LAB NO: 1 **Date:**

INTRODUCTION TO SOCKET PROGRAMMING

Objectives

- To illustrate the significance of socket programming
- To recognize basic socket function calls by performing simple client server operations

Introduction

Computer network is a communication network in which a collection of computers are connected together to facilitate data exchange .The connection between the computers can be wired or wireless. A computer network basically comprises of 5 components as shown in Fig.1.1:

- Sender
- Receiver
- Message
- Transmission medium
- Protocols

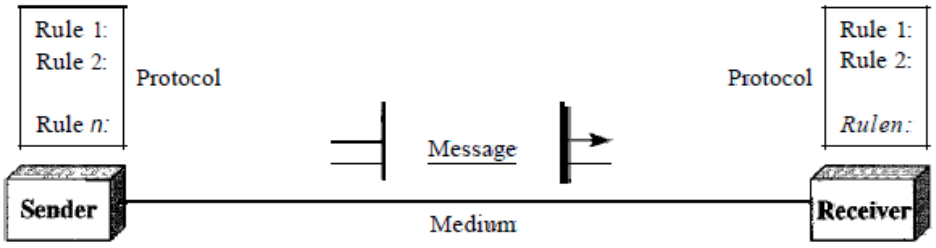

Fig.1.1: Components of data communication

Client Server Architecture

Communication in computer networks follows a client server model as illustrated in Fig. 1.2. Here, a machine (referred as **client**) makes a request to connect to another machine (called as **server**) for providing some service. The services running on the server run on known ports (application identifiers) and the client needs to know the address of the server machine and this port in order to connect to the server. On the other hand, the server does not need to know about the address or the port of the client at the time of connection initiation. The first packet which the client sends as a request to the server contains these information about the client which are further

used by the server to send any information. Client **(Active Open)** acts as the active device which makes the first move to establish the connection whereas the server **(Passive Open)** passively waits for such requests from some client.

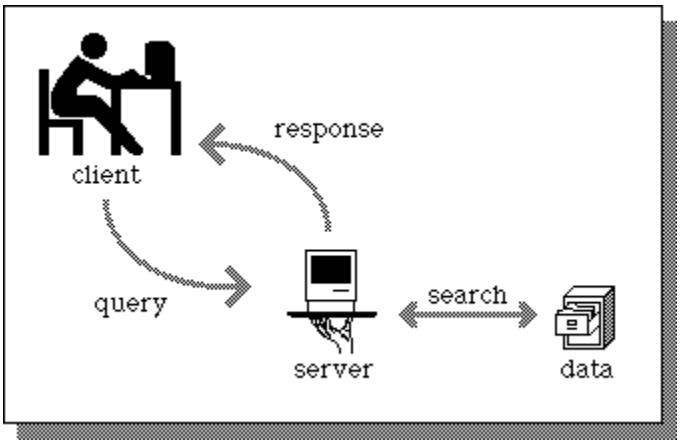

Fig.1.2: Illustration of Client Server Model

Basics of Socket Programming

In UNIX, whenever there is a need for inter process communication within the same machine, we use mechanism like signals or pipes. Similarly, when we desire a communication between two applications possibly running on different machines, we need **sockets**. A network socket is an endpoint for sending or receiving data at a single node in a computer network that supports full duplex transmission.
Sockets are created and used with a set of programming requests or "function calls" sometimes called the sockets Application Programming Interface (API). The most common sockets API is the Berkeley UNIX C interface for sockets which are the interfaces between applications layer and the transport layer that acts as a virtual connection between two processes. Each socket is identified by an address so that processes can connect to them.
Socket Address = IP Address + Port Number

Types of Communication Services

The data transfer between client and server application initiated by socket APIs can be achieved either by using connection oriented or connectionless services.

a. **Connection Oriented Communication :**In connection oriented service a connection has to be established between two devices before starting the communication to allocate the resources needed to aid the data transfer. Then the message transfer takes place until the connection is released. This type of communication is characterized by a high level of

reliability in terms of the number and the sequence of bytes. It is analogous to the telephone network.

b. **Connectionless Communication:** In connectionless the data is transferred in one direction from source to destination without checking that destination is still there or not or if it prepared to accept the message. Authentication is not needed in this. It is analogous to the postal service where Packets (letters) are sent at a time to a particular destination.

Based on the two types of communication described above, two kinds of sockets are used:

a. **Stream sockets:** used for connection-oriented communication, when reliability in connection is desired. Protocol used is TCP (Transmission Control Protocol) for data transmission.
b. **Datagram sockets:** used for connectionless communication, when reliability is not as much as an issue compared to the cost of providing that reliability. For e.g. Streaming audio/video is always sent over such sockets so as to diminish network traffic. Protocol used is UDP (User Datagram Protocol) for data transmission.

Socket System Calls for Connection-Oriented Protocol

Steps followed by client to establish the connection:

a. Create a socket
b. Connect the socket to the address of the server
c. Send/Receive data
d. Close the socket connection

Steps followed by server to establish the connection:

1. Create a socket
2. Bind the socket to the port number known to all clients
3. Listen for the connection request
4. Accept connection request. This call typically blocks until a client connects with the server.
5. Send/Receive data
6. Close the socket connection

The sequence of socket function calls to be invoked for a connection oriented client server communication is depicted in Fig. 1.3 and the significance of these function calls is summarized in Table 1.1.

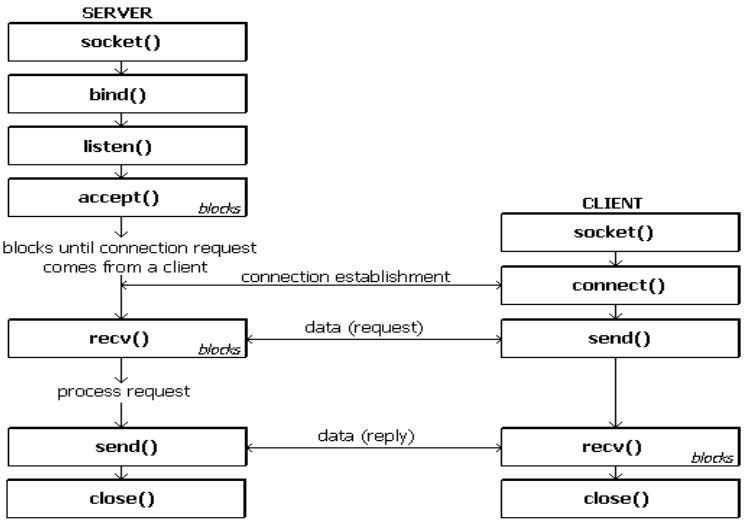

Fig.1.3: Connection oriented Socket Structure

Table 1.1: Significance of each socket function call

Primitive	Meaning
Socket	Create a new communication request
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Socket System Calls for Connectionless Protocol

Steps of establishing a UDP socket communication on the client side are as follows:

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

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

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

Fig. 1.4 shows the interaction between a UDP client and server. The client 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 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.

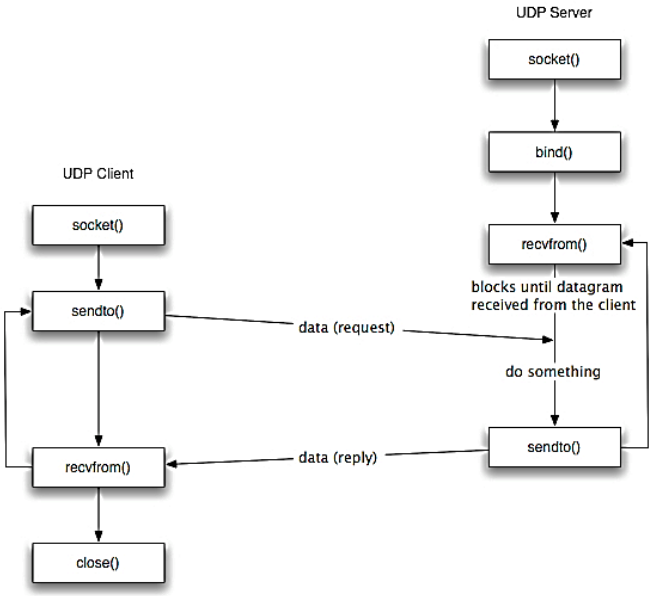

Fig.1. 4: Connectionless Socket Structure

Socket System Calls in detail

Headers

1. **sys/types.h:** Defines the data type of socket address structure in unsigned long.
2. **sys/socket.h:** The socket functions can be defined as taking pointers to the generic socket address structure called sockaddr.
3. **netinet/in.h:** Defines the IPv4 socket address structure commonly called Internet socket address structure called sockaddr_in.

Structures

Various structures are used in Unix Socket Programming to hold information about the address and port, and other information. Most socket functions require a pointer to a socket address structure as an argument.

1. **Sockaddr:** This is a generic socket address structure, which will be passed in most of the socket function calls. It holds the socket information. The table 1.2 provides a description of the member fields.

```
struct sockaddr {
unsigned short sa_family;
char sa_data[14];
```

};

Table 1.2: Generic socket address structure fields

Attribute	Values	Description
sa_family	AF_INET AF_UNIX AF_NS AF_IMPLINK	It represents an address family. In most of the Internet-based applications, we use AF_INET.
sa_data	Protocol-specific Address	The content of the 14 bytes of protocol specific address are interpreted according to the type of address. For the Internet family, we use port number IP address, which is represented by *sockaddr_in* structure.

2. sockaddr_in

This helps to reference to the socket's elements and table 1.3 provides a description of the member fields.

struct sockaddr_in {
short int sin_family;
unsigned short int sin_port;
structin_addrsin_addr;
unsigned char sin_zero[8]; };

Table 1.3: Structure fields of sockaddr_in

Attribute	Values	Description
sa_family	AF_INET AF_UNIX AF_NS AF_IMPLINK	It represents an address family. In most of the Internet-based applications, we use AF_INET.
sin_port	Service Port	A 16-bit port number in Network Byte Order.
sin_addr	IP Address	A 32-bit IP address in Network Byte Order.
sin_zero	Not Used	This value could be set to NULL as this is not being used.

3. in_addr

This structure is used only in the above structure as a structure field and holds 32 bit netid/hostid.

struct in_addr {
unsigned long s_addr;

};
The table 1.4 provides a description of the member fields.

Table 1.4: Structure fields of in_addr

Attribute	Values	Description
s_addr	service port	A 32-bit IP address in Network Byte Order.

4. hostent

This structure is used to keep information related to host.

```
struct hostent {
char *h_name;
char **h_aliases;
int h_addrtype;
int h_length;
char **h_addr_list

#define h_addrh_addr_list[0]
};
```

The table 1.5 provides a description of the member fields.

Table 1.5: Structure fields of hostent

Attribute	Values	Description
h_name	ti.com etc.	It is the official name of the host. For example, tutorialspoint.com, google.com, etc.
h_aliases	TI	It holds a list of host name aliases.
h_addrtype	AF_INET	It contains the address family and in case of Internet based application, it will always be AF_INET.
h_length	4	It holds the length of the IP address, which is 4 for Internet Address.
h_addr_list	in_addr	For Internet addresses, the array of pointers h_addr_list[0], h_addr_list[1], and so on, are points to structure in_addr.

NOTE − h_addr is defined as h_addr_list[0] to keep backward compatibility.

5. servent

This particular structure is used to keep information related to service and associated ports.

```
struct servent {
char  *s_name;
char  **s_aliases;
int s_port;
char  *s_proto;
};
```

The table 1.6 provides a description of the member fields.

Table 1.6: Structure fields of servent

Attribute	Values	Description
s_name	http	This is the official name of the service. For example, SMTP, FTP POP3, etc.
s_aliases	ALIAS	It holds the list of service aliases. Most of the time this will be set to NULL.
s_port	80	It will have associated port number. For example, for HTTP, this will be 80.
s_proto	TCP UDP	It is set to the protocol used. Internet services are provided using either TCP or UDP.

Ports and Services

To resolve the problem of identifying a particular server process running on a host, both TCP and UDP have defined a group of ports. A port is always associated with an IP address of a host and the protocol type of the communication, and thus completes the destination or origination address of a communication session. A port is identified for each address and protocol by a 16-bit number, commonly known as the **port number**.Specific port numbers are often used to identify specific services as listed below.

1. **Ports 0-1023** *(well-known ports)*: reserved for privileged services like for ftp: 21 and for telnet: 23.

2. **Ports 1024-49151** *(registered ports)*: vendors use for some applications.
3. **Ports above 49151** *(dynamic/ephemeral/private ports)*: short-lived transport protocol port for Internet Protocol (IP) communications allocated automatically from a predefined range by the IP stack software. After communication is terminated, the port becomes available for use in another session. However, it is usually reused only after the entire port range is used up.

Constructing Messages - Byte Ordering

In computers, addresses and port numbers are stored as integers of type:

‰ u_short sin_port; (16 bit)

‰ in_addr sin_addr; (32 bit)

Unfortunately, not all computers store the bytes that comprise a multibyte value in the same order. There are 2 types of byte ordering.

- **Little Endian (Host byte order)** − in this scheme, low-order byte is stored on the starting address (A) and high-order byte is stored on the next higher address (A + 1).
- **Big Endian (Network byte order)** − in this scheme, high-order byte is stored on the starting address (A) and low-order byte is stored on the next higher address (A + 1).

To allow machines with different byte order conventions communicate with each other, the internet protocols specify a canonical byte order convention for data transmitted over the network. This is known as Network Byte Order. Table 1.7 describes some of the byte order functions available in UNIX programming.

Table 1.7: Byte order conversion function description

Function	Description
htons()	Host to Network Short (16 bit)
htonl()	Host to Network Long (32 bit)
ntohl()	Network to Host Long (32 bit)
ntohs()	Network to Host Short (16 bit)

IP Address Functions

These functions convert Internet addresses between ASCII strings (what humans prefer to use) and network byte ordered binary values (values that are stored in socket address structures). Some of the commonly used IP address functions are described below.

1. **int inet_aton(const char *strptr, struct in_addr *addrptr)**

This function call converts the specified string in the Internet standard dot notation to a network address, and stores the address in the structure provided. The converted address will be in Network Byte Order (bytes ordered from left to right). It returns 1 if the string was valid and 0 on error.
Following is the usage example:

```
#include <arpa/inet.h>
(...)
int retval;
struct in_addr addrptr
memset(&addrptr, '\0', sizeof(addrptr));
retval = inet_aton("68.178.157.132", &addrptr);
(...)
```

2. in_addr_tinet_addr(const char *strptr)

This function call converts the specified string in the Internet standard dot notation to an integer value suitable for use as an Internet address. The converted address will be in Network Byte Order (bytes ordered from left to right). It returns a 32-bit binary network byte ordered IPv4 address and INADDR_NONE on error.
Following is the usage example:

```
#include <arpa/inet.h>
(...)
struct sockaddr_ indest;
memset(&dest, '\0', sizeof(dest));
dest.sin_addr.s_addr = inet_addr("68.178.157.132");
(...)
```

3. char *inet_ntoa(struct in_addr inaddr)

This function call converts the specified Internet host address to a string in the Internet standard dot notation.
Following is the usage example −

```
#include <arpa/inet.h>
char *ip;
ip = inet_ntoa(dest.sin_addr);
printf("IP Address is: %s\n",ip);
```

---

1. Socket function

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

This call returns a socket descriptor that you can use in later system calls or -1 on error.

Parameters

a. **family** − specifies the protocol family and is one of the constants as given n table 1.7

Table 1.7: Different socket address family

Family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols
AF_ROUTE	Routing Sockets
AF_KEY	Ket socket

b. **type** − It specifies the kind of socket you want. The possible socket types are listed in Table 1.8.

Table 1.8: Socket Types

Type	Description
SOCK_STREAM	Stream socket
SOCK_DGRAM	Datagram socket
SOCK_SEQPACKET	Sequenced packet socket
SOCK_RAW	Raw socket

c. **protocol** − The argument should be set to the specific protocol type given below, or 0 to select the system's default for the given combination of family and type .The default protocol for SOCK_STREAM with AF_INET family is TCP. Other protocol description is given in Table 1.9.

Table 1.9: Different protocols

Protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

Example

```
if ( (sd = socket( AF_INET, SOCK_STREAM, 0 )) < 0 ) {
        cout<<"Socket creation error";
        exit(-1);
}
```

---

2. Connect

The connect function is used by a TCP client to establish a connection with a TCP server. This call normally **blocks** until either the connection is established or is rejected.

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);

This call returns 0 if it successfully connects to the server, otherwise it returns -1 on error.

Parameters

a. **sockfd** − it is a socket descriptor returned by the socket function.
b. **serv_addr** − it is a pointer to struct sockaddr that contains destination (Server) IP address and port.
c. **addrlen** − the length of the address structure pointed to by servaddr. Set it to sizeof(structsockaddr).

---

3. Bind

The bind function assigns a local protocol address to a socket. With the Internet protocols, the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number. This function is called by TCP server only. bind() allows to specify the IP address, the port, both or neither. The table 1.10 summarizes the combinations for IPv4.

int bind(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);

This call returns 0 if it successfully binds to the address, otherwise it returns -1 on error.

Parameters

a. **sockfd** − it is a socket descriptor returned by the socket function.
b. **servaddr**− it is a pointer to struct sockaddr that contains the local IP address and port.
c. **addrlen** − Set it to sizeof(structsockaddr).

The <sys/socket.h> header shall define the type **socklen_t**, which is an integer type of width of at least 32 bits.

Table 1.10: IP address and port number combinations

IP Address	IP Port	Result
INADDR_ANY	0	Kernel chooses IP address and port
INADDR_ANY	non zero	Kernel chooses IP address, process specifies port
Local IP address	0	Process specifies IP address, kernel chooses port
Local IP address	non zero	Process specifies IP address and port

---

4. Listen

The listen function is called only by a TCP server. It converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket.

int listen(int sockfd,int backlog);

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

a. **sockfd** - it is a socket descriptor returned by the socket function.
b. **backlog** - it is the maximum number of connections the kernel should queue for this socket.

---

5. Accept

The accept function is called by a TCP server to return the next completed connection from the front of the completed connection queue.

int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);

This call returns a non-negative descriptor on success, otherwise it returns -1 on error. The returned descriptor is assumed to be a client socket descriptor and all read-write operations will be done on this descriptor to communicate with the client.

Parameters

a. **sockfd** − It is a socket descriptor returned by the socket function.
b. **cliaddr** − It is a pointer to structsockaddr that contains client IP address and port.
c. **addrlen** − Set it to sizeof(structsockaddr).

---

6. Send

The send function is used to send data over stream sockets or CONNECTED datagram sockets. If you want to send data over UNCONNECTED datagram sockets, you must use sendto() function.

int send(int sockfd, const void *msg, int len, int flags);

This call returns the number of bytes sent out, otherwise it will return -1 on error.

Parameters

a. **sockfd** − it is a socket descriptor returned by the socket function.
b. **msg** − it is a pointer to the data you want to send.
c. **len** − it is the length of the data you want to send (in bytes).
d. **flags** − it is set to 0.The additional argument flags is used to specify how we want the data to be transmitted.

---

7. Recv

The recv function is used to receive data over stream sockets or CONNECTED datagram sockets. If you want to receive data over UNCONNECTED datagram sockets you must use recvfrom().

int recv(int sockfd, void *buf, int len, unsigned int flags);

This call returns the number of bytes read into the buffer, otherwise it will return -1 on error.

Parameters

a. **sockfd** − it is a socket descriptor returned by the socket function.
b. **buf** − it is the buffer to read the information into.

c. **len** − it is the maximum length of the buffer.
d. **flags** − it is set to 0. The additional argument flags is used to specify how we want the data to be transmitted.

---

8. Sendto

The sendto function is used to send data over UNCONNECTED datagram sockets. Upon successful completion, sendto() shall return the number of bytes sent. Otherwise, -1 shall be returned and *errno* set to indicate the error.

int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen);

Parameters

a. **sockfd** − it is a socket descriptor returned by the socket function.
b. **msg** − it is a pointer to the data you want to send.
c. **len** − it is the length of the data you want to send (in bytes).
d. **flags** − it is set to 0.
e. **to** − it is a pointer to struct sockaddr for the host where data has to be sent.
f. **tolen** − it is set it to sizeof(struct sockaddr).

---

9. Recvfrom

The recvfrom function is used to receive data from UNCONNECTED datagram sockets. Upon successful completion, *recvfrom*() shall return the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, *recvfrom*() shall return 0. Otherwise, the function shall return -1 and set *errno* to indicate the error.

size_t recvfrom(int sockfd, void *buf, int len, unsigned int flags, struct sockaddr *from, int *fromlen);

Parameters

a. **sockfd** − it is a socket descriptor returned by the socket function.
b. **buf** − it is the buffer to read the information into.
c. **len** − it is the maximum length of the buffer.
d. **flags** − it is set to 0.
e. **from** − it is a pointer to structsockaddr for the host where data has to be read.
f. **fromlen** − it is set it to sizeof(structsockaddr).
g. **ssize_t** - this data type is used to represent the sizes of blocks that can be read or written in a single operation. It is similar to size_t, but must be a signed type.

---

10. Close

The close function is used to close the communication between the client and the server.

int close(int sockfd);

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

sockfd − it is a socket descriptor returned by the socket function.

---

11. Shutdown

The shutdown function is used to gracefully close the communication between the client and the server. This function gives more control in comparison to the *close* function.

int shutdown(int sockfd, int how);

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

a. **sockfd** − it is a socket descriptor returned by the socket function.
b. **how** − Put one of the numbers −
   i. 0 − indicates that receiving is not allowed,
   ii. 1 − indicates that sending is not allowed, and
   iii. 2 − indicates that both sending and receiving are not allowed. When 'how' is set to 2, it's the same thing as close().

Sample exercise

Algorithm to implement an UDP Echo Client/Server Communication

Server:

1. Include the necessary header files.
2. Create a socket using socket function with family AF_INET, type as SOCK_DGRAM.
3. Assign the sin_family to AF_INET, sin_addr to INADDR_ANY, sin_port to SERVER_PORT.
4. Bind the local host address to socket using the bind function.
5. Within an infinite loop, receive message from the client using recvfrom function, print it on the console and send (echo) the message back to the client using sendto function.

Client:

1. Include the necessary header files.
2. Create a socket using socket function with family AF_INET, type as SOCK_DGRAM.
3. Initialize the socket parameters. Assign the sin_family to AF_INET, sin_addr to "127.0.0.1", sin_port to dynamically assigned port number.
4. Within an infinite loop, read message from the console and send the message to the server using the sendto function.
5. Receive the echo message using the recvfrom function and print it on the console.

A simple UPD client-server program where a client connects to the server. Server sends a message to the client which is displayed at the client side.

/*******SERVER CODE**********/**

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

main()
{
	int s,r,recb,sntb,x;
	int ca;
	printf("INPUT port number: ");
	scanf("%d", &x);
	socklen_t len;
	struct sockaddr_in server,client;
	char buff[50];

	s=socket(AF_INET,SOCK_DGRAM,0);
	if(s==-1)
	{
		printf("\nSocket creation error.");
		exit(0);
	}
	printf("\nSocket created.");

	server.sin_family=AF_INET;
	server.sin_port=htons(x);
	server.sin_addr.s_addr=htonl(INADDR_ANY);
	len=sizeof(client);
	ca=sizeof(client);

	r=bind(s,(struct sockaddr*)&server,sizeof(server));
	if(r==-1)
	{
		printf("\nBinding error.");
		exit(0);
	}
	printf("\nSocket binded.");
```

```
while(1){

	recb=recvfrom(s,buff,sizeof(buff),0,(struct sockaddr*)&client,&ca);
	if(recb==-1)
	{
		printf("\nMessage Recieving Failed");
		close(s);
		exit(0);
	}

	printf("\nMessage Recieved: ");
	printf("%s", buff);

	if(!strcmp(buff,"stop"))
		break;

	printf("\n\n");
	printf("Type Message: ");
	scanf("%s", buff);

	sntb=sendto(s,buff,sizeof(buff),0,(struct sockaddr*)&client,len);
	if(sntb==-1)
	{
		printf("\nMessage Sending Failed");
		close(s);
		exit(0);
	}

	if(!strcmp(buff,"stop"))
		break;

}
	close(s);
}
```

/*******Client Code**********/**

```
#include<string.h>
#include<arpa/inet.h>
#include<stdlib.h>
#include<stdio.h>
```

```
#include<unistd.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<fcntl.h>
#include<sys/stat.h>

main()
{
	int s,r,recb,sntb,x;
	int sa;
	socklen_t len;
	printf("INPUT port number: ");
	scanf("%d", &x);
	struct sockaddr_in server,client;
	char buff[50];
	s=socket(AF_INET,SOCK_DGRAM,0);
	if(s==-1)
	{
		printf("\nSocket creation error.");
		exit(0);
	}
	printf("\nSocket created.");

	server.sin_family=AF_INET;
	server.sin_port=htons(x);
	server.sin_addr.s_addr=inet_addr("127.0.0.1");
	sa=sizeof(server);
	len=sizeof(server);

while(1){

	printf("\n\n");
	printf("Type Message: ");
	scanf("%s", buff);

	sntb=sendto(s,buff,sizeof(buff),0,(struct sockaddr *)&server, len);
	if(sntb==-1)
	{
		close(s);
```

```
                printf("\nMessage sending Failed");
                exit(0);
        }
        if(!strcmp(buff,"stop"))
                break;

        recb=recvfrom(s,buff,sizeof(buff),0,(struct sockaddr *)&server,&sa);
        if(recb==-1)
        {
                printf("\nMessage Recieving Failed");
                close(s);
                exit(0);
        }

        printf("\nMessage Recieved: ");
        printf("%s", buff);

        if(!strcmp(buff,"stop"))
                break;
}
        close(s);

}
```

TCP Echo Client/Server Communication

An echo client/server program performs the following:

1. The client reads a line of text from the standard input and writes the line to the server using send() function.
2. The server reads the line from the network input using recv() and echoes the line back to the client using send() function.
3. The client reads the echoed line using recv() and prints it on its standard output.

Fig. 1.5 depicts the echo client/server along with the functions used for input and output.

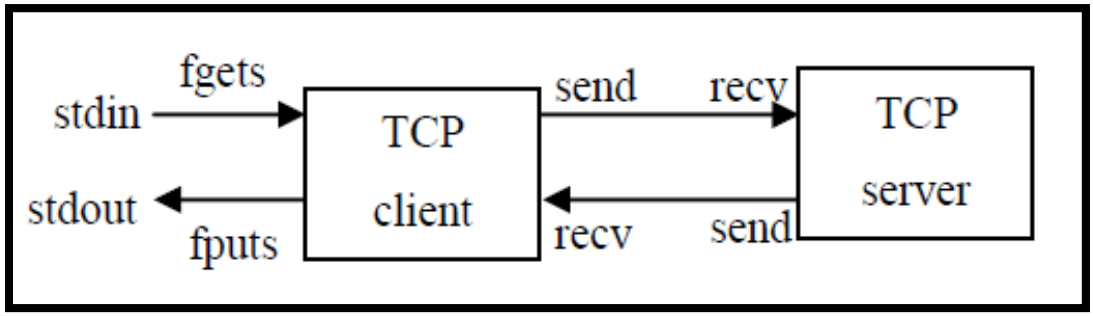

Fig.1.5: Echo client/server

A simple TCP client-server program where a client connects to the server. Server sends a message to the client which is displayed at the client side.

/*******Client Code**********/**

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

main()
{
        int s,r,recb,sntb,x;
        printf("INPUT port number: ");
        scanf("%d", &x);
        struct sockaddr_in server;
        char buff[50];
        s=socket(AF_INET,SOCK_STREAM,0);
        if(s==-1)
        {
                printf("\nSocket creation error.");
                exit(0);
        }
        printf("\nSocket created.");

        server.sin_family=AF_INET;
        server.sin_port=htons(x);
        server.sin_addr.s_addr=inet_addr("127.0.0.1");

        r=connect(s,(struct sockaddr*)&server,sizeof(server));
        if(r==-1)
        {
                printf("\nConnection error.");
                exit(0);
        }
        printf("\nSocket connected.");

        printf("\n\n");
        printf("Type Message: ");
        scanf("%s", buff);
```

```
        sntb=send(s,buff,sizeof(buff),0);
        if(sntb==-1)
        {
                close(s);
                printf("\nMessage Sending Failed");
                exit(0);
        }

        recb=recv(s,buff,sizeof(buff),0);
        if(recb==-1)
        {
                printf("\nMessage Recieving Failed");
                close(s);
                exit(0);
        }

        printf("\nMessage Recieved: ");
        printf("%s", buff);
        printf("\n\n");

        close(s);

}
```

/*******SERVER CODE**********/**

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

main()
{
        int s,r,recb,sntb,x,ns,a=0;
        printf("INPUT port number: ");
        scanf("%d", &x);
        socklen_t len;
        struct sockaddr_in server,client;
        char buff[50];

        s=socket(AF_INET,SOCK_STREAM,0);
        if(s==-1)
        {
```

```
		printf("\nSocket creation error.");
		exit(0);
	}
	printf("\nSocket created.");

	server.sin_family=AF_INET;
	server.sin_port=htons(x);
	server.sin_addr.s_addr=htonl(INADDR_ANY);

	r=bind(s,(struct sockaddr*)&server,sizeof(server));
	if(r==-1)
	{
		printf("\nBinding error.");
		exit(0);
	}
	printf("\nSocket binded.");

	r=listen(s,1);
	if(r==-1)
	{
		close(s);
		exit(0);
	}
	printf("\nSocket listening.");

	len=sizeof(client);

	ns=accept(s,(struct sockaddr*)&client, &len);
	if(ns==-1)
	{
		close(s);
		exit(0);
	}
	printf("\nSocket accepting.");

	recb=recv(ns,buff,sizeof(buff),0);
	if(recb==-1)
	{
		printf("\nMessage Recieving Failed");
		close(s);
		close(ns);
		exit(0);
	}

	printf("\nMessage Recieved: ");
	printf("%s", buff);
```

```
        printf("\n\n");
        scanf("%s", buff);

        sntb=send(ns,buff,sizeof(buff),0);
        if(sntb==-1)
        {
                printf("\nMessage Sending Failed");
                close(s);
                close(ns);
                exit(0);
        }

        close(ns);
        close(s);
}
```

Steps to execute the program

1. Open two terminal windows and open a text file from each terminal with .c extension using command: gedit *filename.c*
2. Type the client and server program in separate text files and save it before exiting the text window.
3. First compile and run the server using commands mentioned below
    a. gcc -o *filename*
    b. ./a.out or ./filename
4. Compile and run the client using the same instructions as listed in 3a & 3b.

Note: The ephemeral port number has to be changed every time the program is executed.

Lab exercises

1. Write two separate C programs (one for server and other for client) using socket APIs for TCP, to implement the client-server model such that the client should send a set of integers along with a choice to search for a number or sort the given set in ascending/descending order or split the given set to odd & even to the server. The server perform the relevant operation according to the choice. Client should continue to send the messages until the user enters selects the choice "exit".
2. Write two separate C programs (one for server and other for client) using UNIX socket APIs for UDP, in which the client accepts a string from the user and sends it to the server. The server will check if the string is palindrome or not and send the result with the length of the string and the number of occurrences of each vowel in the string to the client. The client displays the received data on the client screen. The process repeats until user enter the string "Halt". Then both the processes terminate. (The program should make use of TCP and UDP separately).

Additional exercise

1. Write two separate C programs (one for the Server and the other for Client) using UNIX socket APIs using both connection oriented and connectionless services, in which the server displays the client's socket address, IP address and port number on the server screen.