

# CHAPTER 1

## INTRODUCTION

**CHAT SERVER:** A “SERVER” which monitors the users of a “CHAT ROOM”. When one of the users sends a message to all the participants the server broadcasts the message to all the users. The main function of this type of server is thus reading and broadcasting the messages and responses in a chat room.

Chat server is a standalone application that is made up the combination of two-application, server application (which runs on server side) and client application (which runs on client side). This application is using for chatting in LAN. To start chatting, client must be connected with the server after that its message can broadcast to each and every client.

### 1.1 PURPOSE

As a matter of fact there are several varieties of chatting. The simplest computer chatting is a method of sending, receiving, and storing typed messages with a network of users. This network could be WAN (Wide Area Network) or LAN (Local Area Network). Our chatting system will deal only with LAN's (static IP address) and it is made up of two applications one runs on the server side (any computer on the network you choose it to be the server) while the other is delivered and executed on the client PC. Every time the client wants to chat it runs the client application, enter ip address where the server application is running and start chatting. The system is many-to-many arrangement; every-one is able to “talk” to anyone else. Messages may be broadcasted to all receivers (recipients are automatically notified of incoming messages) or sent to special individuals (private chatting through server). “Windows sockets” is our programming interface to have access to network functionality.

### 1.2 SCOPE

Chat server is an online system developed for the group of members to communicate for the group of members to communicate with each other over the internet. This system solves almost all the limitations of conventional system. Both the students and teachers and the company equally benefited by the proposed system. The system saves a lot of time and effort for both.

This chat program is created to address communication issue for people tied on the phone and sitting in front of their computer. It's very easy to use the company chat program among co-workers. It provides them with an additional way real-time communication in a busy office environment. It can be used internally or over Internet. Chat messages are not being stored on the Chat-Server, but users can save their conversations locally if they choose to do so. All messages are sent in the text format and are not encrypted.

### 1.3 About the project

This particular project is a solution developed to communicate between the users across locally connected network. The concept of sending letters and telegraphs has been reduced due to the new era of Internet Mailing. One such facility is being provided by the Chat Server.

“Chat Server” automates all the aspects stated above related to a communication in a highly secure environment. This project has been developed to receive instant messages and to provide total user satisfaction.

## **CHAPTER 2**

### **SOFTWARE REQUIREMENT SPECIFICATIONS**

#### **2.1 Overall Description:**

Elementary functions are used to write a complete TCP client/server example. For example a chat server performs the following steps:

1. The client reads a line of text from its standard input and writes the line to the server.
2. The server reads the line from its network input and it sends it to the other client.
3. The client reads the input from server and this is displayed to the user at client.



Fig 2.1 Interaction b/w client and server

Fig 2.1 shows interaction between client and server. The two arrows between the client and server, but this is really one full-duplex TCP connection. The `fgets` and `fputs` functions are from the standard I/O library and the `write` and `read` functions.

While developing the chat server, most TCP/IP implementations provide such a server, using both TCP and UDP. Such servers are used with the client.

A client/server that sends input lines is a valid, yet simple, example of a network application. All the basic steps required to implement any client/server are illustrated by this example. To expand this example into an application, there should be change in what the server does with the input it receives from its clients.

Besides running the client and server in their normal mode (type in a line), there is a need to examine lots of boundary conditions, for this example: what happens when the client and server are started; what happens when the client terminates normally; what happens to

the client if the server process terminates before the client is done; what happens to the client if the server host crashes; and so on. By looking at all these scenarios and understanding what happens at the network level, and how this appears to the sockets API, there will be more understanding about what goes on at these levels and how to code applications to handle these scenarios.

### 2.2 Specific Requirements:

The minimum requirement for implementing chat server is described as follows:

The server program should be running in system and the clients on the other system should be able to access the server. For this reliable network (Ethernet LAN) is needed in which server and clients are connected. Each system should have a minimum of 256MB RAM, Pentium processor and 16 bit colour video card.

#### 2.2.1 FUNCTIONALITY

In this project TELNET feature is used. The description of telnet is as follows.

##### TELNET:

**TELNET** is an abbreviation for TEerminaL NETwork. It is standard TCP/IP protocol for virtual terminal service as proposed by International Organization for Standards (ISO). TELNET enables the establishment of a connection to a remote system in such way that the local terminal appears to be at the remote system. TELNET is a general-purpose client/server application program.

When user wants to access an application program or utility located on a remote machine, user performs remote log-in. Telnet client and server programs come into use. The user sends the keystrokes to terminal driver, where the local operating system accepts the characters but does not interpret them. The characters are sent to the telnet client, which transforms the characters to a universal character set called NETWORK VIRTUAL TERMINAL (NVT) characters and delivers to local TCP/IP protocol stack. It is designed to receive characters from a terminal driver.

The main function of the chat server is to receive data from one client and send it to other clients. Initially a socket is created at the server using `socket()` function call. This function will return socket descriptor for each socket. Then at the server `bind()` function is called to bind the wildcard address to the server socket. Then it will call `listen()` function

which will put the socket server in listening state. Then the connection from the clients is accepted using `accept()` function.

Each client using the feature of TELNET, remote logs the server this is done by giving command `telnet <ip address of the server> <port number>`. Now the connection is established between the client and the listening server using three way Handshake. The 3-way handshake is as shown below in fig 2.1.

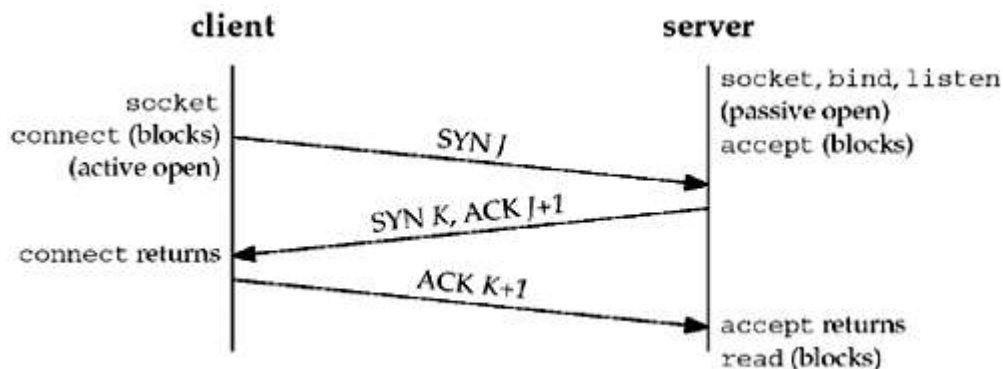


Fig 2.1 The 3-Way handshake between client and server

Since three packets are exchanged between the client and server for connection establishment, this is called 3-way handshake.

The client will send a SYN packet with as sequence number to the server requesting for connection establishment. When the server receives the SYN packet, it will accept the connection using `accept()` function and sends back SYN packet having a sequence number and the acknowledgement. On receiving this packet the client will send an acknowledgement, thus establishing the connection between the client and server.

Once the connection is established between the clients and the server, communication takes place between the clients and the server. The data from the standard input at the client is directly read by the server using `recv()` function. The read data is send to the destination client(or clients) using the `write()` function. The client will read from the network and displays it to the standard output.

Once the connection is finished between the client and the server the connection has to be closed. This is done using the `close()` function call. The 4-way handshake is used to close the connection as shown in the figure 2.2

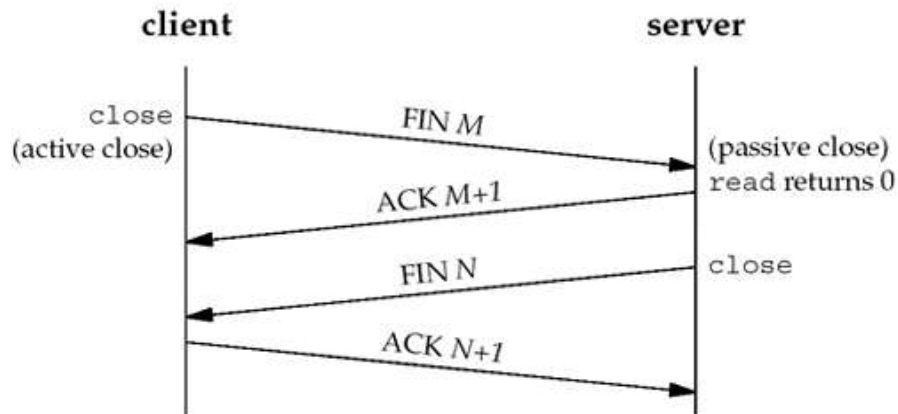


Fig 2.2 The 4-Way handshake between client and server

Since 4 packets are transmitted between the client and the server, this is called 4-way handshake. The termination can be initiated by either the client or server. In the above figure, the client is initiating the termination. The client will send a FIN packet to the server. On receiving this packet the server will send an acknowledgement. This indicates that the client has finished sending the data but it will receive the data from the server. Once server is finished sending data it will send FIN packet to the client. On receiving this packet the client will send an acknowledgement which terminates the connection between the client and the server.

### 2.2.2 DESIGN CONSTRAINTS

Design constraints refer to limitations on the conditions under which the system is being designed or on the requirements of the systems.

One of the design constraints is the underlying network dependencies. The project is dependent on a reliable Ethernet connection, since the Ethernet network is the most common network used. The other design constraints also include network bandwidth. Today the networks come with the bandwidth of several GBps. The project is dependent on network with a minimum of about 10 MBps.

The TCP socket is used instead of the UDP socket because in UDP there is no flow control, hence there are possibility that the server may overwhelm the client hence it is not safe. And however UDP is connectionless and unreliable, hence TCP socket is preferred because of its sliding window protocol, which controls the flow of data between the server and the clients.

The number of client that can simultaneously communicate with the server depends on the backlog value specified in the function listen(). This is used as an argument for backlog function. Generally, a large backlog value cannot be specified. Hence there is a limit in the number of clients that can simultaneously communicate with the server.

### 2.2.3 INTERFACES

Basically, interface is the program or a device which act as a communicating medium between the two computers or programs which want to communicate with each other. This part can be further divided into the following

#### 2.2.3.1 User Interface

This describes the interface which helps the user to communicate with computer. Here the only interface used is the command terminal where the IP address of the server has to be typed to establish the connection between the client and the server.

Here the terminal is required on both the client and the server computers. The terminal on the client's side will ask for the IP address of the server, and read from standard input and puts to the network. Once the server receives the text it will read from the network and send to other clients terminal.

On the server side, it will display whether the binding is done successfully or not and if done correctly, it will show appropriate message and will show the status of transferring of the text. After completion an appropriate message will be displayed.

### 2.2.3.2 Hardware Interface

This describes the hardware which is used to read or write the data or text which is being transferred over.

Here, the only hardware used is the keyboard to type the message or text or data at the client. This data is transferred over the network to the server which it sends to other clients and displayed on the clients terminals.

### 2.2.3.3 Software Interface

This part will describe the software which will be used by the project. Since the project is a simple client-server program, it does not require any extra software which might have to be installed.

The operating system used is FEDORA 10 which will have all the inbuilt libraries and the headers which are used in the program. VI editor, which is inbuilt in FEDORA 10, is used to code the project.

### 2.2.3.4 Communicate interface

This describes the medium used for the communication purposes between the modules of the program.

Here, sockets are used for the inter process communication. To perform the network I/O, the first thing a process must do is call the 'socket' function, specifying the type of communication protocol desired.

The prototype of the socket function is as follows:

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

Here, 'family' specifies the protocol family and is one of the constants which is predefined. This argument is often referred to as domain instead of family. The socket type is one of the constants which is also predefined. The 'protocol' argument to the 'socket' function should be set to the specific protocol type which is also predefined or 0 to select the system's default for the given combination of family and type. Not all combination of family of family and type are valid.



First 'connect' function will be called to establish the connection. After establishing the connection the 'bind' function will be called which assigns the local protocol address to a socket. Then 'listen' will be called to accept the incoming request directed to the socket and also specifies the number of connection the kernel should queue for this socket, then 'accept' function will be called to return the next completed connection from the front of the completed connection queue.

## **CHAPTER 3**

### **DETAILED DESIGN**

This section will give the detailed design of the chat server. This section will describe all the modules used, their purpose and their responsibility. Finally this section will describe the inputs and the outputs.

### **3.1 MODULES**

#### **3.1.1 INTRODUCTION**

We have implemented the entire project as one single program. The program will run in the server and the clients access it via the network. So we are using only one module, i.e. server module.

##### **Server Module**

Function of the server module is to receive the data from one client and send it to the respective client (private message) or to all clients (broadcast). In the beginning a socket is created at the server called server socket. The socket() function is used to create the socket. This function will return a socket descriptor to the socket. The wildcard address and the port number 7989 are bound to the server using the bind function. The listenfd, server socket descriptor is added to the master set. Now the listen() function is called to accept any client connections. Now the server is said to be in listening state. Whenever a client connection comes, server calls accept() function to accept the connection and adds that client to the master set. This process of accepting connections occurs in loop, so that it can accept many connections.

Once the connections are established, data from the clients are handled in round robin fashion. After receiving the data from a client, the server observes the data and takes necessary action like, sending to specific client or to broadcast it. In the mean time the server keeps about all client connections it as accepted by adding them to master set and removing the clients from the master set which have disconnected.

The client get connected to server by giving command telnet <server ip address> 7989. The client can disconnect from server by closing telnet connection. The user at the server can stop the process at any time, when all the clients connection gets disconnected.

Some of the important functions used in this module are as follows.

### **socket() function**

This function is used in both the client and the server, to create sockets before establishing the connection and it returns the socket descriptor of the created socket.

The prototype of this function is as follows

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

The argument family specifies the protocol family and it can take the value such as AF\_INET for IPV4 and AF\_INET6 for IPV6 etc.

The argument type specifies the type of the socket i.e. whether it is a stream socket or raw socket or datagram socket.

The argument protocol gives the type of protocol we are using and it is normally set to zero except for raw socket.

### **bind() function**

This function is used to bind the local protocol address to the socket. This is called at the server side. The prototype of this function is as follows.

**int bind(int sockfd, const struct sockaddr \* myaddr, socklen\_t addrlen);**

The first argument is the socket descriptor. The second argument is a pointer to the server's socket address structure and the third argument is the size of this structure. It returns 0 on success and -1 on error.

### **listen() function**

This function is called only by the server and will turn the socket into listening socket which will accept the connection from the client. The prototype of this function is as follows

**int listen(int sockfd, int backlog);**

The first argument is a socket descriptor. The second argument for this function specifies the maximum number of connections that the kernel should queue for this socket. It returns 0 on success and -1 on error.

### **accept() function**

This function is called by the server and it is when this function is called the connection is established between the client and the server. The prototype of this function is as follows

**int accept(int sockfd, struct sockaddr \*cliaddr, socklen\_t \*addrlen);**

The first argument is a socket descriptor. The second argument is a pointer to the socket address structure of the client and the third argument is a pointer to the size of this structure. On success this function will return the socket descriptor of the connected socket.

### **close() function**

This function is called by the server to terminate the connection. The prototype of this function is as follows.

**int close(int sockfd);**

The argument to this function is the socket descriptor of the socket for which the connection is closed. On success it will return 0 and on error it will return -1.

### **Select () function:**

This function allows the process to instruct the kernel to wait for any one of multiple events to occur and to wake up the process only when one or more of these events occurs or when a specified amount of time has passed. The prototype of this function is as follows.

```
int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset, const
struct timeval *timeout);
```

The maxfdp1 argument gives maximum descriptor value plus 1.

The readset, writeset and exceptset arguments are pointers to read, write and except descriptors set respectively.

The timeout value of type struct timeval tells the kernel how long to wait for one of the specified descriptors to become ready.

### Basic Buffer Functions

#### recv() function

This is used by the server to receive the data either from network or from the standard input. The prototype of the function is as follows.

```
ssize_t recv(int sockfd, void *buff, size_t nbytes, int flags);
```

It will receive data from socket described by sockfd and put it into buff. The third argument will give the no. of bytes the buff can store and this function will return the no of bytes received. The flags value specified will be 0 in our program.

#### write() function

This is used by the server to either write to the network or to the standard output. The prototype of the function is as follows.

```
int write(char buf1, char buf2, int sizeof(buf1), 0);
```

It will write from buf1 to buf2. The third argument gives the no. of bytes to be written to buf2. The function will return the number of bytes successfully written to buf2.

### 3.1.2 PURPOSE

For a chat server to work properly the functions described above just now should be implemented properly. They are the basic functions for a chat server to work. If these functions are not implemented properly the chat server may not run properly.

In addition to the above functions some inbuilt functions are used to handle the error conditions. Without these functions, it becomes difficult to debug any errors which occur during runtime. So proper use of these functions should be done.

### 3.1.3 FUNCTIONALITY (FLOWCHART)

The flowchart showing the functions called at the server and the client and the flow of data between them is as shown in fig 3.1.

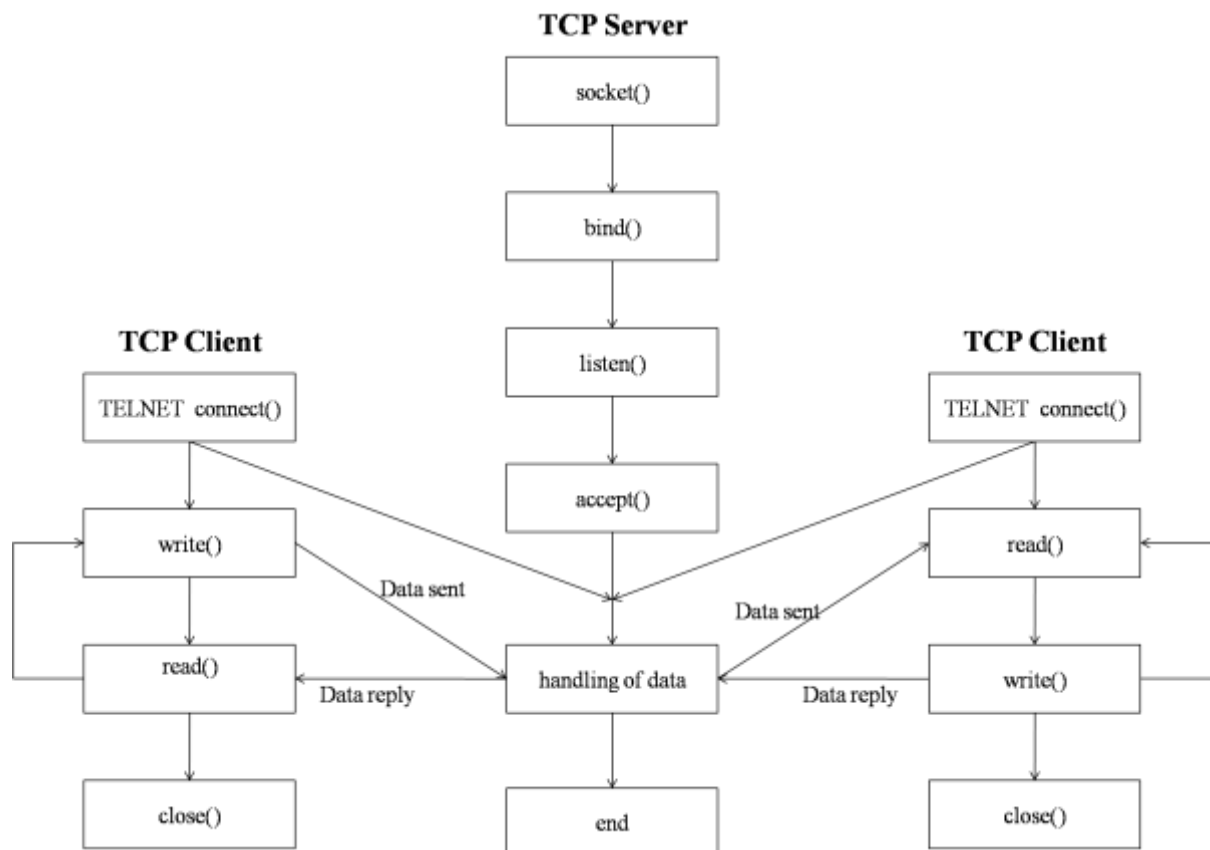


Fig 3.1 Interaction between client and server

### CHAPTER 4

### IMPLEMENTATION

In this section implementation details of the chat server is described. The startup and the termination of the chat server is also described.

#### Normal Startup

The server is started in the background on the host Linux. When the server starts it will call the functions `socket()`, `bind()`, `listen()` and `accept()`, blocking in the call to `accept()`. The socket will be now in the listening state with wildcard for IP address and a port number of 7989.

The client on the same host or different host, specifies `telnet <server ip address> 7989` to get connected to the server. This connection takes place using 3-way handshake.

After the connection is established, the following steps takes place.

- Whenever the user at client types a line in the terminal, it will be read by server using `recv()` function.
- The server looks into the data read and sends data to specified client using `write()` function. The server repeats the process with all clients.
- The data sent by server is directly displayed on the terminal at the client.

#### Termination

At this point, the connection is established and the data is being exchanged between the clients via the server. Now the termination involves the following steps:

- When the client types `[Ctrl+]` and then types `close`, the telnet connection between client and server gets closed which then initiates the TCP termination process. Now the client is no more reachable from server.
- The termination can be even initiated by the server. When the server process is terminated or stopped, all the connections of clients with server gets terminated using normal TCP termination process.

## **CHAPTER 5**

### **TESTING**

Testing is vital to the success of the system. System testing makes a logical assumption that if all parts of the system are correct, the goal will be successfully achieved. In the testing process the actual system in an organization is tested and gather errors from the new system operates in full efficiency as stated. System testing is the stage of implementation, which is aimed to ensuring that the system works accurately and efficiently.

The main objective of testing is to uncover errors from the system. For the uncovering process proper input data should be given to the system. It is important to give correct inputs to efficient testing.

Testing is done for each module. After testing all the modules, the modules are integrated and testing of the final system is done with the test data, specially designed to show that the system will operate successfully in all its aspects conditions. Thus the system testing is a confirmation that all is correct and an opportunity to show the user that the system works.

#### **5.1 UNIT TESTING**

Unit testing verification efforts on the smallest unit of software design, module. This is known as “Module Testing”. The modules are tested separately. This testing is carried out during programming stage itself. In these testing steps, each module is found to be working satisfactorily as regard to the expected output from the module.

Each module is tested for any errors and checked using the loopback address whether they are working properly.

#### **5.2 INTEGRATION TESTING**

This is the process of testing of the whole software after the modules are integrated to make the complete software. Checking is done using both loopback address where the client and the server are run on the same system and in the case where the client and the server are run on two different system.



### **5.3 INTERFACE TESTING**

The interface used for this chat server is the sockets for interprocess communication. Initially socket creation is checked for correctness. If there are any errors in the socket creation suitable message is printed to the terminal.

## **CHAPTER 6**

### **CONCLUSION**

#### **6.1 SUMMARY**

This project was an effort at learning the details on the communication of the systems over the computer networks. Going through this project, effective ways of connecting the computers using the inter process communications process like sockets is understood.

Although a lot of effort has been put in building the project, there might be deficiencies. To the best of our knowledge this system takes care of most of the conditions that may occur at the run time environment.

To make the best of the project the user has to work on the system and to train himself which would help the user in better understanding of the system and in turn increases the user efficiencies and skills in using the system

This project is developed such that anybody who has a basic knowledge about the computer system and networks can easily manage. There might be minor changes in the implementation with respect to the proposed system.

#### **6.2 LIMITATIONS**

- It uses only TCP protocol.
- There is a limit on the number of multiple clients that can simultaneously communicate with the server. It is typically 31 clients in this program.
- It can be run only on those systems which are connected by the local network. That is it involves only static IP address.
- Whenever a client is typing any text, if there is any data from server, it is displayed on the same text field.

#### **6.3 FURTHER ENHANCEMENT**

- The chat server designed works with the TCP protocol. It can be made to work with the UDP protocol
- The number of simultaneous clients that the server can handle can be increased.
- A suitable front end can be added so that user interface becomes more convenient.

## **CHAPTER 7**

### **REFERENCES**

#### **Books referred**

- [1]. Unix Network Programming, W Richard Stevens, Bill Fenner, Andrew M Rudoff.
- [2]. Computer Networks, 4<sup>th</sup> Edition, Andrew S. Tanenbaum
- [3]. Unix system Programming, Terrance Chan.

**CHAPTER 8****APPENDICES****Appendix A: Source Code Listing**

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

#define PORT 1503          // port we're listening on
#define SHOW 1             // command show
#define MESSAGE 2
#define HELP 3
#define NICK 4
#define NEWLINE 5
#define VALIDATE 6
void write_to_socket (int target, char *buff_to_send, int len);
int check_for_command (char *buffer, int *p, int);
int is_fd_valid (int destination);
void type_writer (char *buf, int len, int fd);

typedef struct
{
    int id;
    char name[20];
    int vlid;          /* socket descriptor */
} USER;
USER user[20];
int usercount = 4;

int
main (void)
{
    fd_set master;      // master file descriptor list
    fd_set read_fds;    // temp file descriptor list for select()
    struct sockaddr_in myaddr;    // server address
    struct sockaddr_in remoteaddr; // client address
    int fdmax;          // maximum file descriptor number
    int listener;       // listening socket descriptor
    int newfd;          // newly accept()ed socket descriptor
    char buf[1024], buf2[20]; // buffer for client data
    int nbytes;
    int yes = 1;        // for setsockopt() SO_REUSEADDR, below
    int addrlen;
    int i, j, k, tmp, destination = 0, l;
    int str_len;
    char c;
```

```
char *tmp_ptr;

bzero (user, sizeof (USER));

FD_ZERO (&master);          // clear the master and temp sets
FD_ZERO (&read_fds);

// get the listener
if ((listener = socket (AF_INET, SOCK_STREAM, 0)) == -1)
{
    //perror ("socket");
    exit (1);
}

// lose the "address already in use" error message
if (setsockopt (listener, SOL_SOCKET, SO_REUSEADDR, &yes,
    sizeof (int)) == -1)
{
    //perror ("setsockopt");
    exit (1);
}
// bind
myaddr.sin_family = AF_INET;
myaddr.sin_addr.s_addr = INADDR_ANY;
myaddr.sin_port = htons (PORT);
memset (&(myaddr.sin_zero), '\0', 8);
if (bind (listener, (struct sockaddr *) &myaddr, sizeof (myaddr))
== -1)
{
    //perror ("bind");
    exit (1);
}
// listen
if (listen (listener, 10) == -1)
{
    //perror ("listen");
    exit (1);
}

// add the listener to the master set
FD_SET (listener, &master);

// keep track of the biggest file descriptor
fdmax = listener;          // so far, it's this one

// main loop
for (;;)
{
    read_fds = master;      // copy it
    if (select (fdmax + 1, &read_fds, NULL, NULL, NULL) == -1)
    {
        //perror ("select");
        exit (1);
    }
}
```

---

---

**DEPT OF CSE, RVCE**                  **JAN – MAY 2012**                  **PAGE 22**

```
        "\t\t~ Type /numeric-user/ to send a message
to particular user only\n\n"
        "\t\t ex: /6/ Hi number 6 how r u\n\n"
        "\t\t~ Type /HELP/ to see server
commands\n\n\n"

"*****
*****\n\n\n\n>>");
        str_len = strlen (buf);
        write_to_socket (newfd, buf, str_len);
        bzero (buf, sizeof (buf));
        strcpy (buf, "enter username\n");
        str_len = strlen (buf);
        write_to_socket (newfd, buf, str_len);
        //strcpy(buff, "\tWelecome to chat server\n");
    }
}
else
{
    // handle data from a client
    bzero (buf, sizeof (buf));
    if ((nbytes = recv (i, buf, sizeof (buf), 0)) <= 0)
    {
        // got error or connection closed by client
        if (nbytes == 0)
        {
            // connection closed
            printf ("selectserver: socket %d hung up\n", i);
            user[i].id = 0;
            --usercount;
        }
        else
        {
            //perror ("recv");
        }
        close (i); // bye!
        FD_CLR (i, &master); // remove from master set
    }
    else
    {
        // we got some data from a client

        for (j = 0; j <= fdmax; j++)
        {
            // send to everyone!
            if (FD_ISSET (j, &master))
            {
                // except the listener and ourselves
                if (j != listener && j != i)
                {
                    switch (check_for_command
                            (buf, &destination, i))
                    {
                        case SHOW:

                            k = usercount - 1;
```

```
bzero (buf, sizeof (buf));
sprintf (buf, ">>");
//str_len=strlen(buf);
write_to_socket (i, buf, strlen (buf));
bzero (buf, sizeof (buf));
while (k >= 4)
{
    sprintf (buf, "<%s-%d>\n>>",
            user[k].name, user[k].id);
    printf ("%s\n", user[k].name);

    str_len = strlen (buf);
    write_to_socket (i, buf, 12);
    bzero (buf, sizeof (buf));
    --k;
}
j = 1111; // to end the loop
break;
case VALIDATE:
    //strcpy(user[i].name,buf);
    user[i].valid = 1;
    printf ("%s\n", user[i].name);
    bzero (buf, sizeof (buf));
    sprintf (buf, ">>");
    //str_len=strlen(buf);
    write_to_socket (i, buf, strlen (buf));
    j = 1111;
    break;
case MESSAGE:
    if (is_fd_valid (destination))
    {
        strcpy (buf2, user[i].name);
        strcat (buf2, ">>");
        strcat (buf2, buf + 3);
        str_len = strlen (buf2);
        write_to_socket (destination, buf2,
                        str_len);
    }
    else
    {
        bzero (buf, sizeof (buf));
        strcpy (buf,
            "Invalid user\n>>\r\n");
        str_len = strlen (buf);
        write_to_socket (i, buf, str_len);
    }
    j = 1111; // to end the loop
    break;

case HELP:
    bzero (buf, sizeof (buf));
    strcpy (buf,
```



---

```

        str_len = strlen (buf);
        write_to_socket (i, buf, str_len);
        j = 1111;
        break;
    case NEWLINE:
        bzero (buf, sizeof (buf));
        sprintf (buf, ">>");
        //str_len=strlen(buf);
        write_to_socket (i, buf, strlen (buf));
        j = 1111;
        break;
    default:
        bzero (buf2, sizeof (buf2));
        strcat (buf2, user[i].name);
        strcat (buf2, ">>");
        strcat (buf2, buf);
        write_to_socket (j, buf2, nbytes + 6);
        break;
    }
    //}
}
}
}
}
}
}
}
}
}
}
return 0;
}

```

```
        exit (0);
    }
}

int
check_for_command (char *buffer, int *p, int i)
{
    char tmp[6];
    strncpy (tmp, buffer, 5);
    while (*buffer == ' ')
        ++buffer;
    if (user[i].valid == 0)
    {
        strncpy (user[i].name, buffer, 4);
        user[i].name[4] = '\0';

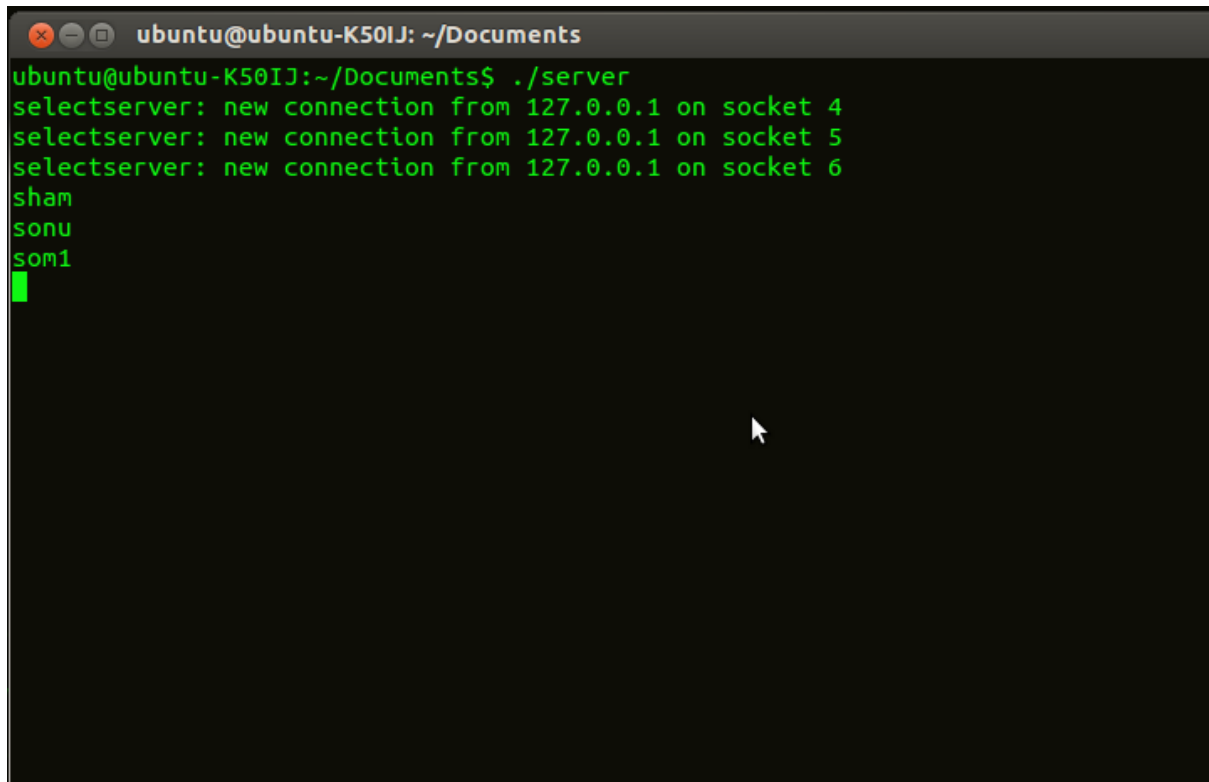
        return VALIDATE;
    }
    if (!strcmp (buffer, "/SHOW/", 6) || !strcmp (buffer, "/show/",
6))
        return SHOW;
    else if (!strcmp (buffer, "/HELP/", 6) || !strcmp (buffer,
"/help/", 6))
        return HELP;
    else if ((*buffer == '/')
        && ((*buffer + 2) == '/') || ((*buffer + 3) == '/'))
    {
        *p = *(buffer + 1) - '0';
        if (*(buffer + 3) == '/')
            *p = (*p) * 10 + *(buffer + 2);

        return MESSAGE;
    }
    else if (*buffer == '\r')
        return NEWLINE;

    else
        return 0;
}

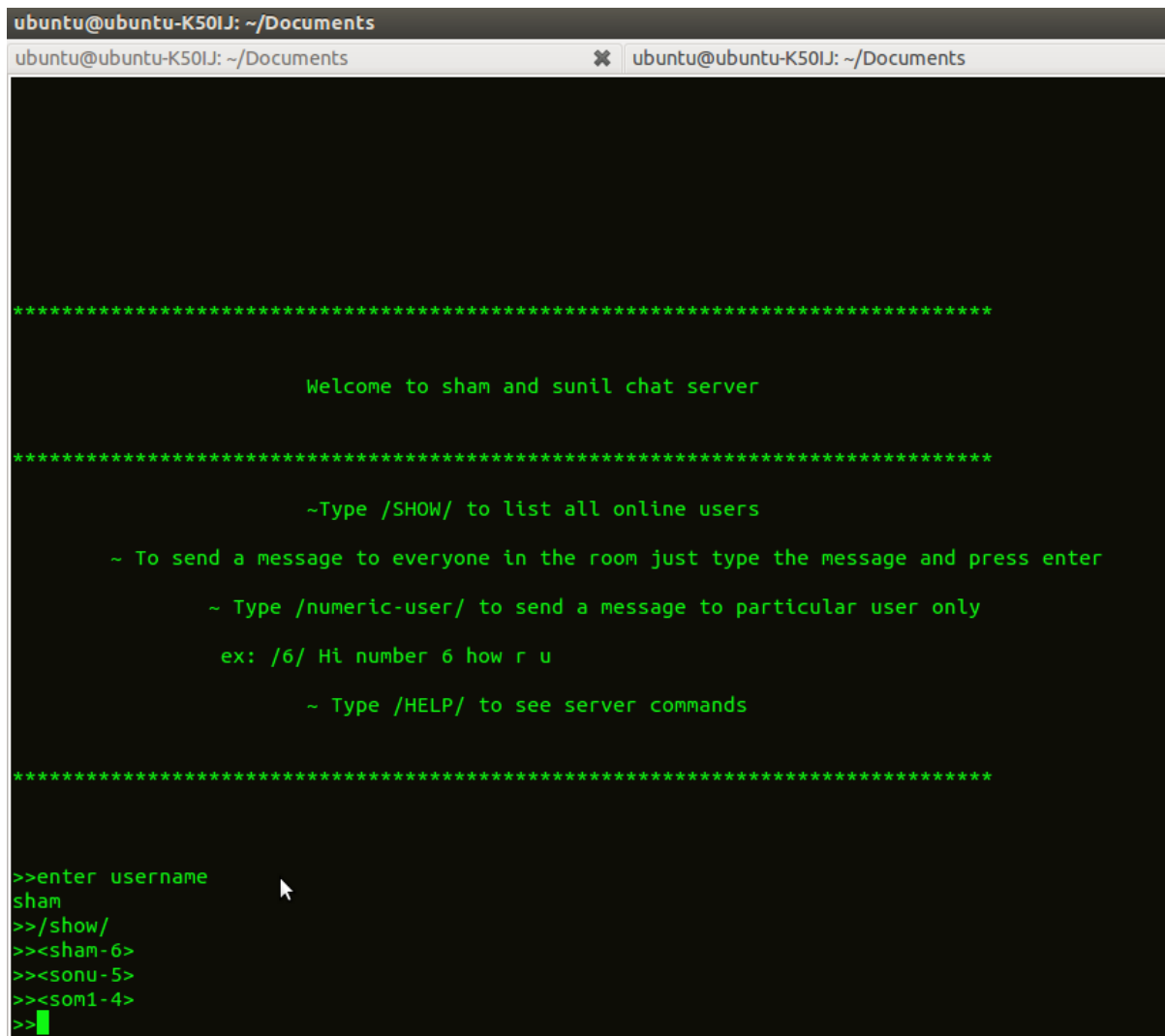
int
is_fd_valid (int destination)
{
    int i;
    for (i = 0; i < usercount; ++i)
        if (user[i].id == destination)
            return 1;
    return 0;
}
```

### Appendix B: Snapshots

A terminal window titled 'ubuntu@ubuntu-K50IJ: ~/Documents' with a dark background and green text. The user has executed './server', resulting in three lines of output: 'selectserver: new connection from 127.0.0.1 on socket 4', 'selectserver: new connection from 127.0.0.1 on socket 5', and 'selectserver: new connection from 127.0.0.1 on socket 6'. Below these, the names 'sham', 'sonu', and 'som1' are printed on separate lines. A green cursor is visible on the line following 'som1'.

```
ubuntu@ubuntu-K50IJ: ~/Documents
ubuntu@ubuntu-K50IJ:~/Documents$ ./server
selectserver: new connection from 127.0.0.1 on socket 4
selectserver: new connection from 127.0.0.1 on socket 5
selectserver: new connection from 127.0.0.1 on socket 6
sham
sonu
som1
█
```

Fig 8.1 The server when executed.



```
ubuntu@ubuntu-K50IJ: ~/Documents
ubuntu@ubuntu-K50IJ: ~/Documents  ✕  ubuntu@ubuntu-K50IJ: ~/Documents

*****

Welcome to sham and sunil chat server

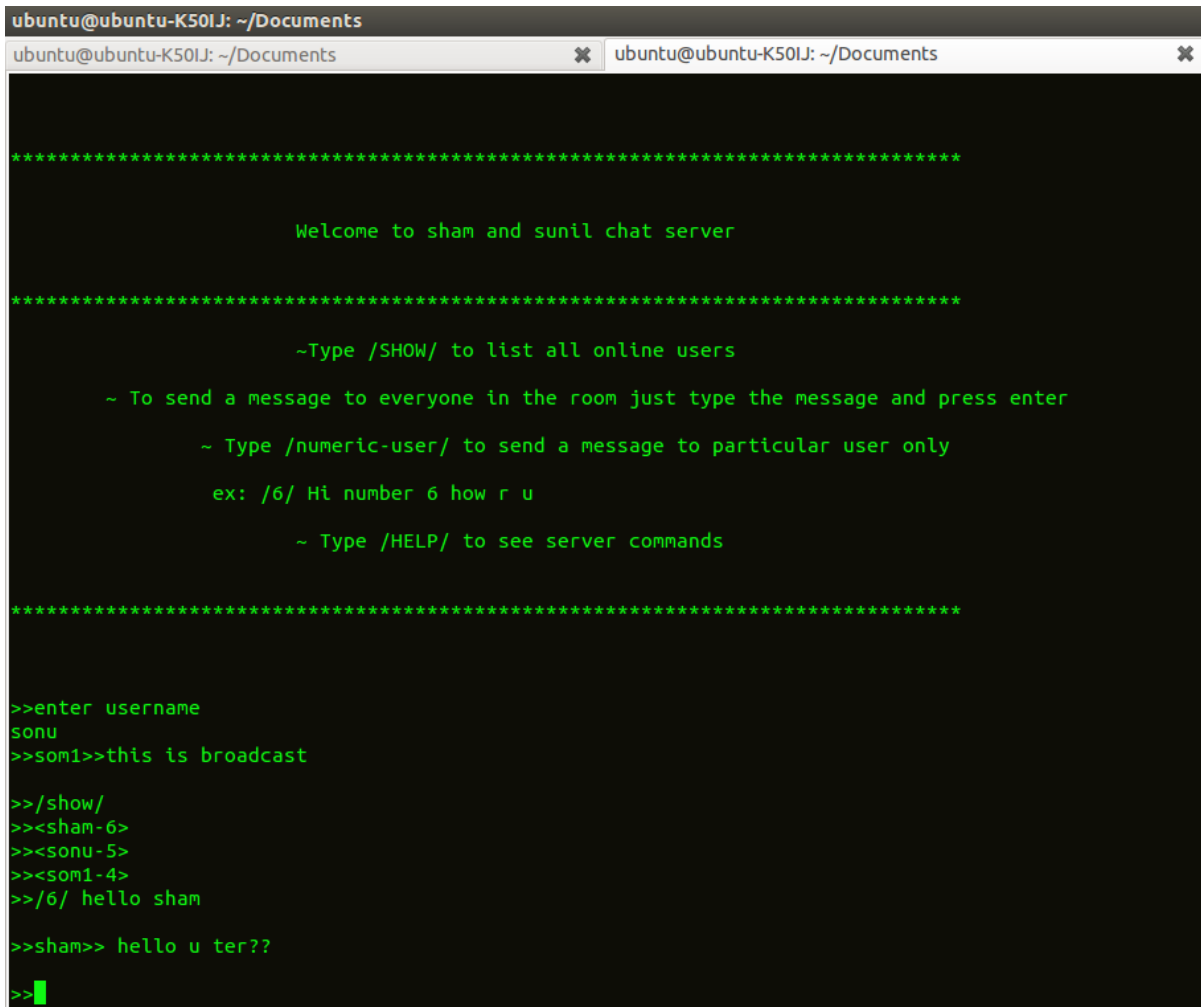
*****

~Type /SHOW/ to list all online users
~ To send a message to everyone in the room just type the message and press enter
~ Type /numeric-user/ to send a message to particular user only
ex: /6/ Hi number 6 how r u
~ Type /HELP/ to see server commands

*****

>>enter username
sham
>>/show/
>><sham-6>
>><sonu-5>
>><som1-4>
>>
```

Fig 8.2 The client window when it is connected to a client, when chatting.



```
ubuntu@ubuntu-K50IJ: ~/Documents
ubuntu@ubuntu-K50IJ: ~/Documents  x  ubuntu@ubuntu-K50IJ: ~/Documents  x

*****

Welcome to sham and sunil chat server

*****

~Type /SHOW/ to list all online users
~ To send a message to everyone in the room just type the message and press enter
~ Type /numeric-user/ to send a message to particular user only
ex: /6/ Hi number 6 how r u
~ Type /HELP/ to see server commands

*****

>>enter username
sonu
>>som1>>this is broadcast

>>/show/
>><sham-6>
>><sonu-5>
>><som1-4>
>>/6/ hello sham

>>sham>> hello u ter??

>> 
```

Fig 8.3 The client window when it is connected to other client, when chatting.