MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

Multi Client Chat Server

Mini Project in Computer Networks

Shriya Hebbar 210905118

Megan Samuel 210905101

Sidhant Sinha 210905120

Upendra Sharma 210905127

Course: CSE 3162

Semester V

2023-11-15

Report

Abstract

In this project, A Multithreaded Client-Server application is implemented to replicate a chat room. The primary goal is to enable multiple clients to connect to a central server. They can then exchange text messages, and transfer files among themselves. The server will run continuously receiving requests from clients waiting for connection and grant them access to chat upon successful authentication. Once connected, clients can exchange text messages with other connected clients in a command line interface. Messages are relayed through the server to ensure communication among all clients. Clients can send files to other clients within the chat system. The server facilitates this process by coordinating the transfer and ensuring the correct delivery of files. Clients can send and receive various types of files. The request to the server will be in format (*receiver_name type_of_message message/filename*). The request is then relayed based on the *receiver_name* to the specified client. The client has two jobs: wait for a message to be sent by the client (keyboard event) or a message to be displayed to the interface (server event).

Acknowledgment

We would like to express our sincere gratitude to our dedicated team members for their unwavering commitment and collaborative spirit in completing this college group project on computer networks. Each member's unique skills and contributions played a pivotal role in the successful development of our project.This endeavor has been a valuable learning experience, and we appreciate the support and encouragement from our peers and the academic community.

Contents

1. Introduction

In today's interconnected world, communication is a fundamental aspect of our daily lives. Whether for business collaboration, social interaction, or information sharing, efficient and secure communication tools are paramount. The project at hand, a Multi-Client Chat Server with File Transfer, addresses this need by offering a robust and versatile solution.

This project is designed and implemented using the C programming language, known for its performance and versatility. It combines the power of networking, file handling, and security to create a comprehensive chat server that caters to the demands of modern communication. The aim is to create an all-in-one solution for intra-network communication which is reliable and fast at the same time. Secure communication is a huge aspect for which user authentication is necessary as well.

1.1 Problem Statement

A Multi-Client Chat Server that allows exchange of messages between clients logged in on the same network. Each registered user logs in to share text messages, files with all other users available in the chat.

1.2 Scope

The Multi-Client Chat Server is a software application that can be applicable in various scenarios. One of main uses of this can be as a communication medium intra organization where clients are logged in on the same network and need to have fast reliable and secure communications. The integrated broadcast function also be used to relay announcements to all the connected clients at once.

1.3 Project Overview

The Multi-Client Chat Server is a software application that facilitates real-time communication between multiple clients over a network. It is built with the following key features:

1. Multi-Client Support The server allows multiple clients to connect simultaneously, enabling a number of users to communicate with one another concurrently.

2. Real-Time Chat Users can send text messages to each other in real time. This feature provides a seamless and responsive chatting experience, making it ideal for instant messaging.

3. Document File Transfer One of the standout features of this project is the ability to transfer documents including audio and video files between clients.

4. Broadcast Functionality

The integrated broadcast function can be used to relay messages to all the connected clients at once.

2. Literature Review

In our review of sections of *Computer Networking: A Top-Down Approach* we delved into the practical implementation of networking concepts using socket programming. This approach allowed us to gain hands-on experience in developing applications that communicate over a network. By understanding the intricacies of socket programming, we were able to grasp the fundamental principles that govern the secure exchange of data between devices. The book provided a holistic understanding of networking enabling us to build a comprehensive skill set in this domain.

Additionally, our study of operating systems, as guided by *Operating System Concepts*, delved into the core mechanisms of concurrent programming. The utilization of mutex locks and threads became central to our exploration of parallelism and resource sharing in operating systems. Through this we gained insights into the challenges and solutions associated with managing multiple threads accessing shared resources concurrently. The study of mutex locks offered a robust method for preventing data corruption and ensuring synchronization, while multithreading enabled us to harness the power of parallel processing for enhanced system performance.

This dual focus on networking and operating systems equipped us with a comprehensive skill set, blending theoretical knowledge with hands-on programming experience in both networking and concurrent systems.

3. Methodology

Since this application problem inherently needed us to come up with network involved concurrent and synchronous operations of multiple clients, C language is used. C has strong support for socket programming Parallelism of which we can take advantage of by implementing multi-threading to run multiple clients at the same time.

Transmission Control Protocol was chosen as the communication medium. We developed a server and client program for the implementation of the project.Leveraging socket programming, we established connections by creating and configuring sockets on the server and client sides. We defined a structure, *client*, to store client information. On the client end we created two threads for sending and receiving messages to facilitate concurrent communication.On connection with a client the server creates a new thread to handle that particular client's requests. Our custom message format, specified using the *message* structure, is the basis for handling communication. On the server side, we maintain a variable called *numUsers* to keep track of the total number of registered users. We've also implemented *addQueue()* and *removeQeueue()* functions to manage adding and removing users from the client list while ensuring thread safety through mutex locks. To enhance the user experience, we utilize *broadcast_join()* and *broadcast_leave()* functions to inform clients when a user joins or leaves the chat.

Our program is designed to facilitate seamless message passing between users, providing a robust chat application.

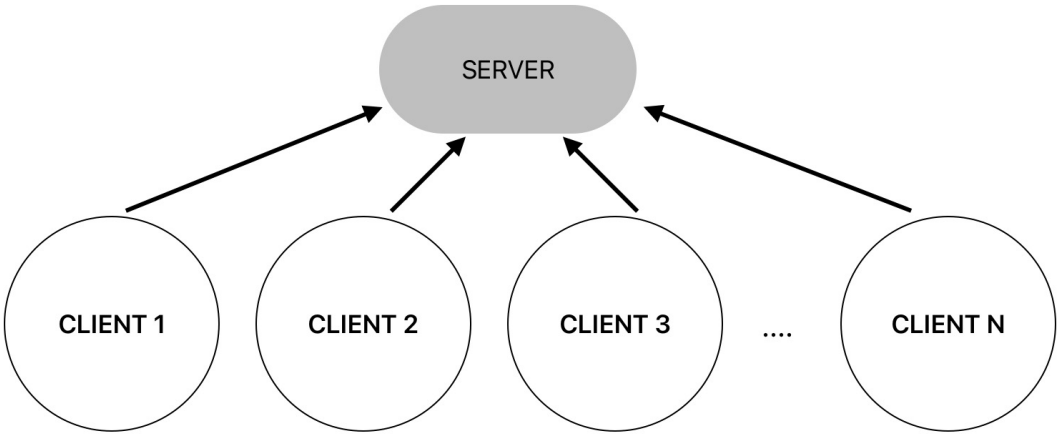

Figure 1: Visualization of clients connecting to the server

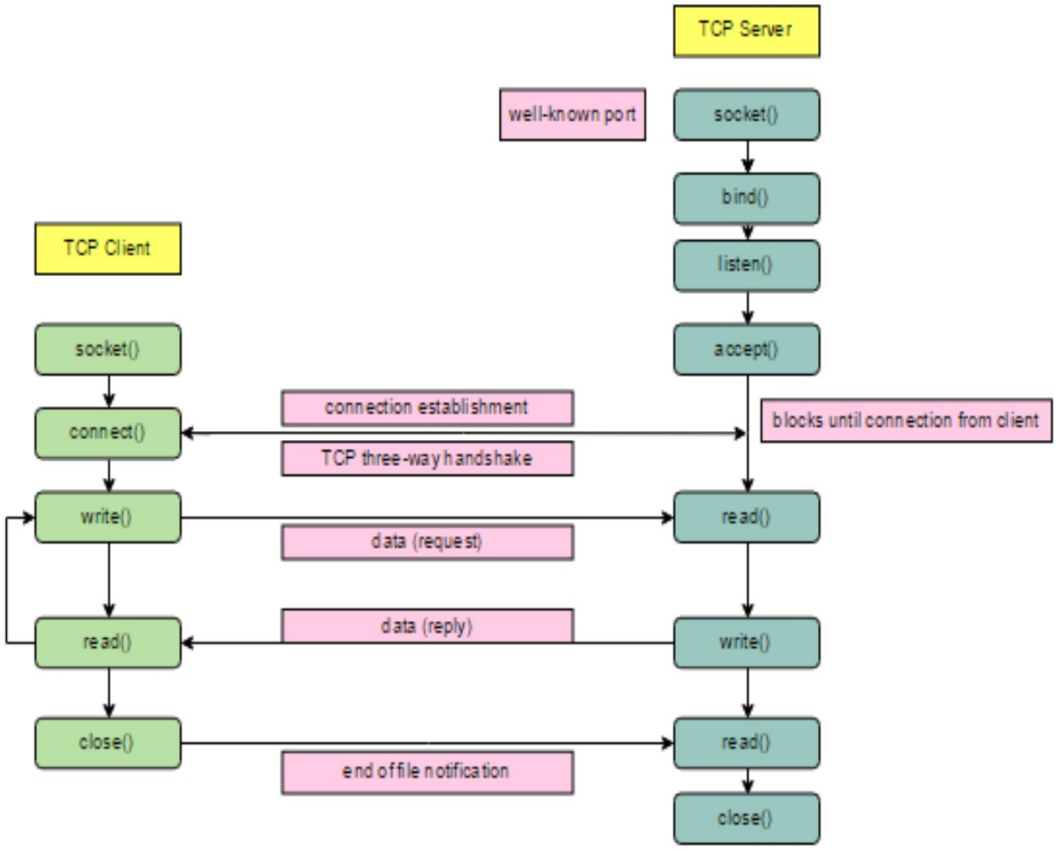

Figure 2: Block Diagram for Socket Programming

4. Implementation

In the implementation of our Multi-Client Chat Server, a central server is established, which creates a primary socket that awaits client connections. Upon the arrival of a new client, a dedicated socket is created to establish a connection with that specific client. The server employs a multi-threaded approach to efficiently handle concurrent client requests. For each connected client, a separate thread is created, and the handleClient() function is invoked.This multi-threaded architecture allows the server to concurrently manage and respond to various client interactions, enhancing the overall responsiveness and efficiency of our Multi-Client Chat Server implementation.

On the Client side of the program, two threads operate concurrently to facilitate a seamless user experience. The first thread is dedicated to process inputs, where users can specify the recipient, type of message, and the content of the message. This information is then transmitted to the server for further processing. Simultaneously, the second thread is responsible for displaying output. This thread continuously retrieves messages from other clients sent via the server.On typing "Close" the chat is ended.

The following subsections show the entire working of the project.

4.1 Registration and Login

The Client registers by creating login creating login credentials which is sent to the server.The server stores this information into database file saved locally on the server machine.

The following is code snippet represents the same :

```
void save_user_db(void)
{
  FILE *fp;
  int ix;

  fp = fopen(user_db_file, "w");
  if (fp == NULL) {
    printf("unable to open user_db_file\n");
    return;
  }
  for (ix=0; ix < numUsers; ix++) {
    fprintf(fp, "%s=%s\n", users[ix]->username, users[ix]->password);
  }
  fclose(fp);
}
```

For convenience the server displays the client registered and the number

Code snippet for the same :

```
void read_user_db()
{
  FILE *fp;
  char *user, *passwd;
  struct idpass ip;
  char buf[120];

  printf("reading user_db\n");

  fp = fopen(user_db_file, "r");
  if (fp == NULL) {
    printf("user_db_file is empty\n");
    return;
  }
  while (fgets(buf, sizeof(buf), fp) != NULL) {
    if ((buf[0] == '#') || (buf[0] == ' ') || (buf[0] == '\t') ||
        (buf[0] == '\n')) {
         continue;
    }
    user   = strtok(buf, "=\n\r");
    passwd = strtok(NULL, "=\n\r");
    if (user == NULL || passwd == NULL) {
        continue;
    }
    if ((strlen(user) > 0) && (strlen(passwd) > 0)) {
      strncpy(ip.id, user, sizeof(ip.id));
      strncpy(ip.pass, passwd, sizeof(ip.pass));
      printf("user=%s\n", user);
      newUser(0, ip);
    }
  }
  printf("user_db: num_users=%d\n", numUsers);
  fclose(fp);
}
```

Upon Login the credentials are sent to the sever and the client is authenticated using the information present in the database.

4.2 Socket Creation

The server is listening port 13055 which is a dedicated port for this network operation. Client calls socket constructor and pass parameters "Internet Address" and "Port number" on which server is running. When the server receives a request from a client it sends back the number after binding it with the client request. Giving the new port number to the client means connections is established. So, at the same time server is listening and writing back a response to the client.

Server Side code :

```
int main(){
    int mainsockfd, clisockfd;
    struct sockaddr_in servadd, cliadd;
    char user[NAME_SIZE],pswd[PWD_SIZE];
    pthread_t tid;

    //creating the main socket that listens for connections.
    mainsockfd=socket(AF_INET, SOCK_STREAM,0);
    if(mainsockfd<0){
        perror("Socket not created\n");
        exit(EXIT_FAILURE);
    }
    servadd.sin_family= AF_INET;
    servadd.sin_addr.s_addr = INADDR_ANY;
    servadd.sin_port=htons(PORT);
    int opt = 1;
    setsockopt(mainsockfd, SOL_SOCKET, SO_REUSEADDR, (void *) &opt,
       sizeof(opt));

    int res= bind(mainsockfd,(struct sockaddr*)&servadd,
       sizeof(servadd));
    if(res==-1){
        perror("Bind error\n");
        exit(EXIT_FAILURE);
    }

    printf("Server listening on port %d\n",PORT);

    listen(mainsockfd,MAX);

    memset(clients,0,sizeof(clients));
    memset(users,0,sizeof(users));
    read_user_db();

    while(1)
    {
        int clilen=sizeof(cliadd);
```

```
        clisockfd = accept(mainsockfd,(struct sockaddr*)&cliadd,
           &clilen);
        if(clisockfd<0){
        perror("Accept error\n");
        }
```

Client Side code :

```
int main(int argc, char * argv[]) {
    int sockfd, reg;
    struct sockaddr_in server_add;
    pthread_t send_thread, recv_thread;
    char *srv_ip;

    if (argc != 2) {
      printf("usage: ./progname <ip_addr>\n");
      exit(0);
    }

    srv_ip = argv[1];

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("Socket not created\n");
        exit(EXIT_FAILURE);
    }

    server_add.sin_family = AF_INET;
    server_add.sin_addr.s_addr = inet_addr(srv_ip);
    server_add.sin_port = htons(PORT);
    int len = sizeof(server_add);
    int res=connect(sockfd,(struct sockaddr *)&server_add,len);
    if (res==-1) {
        perror("Socket not connected");
        exit(1);
    }
```

4.3 Client Operation

The two operations performed by the client are:

- wait for a message to be sent by the client (keyboard event).This is done by the *void send_handler(int *);* function.

```
void send_handler(int *sockfd_ptr) {;
    int sd = *sockfd_ptr;
    message* messageToSend = (message*)malloc(sizeof(message));
    filemsg *data = (filemsg *)messageToSend->msg;
    char buf[MSG_SIZE],top[NAME_SIZE];
    int type;
    struct stat st;
    FILE *file;
    int bytesRead;
    printf("Send message in format: To Type Message\n1:text
       message\n2:text file\n3:audio file\n\n");
    do{
        top[0]=0;
        buf[0]=0;
        type=0;
        scanf("%s %d %[^\n]",top,&type,buf);
        strcpy(messageToSend->from,usn);
        messageToSend->type=type;
        strcpy(messageToSend->to,top);
        switch(type){
            case 1:strcpy(messageToSend->msg,buf);
                int n=write(sd,messageToSend,sizeof(message));
                if(n<0) close(sd);
                printf("Message sent to %s\n",messageToSend->to);
                break;
            case 2:
                strcpy(data->filename,buf);

                if (stat(buf, &st) == 0) {
                    data->size= st.st_size;
                }
                if(data->size>=MSG_SIZE){
                    printf("Limit exceeded\n");
                    return;
                }
                file = fopen(buf, "rb");
                if (file == NULL) {
                    perror("Error opening file");
                    return;
                }
                bytesRead = fread(&(data->buf), 1, data->size,
```

```
                    file);
                if (bytesRead != data->size) {
                    perror("Error reading file");
                    fclose(file);
                    return;
                }
                // Close the file
                fclose(file);
                printf("Sending filename-%s from %s to
                    %s\n",data->filename,messageToSend->from,messageToSend->to);
                write(sd,messageToSend,sizeof(message));
                break;
            case 3:
                strcpy(data->filename,buf);

                if (stat(buf, &st) == 0) {
                    data->size= st.st_size;
                }
                if(data->size>=MSG_SIZE){
                    printf("Limit exceeded\n");
                    return;
                }
                file = fopen(buf, "rb");
                if (file == NULL) {
                    perror("Error opening file");
                    return;
                }
                bytesRead = fread(&(data->buf), 1, data->size,
                    file);
                if (bytesRead != data->size) {
                    perror("Error reading file");
                    fclose(file);
                    return;
                }
                // Close the file
                fclose(file);
                printf("Sending audio filename-%s from %s to
                    %s\n",data->filename,messageToSend->from,messageToSend->to);
                write(sd,messageToSend,sizeof(message));
                break;

        }
    }while(strcmp(buf,"Close")!=0);
    close(sd);


}
```

- Message to be displayed to the interface (server event).This is done by *void recv_handler(int *);* function.

```
void recv_handler(int *sockfd_ptr) {
    int sd = *sockfd_ptr;
    message* messageReceived = (message*)malloc(sizeof(message));
    filemsg *data = (filemsg *)messageReceived->msg;
    struct stat st;
    char *dirname = "shared_folder";
    char path[300];
    FILE *file;
    size_t bytesRead;
    int size,result;
    char command[310];

    while (1) {

        int n=read(sd, messageReceived, sizeof(message));
        if(n<=0){
            close(sd);
            break;
        }
        switch(messageReceived->type){
            case 1:
                if(strcmp(messageReceived->from,"Server")==0){
                    printf("%s\n",messageReceived->msg);
                }
                else{
                    printf("===Message from
                        %s===\n%s\n",messageReceived->from,messageReceived->msg);
                }
                break;
            case 2:
                size=data->size;

                // Create the directory

                if (stat(dirname, &st) != 0) {
                // Directory does not exist, create it
                    if (mkdir(dirname, 0777) == -1) {
                        perror("Error creating directory");
                        return;
                    }
                }

                // Open or create the file within the directory

                snprintf(path, sizeof(path), "%s/incoming_%s",
```

```
            dirname, data->filename);

        file = fopen(path, "wb");
        if (file == NULL) {
            perror("Error opening file");
            return;
        }

        // Read the file into the buffer
        bytesRead = fwrite(&(data->buf), 1, data->size,
            file);
        if (bytesRead != data->size) {
            perror("Error writing file");
            fclose(file);
            return;
        }
        // Close the file
        fclose(file);
        printf("===File from %s===\nFile created in %s
            directory\n",messageReceived->from,dirname);
        break;

    case 3:
        size=data->size;
        // Create the directory

        if (stat(dirname, &st) != 0) {
        // Directory does not exist, create it
            if (mkdir(dirname, 0777) == -1) {
                perror("Error creating directory");
                return;
            }
        }

        // Open or create the file within the directory
        snprintf(path, sizeof(path), "%s/incoming_%s",
            dirname, data->filename);

        file = fopen(path, "wb");
        if (file == NULL) {
            perror("Error opening file");
            return;
        }

        // Read the file into the buffer
        bytesRead = fwrite(&(data->buf), 1, data->size,
            file);
```

```
                if (bytesRead != data->size) {
                    perror("Error writing file");
                    fclose(file);
                    return;
                }
                // Close the file
                fclose(file);

                snprintf(command, sizeof(command), "vlc %s", path);

                // Run the command using system
                result = system(command);

                // Check the result of the command
                if (result == -1) {
                    perror("Error running command");
                    return;
                }
                else {
                    printf("Audio played successfully\n");
                }
                break;

        }

    }
}
```

4.4 Client Handling

Upon Authentication new thread is created for each client witth the *handleclient()* function.This reads the message sent by the client and calls the forward function.

```
void handleClient(void* arg){

    client *thisClient = (client *) arg;
    message *receivedMessage = (message*)malloc(sizeof(message));


    while (1) {

        int n=recv(thisClient->sockfd, receivedMessage,
           sizeof(message),MSG_WAITALL);
        if(n<=0) break;
        if (strcmp(receivedMessage->msg,"Close")==0) {
            broadcast_leave(thisClient->name);
            break;
        }
        if(strcmp(receivedMessage->to,"all")==0){
            broadcast(receivedMessage);
        }
        else {
            forward(receivedMessage);
        }
    }

    close(thisClient->sockfd);
    removeQueue(thisClient->sockfd);
    free(thisClient);
    pthread_detach(pthread_self());
    return;
}

char *user_db_file = "user_db.txt";
```

The *forward()* function sends the message to the specific receiver client.

```
    void forward(message *msg){

    pthread_mutex_lock(&clients_mutex);
    printf("Forwarded from %s to %s\n",msg->from,msg->to);
    for(int i=0;i<numUsers;i++){
        if(clients[i]!=NULL){
        if(strcmp(clients[i]->name, msg->to)==0){
            write(clients[i]->sockfd, msg, sizeof(message));
            break;

        }}
    }
    pthread_mutex_unlock(&clients_mutex);
}
```

4.5 Broadcasting

- *Broadcast join* function informs all connected clients when a new client joins the chat.

```
        void broadcast_join(char nam[50]){

    pthread_mutex_lock(&clients_mutex);
    char buf[MSG_SIZE];
    for(int i=0;i<numUsers;i++){
        if(clients[i]!=NULL && strcmp(clients[i]->name,nam)!=0){
            sprintf(buf, "=== %s has joined the chat ===\n", nam);
            message* msg = (message*)malloc(sizeof(message));
            strcpy(msg->from,"Server");
            strcpy(msg->to,clients[i]->name);
            msg->type=1;
            strcpy(msg->msg,buf);
            write(clients[i]->sockfd, msg, sizeof(message));
        }
    }
    pthread_mutex_unlock(&clients_mutex);
}
```

- *Broadcast leave* function informs all connected clients when a client leaves the chat.

```
        void broadcast_leave(char name[NAME_SIZE]){
    pthread_mutex_lock(&clients_mutex);
    char buf[MSG_SIZE];
    printf("%s logged out\n",name);
    for(int i=0;i<numUsers;i++){
        if(clients[i]!=NULL && strcmp(clients[i]->name,name)!=0){
            sprintf(buf, "=== %s has left the chat ===\n", name);
            message* msg = (message*)malloc(sizeof(message));
            strcpy(msg->from,"Server");
            strcpy(msg->to,clients[i]->name);
            msg->type=1;
            strcpy(msg->msg,buf);
            write(clients[i]->sockfd, msg, sizeof(message));

        }
    }
    pthread_mutex_unlock(&clients_mutex);
}
```

- *Broadcast Message* is a function to send a message to all connected clients except the sender.

```
        void broadcast(message *msg){

    pthread_mutex_lock(&clients_mutex);
    for(int i=0;i<numUsers;i++){
        if(clients[i]!=NULL){
        if(strcmp(clients[i]->name, msg->from)!=0 ){
            write(clients[i]->sockfd, msg, sizeof(message));
        }   }
    }
    pthread_mutex_unlock(&clients_mutex);
}
```

5. Discussion

During the making of this project numerous problems were faced and handled.Some of these are :

- The size of the messages were too big. As a result the messages were sent in chunks.This was specially a problem when sending larger files be it text, audio or video.This was solved by replacing the read command to receive.
- Initially the Broadcast function was not working as intended.It would not send the broadcasted message to all connected clients.This was fixed by checking if the client exists before forwarding the message.

6. Conclusion

The conclusion of this project is a comprehensive Multi-Client Chat Server application with advanced file transfer capabilities, implemented using the high-performance C programming language. The successful outcome establishes a dependable and secure communication platform, allowing multiple clients to connect seamlessly to a central server. Users undergo an authentication process, ensuring the privacy and integrity of communication.

In practical terms, this application serves as a reliable intra-organizational communication tool, particularly suited for scenarios where multiple clients within the same network need fast, dependable, and secure interactions. The integrated broadcast function efficiently relays announcements to all connected clients simultaneously, enhancing the overall communicative experience.

The project's key features, including multi-client support, real-time chat functionality, and document file transfer capabilities for various file types, contribute to its versatility and utility. The integration of networking, file handling, and security aspects positions this project as a comprehensive solution that addresses the modern communication needs of organizations and users alike.

7. Future Works

Strengthening the authentication process is critical for bolstering the security and integrity of the system. One avenue for future work involves the implementation of advanced authentication protocols, such as multi-factor authentication (MFA), which adds an additional layer of security by requiring users to provide multiple forms of identification. This could include a combination of passwords, bio-metric data, or one-time codes, significantly reducing the vulnerability to unauthorized access.

For the future development of our project the creation of an interactive user interface, potentially in the form of a dedicated application would be a necessity. This evolution would enhance the accessibility and user-friendliness of our system, providing a more intuitive platform for users to interact with the functionalities we've implemented.

Developing an app could streamline the user experience, allowing individuals to seamlessly navigate through the features and access the network or operating system functionalities with ease. Such an interface could feature graphical representations, real-time status updates, and perhaps even interactive controls, significantly augmenting the overall usability of the system.

8. References

1. https://github.com/coduri/SocketChatRoom/tree/main
2. Computer Networking: A Top-Down Approach (6E), Jim Kurose
3. Operating System Concepts (9E), Abraham Silberschatz, Greg Gagne, Peter Baer Galvin