

B.M.S College of Engineering

**P.O. Box No.: 1908 Bull Temple Road,
Bangalore-560 019**

DEPARTMENT OF INFORMATION SCIENCE & ENGINEERING



Course – Unix System Programming

Course Code – 19IS4PWUSP

Academic Year – 2020-21

Report on Unix System Programming Project

MULTI CHAT SYSTEM USING SOCKET PROGRAMMING

Submitted by

Ayman Shakil – 1BM19IS192

Mahan Khanal – 1BM19IS200

Nabeel Saleem – 1BM19IS202

Submitted to
Sandeep Verma N

B.M.S College of Engineering

P.O. Box No.: 1908 Bull Temple Road,
Bangalore-560 019

DEPARTMENT OF INFORMATION SCIENCE & ENGINEERING



CERTIFICATE

Certified that the Project has been successfully presented at **B.M.S College Of Engineering** by **Ayman Shakil, Mahan Khanal, Nabeel Saleem** bearing USN: **1BM19IS192, 1BM19IS200 and 1BM19IS202** in partial fulfilment of the requirements for the IV Semester degree in **Bachelor of Engineering in Information Science & Engineering** of **Visvesvaraya Technological University, Belgaum** as a part of project for the course **Unix System Programming (19IS4PWUSP)** during academic year 2020-2021.

Faculty Name – Sandeep Verma N
Designation – Assistant Professor
Department of ISE, BMSCE

TABLE OF CONTENTS

S.NO	TOPIC	PAGE
1	Abstract	4
2	Introduction	5
3	Block Diagram	6
4	Problem Statement	7
5	API used in the project	7
6	Explanation about the API's	7
7	Implementation / code	11
8	Results/Snapshots	26

ABSTRACT

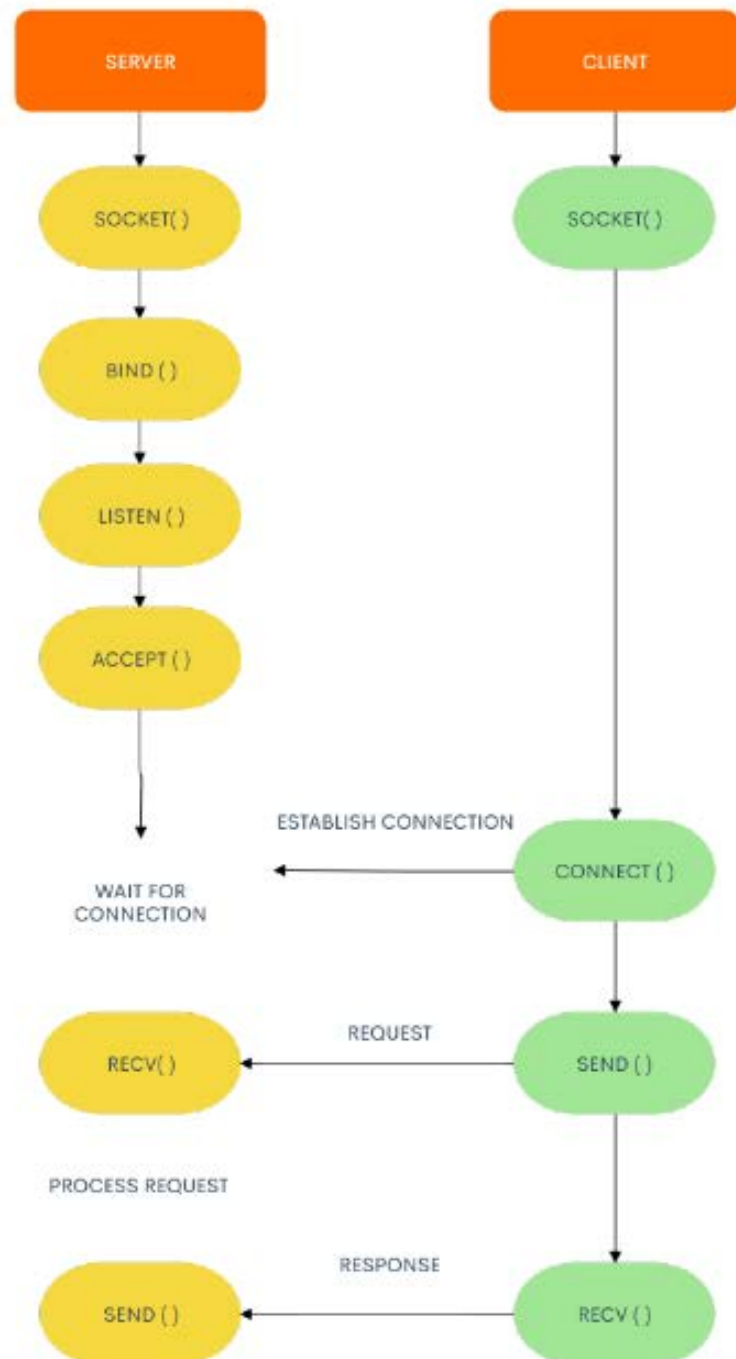
The project Multi Chat system is a client server chat application which is implemented using the concepts of socket Programming and is written in C language. A socket is a communications connection point (endpoint) that you can name and address in a network. Socket programming shows how to use socket APIs to establish communication links between remote and local processes. A chat application should allow both sending and receiving process in simultaneous way. This is achieved in this application with multi-threading concept. Another important feature in chat application is group chat which is implemented in this application. It allows people to chat. Message will be sent to all the users in chat room along with the name of the user who has sent the message Users who are available in the chat room will receive the message.

INTRODUCTION

The chatting application has huge impact on day to day life. There are numerous chatting application available in this world. Each application has different additional features varying from other applications. These application organizations compete with each other and add some competing features during each release. They have reached people much and have an impact on people's life. People find a better application from an available internet application which they feel much reliable and secure. Some of the available chatting applications that are available in these days are WhatsApp, Facebook, Instagram, Hike, etc...The above mentioned applications have billion users all over the world. Those companies are one of the top companies in the world. They have higher revenue per year and have many employees for their organizations developing additional features to compete with other organizations during each release. These applications have different features and follows different ways to ensure security of their user data. Today a data theft is the major crime and most people are involved in it. There are many cases being filed these days about personal data loss. So the organizations have to ensure the security from data loss by the third party data crisis. The basic chatting system should involve both sending and receiving processes simultaneously. In this application both sending and receiving messages simultaneously happens through multi-threading concept.

BLOCK DIAGRAM

The system architecture / block diagram of the events that happen between the server and the client is shown below



PROBLEM STATEMENT

Private organizations aim at having separate chat system to communicate with people and to share resources securely to them. Sharing resources through other public applications are not much reliable and secure, there are several restrictions in public applications because public applications are easily affected by network threats, etc. This application extends one-way messaging to multi way communication minimizing risks of network threats, data security and time complexity.

API'S USED IN THE PROJECT

An application programming interface is a connection between computers or between computer programs. It is a type of software interface, offering a service to other pieces of software. A document or standard that describes how to build such a connection or interface is called an API specification.

The API's used in our project are :

- SOCKET API
- PTHREAD API

EXPLANATION OF API'S USED IN THE PROJECT

Socket API

The socket API is a collection of socket calls that enable you to perform the following primary communication functions between application programs: Set up and establish connections to other users on the network. Send and receive data to and from other users. Close down connections.

The socket() API creates an endpoint for communications and returns a socket descriptor that represents the endpoint. ... The client application uses a connect() API on a stream socket to establish a connection to the server. The server application uses the accept() API to accept a client connection request.

Socket creation:

- `int sockfd = socket(domain, type, protocol)`

The parameters are defined as follows:

sockfd: socket descriptor, an integer (like a file-handle)

domain: integer, communication domain

type: communication type

protocol: Protocol value for Internet Protocol(IP), which is 0. This is the same number which appears on protocol field in the IP header of a packet.(man protocols for more details)

System calls used in our project

- **bind()** - The **bind()** system call associates an address with the socket descriptor. The second parameter is a pointer to the socket address structure, which is generalized for different protocols. The **sockaddr** structure is defined in `<sys/socket.h>`.
- **listen()** - The **listen()** system call prepares a connection-oriented server to accept client connections.
- **accept()** - The **accept()** system call is used with connection-based socket types (**SOCK_STREAM**, **SOCK_SEQPACKET**). It extracts the first connection request on the queue of pending connections for the listening socket, **sockfd**, creates a new connected socket, and returns a new file descriptor referring to that socket.
- **recv()** – The **recv()** system calls are used to receive messages from a socket.
- **send()** – The **send()** system call is used to transmit a message to another socket. The **send()** call may be used only when the socket is in a connected state (so that the intended recipient is known).
- **connect()** - The **connect()** system call connects the socket referred to by the file descriptor **sockfd** to the address specified by **addr**.

Pthread API

In a Unix/Linux operating system, the C/C++ languages provide the POSIX thread(pthread) standard API(Application program Interface) for all thread related functions. It allows us to create multiple threads for concurrent process flow. Pthreads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library - though this library may be part of another library, such as libc, in some implementations.

In a Unix/Linux operating system, the C/C++ languages provide the POSIX thread(pthread) standard API(Application program Interface) for all thread related functions. It allows us to create multiple threads for concurrent process flow. It is most effective on multiprocessor or multi-core systems where threads can be

implemented on a kernel-level for achieving the speed of execution. Gains can also be found in uni-processor systems by exploiting the latency in IO or other system functions that may halt a process.

We must include the pthread.h header file at the beginning of the script to use all the functions of the pthreads library. To execute the c file, we have to use the -pthread or -lpthread in the command line while compiling the file.

- cc -pthread file.c or
- cc -lpthread file.c

The functions defined in the pthreads library include:

- a. **pthread_create**: used to create a new thread

Syntax:

```
int pthread_create(pthread_t * thread,  
                  const pthread_attr_t * attr,  
                  void * (*start_routine)(void *),  
                  void *arg);
```

Parameters:

- thread: pointer to an unsigned integer value that returns the thread id of the thread created.
- attr: pointer to a structure that is used to define thread attributes like detached state, scheduling policy, stack address, etc. Set to NULL for default thread attributes.
- start_routine: pointer to a subroutine that is executed by the thread. The return type and parameter type of the subroutine must be of type void *. The function has a single attribute but if multiple values need to be passed to the function, a struct must be used.
- arg: pointer to void that contains the arguments to the function defined in the earlier argument

- b. **pthread_exit**: used to terminate a thread

Syntax:

- void pthread_exit(void *retval);

Parameters: This method accepts a mandatory parameter retval which is the pointer to an integer that stores the return status of the thread terminated. The scope of this variable must be global so that any thread waiting to join this thread may read the return status.

- c. **pthread_join**: used to wait for the termination of a thread.

Syntax:

- int pthread_join(pthread_t th, void **thread_return);

Parameter: This method accepts following parameters:

- **th:** thread id of the thread for which the current thread waits.
- **thread_return:** pointer to the location where the exit status of the thread mentioned in th is stored.

d. **pthread_self:** used to get the thread id of the current thread.

Syntax:

- `pthread_t pthread_self(void);`

e. **pthread_equal:** compares whether two threads are the same or not. If the two threads are equal, the function returns a non-zero value otherwise zero.

Syntax:

- `int pthread_equal(pthread_t t1, pthread_t t2);`

Parameters: This method accepts following parameters:

- **t1:** the thread id of the first thread
- **t2:** the thread id of the second thread

f. **pthread_cancel:** used to send a cancellation request to a thread

Syntax:

- `int pthread_cancel(pthread_t thread);`

Parameter: This method accepts a mandatory parameter thread which is the thread id of the thread to which cancel request is sent.

g. **pthread_detach:** used to detach a thread. A detached thread does not require a thread to join on terminating. The resources of the thread are automatically released after terminating if the thread is detached.

Syntax:

- `int pthread_detach(pthread_t thread);`

Parameter: This method accepts a mandatory parameter thread which is the thread id of the thread that must be detached.

IMPLEMENTATION/CODE

A server may be a computer dedicated to running a server application. Organizations have dedicated computer for server application which has to be maintained periodically and has to be monitored continuously for traffic loads would never let them go down which affects the company's revenue. Most organizations have a separate monitoring system to keep an eye over their server so that they can find their server downtime before its clients. These server computers accept clients over network connections that are requested. The server responds back by sending responses being requested. There are many different server applications that vary based on their dedicated work. Some are involved for accepting requests and performing all dedicated works like business application servers while others are just to bypass the request like a proxy server. These server computers must have a faster Central processing unit, faster and more plentiful RAM, and bigger hard disc drive. More obvious distinctions include redundancy in power supplies, network connections, and RAID also as Modular design.

A client is a software application code or a system that requests another application that is running on dedicated machine called Server. These clients need not be connected to the server through wired communication. Wireless communication takes place in this process. Client with a network connection can send a request to the server.

Server and Client connection

- A static Server socket is created in beginning which is then bind with host and port .
- After server instantiation Socket in particular host, it begins to listen in the particular port. Then the server is made to accept the request from the client through the particular port.
- After starting the server, it can accept the requests from clients.
- The socket is instantiated in client side to connect to the server.
- A new Server Thread using socket is created to accept all the requests from multiple clients.
- After accepting the request both read and write operation occurs simultaneously, clients who request the server can communicate with each other and share resources.
- After finishing the communication the socket is closed both in the client and server side.

SERVER.C SOURCE CODE

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

#define MAX_CLIENTS 100
#define BUFFER_SZ 2048

static _Atomic unsigned int cli_count = 0;
static int uid = 10;

/* Client structure */
typedef struct{
    struct sockaddr_in address;
    int sockfd;
    int uid;
    char name[32];
} client_t;
```

```
client_t *clients[MAX_CLIENTS];
```

```
pthread_mutex_t clients_mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
void str_overwrite_stdout() {
```

```
    printf("\r%s", "> ");
```

```
    fflush(stdout);
```

```
}
```

```
void str_trim_lf (char* arr, int length) {
```

```
    int i;
```

```
    for (i = 0; i < length; i++) { // trim \n
```

```
        if (arr[i] == '\n') {
```

```
            arr[i] = '\0';
```

```
            break;
```

```
        }
```

```
    }
```

```
}
```

```
void print_client_addr(struct sockaddr_in addr){
```

```
    printf("%d.%d.%d.%d",
```

```
        addr.sin_addr.s_addr & 0xff,
```

```
        (addr.sin_addr.s_addr & 0xff00) >> 8,
```

```
        (addr.sin_addr.s_addr & 0xff0000) >> 16,
```

```
        (addr.sin_addr.s_addr & 0xff000000) >> 24);
```

```
}
```

```
/* Add clients to queue */
void queue_add(client_t *cl){
    pthread_mutex_lock(&clients_mutex);

    for(int i=0; i < MAX_CLIENTS; ++i){
        if(!clients[i]){
            clients[i] = cl;
            break;
        }
    }

    pthread_mutex_unlock(&clients_mutex);
}
```

```
/* Remove clients to queue */
void queue_remove(int uid){
    pthread_mutex_lock(&clients_mutex);

    for(int i=0; i < MAX_CLIENTS; ++i){
        if(clients[i]){
            if(clients[i]->uid == uid){
                clients[i] = NULL;
                break;
            }
        }
    }
}
```

```

        pthread_mutex_unlock(&clients_mutex);
    }

    /* Send message to all clients except sender */
    void send_message(char *s, int uid){
        pthread_mutex_lock(&clients_mutex);

        for(int i=0; i<MAX_CLIENTS; ++i){
            if(clients[i]){
                if(clients[i]->uid != uid){
                    if(write(clients[i]->sockfd, s, strlen(s)) < 0){
                        perror("ERROR: write to descriptor failed");
                        break;
                    }
                }
            }
        }

        pthread_mutex_unlock(&clients_mutex);
    }

    /* Handle all communication with the client */
    void *handle_client(void *arg){
        char buff_out[BUFFER_SZ];
        char name[32];
        int leave_flag = 0;

```

```

cli_count++;

client_t *cli = (client_t *)arg;

// Name
if(recv(cli->sockfd, name, 32, 0) <= 0 || strlen(name) < 2 || strlen(name)
>= 32-1){
    printf("Didn't enter the name.\n");
    leave_flag = 1;
} else{
    strcpy(cli->name, name);
    sprintf(buff_out, "%s has joined\n", cli->name);
    printf("%s", buff_out);
    send_message(buff_out, cli->uid);
}

bzero(buff_out, BUFFER_SZ);

while(1){
    if (leave_flag) {
        break;
    }

    int receive = recv(cli->sockfd, buff_out, BUFFER_SZ, 0);
    if (receive > 0){
        if(strlen(buff_out) > 0){
            send_message(buff_out, cli->uid);

```



```

        str_trim_lf(buff_out, strlen(buff_out));
        printf("%s -> %s\n", buff_out, cli->name);
    }
} else if (receive == 0 || strcmp(buff_out, "exit") == 0){
    sprintf(buff_out, "%s has left\n", cli->name);
    printf("%s", buff_out);
    send_message(buff_out, cli->uid);
    leave_flag = 1;
} else {
    printf("ERROR: -1\n");
    leave_flag = 1;
}

bzero(buff_out, BUFFER_SZ);
}

/* Delete client from queue and yield thread */
    close(cli->sockfd);
    queue_remove(cli->uid);
    free(cli);
    cli_count--;
    pthread_detach(pthread_self());

    return NULL;
}

int main(int argc, char **argv){

```

```

    if(argc != 2){
        printf("Usage: %s <port>\n", argv[0]);
        return EXIT_FAILURE;
    }

    char *ip = "127.0.0.1";
    int port = atoi(argv[1]);
    int option = 1;
    int listenfd = 0, connfd = 0;
    struct sockaddr_in serv_addr;
    struct sockaddr_in cli_addr;
    pthread_t tid;

    /* Socket settings */
    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(ip);
    serv_addr.sin_port = htons(port);

    /* Ignore pipe signals */
    signal(SIGPIPE, SIG_IGN);

    if(setsockopt(listenfd, SOL_SOCKET, (SO_REUSEPORT |
SO_REUSEADDR), (char*)&option, sizeof(option)) < 0){
        perror("ERROR: setsockopt failed");
        return EXIT_FAILURE;
    }

```

```

        /* Bind */
if(bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0) {
    perror("ERROR: Socket binding failed");
    return EXIT_FAILURE;
}

/* Listen */
if (listen(listenfd, 10) < 0) {
    perror("ERROR: Socket listening failed");
    return EXIT_FAILURE;
}

printf("=== WELCOME TO THE CHATROOM ===\n");

while(1){
    socklen_t clilen = sizeof(cli_addr);
    connfd = accept(listenfd, (struct sockaddr*)&cli_addr, &clilen);

    /* Check if max clients is reached */
    if((cli_count + 1) == MAX_CLIENTS){
        printf("Max clients reached. Rejected: ");
        print_client_addr(cli_addr);
        printf(":%d\n", cli_addr.sin_port);
        close(connfd);
        continue;
    }
}

```

```
    /* Client settings */
    client_t *cli = (client_t *)malloc(sizeof(client_t));
    cli->address = cli_addr;
    cli->sockfd = connfd;
    cli->uid = uid++;

    /* Add client to the queue and fork thread */
    queue_add(cli);
    pthread_create(&tid, NULL, &handle_client, (void*)cli);

    /* Reduce CPU usage */
    sleep(1);
}

return EXIT_SUCCESS;
}
```

CLIENT.C SOURCE CODE

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

#define LENGTH 2048

// Global variables
volatile sig_atomic_t flag = 0;
int sockfd = 0;
char name[32];

void str_overwrite_stdout() {
    printf("%s", "> ");
    fflush(stdout);
}

void str_trim_lf (char* arr, int length) {
    int i;
    for (i = 0; i < length; i++) { // trim \n
        if (arr[i] == '\n') {
```

```

        arr[i] = '\0';
        break;
    }
}
}

void catch_ctrl_c_and_exit(int sig) {
    flag = 1;
}

void send_msg_handler() {
    char message[LENGTH] = {};
    char buffer[LENGTH + 32] = {};

    while(1) {
        str_overwrite_stdout();
        fgets(message, LENGTH, stdin);
        str_trim_lf(message, LENGTH);

        if (strcmp(message, "exit") == 0) {
            break;
        } else {
            sprintf(buffer, "%s: %s\n", name, message);
            send(sockfd, buffer, strlen(buffer), 0);
        }

        bzero(message, LENGTH);
        bzero(buffer, LENGTH + 32);
    }
}

```

```

    catch_ctrl_c_and_exit(2);
}

void recv_msg_handler() {
    char message[LENGTH] = {};
    while (1) {
        int receive = recv(sockfd, message, LENGTH, 0);
        if (receive > 0) {
            printf("%s", message);
            str_overwrite_stdout();
        } else if (receive == 0) {
            break;
        } else {
            // -1
        }
        memset(message, 0, sizeof(message));
    }
}

int main(int argc, char **argv){
    if(argc != 2){
        printf("Usage: %s <port>\n", argv[0]);
        return EXIT_FAILURE;
    }

    char *ip = "127.0.0.1";
    int port = atoi(argv[1]);

    signal(SIGINT, catch_ctrl_c_and_exit);

```

```
    printf("Please enter your name: ");
fgets(name, 32, stdin);
str_trim_lf(name, strlen(name));

if (strlen(name) > 32 || strlen(name) < 2){
    printf("Name must be less than 30 and more than 2 characters.\n");
    return EXIT_FAILURE;
}

struct sockaddr_in server_addr;

/* Socket settings */
sockfd = socket(AF_INET, SOCK_STREAM, 0);
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = inet_addr(ip);
server_addr.sin_port = htons(port);

// Connect to Server
int err = connect(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr));
if (err == -1) {
    printf("ERROR: connect\n");
    return EXIT_FAILURE;
}

// Send name
send(sockfd, name, 32, 0);
```



```
printf("=== WELCOME TO THE CHATROOM ===\n");

pthread_t send_msg_thread;

if(pthread_create(&send_msg_thread, NULL, (void *) send_msg_handler, NULL) !=
0){

    printf("ERROR: pthread\n");

    return EXIT_FAILURE;

}

pthread_t recv_msg_thread;

if(pthread_create(&recv_msg_thread, NULL, (void *) recv_msg_handler, NULL) !=
0){

    printf("ERROR: pthread\n");

    return EXIT_FAILURE;

}

while (1){

    if(flag){

        printf("\nBye\n");

        break;

    }

}

close(sockfd);

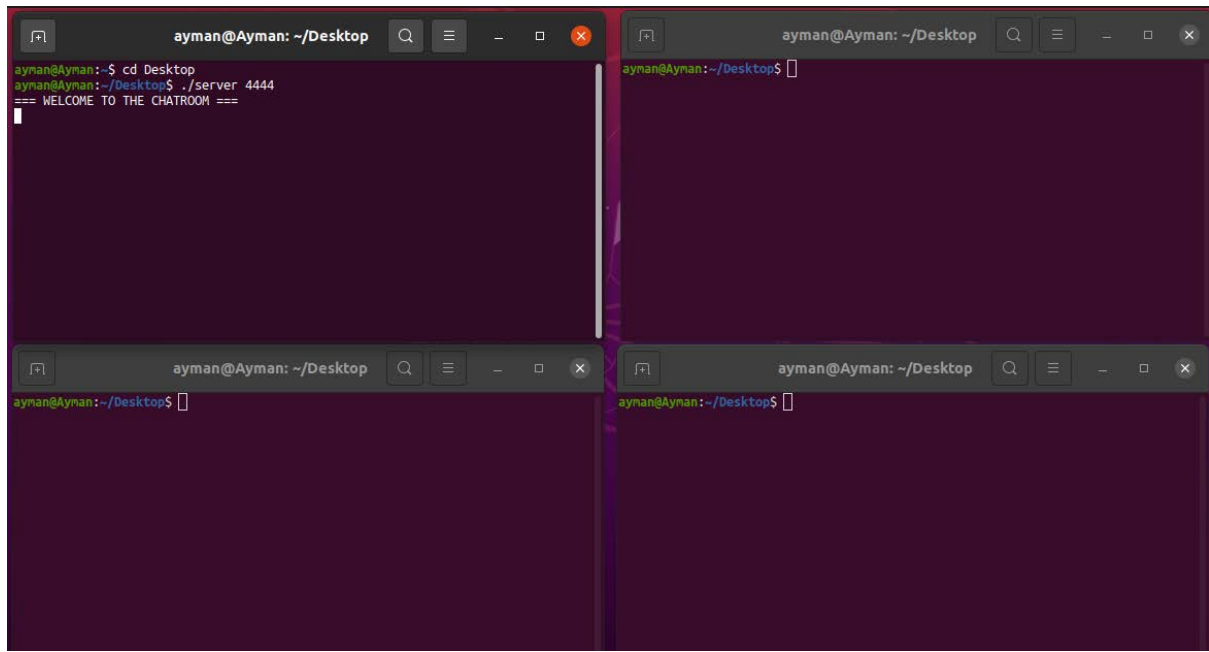
return EXIT_SUCCESS;

}
```

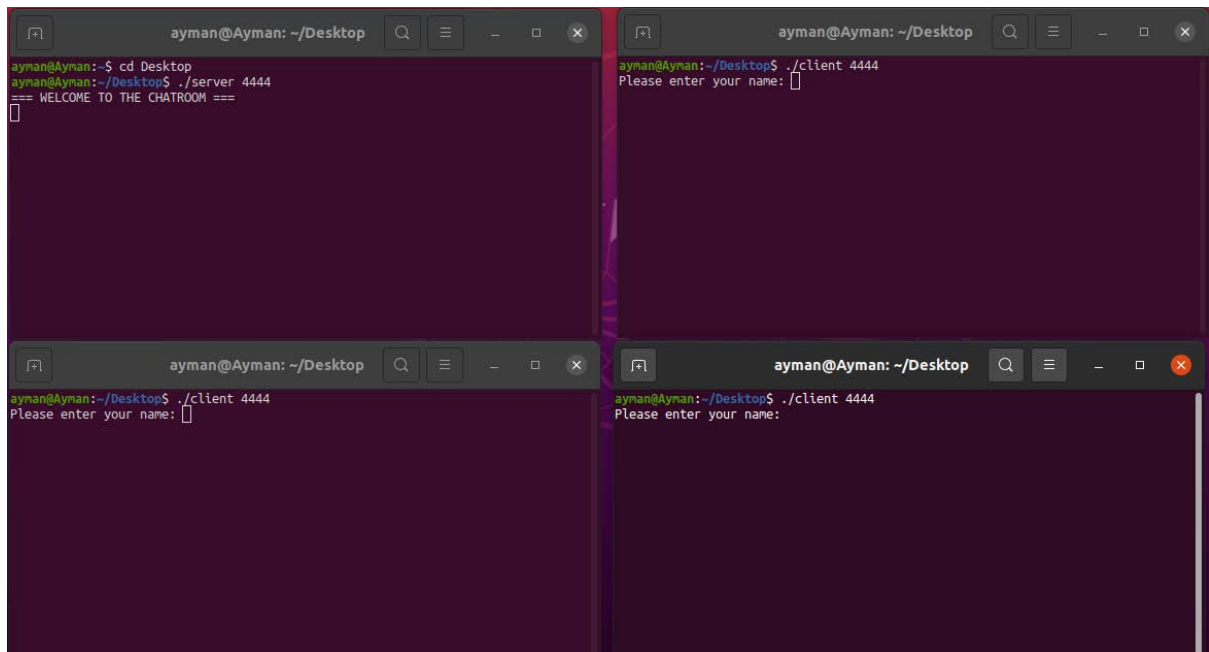
RESULTS AND SNAPSHOTS

For demonstrating our project , we have opened 4 terminals , one is the main server and the other three are clients / users.

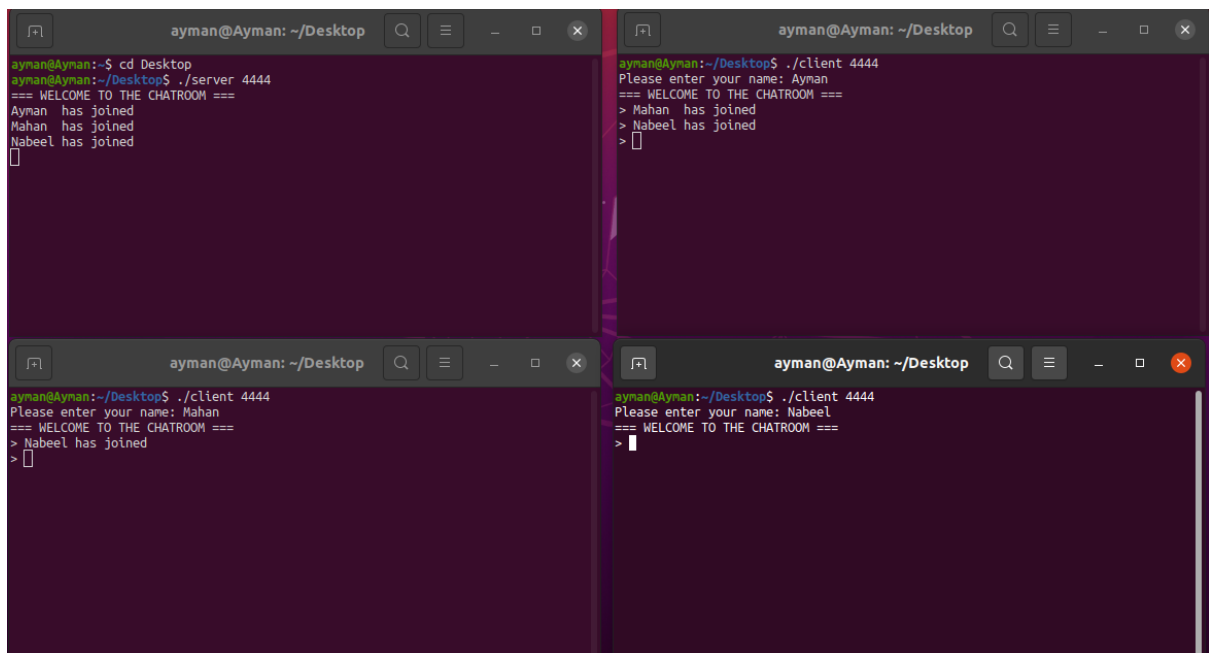
`./server 4444` is the command used to start the server.



Now we connect the clients / users to our chat room using the `./client 4444` command since the server is running at port 4444.



As we can see the connection between the client and server is successful and now the user is asked to enter his/her name.



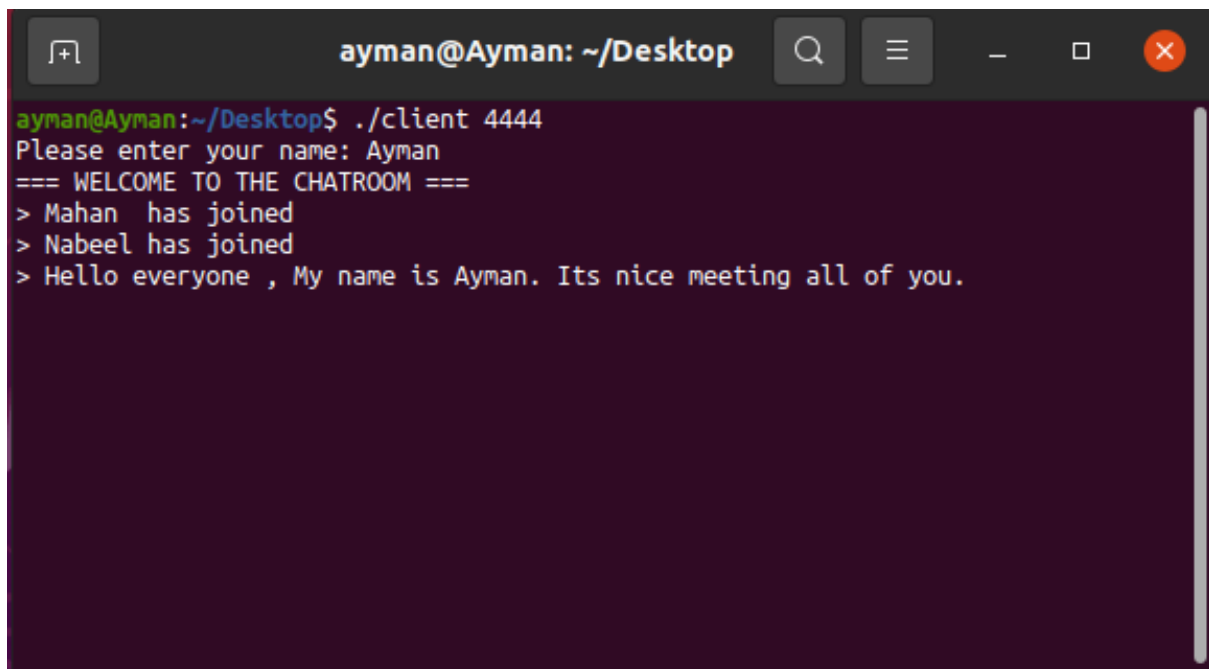
The image shows four terminal windows arranged in a 2x2 grid, all with the title bar 'ayman@Ayman: ~/Desktop'.
Top-left window: Shows the server running. The prompt is 'ayman@Ayman:~\$ cd Desktop'. Then 'ayman@Ayman:~/Desktop\$./server 4444' is entered. The output is '=== WELCOME TO THE CHATROOM ===', followed by 'Ayman has joined', 'Mahan has joined', and 'Nabeel has joined'.
Top-right window: Shows a client running. The prompt is 'ayman@Ayman:~/Desktop\$./client 4444'. The output is 'Please enter your name: Ayman', followed by '=== WELCOME TO THE CHATROOM ===', and then '> Mahan has joined' and '> Nabeel has joined'.
Bottom-left window: Shows another client running. The prompt is 'ayman@Ayman:~/Desktop\$./client 4444'. The output is 'Please enter your name: Mahan', followed by '=== WELCOME TO THE CHATROOM ===', and then '> Nabeel has joined'.
Bottom-right window: Shows another client running. The prompt is 'ayman@Ayman:~/Desktop\$./client 4444'. The output is 'Please enter your name: Nabeel', followed by '=== WELCOME TO THE CHATROOM ===', and then '>'.

As we can see that upon entering the name , subsequently the server displays the name of the users who has entered the chat room.

The user also gets a message as to who joined the chat room with them.

Lets now see the functionality of our multi chat system.

Suppose the user wishes to say the following in the chat room



The image shows a single terminal window with the title bar 'ayman@Ayman: ~/Desktop'. The prompt is 'ayman@Ayman:~/Desktop\$./client 4444'. The output is 'Please enter your name: Ayman', followed by '=== WELCOME TO THE CHATROOM ===', and then '> Mahan has joined', '> Nabeel has joined', and '> Hello everyone , My name is Ayman. Its nice meeting all of you.'.

As we can see down below the message entered by the user is displayed on the server and the clients who have entered the chat room.

```
ayman@Ayman: ~/Desktop
ayman@Ayman:~$ cd Desktop
ayman@Ayman:~/Desktop$ ./server 4444
=== WELCOME TO THE CHATROOM ===
Ayman has joined
Mahan has joined
Nabeel has joined
Ayman : Hello everyone , My name is Ayman. Its nice meeting all of you. -> Ayman

```

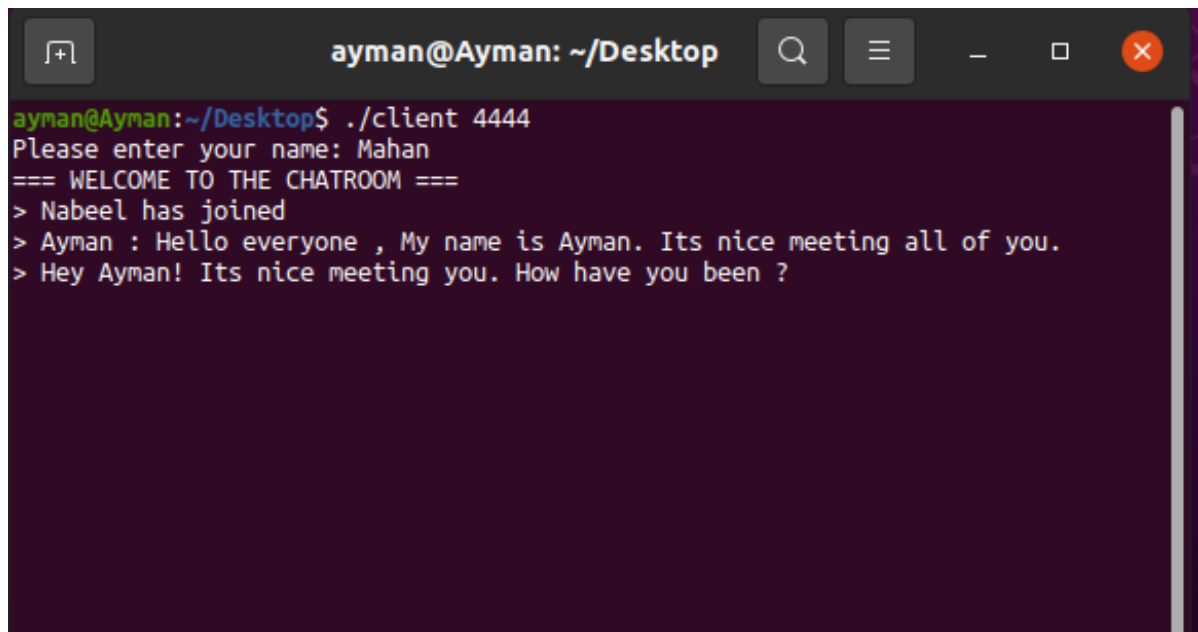
```
ayman@Ayman: ~/Desktop
ayman@Ayman:~/Desktop$ ./client 4444
Please enter your name: Mahan
=== WELCOME TO THE CHATROOM ===
> Nabeel has joined
> Ayman : Hello everyone , My name is Ayman. Its nice meeting all of you.
>

```

```
ayman@Ayman: ~/Desktop
ayman@Ayman:~/Desktop$ ./client 4444
Please enter your name: Nabeel
=== WELCOME TO THE CHATROOM ===
> Ayman : Hello everyone , My name is Ayman. Its nice meeting all of you.
>

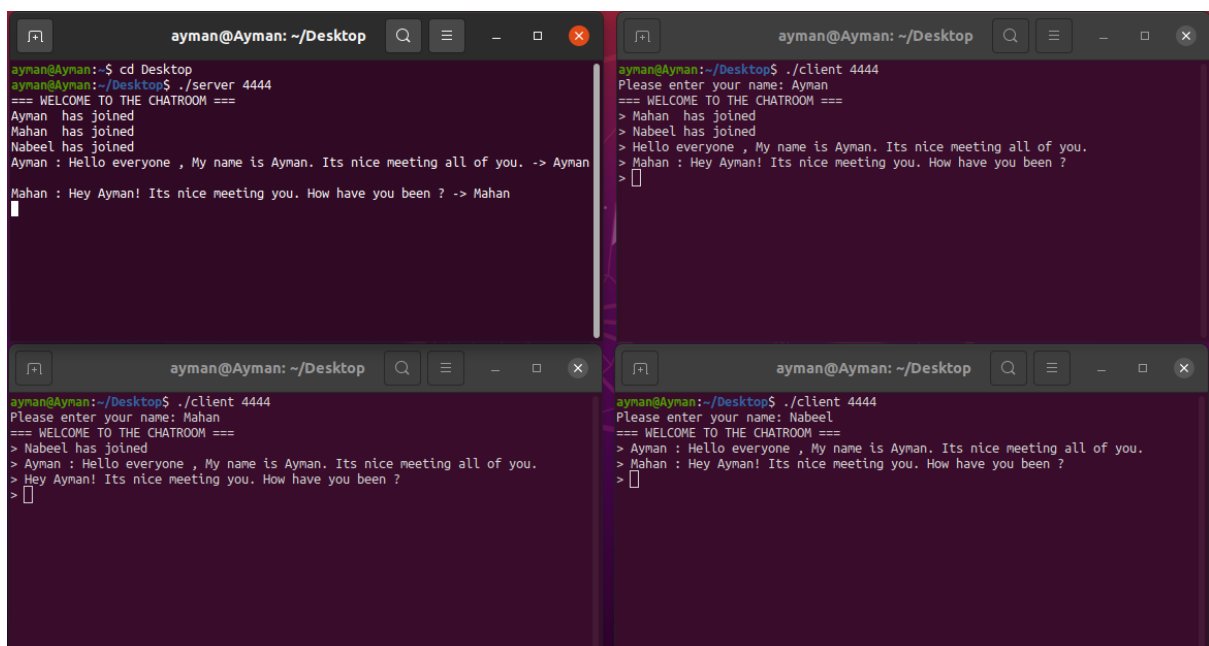
```

Now suppose the other user wishes to reply to the user who sent the message, he can accomplish it by typing the message he desires to send

A terminal window titled 'ayman@Ayman: ~/Desktop' with standard window controls. The prompt is 'ayman@Ayman:~/Desktop\$'. The user has entered './client 4444'. The output shows a chatroom interface: 'Please enter your name: Mahan', '=== WELCOME TO THE CHATROOM ===', and a list of messages: '> Nabeel has joined', '> Ayman : Hello everyone , My name is Ayman. Its nice meeting all of you.', and '> Hey Ayman! Its nice meeting you. How have you been ?'.

```
ayman@Ayman:~/Desktop$ ./client 4444
Please enter your name: Mahan
=== WELCOME TO THE CHATROOM ===
> Nabeel has joined
> Ayman : Hello everyone , My name is Ayman. Its nice meeting all of you.
> Hey Ayman! Its nice meeting you. How have you been ?
```

The same message is now then displayed across the server and the clients present in the chat room

Four terminal windows are shown, each titled 'ayman@Ayman: ~/Desktop'. The top-left window shows the server running './server 4444', displaying join notifications for Ayman, Mahan, and Nabeel, and the start of a message from Ayman. The top-right window shows a client running './client 4444' with the name 'Ayman', displaying the same chatroom messages. The bottom-left window shows another client running './client 4444' with the name 'Mahan', displaying the same chatroom messages. The bottom-right window shows a client running './client 4444' with the name 'Nabeel', displaying the same chatroom messages. This demonstrates message distribution to all clients.

```
ayman@Ayman:~/Desktop$ ./server 4444
=== WELCOME TO THE CHATROOM ===
Ayman has joined
Mahan has joined
Nabeel has joined
Ayman : Hello everyone , My name is Ayman. Its nice meeting all of you. -> Ayman
Mahan : Hey Ayman! Its nice meeting you. How have you been ? -> Mahan

ayman@Ayman:~/Desktop$ ./client 4444
Please enter your name: Ayman
=== WELCOME TO THE CHATROOM ===
> Mahan has joined
> Nabeel has joined
> Hello everyone , My name is Ayman. Its nice meeting all of you.
> Mahan : Hey Ayman! Its nice meeting you. How have you been ?
>

ayman@Ayman:~/Desktop$ ./client 4444
Please enter your name: Mahan
=== WELCOME TO THE CHATROOM ===
> Nabeel has joined
> Ayman : Hello everyone , My name is Ayman. Its nice meeting all of you.
> Hey Ayman! Its nice meeting you. How have you been ?
>

ayman@Ayman:~/Desktop$ ./client 4444
Please enter your name: Nabeel
=== WELCOME TO THE CHATROOM ===
> Ayman : Hello everyone , My name is Ayman. Its nice meeting all of you.
> Mahan : Hey Ayman! Its nice meeting you. How have you been ?
>
```