# Development of a Client-Server Service: Multi-client Chatroom

**Student** Samantha Licea Dominguez
**Teacher** Sugaya Midori
**Class** Operating Systems and Excercises

## Introduction

Nowadays, we are so used to use chat applications (such as WhatsApp, Line or WeChat) in our daily lives: either to communicate with friends, relatives, co-workers or people in general. For this project, we were asked to develop a client-server service in C language. That is why, the motivation for this project is to develop a multi-client chatroom that allows clients to connect to the server and communicate with each other using the client-server architecture. Also, this is one of the easiest implementations of client-server applications. However, there are several ways to implement such application, so that is why this project uses *treads* to accomplish the exchange of messages.

As it is known, the Internet is divided into some layers due to its high complexity. One of those layers, the *Application Layer*, is implemented all in software and is run in multiple End Systems. End Systems (ESs), also known as hosts, are devices that are connected to the Internet and within each other to exchange information. ESs are defined by an *IP address*, which is the ES's general address, and a *port number*, which allows to specify the process that is being run in the ES. Application structures describe the structure and distribution of the application's functions among hosts. In this case, the Client-Server architecture is developed. A *server* is a host that is "always" online, and provides services or fulfills requests. On the other hand, a *client* is a host that requires services from the server. Some of the properties of this architecture include that the server has an established address, clients connect to the server and not to each other, and all clients connect to the server and the server accomplishes their requests. Another thing worth mentioning is the concept of process. A *process*, similarly to an application, is a program that runs also on ESs. A process that sends information creates the application layer and sends data to the network, whilst a process that receives information receives data from the network. A *thread* manages the execution of processes as a sequence, and is implemented in a parallel way to the program. Lastly, a *socket* is a software interface used to send and receive processes among ESs.

The structure of this assignment goes as follows: Section Methodology describes the flow chart of the programs to be implemented (there are 2 flowcharts, one for the client's code and one for the server's) as well as the design and the sources from which the information was retrieved; Section Source Code displays and explains the code thoroughly so that the reader is able to understand the concepts explained above; and Section Conclusion encompasses all the development of the project as well as the contents learned during the class and impressions of the assignment.

## Methodology

For this project, several electronic resources were consulted and used. First, the class presentations were helpful to refresh the knowledge obtained from the classes, as well as for the implementation of the code for both the client and the server codes [1, 2], especially the PDF presentations corresponding to Client-Server socket programming, and the one of thread and fork contents [3]. Also, since this task was a little bit challenging, another sources were consulted [4, 5] for better development as well as for checking possible errors in the code and comparing it for improvements.

For a better understanding of the code in Section Source Code, flowcharts of both the code of the client and the server were made, as it can be seen in **Figure 1**. The following is the explanation for the server's code:

1. The program first starts by initializing the necessary libraries to work properly, as well as global variables such as MAX_CLIENTS, BUFFER_SIZE and NAME_LEN. Also, it defines a client structure to store the client's information such as their address, description of the socket, their unique user ID, and name. Additionally, the program uses *pthread_mutex_t*, which protects all the arrays of characters containing the messages when sending them.

2. In the main function, the server is initialized and bound to a socket.

3. Then it listens any incoming connections.

4. If a new client connects, the program creates a new thread (for each client) and assigns a unique ID to it. This thread manages the receiving messages for the client and broadcasts them to other clients connected to the server.

This is a general description of what the server actually does. If will be explained in detail in the next section, so meanwhile the client code's flow goes as follows:

1. The program, similarly to the server program, starts by initializing the necessary libraries to work properly, as well as global variables such as MAX_CLIENTS, BUFFER_SIZE and NAME_LEN.

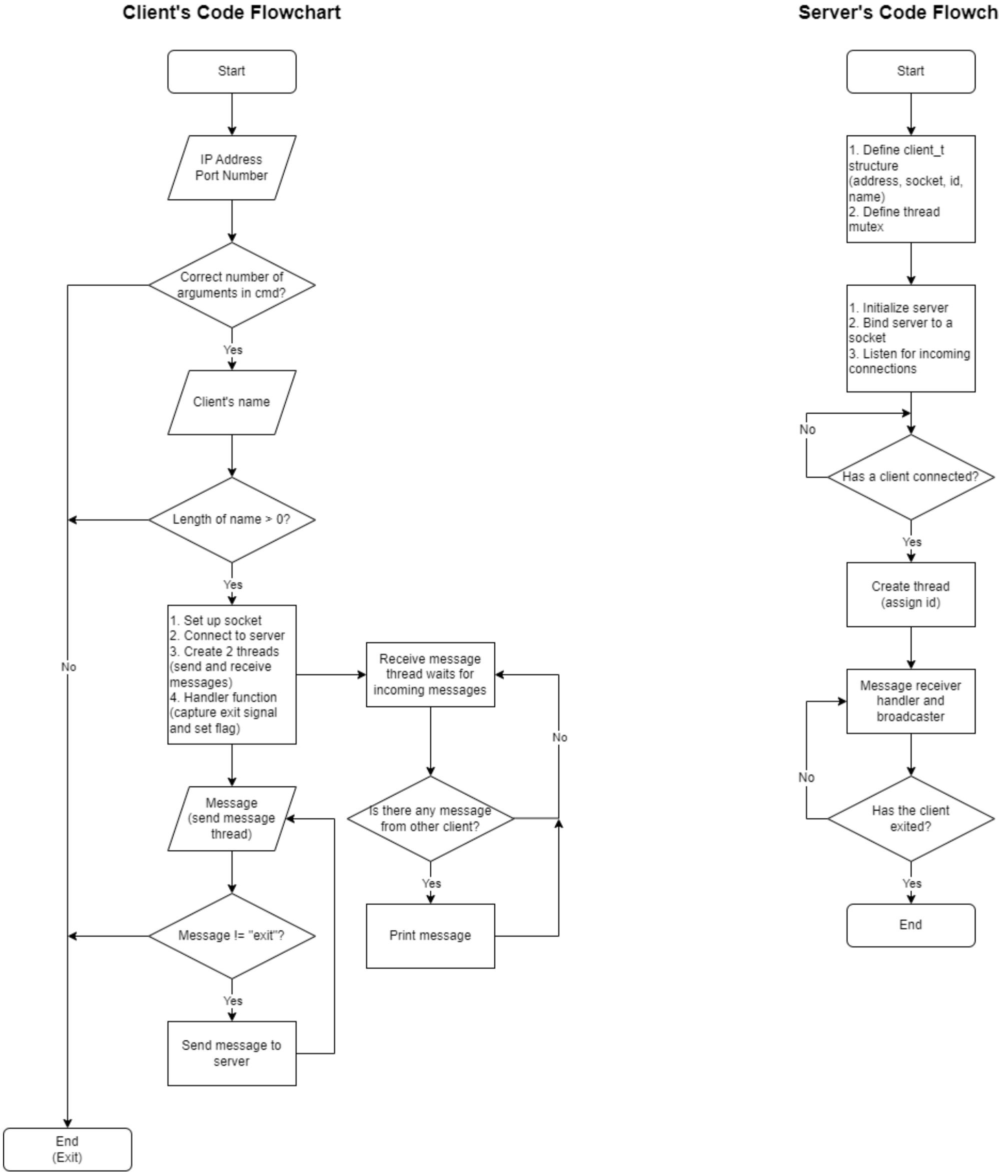
2. In the main function, the program waits for the input of the client's name. If the name's length is greater than 0 and smaller than the NAME_LEN, then it proceeds. If not, the program exits with an error.

3. Then, the program sets a socket for the client, connects to the server and creates 2 threads: one to handle incoming messages from other clients connected to the server, and one to handle the transmission of messages. It also has a handler function, which is in charge of detecting the exiting flag.

4. Then, the sending message thread waits for the user to type a message. If the message is different from "exit" (which is the exiting code word for this assignment), it sends the message to the server. If not, it closes the connection with the server.

5. At the same time, the receiving message thread waits for incoming messages from the server. When there is a new message, it prints it. If not, it keeps waiting, until the connection with the server is closed.

**Figure 1**



Client's Code Flowchart

Server's Code Flowchart

Start
IP Address
Port Number
Correct number of arguments in cmd?
Yes
Client's name
Length of name > 0?
Yes
1. Set up socket
2. Connect to server
3. Create 2 threads (send and receive messages)
4. Handler function (capture exit signal and set flag)
No
Receive message thread waits for incoming messages
No
Is there any message from other client?
Yes
Print message
Message (send message thread)
Message != "exit"?
Yes
Send message to server
End (Exit)
Start
1. Define client_t structure (address, socket, id, name)
2. Define thread mutex
1. Initialize server
2. Bind server to a socket
3. Listen for incoming connections
No
Has a client connected?
Yes
Create thread (assign id)
Message receiver handler and broadcaster
No
Has the client exited?
Yes
End

Once the general perspective of this service's design has been understood, the codes for both client and server is explained thoroughly in the next section.

## Source Code

In this section, the program for this service is explained in a detailed way. However, before that, a very general explanation should be introduced. The idea of this program is that the server will be bound to a specific address and port (given by the user as an input in the terminal). After performing several checks, the server connects and starts a thread for each of the clients that connect. On the other hand, the client connects to the address and port number of the server, and when done so, 2 main threads start: sending and receiving messages. This will continue infinitely until the program receives the defined exiting key word, where the client will stop the threads and close the connection with the server.

Knowing the general operation of the code, the code of the server is explained as follows:

1. First, all the necessary libraries are imported in order to be able to work, especially the libraries for the socket (<sys/socket.h>) and the thread (<pthread.h>).

2. Then, 3 global variables, which correspond to the maximum number of clients that can be connected to the server (set to 100 in this case), the buffer's size which will carry information (set to 2048) and the length of the user's name array (up to 32 characters) are defined.

3. In the main function, it is established that the user will input both the IP address and Port Number to which it will connect, and those values are stored in their corresponding variables (char *ip = argv[0] and int port). Also, there are 2 structures of type *sockaddr_in*, which is a library from the <arpa/inet.h> library, in charge of dealing with network addresses, so there is one for the client and one for the server. And lastly, the thread (pthread_t tid) is also initialized.

4. Continuing in the main, the input of the IP address and the Port number is checked to be correct (3 arguments). If it is not correct, it exits with an error of number of arguments.

5. Then the endpoint connection with the *socket()* function is established by defining its domain, type and protocol; and the settings of the server such as family, address and port are defined as well.

6. A signal is set in order to manage errors in the code (ignoring SIGPIPE). Then, the socket properties are checked, and if there was to be an error, the program exits with an error. If there isn't one, the program proceeds.

7. The address to the server with the *bind()* function is bound, and if this is successful, then the server starts listening for connections in the socket.

8. The program listens for any client connection endlessly (until there is an exiting command). A client is accepted to the server as long as the capacity of clients connected doesn't reach the maximum amount set. To manage that, there is a counter that keeps record of the amount of clients connected. So if the maximum amount o clients is reached, and there is a client who wants to connect, there is an error and the connection closes with that client.

9. However, if the connection is successful, the client's information structure is built by gathering its address, socket, unique ID and name (established by the client). After, the client is added to the queue, and a thread for it starts.

   (a) In the clientHandler function, the buffer is established and the name of the client is obtained. Also, the flag for when a client exits the chatroom is set.

   (b) First, the client has to input its name. The length of the name is checked to be correct according to the length, and if it's not correct there is said to be an error and the client must provide a correct name.

   (c) If there is no error in the name, the client joins the server and is able to start sending messages. Several code is implemented to manage the messages, so that when a message is sent, it shows the other clients it's name and the message. Each time a message is sent, the buffer is set empty again. Also, the client can receive messages at any time, so another bunch of code is computed to receive incoming messages from other clients and printing them with the client's name.

(d) At last, if the message written by the client is equal to the *”exit”* string, then the client is taken off from the queue (since it is the key word for exiting the server). This is also announced to the rest of the clients connected to the server. After the client is off the queue, everything is updated (client counter and buffer), and both the connection and the thread closes and ends.

It can be noted that there are functions that manage the sending and receiving of the messages, and there is the use of *pthread_mutex_unlock*, which is in charge of the protection of transference of messages. The code can be observed as follows.

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

#define MAX_CLIENTS 100
#define BUFFER_SIZE 2048
#define NAME_LEN 32

//client's structure
typedef struct{
  struct sockaddr_in address;
  int sockfd;
  int uid; //user id
  char name[NAME_LEN];
} client_t;

client_t *clients[MAX_CLIENTS];
pthread_mutex_t clients_mutex = PTHREAD_MUTEX_INITIALIZER; //transferring messages

//global variables
static _Atomic unsigned int ccont = 0; //client cont
static int uid = 10;

//functions
//Sending, receiving and overriding the buffer. Also, in and off the queue.
void override_str_stdout(){
  printf("\n%s","> ");
  fflush(stdout);
}

void trim_str_lf(char* arr,int len){
  for(int i = 0; i < len;i++){
    if(arr[i] == '\n'){
      arr[i] = '\0';
      break;
    }
  }
}

void intoQueue(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);
}

void offQueue(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 everyone except the sender
void sendMessage(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){
          printf("[-]ERROR: write to descriptor failed\n");
          break;
        }
      }
    }
  }

  pthread_mutex_unlock(&clients_mutex);
}
//client handler (thread)
void *clientHandler(void *arg){
  char buffer[BUFFER_SIZE];
  char name[NAME_LEN];
  int lflag = 0; //leave flag
  ccont++;

  client_t *cli = (client_t*)arg;

  //nombre del cliente
  if(recv(cli->sockfd,name,NAME_LEN,0) <= 0 || strlen(name) < 2 || strlen(name) >= NAME_LEN - 1){
    printf("[-]Enter name correctly\n");
    lflag = 1;
  }else{
    //el cliente se unio al servidor
    strcpy(cli->name,name);
    sprintf(buffer,"[+]%s has joined the chatroom\n",cli->name);
    printf("%s",buffer);
    sendMessage(buffer,cli->uid);
  }

  bzero(buffer,BUFFER_SIZE);

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

    int receive = recv(cli->sockfd,buffer,BUFFER_SIZE,0);

    if(receive > 0){
      if(strlen(buffer) > 0){
        sendMessage(buffer,cli->uid);
        trim_str_lf(buffer,strlen(buffer));
        printf("%s1\n",buffer);
      }
    }else if(receive == 0 || strcmp(buffer,":exit") == 0){
      sprintf(buffer,"[+]%s has left the chatroom\n",cli->name);
      printf("%s",buffer);
      sendMessage(buffer,cli->uid);
      lflag = 1;
    }else{
      printf("[-]ERROR: -1\n");
      lflag = 1;
    }

    bzero(buffer,BUFFER_SIZE);
  }
  close(cli->sockfd);
  offQueue(cli->uid);
  free(cli);
  ccont--;
  pthread_detach(pthread_self());

  return NULL;
}

int main(int argc,char *argv[]){
  char *ip = argv[0];
  int port = strtol(argv[2],NULL,10),option = 1,listenfd = 0,connfd = 0;
  struct sockaddr_in servAddr;
  struct sockaddr_in cliAddr;
  pthread_t tid;

  if(argc != 3){
    printf("[-]Usage: %s <port>\n",argv[0]);
    return EXIT_FAILURE;
  }

  //socket settings
  listenfd = socket(AF_INET,SOCK_STREAM,0);
  servAddr.sin_family = AF_INET;
  servAddr.sin_addr.s_addr = inet_addr(ip);
  servAddr.sin_port = htons(port);
  inet_aton(argv[1],&servAddr.sin_addr);

  //signals
  signal(SIGPIPE,SIG_IGN);
  if(setsockopt(listenfd,SOL_SOCKET,(SO_REUSEPORT | SO_REUSEADDR),(char*)&option,sizeof(option)) < 0){
    printf("[-]ERROR: set socket option\n");
    return EXIT_FAILURE;
  }

  //binding
  if(bind(listenfd,(struct sockaddr*)&servAddr,sizeof(servAddr)) < 0){
    printf("[-]ERROR: binding\n");
    return EXIT_FAILURE;
  }

  //if bind == success, listen
  if(listen(listenfd,10) < 0){
    printf("[-]ERROR: listening\n");
    return EXIT_FAILURE;
  }

  //if everything == success, server listens
  printf("OSProject===******= WELCOME TO THE CHATROOM : SERVER ===******===OSProject\n");

  //infinite loop to connect clients
  while(1){
    socklen_t clilen = sizeof(cliAddr);
    connfd = accept(listenfd,(struct sockaddr*)&cliAddr,&clilen);

    //check maximum amount of clients
    if((ccont +1) == MAX_CLIENTS){
      printf("[-]MAXIMUM amount of clients reached. Connection Rejected: %s\n",inet_ntoa(servAddr.sin_addr));
      close(connfd);
      continue;
    }

    //client settings
    client_t *cli = (client_t *)malloc(sizeof(client_t));
    cli->address = cliAddr;
    cli->sockfd = connfd;
    cli->uid = uid++;

    //add client to the queue and start thread
    intoQueue(cli);
    pthread_create(&tid,NULL,&clientHandler,(void*)cli);

    //reduce CPU use
    sleep(1);
  }

  return EXIT_SUCCESS;
}
```

Now that the code of the server has been explained, the code of the client must become easier to understand. Also, it uses several functions and features similar to the server's code. The code for the client can be observed as follows:

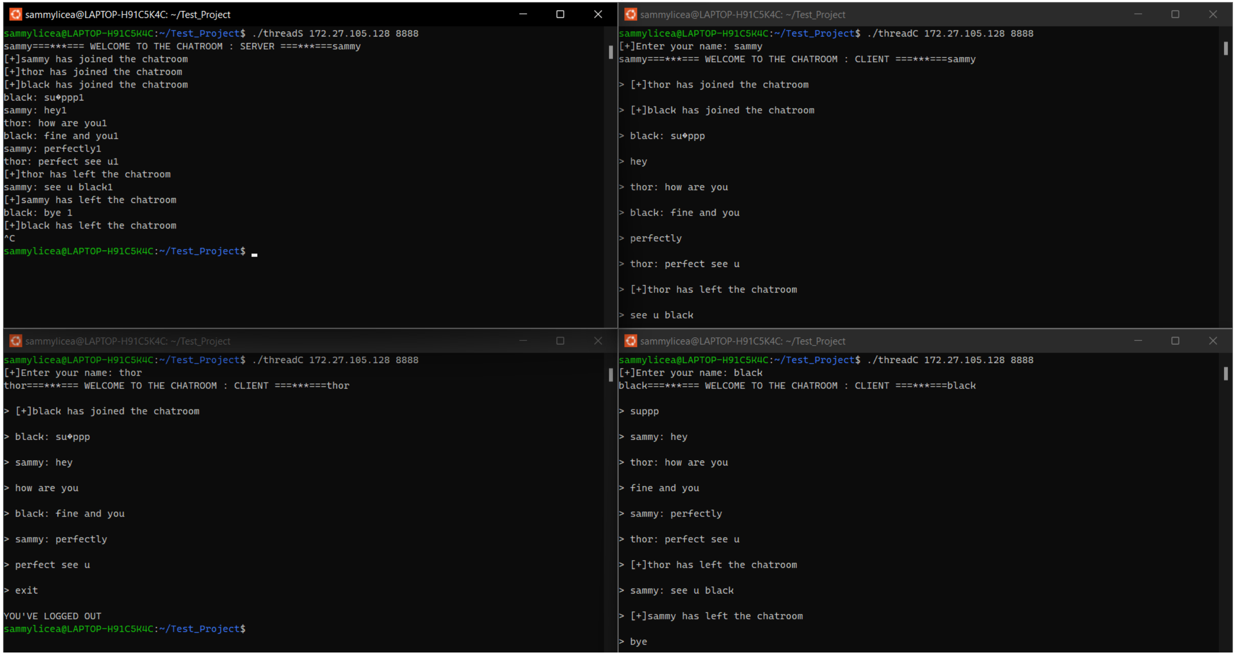
1. First, the proper libraries are installed (same as in the server) to be able to set the socket and threads. Also, the maximum amount of clients is defined, as well as the buffer size and name length, globally. We have the same functions for overriding, trimming, and catching the exit flag.
2. In the main function, same as in the server's code, the user inputs the IP address and Port Number to be connected to (in this case, the one pertaining to the server). So if the correct amount of arguments is given (3) then the program proceeds, and if it's not, then there is said to be an error and the program exits with an error.
3. Then, the same signal as the server is set. If it is not set correctly, then there is said to be an error.
4. If there is no error, then the program proceeds to the socket's settings and to the connection from the client to the server. If there is an error regarding the address, port number, etc., then there is to be an error in the connection.
5. If there is no error, then the client's name is sent altogether with the name's length.
6. If there are no errors, then the client is connected and the server is listening to it. Then 2 threads are created: one for sending messages and one from receiving them.
   (a) The *sendMess_handler()* thread is in charge of setting the buffer to 0 (clearing it) after sending each message. It contains the name of the client and the message, so that when it is sent, all clients connected know who sent the message. If the message to be sent is equal to the *"exit"* string, then the thread stops.
   (b) On the other hand, the *recvMess_handler()* thread is in charge of receiving incoming messages from other clients and printing them to the client. If the connection is lost, then the thread stops.
7. An infinite loop is set, where the exit flag is constantly being checked. If the client exits the chatroom, the connection is closed and the threads both stop.

The code for the client can be seen as follows.

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

#define MAX_CLIENTS 100
#define BUFF_SIZE 2048
#define NAME_LEN 32

volatile sig_atomic_t flag = 0;
int sockfd = 0;
char name[NAME_LEN];

```

```c
void override_str_stdout(){
  printf("\n%s","> ");
  fflush(stdout);
}

void trim_str_lf(char* arr,int len){
  for(int i = 0; i < len;i++){
    if(arr[i] == '\n'){
      arr[i] = '\0';
      break;
    }
  }
}

void catchExit(){
  flag = 1;
}

void recvMess_handler(){
  char message[BUFF_SIZE] = {};
  while(1){
    int receive = recv(sockfd,message,BUFF_SIZE,0);

    if(receive > 0){
      printf("%s ",message);
      override_str_stdout();
    } else if(receive == 0){
      break;
    }
    bzero(message,BUFF_SIZE);
  }
}

void sendMess_handler(){
  char buffer[BUFF_SIZE] = {};
  char message[BUFF_SIZE + NAME_LEN] = {};

  while(1){
    override_str_stdout();
    fgets(buffer,BUFF_SIZE,stdin);
    trim_str_lf(buffer,BUFF_SIZE);

    if(strcmp(buffer,"exit") == 0){
      break;
    }else{
      sprintf(message,"%s: %s\n",name,buffer);
      send(sockfd,message,strlen(message),0);
    }
    bzero(buffer,BUFF_SIZE);
    bzero(message,BUFF_SIZE + NAME_LEN);
  }
  catchExit(2);
}

int main(int argc,char *argv[]){
  if(argc != 3){
    printf("[-]Usage: %s <port>\n",argv[0]);
    return EXIT_FAILURE;
  }

  char *ip = argv[0];
  int port = strtol(argv[2],NULL,10);

  //signal
  signal(SIGINT,catchExit);
  printf("[+]Enter your name: ");
  fgets(name,NAME_LEN,stdin);
  trim_str_lf(name,strlen(name));

  if(strlen(name) > NAME_LEN - 1 || strlen(name) < 2){
    printf("[-]ERROR: Enter name correctly again: \n");
    return EXIT_FAILURE;
  }

  struct sockaddr_in servAddr;
  //socket setting
  sockfd = socket(AF_INET,SOCK_STREAM,0);
  servAddr.sin_family = AF_INET;
  servAddr.sin_addr.s_addr = inet_addr(ip);
  servAddr.sin_port = htons(port);
  inet_aton(argv[1],&servAddr.sin_addr);

  //Connection to the server
  int err = connect(sockfd,(struct sockaddr*)&servAddr,sizeof(servAddr));
  if(err == -1){
    printf("[-]ERROR: connection\n");
    return EXIT_FAILURE;
  }

  //send name
  send(sockfd,name,NAME_LEN,0);

  //if everything == success, server listens
  printf("%s===******=== WELCOME TO THE CHATROOM : CLIENT ===******===%s\n",name,name);

  pthread_t sendMess;
  pthread_t recvMess;
  if(pthread_create(&sendMess,NULL,(void*)sendMess_handler,NULL) != 0){
    printf("[-]ERROR: pthread Send\n");
    return EXIT_FAILURE;
  }
  if(pthread_create(&recvMess,NULL,(void*)recvMess_handler,NULL) != 0){
    printf("[-]ERROR: pthread Receive\n");
    return EXIT_FAILURE;
  }

  while(1){
    if(flag){
      printf("\nYOU'VE LOGGED OUT\n");
      break;
    }
  }

  close(sockfd);

  return EXIT_SUCCESS;
}
```

## Image 1


sammylicea@LAPTOP-H91C5K4C: ~/Test_Project
sammylicea@LAPTOP-H91C5K4C: ~/Test_Project
sammylicea@LAPTOP-H91C5K4C: ~/Test_Project
sammylicea@LAPTOP-H91C5K4C: ~/Test_Project

```
sammylicea@LAPTOP-H91C5K4C:~/Test_Project$ ./threadS 172.27.105.128 8888
sammy===******=== WELCOME TO THE CHATROOM : SERVER ===******===sammy
[+]sammy has joined the chatroom
[+]thor has joined the chatroom
[+]black has joined the chatroom
black: su�ppp1
sammy: hey1
thor: how are you1
black: fine and you1
sammy: perfectly1
thor: perfect see u1
[+]thor has left the chatroom
sammy: see u black1
[+]sammy has left the chatroom
black: bye 1
[+]black has left the chatroom
^C
sammylicea@LAPTOP-H91C5K4C:~/Test_Project$ 
```

```
sammylicea@LAPTOP-H91C5K4C:~/Test_Project$ ./threadC 172.27.105.128 8888
[+]Enter your name: sammy
sammy===******=== WELCOME TO THE CHATROOM : CLIENT ===******===sammy

> [+]thor has joined the chatroom

> [+]black has joined the chatroom

> black: su�ppp

> hey

> thor: how are you

> black: fine and you

> perfectly

> thor: perfect see u

> [+]thor has left the chatroom

> see u black
```

```
sammylicea@LAPTOP-H91C5K4C:~/Test_Project$ ./threadC 172.27.105.128 8888
[+]Enter your name: thor
thor===******=== WELCOME TO THE CHATROOM : CLIENT ===******===thor

> [+]black has joined the chatroom

> black: su�ppp

> sammy: hey

> how are you

> black: fine and you

> sammy: perfectly

> perfect see u

> exit

YOU'VE LOGGED OUT
sammylicea@LAPTOP-H91C5K4C:~/Test_Project$
```

```
sammylicea@LAPTOP-H91C5K4C:~/Test_Project$ ./threadC 172.27.105.128 8888
[+]Enter your name: black
black===******=== WELCOME TO THE CHATROOM : CLIENT ===******===black

> suppp

> sammy: hey

> thor: how are you

> fine and you

> sammy: perfectly

> thor: perfect see u

> [+]thor has left the chatroom

> sammy: see u black

> [+]sammy has left the chatroom

> bye
```

**Image 1** shows a picture of the successful running of the program using 3 clients. At the time of the test of the program, the IP address used was the one of SIT Global's Dorm, and the Port Number chosen was 8888. As it can be observed, the server displays the clients who join or exit the chatroom, as well as the messages sent and the client who sends them. On the other hand, it can be observed that each of the clients join the server after inputting their names correctly. After that, they send and receive messages between them until they write *"exit"* (or of course press Ctrl+C). This is the overall implementation of the chatroom service using threads in C language.

## Conclusion

The impression for this project was that client-server architectures are pretty interesting, and at some point fun to develop. However, a very precise understanding of the concepts is needed, since each one of the terms is related to one another. Also, this project was developed in C language in Linux, so a lot of knowledge about libraries and commands is needed to succeed in this type of assignments.

Further, the topics covered in this class were pretty interesting overall. Learning from scratch how to use Linux to build a chatroom application is pretty amazing. Personally, I think that the contents of this class are very useful and common in human's daily lives. So that is why, altogether with the class of Introduction to Computer Networks, I had a great time learning.

## References

[1] Midori Sugaya. Operating systems and system programming 9th network socket (client). University Lecture, 2022. Accessed: 2023–01-06.

[2] Midori Sugaya. Operating systems and system programming 10th network socket (server), 2022. Accessed: 2023–01-16.

[3] Midori Sugaya. Operating systems and system programming 8th session : Threads and deadlock. University Lecture, 2022. Accessed: 2023–01-06.

[4] Idiot Developer. Chatroom in c using threads — socket programming in c, May 2019.

[5] Idiot Developer. Multiple client server program in c using fork — socket programming, Dic. 2017.