TCP and UDP Architectures 39

2.6 Networking Example 41

2.7 Summary 43

2.8 Answers to Check Your Progress 44

2.9 Further Readings 45

2.0 INTRODUCTION

Socket is considered as an end-point, which is used by a process for bi-directional communication in which another socket is associated with another process. As we discussed earlier a socket is like, a file descriptor available in Unix for file communication (I/O) like, open (), create (), close (), read () and write (). Similarly for network communication (I/O) socket behaves like, socket descriptor by which we can read () write () data. File is like a sequence of characters which we can read using repeated read operation in connection oriented mode and in connectionless mode we have to get whole message in a single read operation. But because of the complexity and the requirements of network communication we need many more system calls that we will discuss in this unit.

2.1 OBJECTIVES

Our objective is to introduce you, with the elementary socket calls. After successful completion of this unit, you should be able to:

- have a reasonable understanding of the Elementary System Calls;
- understand the use of basic data transfer calls;
- describe the TCP and UDP Client/Server Architecture; and
- demonstrate an understanding the simple network programs.

2.2 ELEMENTARY SOCKET SYSTEM CALLS

As we discussed in unit 1 when a socket is created, it is allocated a slot in the file descriptor table exactly the same way a file is. Once this socket is associated with a file, we can do I/O such as read and write. Recall that generally sockets are for network communication and network I/O deals with a completely different set of problems than file I/O, but wherever possible, actions corresponding to file I/O are

accomplished. But what is the difference between file I/O and networking I/O? let's find out:

- The client-server relationship is not symmetrical. The client and server processes must be assigned the responsibilities.
- Connectionless protocols do not have an open command because any operation could come from any process.
- Peer process names and file names are important for authority uses.
- More parameters must be specified (protocol, local address, local post server address, server process).
- All computers do not share the same data format (e.g., little endian versus big endian). For details see unit 3.

We now describe the elementary system calls which require to develop Network programs.

2.2.1 Socket System Call

In the early 1980s, with ARPA project, University of California at Berkeley had the responsibility to transport the TCP/IP protocol suit to Unix operating system. It was decided to use Unix system calls with addition to new system calls, if required, as the result new socket interface developed, which become popular as Berkeley UNX or BSD (Berkeley Software Distribution) version 4.1. We are going to discuss BSD Unix system calls with you in this section.

The socket system calls is used by any process to create a socket for doing any network I/O. The structure of socket, we have already discussed as general but here we will discuss in detail with programming concept. The structure of this is given below.

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

int socket (into family, int type, int protocol);

Condition	Returns
Successfully	**Small integer value called the socket descriptor**
Unsuccessful	**Error (-1)**

Socket function creates a socket, it sets values for only family, type and protocol of socket structure the other fields are set by other functions or by operating system. If socket system call creates socket successfully then it returns small integer value called the socket descriptor sockfd (which is similar to a file descriptor) which uniquely identifies the socket. If socket is not created it means there is an error and it returns –1.This socket descriptor sockfd, is used by other functions to refer the socket. (see the socket description tables and its contents in the *Figure1*).

Whenever you will use socket system call you should include both of these header files sys/types.

#include<sys/types.h>: This header file contains definitions of a number of data types used in system calls.

#include<sys/socket.h>: The header file socket.h includes a number of definitions of structures needed for sockets.

Socket header files contain data definitions, structures, constants, macros, and options used by socket subroutines. An application program must include appropriate header files to make use of structures or other information a particular socket subroutine requires. Some other header files are also important for Unix socket programmers: These are given in *Table 1*. You should remember that header files required for some particular calls or API may differ from system to system. So you should always check it with the online manual available in Unix.

Table1: Socket Header Files

Socket Header File	Description
inet.h	takes the place of both in.h and inet.h header files. It defines values common to the Internet.
ip.h	defines values used by the Inter-network Protocol (IP) and details the format of an IP header and options associated with IP.
netdb.h	defines the structures used by the "get" services.
errno.h	defines the errors that can be returned by the socket library when requests are made to it.
sockcfg.h	describes the socket configuration structure
socket.h	defines most of the variables and structures required to interface properly to the socket library.
sockvar.h	is needed when compiling the socket configuration file
tcp.h	describes those options that may be set for a socket of type SOCK_STREAM.
uio.h	describes the structures necessary to use vectored buffering of the socket library.

Socket Descriptor

As you know, in Unix, if some application needs to perform input/output function, it calls the "open" function to create the file descriptor which has further access to the file. Unix uses descriptor as an index into process descriptor table, which follows the pointers to the data structure that holds all details about file. Similarly, when some application calls "socket" the operating system allocates a new data structure as shown in Figure 1(a) and 1(b) to hold the details required for communication and enter the pointer into the description table.

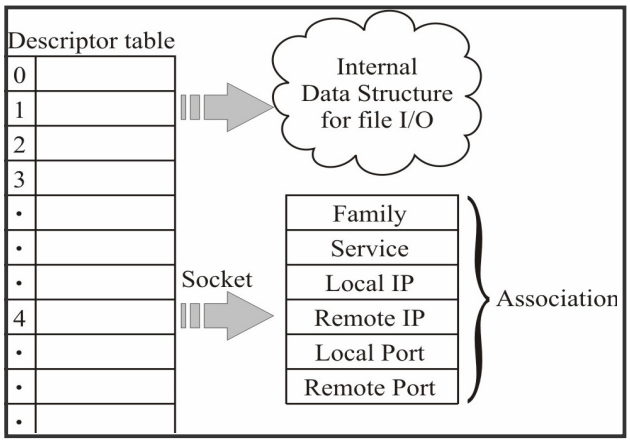


Figure 1 (a)

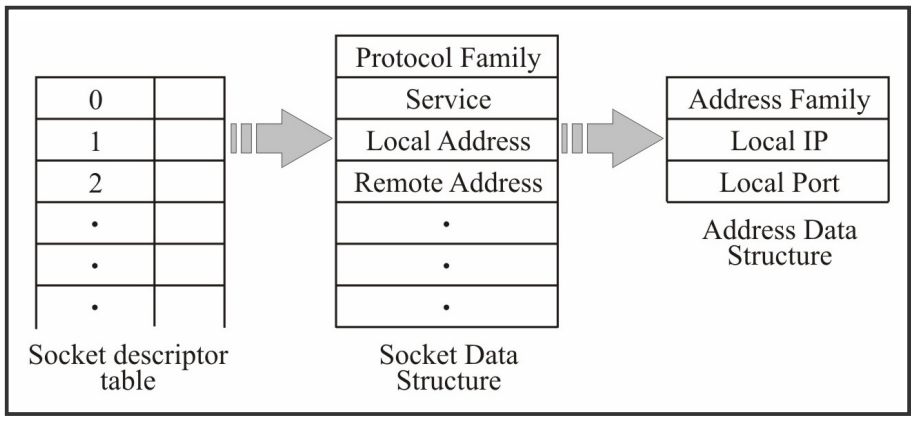


Figure 1 (b)

Figure 1: Socket Descriptor

Socket descriptor defines an address family to provide the freedom to different protocol families for choosing their own representation for their address. A single protocol family can use more than one address families to define address representation. TCP/IP protocol family uses a single address representation; this address family is symbolically represented by AF_INET (Address family internet). *Don't get confused with the PF-NET, both denote the address family in TCP/IP.*

Socket family: As you know socket family defines the protocol group needed for communication. At the time of programming you should choose one of the given options but because we are concerned about TCP/IP you need to remember AF_INET.

```
AF_UNIX /*Unix internal protocols */
AF_INET /* internet protocols */
AF_NS /* Xerox NS protocols */
AF_IMPLINK /* IMP link layer */
```

Socket types: The Type parameter in socket system call specifies the semantics of communication. Sockets are typed according to the communication properties visible to a user. Mostly Processes can communicate only between sockets of the similar type. If underlying communication protocols support, communication between different types of sockets can happen. As we have discussed earlier also you have following options available with "socket",

/*Standard socket types available*/

```
#define  SOCK_STREAM          /* stream socket */
#define  SOCK_DGRAM           /*datagram socket*/
#define  SOCK_RAW             /*raw socket*/
#define  SOCK_SEQPACKET       /*sequence packet socket */
#define  SOCK_RDM             /*reliably-delivered message*/
```

Protocol: The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family.

Protocol is dependent on the services we are using in our communication. For example, if we are using Stream Socket type we must use a protocol that provide a connection-oriented service, like, TCP. But if we are using Datagram Socket type

which provide connectionless services then available protocol is UDP. Please see the Table 2 given below):

Table 2: Protocol corresponding to socket type and socket family

	AF_UNIX	AF_INET	AF_NS
SOCK_STREAM	YES	TCP	SPP
SOCK_DGRAM	YES	UDP	IDP
SOCK_RAW	NOT DEFINED	IP	YES
SOCK_RDM	NOT DEFINED	NOT DEFINED	SPP

All combinations of socket family and type are not valid. The *Table 2* (reference from Unix Network Programming by Richard Stevens) server shows the valid combinations along with the actual protocol that is selected by pair.

To create a TCP socket:

```
int sockfd;
sockfd = socket (AF_INET,
SOCK_STREAM, 0);
```

To create a UDP socket:

```
int Sockfd;
sockfd = socket(AF_INET,
SOCK_DGRAM, 0);
```

2.2.2 Bind System Call

After creating "socket" and before Sending/Receiving data using socket, it must be associated with a local port and a network interface address. That's why we can say mapping of a socket to a port number and IP address is called a "binding". Why binding is needed, as you know server use socket so it can attend client request on specific port. Socket must be associated with particular port on one network interface. But think when we can have multiple network addresses on one host and each network, we have some port addresses. In this case with the help of bind we can define, for which network address (IP address + port address) with which port is associated, otherwise imagine how much confusion can happen.

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr_in *localaddr, int addrlen);
```

Here in the above prototype of bind, sockfd is File descriptor of local socket, as created by the socket function. Localaddr is a pointer to protocol address structure of local socket. The special address ANY_ADDR can be used to allow the connection to be made on any of the host's interfaces and addrlen is indicating length in bytes of structure referenced by address. On success, bind () returns a zero. On failure, it returns -1 with an error number.

Use of Bind

During networking programming we find that there are three user of Bind system call.

- A specific network address and port address can be registered to a client.
- Server needs to register a specific network & port address with its socket, basically it informs the system that "The address associated with me is this, and if you get any message on this address just transfer it to me".

- In case of connectionless client (see your unit 1 of block 1 to know about connectionless and connection oriented services), it needs to assure that system has provided some valid unique addresses, so that when server sends some message it will reach the desired client. This approach of client is same like us, when we write letter we always check whether we have written own address on that envelop or not so that we can get reply on that address.

The bind system call fills the two tuple of association, Local address and Local process elements. When we will use Bind function the client needs to call socket system call because it needs to use the returned value as socket descriptor.

When binding is over, then in-case of UDP socket it is ready to send and receive datagram's. For TCP sockets, the socket is ready to connect or accept calls. Let's see what are these accept and connect call.

Let's have an example:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
#define CLIENTPORT 8090

main()
{
   int sockfd;
   struct sockaddr_in local_addr;

   sockfd = socket(AF_INET, SOCK_STREAM, 0);
   client_addr.sin_family = AF_INET;     /* host byte order */
   client_addr.sin_port = htons(CLIENTPORT); /* short, network byte order */
   client_addr.sin_addr.s_addr = inet_addr("192.10.10.10");
   bzero(&(client_addr.sin_zero), 8);    /* zero the rest of the struct */

   /* error checking for bind(): */
   bind(sockfd, (struct sockaddr *)&client_addr, sizeof(struct sockaddr));
   .
   .
   .
  }
```

In socket programming we have different methods which may be useful to you, when you want to choose an unused port at random you can write /* choose an unused port at random */

and for choosing IP address you can use /* use client IP address */

2.2.3 Connect System Call

Generally "connect" function is used by client program to establish a connection to a remote machine (server).

Syntax of connect is given below:

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

int connect(int sockfd, struct sockaddr_in *server_addr, int serv_addrlen);

connect() returns 0 if successful or returns -1 on unsuccessful and sets errno

In the connect call, sockfd is a socket file descriptor returned by socket(), server_addr points to a structure with *destination* port and IP address and serv_addrlen set to sizeof (struct sockaddr).

Initial code necessary for a TCP client:

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define SERVER_IP   "190.20.10.12"
#define SERVER_PORT 1729

main()
{
 int sockfd;
 struct sockaddr_in ser_addr;   /* will hold the destination addr */

 if ((sockfd = socket(AF_INET, SOCK_STREAM, 0))<0)
 {
  perror("socket error ");
  exit(1);
 }

 Ser_addr.sin_family      = AF_INET;           /* inet address family */
 Ser_addr.sin_port        = htons(SERVER_PORT);   /* destination port */
 Ser_addr.sin_addr.s_addr = inet_addr(SERVER_IP); /* destination IP address */
 memset(&(dest_addr.sin_zero), '\0', 8);   /* zero the rest of the struct */
 /* In function void *memset(void *s, int c, size_t n); The  memset() function
fills the first n bytes of the memory area pointed to be s with the constant byte c.*/

 if (connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr))<0)
 {
  perror("connect error");
  exit(1);
 }
```

During the bind call example earlier we have ignored the local port number, now we are concerned about the remote port. The kernel will choose a local port, and the site we connect to, will automatically get this information from client.

If connectionless client (UDP) use the connect (), then the system call just stores the server address specified by the process, so that the system knows where to send any future data the process writes to the sockfd descriptor. Also, the socket will receive only datagrams from this address. Note that the destination address is not necessary for each Datagram sent, if we are connecting a UDP socket.

Connect system call result in actual connection establishment between the local and remote system in case of connection-oriented protocol. At this point some agreement for the future data exchange happens like buffer size, amount of data, acknowledgement etc., as we have discussed earlier in Unit 3 of Block 1. This exchange of information between client and server are known as 3-way handshake,

To use connect function, first of all client needs to call the socket (), then connect () function sets value of server socket address and client socket address is either provided by bind system call or set by the operating system.

2.2.4 Listen System Call

TCP server, binds to a well-known port and waits for a client to try to connect to it. This process is called as "listening" and it is performed by calling the listen () system call.

The Listen system call is used in the server in the case of connection-oriented communication to prepare a socket to accept messages from clients. It creates a passive socket (recall the concept of passive and active mode of socket) from an unconnected socket. 'Listen' initializes a queue for waiting connections. Before calling listen, socket must be created and its address field must be set.

Usually Listen() is executed after both socket() and bind() calls and before accept() [accept system call will be discussed in next sections].

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

int listen(int sockfd, int Max_conn)

On success, listen() returns "0"
On failure, it returns "-1" and the error is in errno.

Listen takes two parameters, first the socket you would like to listen on and another is the Maximum number of request that will be accepted on the socket at any given time. Socket you would like to listen on is represented here as **sockfd** is the usual socket file descriptor from the socket() system call. Maximum number of connections allowed on the incoming queue at any given time is represented by **backlog.** On the server all incoming connections requests from clients come and wait in one special **queue** (from this queue server choose the connection and accept the request). We have to define the limit on queue that how many connections can wait in a queue. Generally this limit is set to 20. But you can set it with any other number also, for example,

listen (socket, 5), call will inform the operating system that server can only allow five client sockets to connect at any one given time.

If the queue is full, then the new connection request will be rejected, which will result in an error in the client application. Queued connections are removed from the queue and completed with the accept() function, which also creates a new socket for communicating with the client.

Example:

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

#define SERVERPORT 1729

int main()
{
  int sockfd;
  struct sockaddr_in ser_addr;

  if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
  {
    perror("socket error");
```

```
  exit(1);
}

ser_addr.sin_family      = AF_INET;
ser_addr.sin_port        = htons(SERVERPORT);
ser_addr.sin_addr.s_addr = inet_addr(INADDR_ANY);
memset(&(ser_addr.sin_zero), '\0', 8);   /* zero the rest of the struct */

if ( bind(sockfd, (struct sockaddr *)& ser_addr, sizeof(struct sockaddr)) < 0 )
{
  perror("bind");
  exit(1);
}
if ( listen(sockfd, 5) < 0 ) / * Inform the operating sytem that server cn allow
only 5 clients*/
{
  perror("listen error");
  exit(1);
}
  .
  .
  .
 }
```

2.2.5 Accept () System Call

After listen system call accept() completes the process of establishing connection between the server and the client. Remember accept system call is used for stream sockets server like TCP server. It removes the first waiting connection request from the queue of pending connections, creates a new socket with the same properties of old socket (which you specified in accept call) and allocates a new file descriptor for the socket. But think what will happen if there is no pending connection waiting in the queue?

In this case we can have two possibilities one *socket is marked as non-blocking* another *socket is not marked as non-blocking.* In this first case **accept()** blocks the caller until a connection is present and if the socket is marked non-blocking and no pending connections are present on the queue, **accept()** returns an error.

Let's see the syntax of accept() system call

```
#include "sys/types.h"
#include "sys/socket.h"

int accept(int socket, struct sockaddr *clientAddress, int *addressLength);

On success, accept() returns the new socket descriptor.
On failure, it returns -1 and the error is in errno.
```

Here in the code accept() takes three arguments first one is **socket** which represent socket on which server is listening the requests, that's why this is also called as **Listening –socket**. Socket on which the server has successfully completed socket(), bind(), and listen() system call. Second is ***clientAddress** which point to an address (struct sockaddr_in). We can use this address to determine the IP address and port of the client. Third and last one ***addressLength** is generally an integer that will contain the actual length of address structure of client (*addressLength should be set to size of (struct sockaddr_in). Accept() returns -1 on error. If it succeeds, it returns a non-

negative integer that is a descriptor for the accepted socket. Let's see one example, which contains the coding showing all the steps we follow till accept().

Example code for accept ().

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define SERVERPORT 1729

int main()
{
 int sockfd, newsock, len;
 struct sockaddr_in ser_addr, client_address;

 if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
 {
   perror("socket error");
   exit(1);
 }
 if ( bind(sockfd, (struct sockaddr *)& ser_addr, sizeof(struct sockaddr)) < 0 )
 {
   perror("bind error");
   exit(1);
 }
 if ( listen(sockfd, 5) < 0 )
 {
   perror("listen error");
   exit(1);
 }
 len = sizeof(client_address);
 while(1)
 {
  if ((newsock = accept ( sockfd, (struct sockaddr *)&cliaddr, &len))<0)
  {
    perror("accept error");
    exit(1);
  }
```

If accept is successful it returns new socket descriptor returned by accept system call which completes all the five elements of 5-tuple associations while the old socket which we passed in accept () contains only three elements of 5 tuple association (except remote address and remote process). As you know most of the server are concurrent in practical situation so all go for new socket automatically as a part of accept () as given in the following code:

Concurrent Service

```
int sockfd, newsockfd;
if ((sockfd = socket(...)) < 0)
        error_handle("socket error");
if (bind(sockfd, ...) < 0)
        error_handle("bind error");
if (listen(sockfd, ...) < 0)
```

```
        error_handle("listen error);
for ( ; ; ) {
        newsockfd=accept(sockfd, ...); /* blocks */
        if (newsockfd < 0)
                error_handle("accept error");
        if (fork() == 0) {
                close(sockfd);                  /* child */
                do_process(newsockfd);          /* process the request */
                exit(0);
        }
        close(newsockfd);               /* parent */
}
```

When a connection request is received and accepted, the process forks, with the child process serving the connection and the parent waiting for another connection request. Let's see the code for iterative server also:

The iterative server

```
int sockfd, newsockfd;
if ((sockfd = socket(...)) < 0)
        error_handle("socket error");
if (bind(sockfd, ...) < 0)
        error_handle("bind error");
if (listen(sockfd, ...) < 0)
        error_handle("listen error);
for ( ; ; ) {
        newsockfd=accept(sockfd, ...); /* blocks */
        if (newsockfd < 0)
                error_handle("accept error");

        do_process(newsockfd);          /* process the request */
        close(newsockfd);
          }
```

Here the server handles the request using the connected socket descriptor, newsockfd. It then terminates the connection with a close and waits for another connection using the original descriptor, sockfd, which still has its remote address and remote process unspecified.

☞ Check Your Progress 1

1) Which of the following is returned on success in socket system call?
   a) Negative integer value
   b) Big integer value
   c) Small integer value
   d) Text "success"

……

……

……

2) Which of the following header file contains the definition of data types?

a) Socket. h

b) type. h

c) type. h

d) data type. H

……

……

……

3) Listen system call is used on the server in the case of

a) Connection–less common

b) Connection-oriented comma

c) Simplex Comma

d) Duplex Comma full

……

……

……

2.3 ELEMENTARY DATA TRANSFER CALLS

Till now we have studied about connection establishment between client and server, let's start the discussion about data transfer between them.

Once a connection is established between sockets, an application program can send and receive data. Sending and receiving data can be done with any one of the several system calls given in this section. The system calls vary according to the amount of data to be transmitted and received and the state of the socket being used to perform the data transfers. The system call pairs (read, write), (send, recv), (sendto, recvfrom) can be used to transfer data (or communicate) on sockets. Let's discuss each pair of system call one by one. The read() system calls allows a process to receive data on a connected socket without receiving the sender's address. The write system calls can be used with a socket that is in a connected state, as the destination of the data is implicitly specified by the connection. The sendto() subroutine allows the process to specify the destination for a message explicitly. The recvfrom() subroutine allow the process to retrieve the incoming message and the sender's address. The use of the above system call varies from protocol to protocol.

read() and write()

read()

This is used to receive data from the remote machine, it assumes that there is already an open connection present between two machines and it is possible only in case of TCP (the connection oriented protocol in TCP/IP).

Prototype syntax for read system call is

```
#include "sys/types.h"
#include "sys/socket.h"

int read(int sockfd, char void *buff, int buff_len );
```

Condition	returns
If successful	Numbers of bytes read
If EOF	0
If Error	-1

Here in the above syntax of read() first *sockfd* is socket descriptor, *buff* is a pointer to the buffer where we can store the data and *buff_len* is length of buffer or capacity of buffer.

As given above on success, read() return the number of bytes read, if error occurs it returns –1 (and *errno* is set appropriately) and if it returns zero that means it read all data in file and EOF (end of file) has come.

write

When we want to a send some data to a process running on remote machine we use write() system call. Write() assumes that connection is already open between sender and receiver machine, that's why this is limited to process using TCP. Write () function attempts to write the specified number of bytes from the specified buffer to the interface buffer specified socket descriptor (sockfd).

```
#include "sys/types.h"
#include "sys/socket.h"

int write(int sockfd, const void *buff, int buff_len );
```

Condition	returns
If successful	Numbers of bytes written
If Error	-1

Here in the above syntax of **write ()** first argument *sockfd* is socket descriptor, buff is a pointer to the buffer from where we can write the data and *buff_len* is length of buffer or capacity of buffer. As given above in figure on success, the numbers of bytes written are returned (where zero indicates nothing was written). On error, -1 is returned, and errno is set appropriately.

send, sendto, recv, and recvfrom

send, sendto, recv, and recvfrom system calls are similar to the standard read and write system calls but these calls require some additional arguemnts like *flag ,*dest_addr, addrlen and *sour_addr as given below :*

Name of additional Argument(s)	Name of system call
flags	send() and recv().
*flag ,*dest_addr, addrlen*	sendto()
*flag ,*sour_addr, *addrlen*	recvfrom()

You can note *addrlen* argument in *sendto* is an integer value while in *recvfrom* it is a pointer to an integer value.You can compare the syntax of these system calls in the

code given below. You can easily note that the first three arguments *sockfd ,*buff* and *nbytes* are similar to first three arguments of read() and write() argument.

```
#include "sys/types.h"
#include "sys/socket.h"
int send(int sockfd, char *buff, int nbytes, int flags);

int recv(int sockfd, char *buff, int nbytes, int flags);

int sendto(int sockfd, char *buff, int nbytes, int flags,
            struct sockaddr *dest_addr, int addrlen);

int recvfrom(int sockfd, char *buff, int nbytes, int flags,
              struct sockaddr sour_addr, int* addrlen);
```

The use of these system calls depends on the protocol used in your socket association.When we use TCP commonly we use *send()* and *recv()*, allthough *sendto()*, *recvfrom()*, and read() and write() can also work. In case of UDP , if you have bound a IP address and Port, you should use *recv()* for reading, and *send()* for sending. If you have not used bind on the UDP socket it is better to use *sendto()* and *recvfrom()*.

The different flag values are defined in the **sys/socket.h** header file. Flag can be defined as a nonzero value if the application program requires one or more of the following flag values:

MSG_OOB	**Sends or receives out-of-band data.**
MSG_PEEK	**Looks at data without reading.**
MSG_DONTROUTE	**Sends data without routing packets**
MSG_MPEG2	**Sends MPEG2 video data blocks**

Send ()

The send() function initiates transmission of a message from the specified socket to its peer. The send() function sends a message only when the socket is connected.

sockfd is socket descriptor, note that socket must be in connected state before sending data, **buff* is a pointer to data to be transmitted, *nbytes* indicates number of bytes to be sent and *flags toolbars* control flags for special protocol features; set to 0 in most cases. On success, send() returns the number of bytes actually sent.On failure, it returns -1 and the errno.

The following code example shows how you can send data to a connected socket.

```
int i;
  /*  sockfd is the connected socket file descriptor  */
  i = send( sockfd, "Hello World!\n", 13, 0);
  if( i > 0 ){
     printf("sent %d bytes\n", i);
  }
```

sendto()

int sendto(int sockfd, char *buff, int nbytes, int flags, struct sockaddr *dest_addr, int addrlen);

The sendto() function sends a message through a connection oriented or connectionless socket. If sockfd specfic is a connectionless socket, the message is

sent to the address specified by *dest_addr. If sockfd specifies is a connection oriented socket, *dest_addr and addrlen parameters are ignored.

*dest_addr indicate destination address for data to busiest, addrlen represent length of destination address structure and other arguments are similar to send() function explained above. On success, sendto() returns the number of bytes sent. On failure, it returns -1 and the error is in errno.

recv()

int recv(int sockfd, char *buff, int nbytes, int flags);

The recv() function receives a message from a socket. The recv() call can be used on a connection oriented socket and, connectionless socket. If no messages are available at the socket, the recv() call waits for a message to arrive if the socket is blocking. If a socket is nonblocking, -1 is returned and the external variable errno is set.

The flags parameter can be set to MSG_PEEK, MSG_OOB, both, or zero. If it is set to MSG_PEEK, any data returned to the user still is treated as if it had not been read, i.e., the next recv() re-reads the same data.

If successful, recv() returns the number of bytes received, otherwise, it returns -1 and sets errno to indicate the error. recv() returns 0 if the socket is blocking and the connection to the remote node failed.

The following code example shows how you would use recv on a connected socket.

```
int i;
char buff[ 250 ];
/*  clear out buff  */
memset( buff, 0, sizeof( buff ) );
/*  sockfd is the connected socket file descriptor  */
i = recv( sockfd, buff, sizeof(buff), 0);
if( i < 0 )
        {
           printf("error in recving data %d\n", i);
        }
        if( i == 0 )
        {
           printf("socket closed remotely\n");
        }
        if( i > 0 )
        {
    printf("received %d bytes\n", i);
    printf("data :\n%s", buff);
}
```

recvfrom ()

int recvfrom(int sockfd, char *buff, int nbytes, int flags, struct sockaddr *from, int *addrlen);

The recvfrom() system call receives a message from a socket and capture the address from which the data was sent. Unlike the recv() call, which can only be used on a connected stream socket or bound datagram socket, recvfrom() can be used to receive data on a socket whether or not it is connected. If no messages are available at the socket, the recvfrom() call waits for a message to arrive if the socket is blocking. If a socket is nonblocking, -1 is returned and the external variable errno is set.

2.4 CLOSING A SOCKET

There are two ways to close a socket . First is **close(sockfd)** and another is **shutdown(socked, prio).** Let's check what is the difference between both. In close() you can close the socket imeediately, while in shutdown() you can close the socket, after flushing buffers. You can indicate what buffers shutdown should flush with an integer **prio**. The following example(s) show how to use shutdown, and close, on a connected socket.

```
/*  close socket imeediately */
  • close( sockfd );
  • shutdown( sockfd, prio);
/*  close socket, and dont recieve any more */
shutdown( sockfd, 0);
/*  close socket, and dont send any more */
shutdown( sockfd, 1);
/*  close socket, and dont recieve, or send any more */
shutdown( sockfd, 2);
```

close(): The close() function is used to delete a socket descriptor created by the socket() function.

```
#include <socket.h>
#include <uio.h>
int close (socket )
int socket;
```

close() deletes the socket descriptor, sockfd, from the internal descriptor table maintained for the application program and terminates the existence of the communications endpoint. If the socket was connected, the connection is terminated.

shutdown() : The shutdown() function is used to gracefully shut down a socket.

```
#include <socket.h>
#include <uio.h>
int shutdown (sockfd, prio)
int sockfd, prio;
```

The input path can be shut down while continuing to send data, the output path can be shut down while continuing to receive data, or the socket can be shut down in both directions at once but the data queued for transmission is not lost in shutdown(). If prio is 0, further receives are disallowed. If prio is 1, further sends are disallowed. If how is 2, further sends and receives are disallowed.

☞ Check Your Progress 2

1) When read () system calls returns 0 (zero) it means?

   a) Error has occurred

   b) Buffer is empty

   c) Buffer is full

   d) End of file has occurred

   ……

2) Which of the following header file contains the definition of data types?

   a) Disconnected
   b) Ended
   c) Listing
   d) Connected

   ……

   ……

   ……

3) The recvfrom() function receives a message from socket and capture the

   a) Message from pear socket
   b) Address from which the data was sent
   c) Size of buffer from which the data was sent
   d) Port number from the pear socket.

   ……

   ……

   ……

2.5 TCP AND UDP ARCHITECTURES

UDP architectures

As you know broadly the services in TCP/IP are divided in connection oriented and connection-less services, for which TCP and UDP protocols are responsible. It is important for you to understand the architecture of these UDP and TCP communication.

UDP Client algorithm

To understand the communication architecture between UDP client and server, let use different socket calls.

Here we have explained you the step of UDP client algorithm, first you create a socket, then bind it to a local port remember if bind is not used, the kernel will select a free local port), establish the address of the server, write and read from it, and then terminate. In case, client is not interested in a giving reply then there is no need to use bind.

UDP server algorithm

These are steps generally involved in UDP server, here first you create a socket, bind it to a local port, accept and reply to messages from client, and if you want terminate.

The UDP client/server architecture

The system calls are different for a client-server using a connectionless protocol, from the connection-oriented case. The diagram given below in *Figure 2* shows a typical sequence of systems calls for both the client and the server in connection-less (UDP) communication:

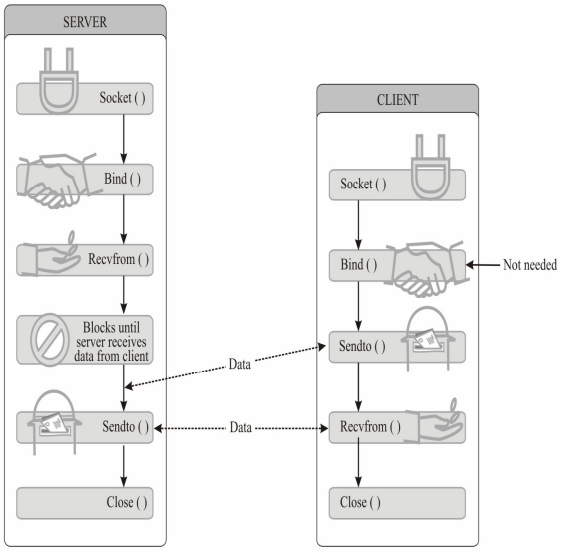


Figure 2: UDP Architecture

TCP architectures

Similarly to understand the architectural communication between TCP client and server we have different socket call in the following section.

TCP Client Algorithm

Here the sequence of steps for connection-oriented client are given. First you will create a socket, bind it to a local port (we usually do not call bind), establish the address of the server, communicate with it, if you want, terminate.

TCP Server Algorithm

These are algorithm sequence for connection oriented server, in this case you first create a socket, bind it to a local port, set up service with indication of maximum number of concurrent services, accept requests from connection oriented clients, receive messages and reply to them and then, finally terminate.

TCP Client/Server Architecture

Typical sequence of system calls to implement TCP clients and servers are given below in the *Figure 3*.

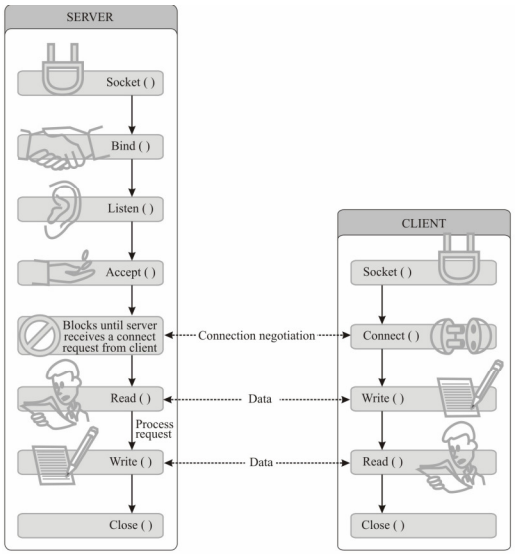


Figure 3:TCP Architecture

2.6 NETWORKING EXAMPLE

An example of an UDP echo client and server is given in this section. The client code is as follows:

UDP echo client

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
```

Include all the necessary header files.

```
#define LINES 128
#define PORT 7200

int main(int argc, char *argv[])
{
 int sockfd;
 int bytesent;
 int byterec;
 struct sockaddr_in ser_addr;
 struct sockaddr_in echo_addr
 socklen_t len= sizeof(ser_addr);
 char sendline[LINES];
 char recvline[LINES + 1];
 if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
 {
  perror("socket error");
  exit(1);
 }
 /* fill address completely with 0's */
 memset(& ser_addr, '\0', sizeof(ser_addr));
 ser_addr.sin_family = AF_INET;
 ser_addr.sin_addr.s_addr = htonl(INADDR_ANY);
 ser_addr.sin_port = htons(PORT); ser_addr

 while (1)
 {
  printf("==>");
  fflush(stdout);
  fgets(sendline, LINES, stdin);
  if (strcmp(sendline, "exit\n") == 0)
  break;
  bytesent = sendto(sockfd, sendline, strlen(sendline), 0,
          (struct sockaddr *) &ser_addr, sizeof(ser_addr));
  if (bytesent == -1)
  {
   perror("sendto error");
   exit(1);
  }
  if (strcmp(sendline, "Terminate!\n") != 0)
  {
   if ((byterec = recvfrom(sockfd, recvline, LINES, 0, & echo_addr, &len)) < 0)
   {
```

```
      perror("recvfrom error");
      exit(1);
     }
     recvline[byterec] = '\0';
     printf(" from server: %s\n", recvline);
    }
    }
   close(sockfd);
    printf("echo client normal end\n");
  return 0;
}
```

UDP echo server

The server code is as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>

#define LINES 128
#define PORT  7200
int
main(int argc, char *argv[])
{
  int sockfd
  int byterec;
  struct sockaddr_in ser_addr;
  struct sockaddr_in cli_addr;
  socklen_t len=sizeof(cli_addr);
  char recvline[LINES + 1];

   if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
  {
    perror("socket error");
    exit(1);
  }
  memset(& ser_addr, '\0', sizeof(ser_addr));
  ser_addr.sin_family = AF_INET;
  ser_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  ser_addr.sin_port = htons(PORT);
  bind(sockfd, (struct sockaddr *) & ser_addr, sizeof(ser_addr));
  while (1)
  {
   if ( (numrec = recvfrom(sockfd, recvline, LINES, 0,
                (struct ser_addr*) & cli_addr,& len)) < 0)
    {
      perror("recvfrom error");
      exit(1);
    }
    fprintf(stdout, "connection from %s, port %d. Received %d bytes.\n",
          inet_ntoa(cli_addr.sin_addr),
          ntohs(cli_addr.sin_port),
          byterec);
```

```
    fflush(stdout);
    recvline[byterec]='\0';
    if (strcmp(recvline, "Terminator!\n") == 0)
    {
      fprintf(stdout, "a client want me to terminate\n");
      fflush(stdout);
      break;
    }
     if (sendto(sockfd, recvline, byterec, 0, &cli_addr, len) < 0)
    {
      perror("sendto error");
      exit(1);
    }
  }
  fprintf(stdout, "server normal end\n");
  fflush(stdout);
  return 0;
}
```

Check Your Progress 3

1) Draw the sequence of system calls required for implementing TCP Clients and server.

……

……

……

……

……

……

……

2) Draw the sequence of system calls required for implementing UDP clients and server.

……

……

……

……

……

……

……

2.7 SUMMARY

We have discussed different elementary system call and their uses in connection establishment, termination and data transfer in TCP and UDP client server architecture. This unit describes the sockets programming interface. The special needs of network communication are discussed in general terms, and socket types and addressing schemes are introduced. In the first section we have covered the elementary system call to provide you basic knowledge about socket programming. The system calls for data transfer were discussed in detail with an example. We have

also covered the different techniques to close the socket, which will definitely help you during programming. The unit concludes with annotated algorithms of client and server communication programs. In the next unit *Socket Programming* we have given advanced socket calls, which will provide you the in-depth knowledge of the functions and their characteristics that are required for developing network applications.

2.8 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

1) C

2) C

3) B

Check Your Progress 2

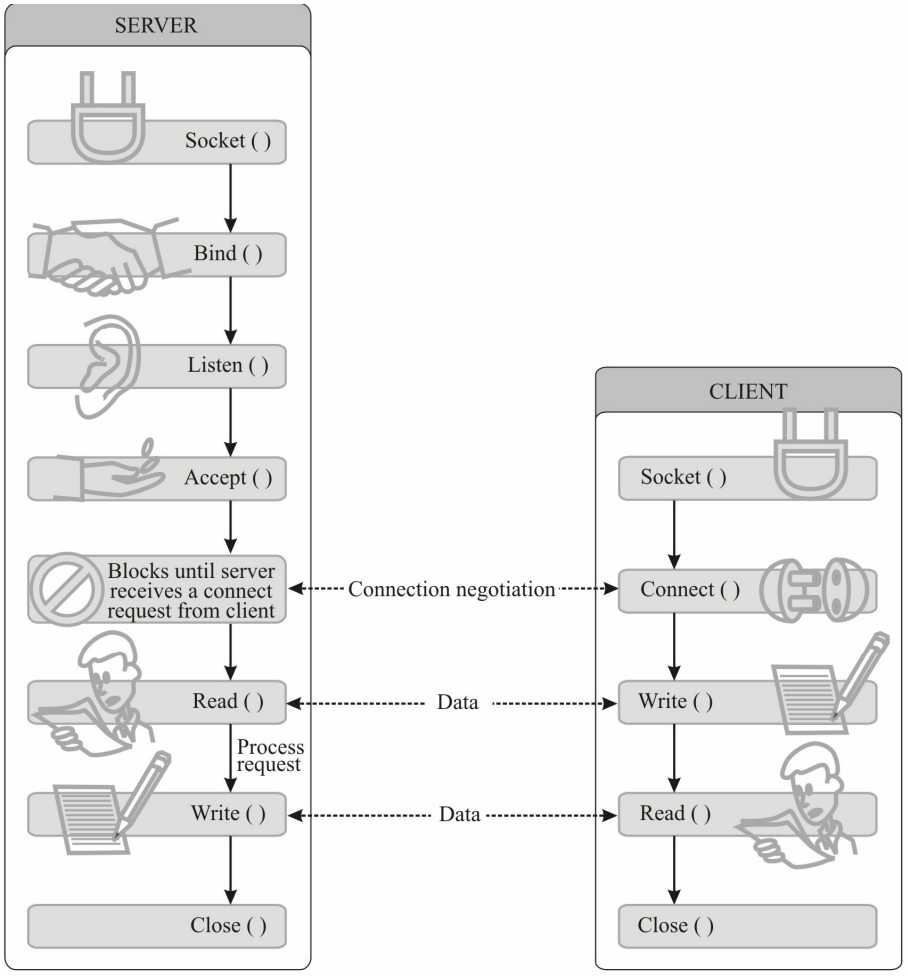
1) D

2) D

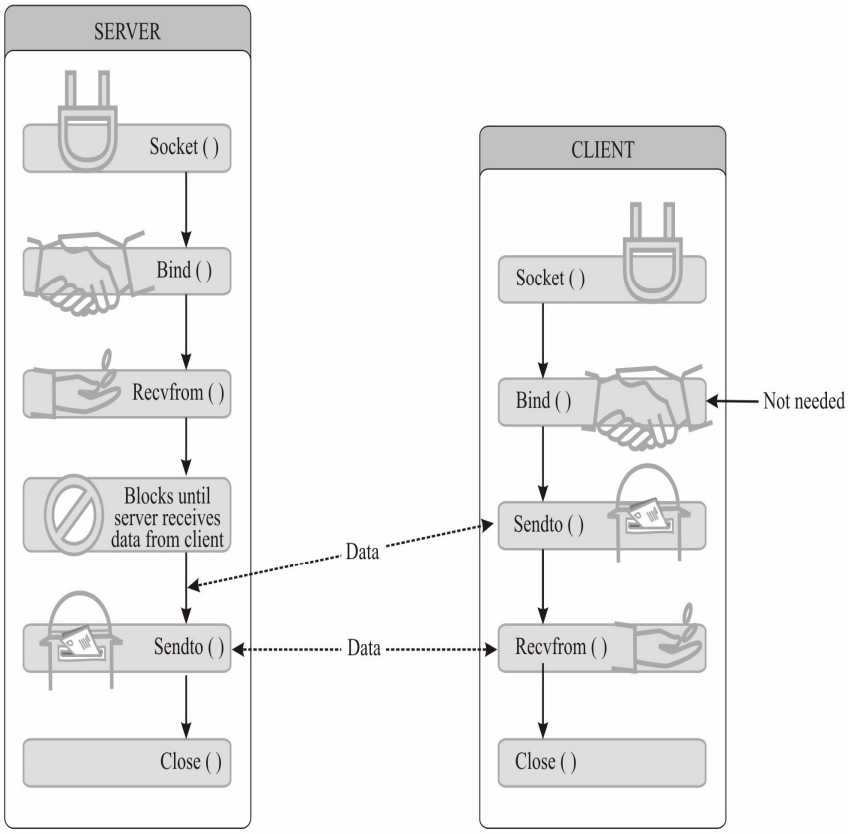
3) B

Check Your Progress 3

1)



2)



2.9 FURTHER READINGS

1) Andrew S. Tanenbaum, *Computer Networks*, Third edition.

2) Behrouz A. Forouzan, *Data Communications and Networking*, Third edition.

3) Douglas E. Comer, *Internetworking with TCP/IP* Vol.1: Principles, Protocols, and Architecture (4th Edition).

4) William Stallings, *Data and Computer Communications*, Seventh Edition.

5) W. Richard Stevens, *The Protocols (TCP/IP Illustrated*, Volume 1).

6) W. Richard Stevens, "*UNIX Network Programming"*, Prentice Hall.