Ain Shams University - Faculty of Engineering

Department of Computer Engineering

Computer Networks – CSE351

Peer-to-Peer Multi-User Chatting Application

Abdelrahman Khaled Sallam	20P8307
Jumana Yasser Mahrous Mostafa	20P8421
Adham Hatem Hanafy	20p8384
Ahmed Tarek Abdellatif Shendy Amer	20P8417

Table of Contents

Table of Figures and Tables

I. PHASE 1

1.User Authentication Mechanism

1.1 User Login:

- **Functionality**: Users can log in using their registered username and password.
- **Implementation**:
    1. User provides credentials (username, password).
    2. Verify the credentials against stored and encrypted passwords.
    3. Generate and issue a session token upon successful authentication.

1.2 User Registration:

- **Functionality**: Users can create an account with a unique username and password.
- **Implementation**:
    1. User provides credentials (username, password).
    2. Store encrypted passwords and associated usernames securely (using hashing algorithms like bcrypt).
    3. Check for username uniqueness during registration.

1.3 Password Security:

- **Functionality**: Ensure password security.
- **Implementation**:
    1. Encourage strong password policies (length, complexity).
    2. Store passwords securely using salted hashing algorithms (e.g., bcrypt).
    3. User authentication will involve a secure username and password exchange over TCP.

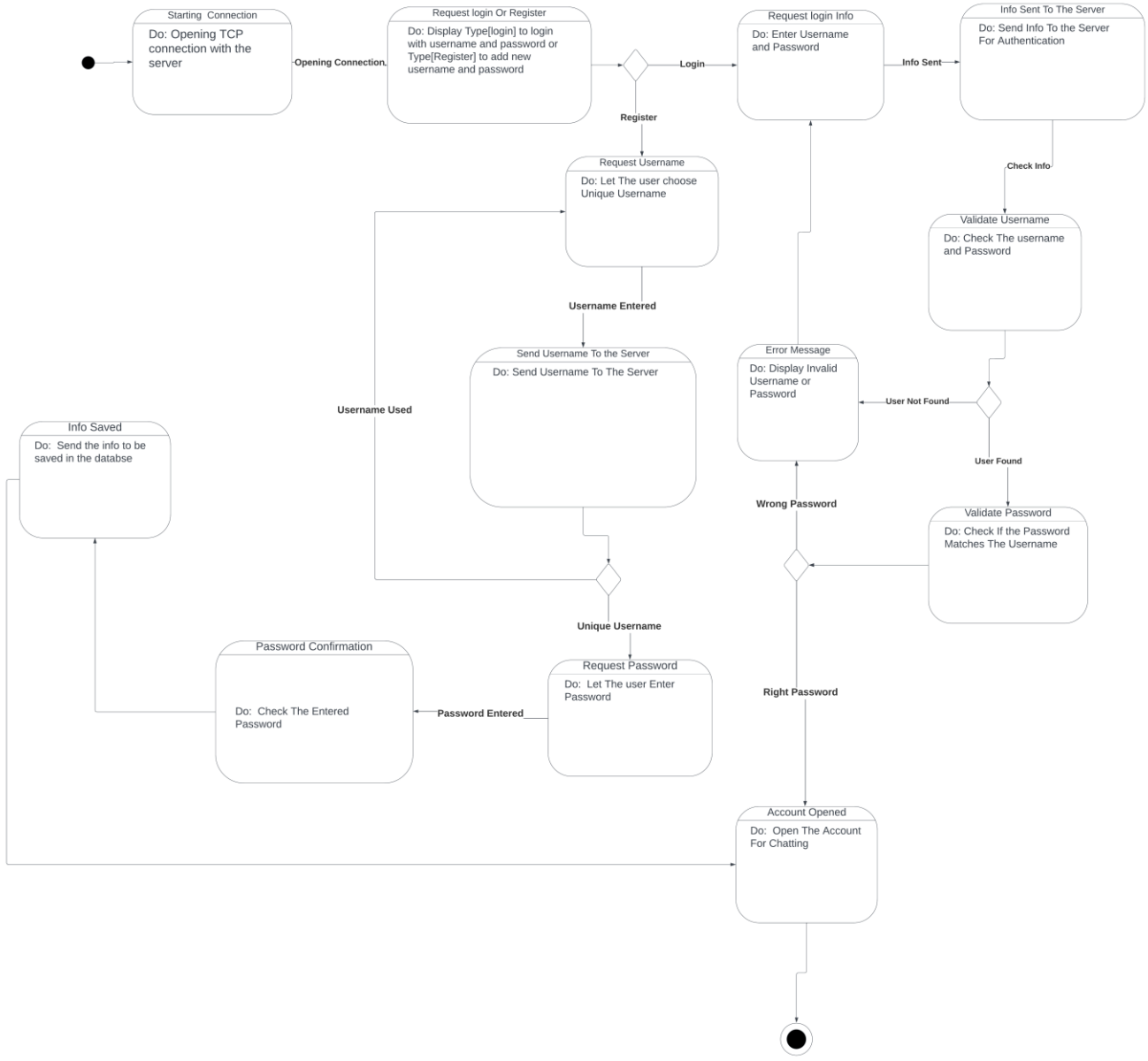


Figure 1- User Authentication State Diagram

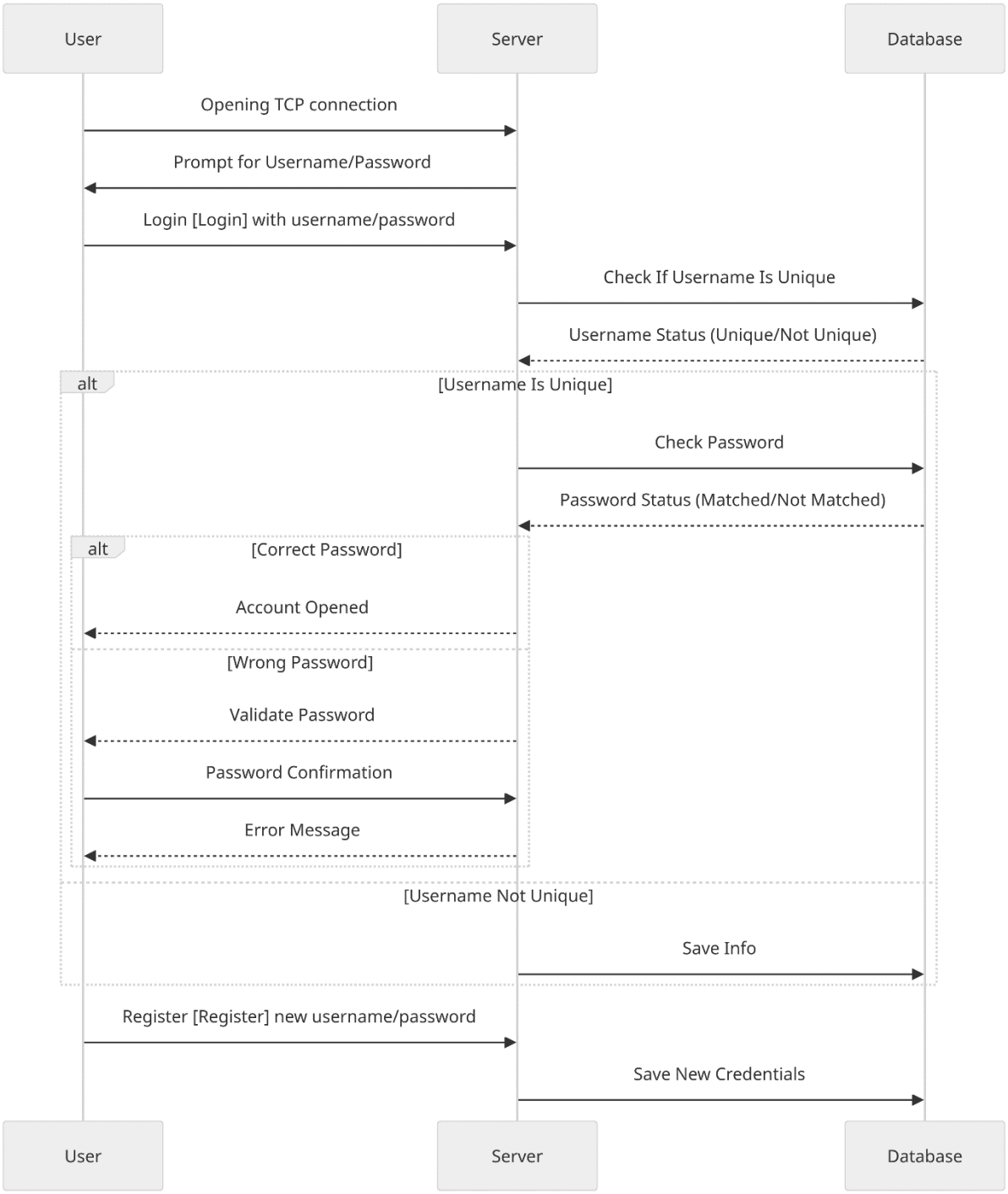


Figure 2: Sequence Diagram

2. Outline the overall project scope, including the key functions

The project scope includes the development of a robust Peer-to-Peer Multi-User Chatting Application with user authentication, client-server architecture, chat room functionality, group messaging, one-to-one chat sessions, message formatting, error handling, UI enhancements, documentation, testing, and scalability considerations. By implementing these key functionalities, the application aims to provide users with an intuitive and reliable platform for text-based communication.

The overall project scope for the Peer-to-Peer Multi-User Chatting Application includes the following key functionalities:

2.1. User Authentication:

- Users can create an account with a unique username and password.
- Users can log in using their registered username and password.

2.2. Basic Client-Server Setup:

- Implement a server application to handle multiple client connections.
- Clients can connect to the server using a client application.
- Users can see a list of online users.

2.3. Chat Room Functionality:

- Users can create new chat rooms.
- Users can join existing chat rooms.
- Users can see a list of available chat rooms.

2.4. Group Messaging in Chat Rooms:

- Users can send messages to everyone in a chat room.
- Users can receive messages from other users in the chat room.
- Users receive notifications for new messages.

2.5. One-to-One Chat Functionality:

- Users can initiate one-to-one chat sessions with other users.
- Users can send private messages to other users.
- Users can receive private messages from other users.
- Users receive notifications for new private messages.

2.6. Message Formatting and Features:

- Support basic text formatting (e.g., bold, italics) in messages.
- Users can format their messages to emphasize certain words.
- Users can share hyperlinks in messages, which can be clicked to open a browser.

2.7. Error Handling and Resilience:

- Implement robust error handling for unexpected scenarios.
- Users receive meaningful error messages for troubleshooting.
- Automatically reconnect users in case of network interruptions.

2.8. User Interface (UI) Enhancements:

- Develop a command-line interface for simplicity.
- Provide a clean and intuitive command-line interface.
- Use color-coded messages for better visual distinction.

2.9. Documentation:

- Create user documentation covering installation, configuration, and usage.
- Provide a comprehensive guide for users to set up and use the application.
- Create technical documentation detailing system architecture, protocols, and codebase structure.

3. System architecture, specifying components and their interactions:

3.1 User Interface:

Functionality: Handles user interface and user input/output

Components:

- Command-line interface:
- Interface for users to interact with the application.
- User Interface Controller: Manages user input and output.

3.2 Application Logic Layer:

Functionality: Manages the application's core logic and business rules.

Components:

- Chat Room Manager: Manages the creation and joining of chat rooms.
- Message Manager: Handles group messaging and one-to-one messaging.
- Error Handler: Manages error handling and resilience

3.3 Security Layer:

Functionality: Ensures the security and integrity of the application, protecting against unauthorized access, data breaches, and other security threats

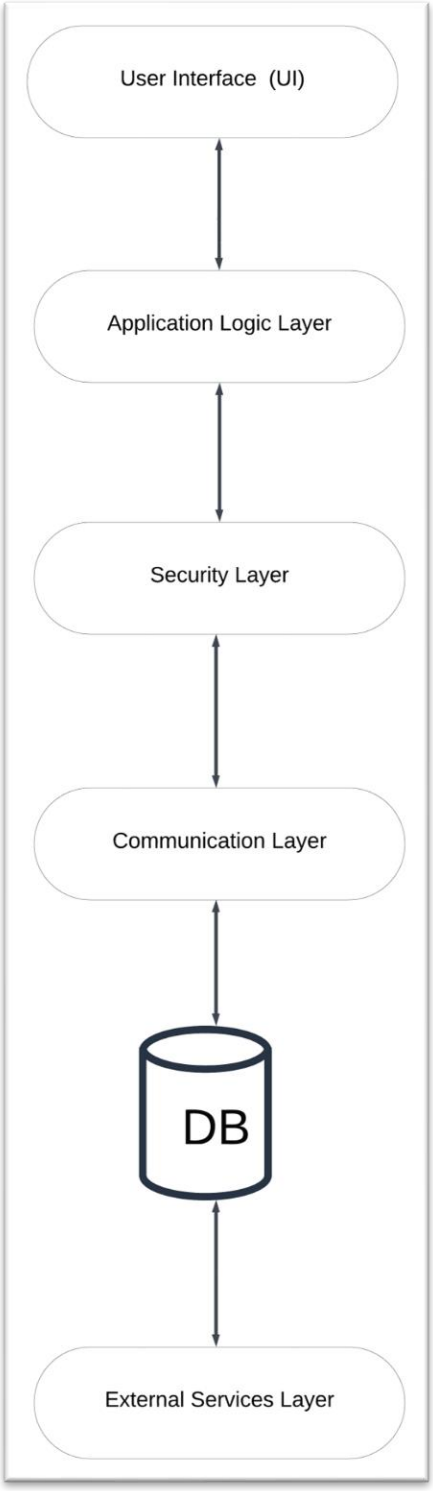


Figure 3: System Architecture

Components:

- Authentication Manager: Verifies the identity of users and ensures that only authorized users can access the system.
- Encryption and Decryption Module: Ensures the confidentiality of data during transmission and storage

3.4 Communication Layer:

Functionality: Handles communication between clients and the server.

Components:

- Establishes connections to the server and communicates user requests
- Listens for incoming connections and manages client connections
- Manages the communication protocol between clients and the server
- Routes messages to the appropriate destinations based on the message type

3.5 Data Access Layer:

Functionality: Manages data storage and retrieval

Components:

- User Database: Stores user credentials for authentication.
- Chat Room Database: Stores information about chat rooms and their participants.
- Message Database: Stores chat messages.

3.6 External services Layer:

Functionality: Manages external services if needed

Components:

- Integration with external authentication services

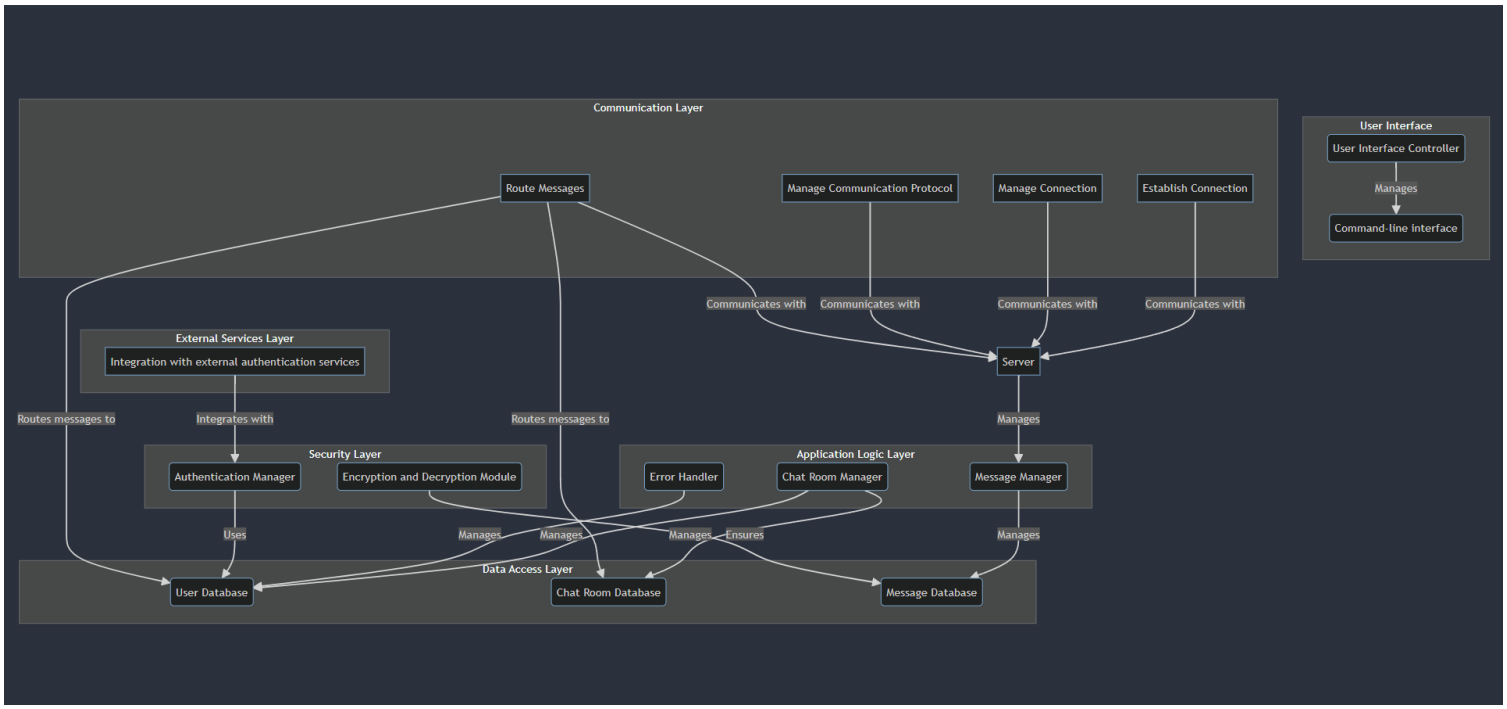


Figure 4- Component Diagram

4. Communication Protocols for Client-Server and Peer-to-Peer Interactions.

4.1 Client-Server Interactions

4.1.1. User Registration and Login:

- Client: sends a registration request with the user's credentials to the server in case an account doesn't already exist. If the user already has an existing account, the client sends a login request with the user's credentials in a JSON format.
- Server: the server validates the credentials if an account already exists. If an account doesn't already exist, the server verifies the username is unique and saves the credentials. The server then responds with a success or failure message is JSON format.

4.1.2. User Discovery and Management:

- Client: the client can send a search request to the server looking for a certain user.
- Server: the server can then respond to this request connecting the first client to the other client he/she wants to connect to. The server can also provide the client with a list of online users and their status. This message can be sent in a JSON format. The server can store this information in a data structure such as a dictionary with keys for each user and their status as values in the data structure used (a dictionary in our case)

4.1.3. Message Exchange:

- Client: when a user sends a message, the client serialized=s the message in a JSON format then proceeds to send it to the server using a TCP connection for maximum security.
- Server: The server will then deserialize the message to extract the recipient's username, the sender's username, message and message content and send the message to the appropriate intended recipient using another TCP connection.

4.1.4. Status Updates and Notifications:

- Client: whenever a user's status gets updates (like their availability status or typing indicators), the client sends a status update to the server in JSON format.
- Server: the server will then broadcast these updates in the data structure used to store this information (a dictionary in our case) then send notifying messages to other clients in JSON formats. The server also sends notifying messages to clients who receive messages from other clients.

4.1.5. Error Handling and Security:

- When an issue faces the server regarding any request made by any of the clients, the server sends error messages to the affected clients. These issues may include errors in message delivery, errors in authentication, connection problems, or other issues.
- Encryption and authentication are incorporated in the protocol to ensure maximum privacy and security and to maintain users' data integrity.

4.1.6. User Logout:

- Client: can send a logout request to the server when they want to leave.
- The server: removes the user from the chatroom and updates others.

4.1.7. Data Format:

- Peers exchange messages in a specific data format, such as JSON or XML, to ensure structured and easy-to-parse data.

4.2 Peer-to-Peer Interactions

4.2.1. Centralized Index Approach (Registry):

- A dedicated server or a set of servers act as a centralized registry that maintains information about the available resources, such as peers, files, or services in the network. Consists of a centralized 'registry' and a chatting application between users.

4.2.2. Peer Discovery:

- Peers: register with the registry upon login or signup. When peers are logged in, they essentially announce their availability and proceed to open ports for communication with other peers in JSON format.
- The centralized registry we use will maintain a list of all online (logged-in) peers and store their open ports and IP address in a data structure like a dictionary.

4.2.3. Message Exchange:

- Peers establish a UDP connection for peer-to-peer communication.
- Peers can initiate a chat by sending a connection request to the intended recipient's open port and IP address in a JSON format.
- Once the connection is established, peers can exchange messages directly in JSON format.

4.2.4. Signaling Server:

- A signaling server can be used to facilitate the initial connection between peers, after which direct communication is established using UDP.
- The signaling server sends connection requests and connection confirmation messages to peers in JSON format.

4.2.5. Data Format:

- Peers exchange messages in a specific data format, such as JSON or XML, to ensure structured and easy-to-parse data.

4.2.6. Security Measures:

- Encryption and authentication measures are taken for maximum privacy and security.

4.2.7. Graceful Disconnection:

- When a peer-to-peer chat ends, peers send a disconnect message.
- The registry updates the user status and informs other peers about the disconnection.

5. Cost Analysis:

Cost Factor	Description	Estimated Cost	Notes
Infrastructure Cloud Services	Virtual machines, storage, and data transfer	$650/month	Based on Chosen Cloud Provider
Database Database Services	Managed database service usage	$250/month	Consider red/Write Operations
Authentication Services Third Party Authentication	API usage for authentication	$70/month	If Using external Authentication
Message Queue Message Queue Service	Managed message queue service	$100/month	Based on Message throughput
Security and Encryptions SSL Certificates	Secure data transmission with SSL/TLS	$100/year	
Notification Service	Push notification service	$50/month	If Implemented
User Interface Frontend Hosting	Hosting and data transfer for the UI	$40/month	If Hosted Seperatley
Development and Maintenance Development Tools and Salaries	Software licenses, tools, and libraries, In-house developers' salaries	$700/month 15,000/month	
Monitoring and Analytics Monitoring Tools	Tools for tracking system performance	$50/month	
Backup and Disaster Recovery Backup Services	Implementation of backup solutions	$70/month	
Scaling Auto-Scaling	Implementation of auto-scaling	$50/month	If Implemented
Network Costs Data Transfer	Costs for data transfer between components and users	$30/month	

Licensing Third-Party Software	Licensing fees for third-party software	$200/month	If Applicable

Table 1: Cost Analysis

Total Estimated Monthly Cost: $17,360

Notes:

- Prices are assumed and do not reflect actual market rates.
- Ensure to review and validate pricing based on the chosen cloud provider and services.
- Actual costs may vary based on usage patterns, optimization strategies, and any changes in pricing by the service providers.

II. PHASE 2

1. Server

First, these are the libraries we used:

```
import json
import socket
import sqlite3
import threading
import colorama
import bcrypt
from colorama import *
```

Figure 5: imports

Json → for the conversion of JSON files.
Socket → to establish the connection between the client and server.
Sqlite3 → for the database.
Threading →to be able to handle multiple users at the same time.
Colorama →for the color-coded CLI.
Bcrypt → used for hashing.

```
HEADER_LENGTH = 10
Flag = True
colorama.init(autoreset=True)

  # Set up the Connection for the Server Using Sockets
  IP = socket.gethostbyname(socket.gethostname())
  PORT = 1234
  server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
  server_socket.bind((IP, PORT))
  server_socket.listen()

  socket_list = [server_socket]
  clients = {}
  print(f'{Fore.RED}Listening for connection on {IP}:{PORT}{Fore.RESET}')
```

Figure 6: setting up the connection for the server using sockets

In this code snippet (figure 2), we establish the TCP connection from the server's side using the port number and the IP address of the device. We do this by first obtaining the host's IP address and setting the port number to be 1234. We then create a TCP socket using the IPV4 address family and set a socket option to allow the reuse of that same address. We do this to ensure that the socket is still able to bind to the address even if it's in a TIME_WAIT state. We then bind the server socket to the IP address and port number we already obtained/defined. After that, we set the server socket to listening mode, waiting for client connections. To keep track of all sockets, we create a llist called "socket_list" that stores all these sockets and another list called "clients" to store information about the client's connection details. Finally, we print a message that indicates that the server is listening for connections in red.

```
HEADER_LENGTH = 10

# Function Used to Receives Msgs From the Client
def receive_message(client_socket):
    try:
        message_header = client_socket.recv(HEADER_LENGTH)
        if not len(message_header):
            return False
        message_length = int(message_header.decode('utf-8').strip())
        return {'header': message_header, 'data': client_socket.recv(message_length)}
    except:
        return False
```

Figure 7: a function used to receive messages from the client

In the code snippet above (figure 3), we created a function that receives messages from the client. We do this by first taking the client socket object as an argument. Then, in a try-except block for exception handling, we receive the message header in the "message_header" variable. We have predefined our HEADER_LENGTH variable to be 10. We then check for an empty header using an if condition that returns "false" when the message header in empty, indicating that no message was received. Afterwards, we decode the received message header from bytes to string using UTF-8 encoding then remove any leading or trailing whitespace and convert the message length into an integer. We then receive the actual data using the message length we extracted and return a dictionary containing two important key pairs: the original header message received as "header" and the actual message data received as "data". In case any exceptions are raised, the function returns "false" to indicate that we failed to receive the message.

```python
# Function To Add the user in the Database in  case of Register
def add_user(username, password):
    # Connecting the Database File
    connection = sqlite3.connect("AppData.db")
    cursor = connection.cursor()

    # Generating Salt For The salted Hashing using bcrypt
    salt = bcrypt.gensalt()
    cursor.execute("INSERT INTO AppData(username, password,salted_password) VALUES (?, ? ,?)",
                   (username, bcrypt.hashpw(password.encode("utf-8"), salt), salt))
    connection.commit()
    connection.close()
```

Figure 8: a function that adds users to the database when registering

In the code snippet above (figure 4), we created a function that adds users to the database when the user registers for a new account. To do this, we took the username and password as parameters then established a connection to the SQLite database file called “AppData.db.” then, to allow interaction with the database, we created a cursor object. Afterwards, we generated a salt (a random value used to enhance the security of password hashing) for hashing using the “bcrypt” library. We then hash the actual password using the generated salt and store the resulting hashed password in the “hashed_password” variable. After that, we execute an SQL query that inserts the user’s information into the AppData table. We then commit these changes to the database and close the connection with the database.

```python
# Functions used to handle single Client
def handle_client(client_socket, address):
    global Flag
    user = receive_message(client_socket)
    if user is False:
        return
    clients[client_socket] = user
    Flag = False
    print(f'{Fore.RESET}Accepted connection from {Fore.LIGHTBLUE_EX}{address} {Fore.RESET}username: '
          f'{Fore.LIGHTBLUE_EX}{user["data"].decode("utf-8")}')

    # Accepting messages algorithm
    while True:
        if client_socket in clients and not Flag:
            password = receive_message(client_socket)
            user_object = clients[client_socket]
            result = client_socket.recv(1024).decode('utf-8')
            if result == 'register':
                add_user(user_object['data'].decode('utf-8'), password['data'].decode('utf-8'))
                print('Registered for user:{0}'.format(user_object['data'].decode('utf-8')))
            Flag = True
        else:
            message = receive_message(client_socket)
            # if no message then the connection is closed betwwen this client  and the server
            if message is False:
                print(f'{Fore.RED}Connection closed from: '
                      f'{clients[client_socket]["data"].decode("utf-8")} {Fore.RESET}')
                try:
                    socket_list.remove(client_socket)
                except ValueError:
                    pass
                del clients[client_socket]
                break

            user_object = clients[client_socket]
            sender_username = user_object['data'].decode('utf-8')
            # Exchanging the messages in a JSON format
            received_json = message['data'].decode('utf-8')
            received_data = json.loads(received_json)
            received_username = received_data['username']
            received_message = received_data['message']

            print(f"{Fore.RESET}Received message from {received_username}: {Fore.RED}{received_message} {Fore.RESET}")

            json_message = json.dumps({'username': sender_username, 'message': received_message})

            for client_sock in clients:
                if client_sock != client_socket:
                    client_sock.send(
                        f'{len(json_message):<{HEADER_LENGTH}}'.encode('utf-8') + json_message.encode('utf-8')
                    )

    client_socket.close()
```

Figure 9: a function that handles receiving messages from each individual client in multi-client programs

In the code snippet above (figure 5), we created a function that handles receiving messages from a single client in our multi-client server. To do this, we first took the client object and the client's IP address as arguments and we receive the message sent from the client with the client's information, including the client's username. We then store this information in a dictionary called "clients" with the client socket as the dictionary's key. After that, to indicate that a connection

has been accepted, we print a message that displays the client's IP address and the received username. Afterwards, we enter a message-handling loop where we handle message exchange with the client. In this loop, we check if the client is in the "clients" dictionary and that a flag isn't set. If true, we'll receive the client's password and if the result is "register", we call the "add_user" function to register the user in the database. When no special action in required, we will continue to handle messages regularly from the client. If no messages are received, we assume the connection is closed, and we remove the client from the server's tracking list and break out of the loop. When a message is received, we parse the received JSON message and print the sender information along with the broadcasted message to all other connected clients. Finally, we close the connection with the client socket when we exit the loop.

```
# Handling the Multithreading in The file to make every client on a Separate thread
while True:
    client_socket, client_address = server_socket.accept()
    thread = threading.Thread(target=handle_client, args=(client_socket, client_address))
    thread.start()
```

Figure 10: handling multithreading- making every client on a separate thread

In the code snippet above (figure 6), we handle multithreading to enable multiple users to send messages. To do this, we get into the while loop and create client sockets one at a time and then we create a thread for each client socket created by calling the function "handle_client" with the arguments "client_socket" and "client_address". Finally, we start the thread so that every client will be running on his own thread.

2. Client

```
import json
import socket
import sqlite3
import sys
import msvcrt
import errno
import bcrypt
import colorama
from colorama import *
```

Figure 11: imports

Json → converting and decoding the messages into Json files
Socket → to create client objects
Sqlite3 → to connect with the database
Sys → to exit when an exception is raised
Msvcrt → ensures that when a message is sent, it is received on all clients' ends and not just a single client.
Errno → for exception-handling
Bycrypt → used for hashing
Colorama → to color-code the messages printed in the cmd

```
colorama.init(autoreset=True)
HEADER_LENGTH = 10
# Set up the connection between the client and the server
IP = socket.gethostbyname(socket.gethostname())
PORT = 1234
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((IP, PORT))
client_socket.setblocking(False)
```

Figure 12: setting up the connection between the client and the server

In the code snippet above (figure 8), we first set up the connection between the client and the server using the IP address and port number specified. We connect the client to the server using the IP address and port number of the server.

```
# Connecting the Database
connection = sqlite3.connect("AppData.db")
cursor = connection.cursor()
```

Figure 13: connecting to the database

In the code snippet above (figure 9), we just connect to the database and allow interactions with it.

```python
# The Register Function that checks that the username is Unique
def user_register(username, password):
    cursor.execute("SELECT * FROM AppData WHERE username=?", (username,))
    if cursor.fetchone() is None:
        username_header_register = f'{len(username):<{HEADER_LENGTH}}'.encode('utf-8')
        username_text = username.encode('utf-8')
        client_socket.send(username_header_register + username_text)
        Password_header_Register = f'{len(password):<{HEADER_LENGTH}}'.encode('utf-8')
        password_text = password.encode('utf-8')
        client_socket.send(Password_header_Register + password_text)
        client_socket.send('register'.encode('utf-8'))
        print("Welcome "+ username + "!")
        return True
    else:
        print(f'{Fore.RED} Username is not unique!')
        return False
```

Figure 14: a function that checks if the username is unique

in the code snippet above (figure 10), we created a user_regsiter function that checks whether or not the username is unique. To do this, we execute an SQL query that queries the given username. If the username already exists, we print an error message indicating that the username is not unique. However, if the username is unique, we will send the username and password of the client to the server by encoding them in messages with appropriate headers. If the username is unique, we return true. If not, we return false.

```python
# The login function that checks the username and the Password
def user_login(username, password):
    cursor.execute("SELECT password, salted_password FROM AppData WHERE username=?", (username,))
    result = cursor.fetchone()
    if result:
        stored_hashed_password = result[0]
        stored_salt = result[1]
        entered_hashed_password = bcrypt.hashpw(password.encode('utf-8'), stored_salt)
        if entered_hashed_password == stored_hashed_password:
            username_header_login = f'{len(username):<{HEADER_LENGTH}}'.encode('utf-8')
            username_text = username.encode('utf-8')
            client_socket.send(username_header_login + username_text)
            Password_header_login = f'{len(password):<{HEADER_LENGTH}}'.encode('utf-8')
            password_text = password.encode('utf-8')
            client_socket.send(Password_header_login + password_text)
            client_socket.send('login'.encode('utf-8'))
            print("Welcome "+ username + "!")
            return True
        else:
            print(f'{Fore.RED} invalid username or password!')
            return False
    else:
        print(f'{Fore.RED}invalid username or password!')
        return False
```

Figure 15: a function that validates the credentials inserted while logging in

In the code snippet above (figure 11), we created a function that checks the username and password when the client wants to login. To do this, we first take the client's username and password as arguments and execute an SQL query that queries for the password for the corresponding username given. If a result to the query exists, we will store the hashed password in a variable called "stored_hashed_password" and the salt in another variable called "stored_salt". We then check if the stored hashed password is the exact same hashed password

entered. If they are equal, we will send the username and password in messages to the server so that the server could display the username it got connected to. We also send “login” to the server to log the user in. however, if the username or password are invalid, we print an error message indicating that they’re invalid.

```
# Choosing the Login Or Register For the Client
choice = input("1.login\n2.register\n")

if choice == '1':
    # This While Loop makes enter the username and Password again if the data is not correct
    LogingResult = False
    while not LogingResult:
        ClientUsername = input(f"{Fore.RESET}Enter Username: {Fore.LIGHTBLUE_EX}")
        ClientPassword = input(f"{Fore.RESET}Enter Password: {Fore.LIGHTBLUE_EX}")
        LogingResult = user_login(ClientUsername, ClientPassword)

else:
    RegisterResult = False
    while not RegisterResult:
        ClientUsername = input(f"{Fore.RESET}Enter Unique Username:{Fore.LIGHTBLUE_EX} ")
        ClientPassword = input(f"{Fore.RESET}Create Password: {Fore.LIGHTBLUE_EX}")
        RegisterResult = user_register(ClientUsername, ClientPassword)
```

Figure 16: allowing the user to choose whether he wants to register or login

In the code snippet above (figure 12), we let the user choose if he wants to login or register. If he wants to login, we first initialize a variable called “LogingResult” to false. We then enter a while loop that keeps looping until LogingResult is true. Inside the loop, we let the user enter his username and password. We then call the user_login functions and pass the username and password as parameters to check if login is successful. If login is successful, LogingResult is set to true and we exit the while loop. However, if login was unsuccessful, LogingResult is set to false and we iterate into the while loop once more so that the user keeps trying to login till logging in is successful. Similarly, if the user chooses to register, we set a variable called “RegisterResult” to false. We then enter a while loop that could only be exited when RegisterResult is true. We then let the user enter the username and password he wants to register with. Afterwards, we call the user_register function to validate the user’s credentials (to make sure that the username is unique). If registering was successful, RegisterResult is set to true and we exit the while loop. If it was unsuccessful, RegisterResult is set to false and we iterate once more in the while loop allowing the user to enter a different username and password until registration is successful.

```
while True:
    # Handling the sending of the Messages in JSON format
    if msvcrt.kbhit():
        message = input(f'{Fore.RESET}{ClientUsername} ->{Fore.LIGHTMAGENTA_EX} ')
        if message:
            json_message = json.dumps({'username': ClientUsername, 'message': message})
            message_header = f'{len(json_message):<{HEADER_LENGTH}}'.encode('utf-8')
            client_socket.send(message_header + json_message.encode('utf-8'))

    try:
        while True:
            message_header_received = client_socket.recv(HEADER_LENGTH)
            if not message_header_received:
                print('Connection closed by the server')
                sys.exit()

            message_length = int(message_header_received.decode('utf-8').strip())
            received_json_message = client_socket.recv(message_length).decode('utf-8')
            received_data = json.loads(received_json_message)
            print(f'{received_data["username"]} -> {received_data["message"]}')

    # Handling Exceptions Like if the Server Closed
    except IOError as e:
        if e.errno != errno.EAGAIN and e.errno != errno.EWOULDBLOCK:
            print(f'{Fore.RED}Reading error:{str(e)} ')
            sys.exit()
        continue

    except Exception as e:
        print(f'{Fore.RED}Reading error:{str(e)} ')
        sys.exit()
```

Figure 17: handling sending messages to all clients through the server

in the code snippet above (figure 13), we handle sending the messages to all connected clients through the server. To do this, we first check if the client is sending the messages to all clients. If so, the client sends the message to the server and the server then sends the message to all other clients. If a message is received to the client, we first check if a message is received. If a message is not received, we print a message that indicates that the connection was closed by the server. If a message is received, we calculate the length of the message and decode the JSON file of the message we received and print the received message along with the corresponding sender's username on the cmd. This is all done in a try-except block so that if any exception is raised (i.e an IOError), we print the error on the console.

3. Database

```python
import sqlite3

# Creating The Database Table
connection = sqlite3.connect("AppData.db")
cursor = connection.cursor()

cursor.execute("""
CREATE TABLE IF NOT EXISTS AppData(
id INTEGER PRIMARY KEY,
username VARCHAR(255) NOT NULL,
password VARCHAR(255) NOT NULL,
salted_password VARCHAR(255) NOT NULL
)
""")

connection.commit()
connection.close()
```

Figure 18: creating the database table

In the code snippet above, we just create the database.

4. Outputs

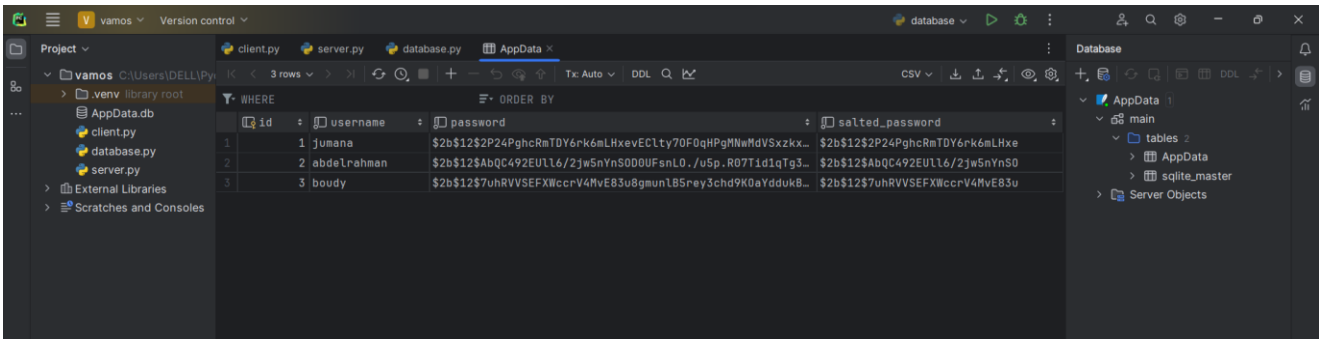


Figure 19: A sample of the database contents

This is the database table before registering any new clients. The current clients in the database were registered in a previous run.

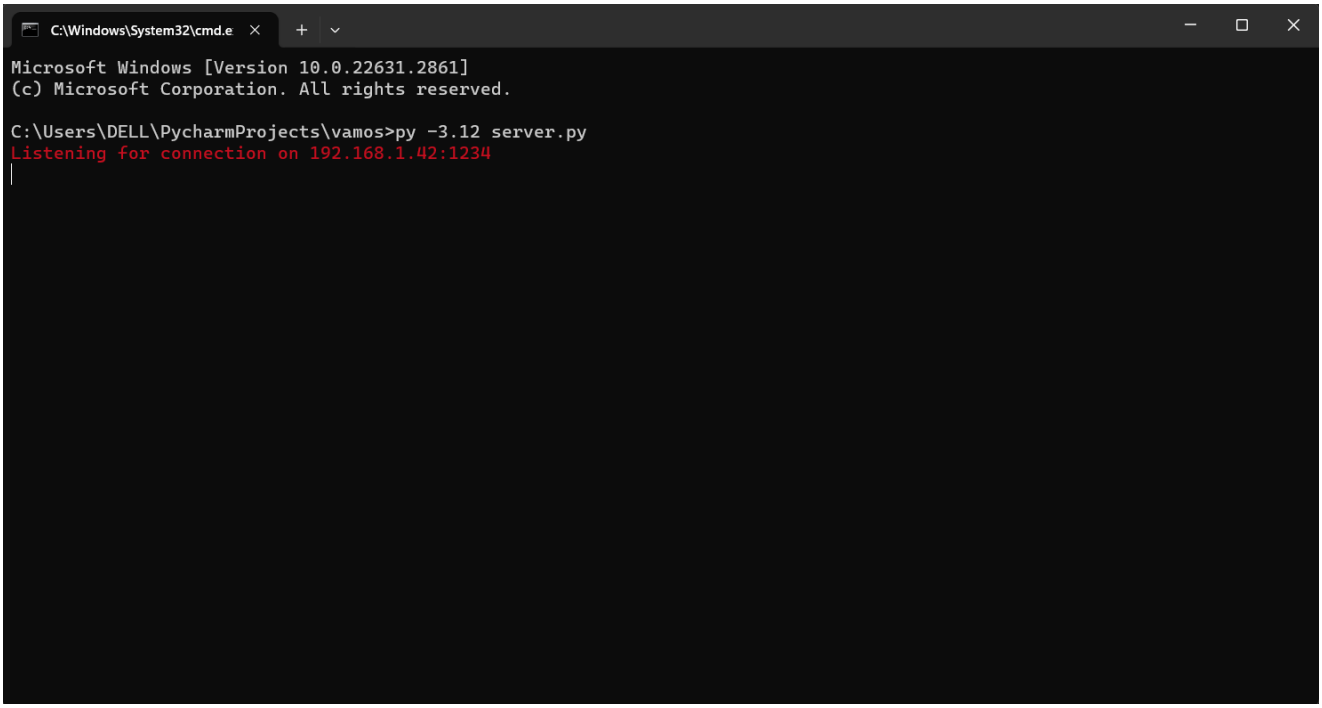


Figure 20: Running the server

Here we ran the server. The server is currently waiting for any client to connect.

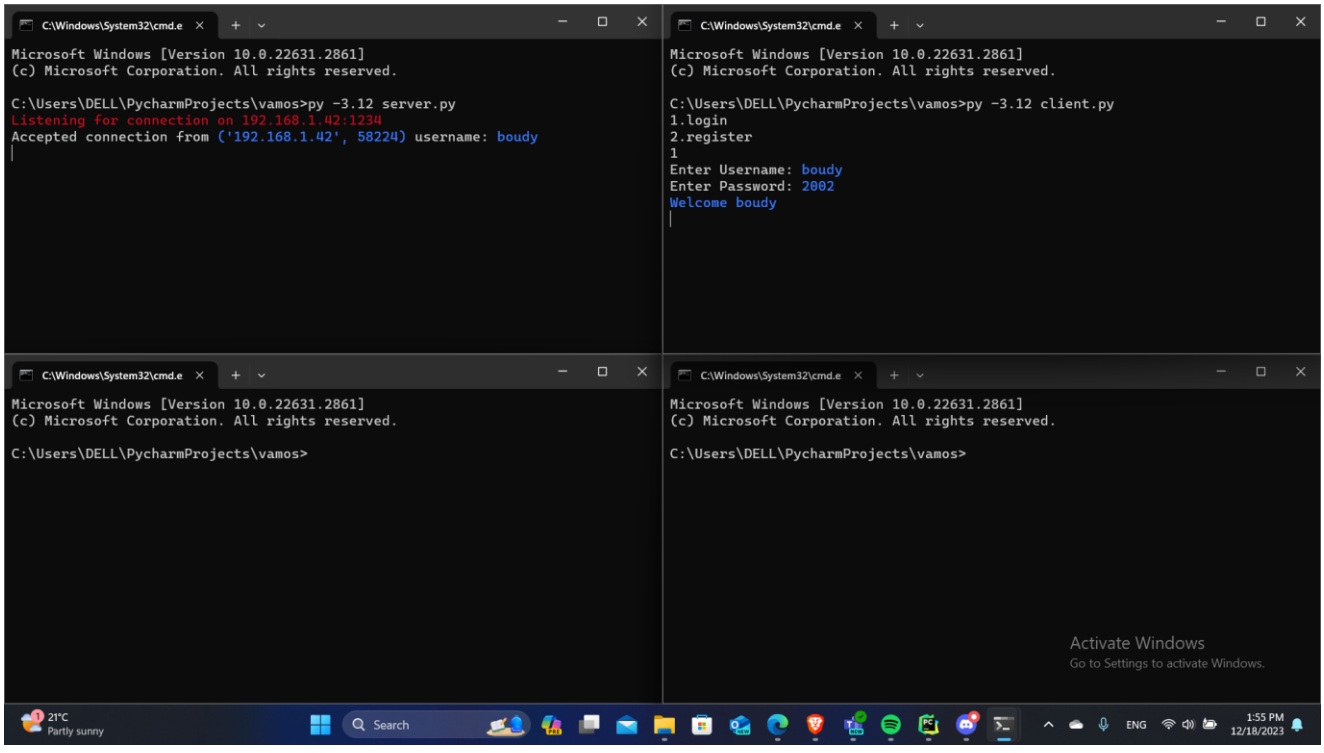
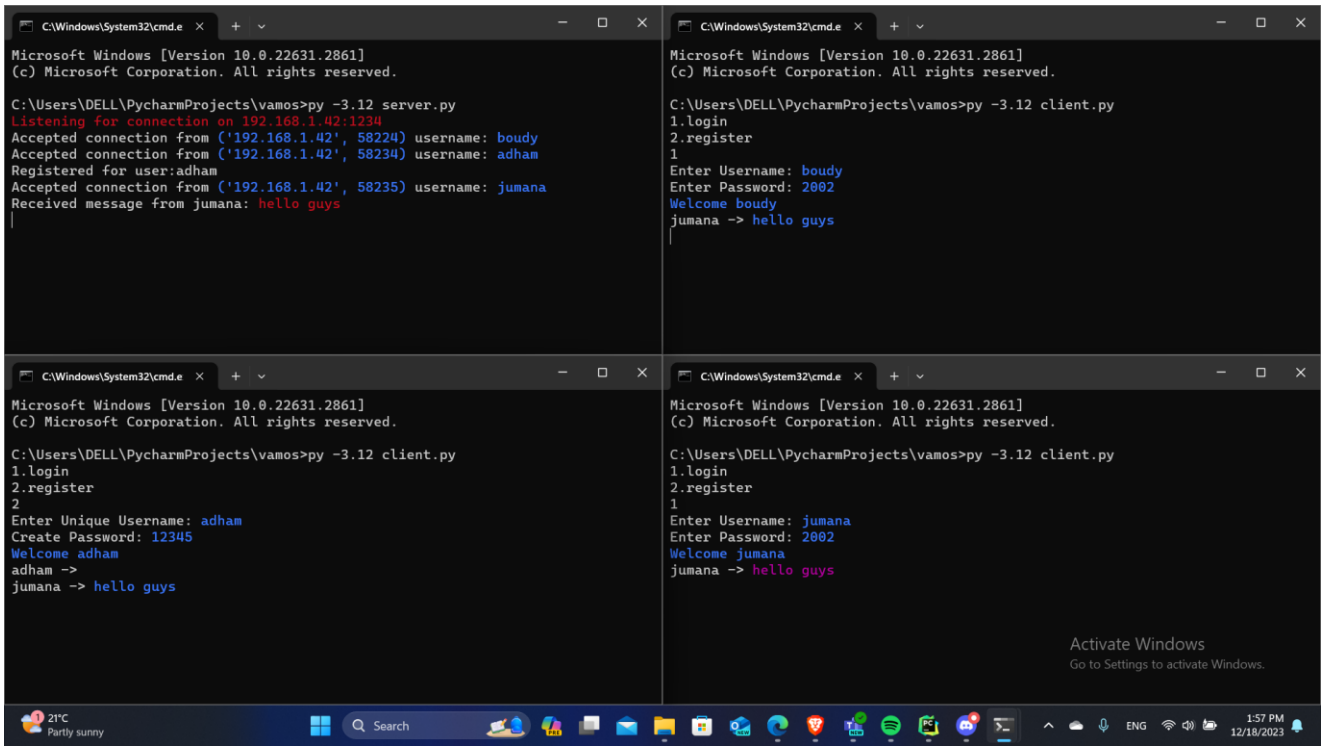


Figure 21: Logging in with a client and the server accepting the connection

Here we logged in using one client's credentials and the server accepted the connection.



Here we registered with a new username (Adham) and password and logged in with an old pre-registered username (Jumana). At the same time, the server accepted their connections which indicates that multithreading is working because we can handle multiple users at the same time. Also, we made Jumana send a message to the entire group chat and all other clients like Adham received it. The message Jumana sent appeared to all the clients and the server as well.

Figure 23: A user registering and another user sending a message that all other users in the group chat received.

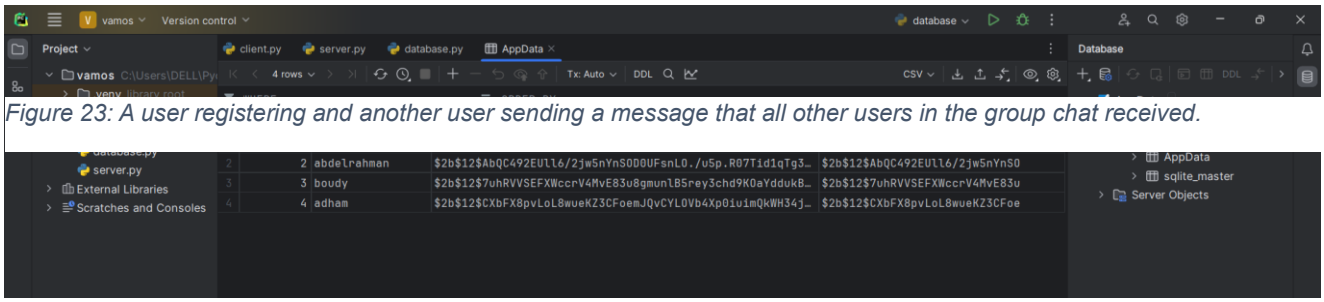


Figure 22: The updated database

This is the updated database after we registered the new user with the username “Adham”. The password is being hashed using salted hashing so that if a hacker gets access into our database, he won’t be able to access the actual password, he will only see the hashed password bearing in mind he doesn’t have the salt of the hashing so he won’t be able to decode the password.

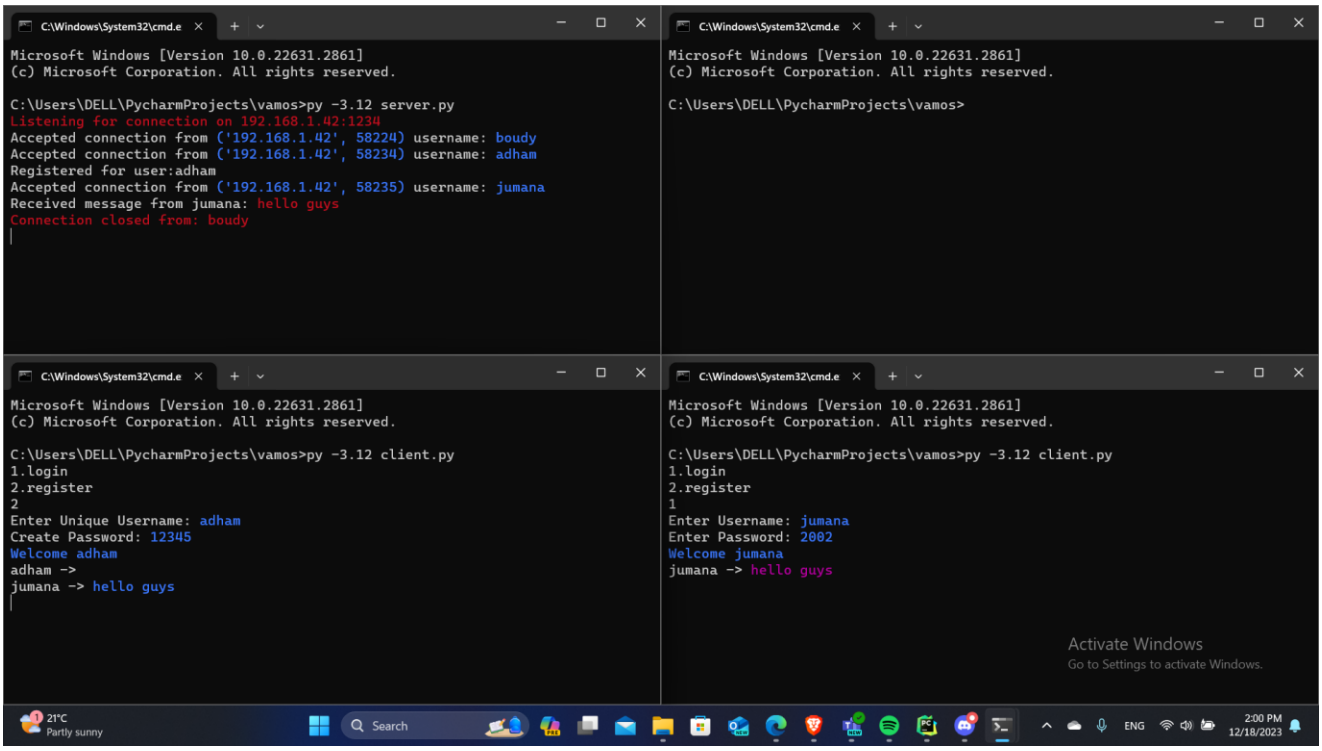


Figure 24: A client closing his connection and the server printing a closing notification

When a client closed the connection, the server didn't raise an exception. We handled this by printing that the connection closed from the corresponding client on the server's terminal.

5. Repo Link

https://github.com/AbdelrahmanKhaled18/Peer_to_Peer_Multi_Chatting_App

Figure 25: GitHub Repository link

6. Appendix 1

i. Client Code

```
import json
import socket
import sqlite3
import sys
import msvcrt
import errno
import bcrypt
import colorama
from colorama import *

colorama.init(autoreset=True)
HEADER_LENGTH = 10
# Set up the connection between the client and the server
IP = socket.gethostbyname(socket.gethostname())
PORT = 1234
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((IP, PORT))
client_socket.setblocking(False)

# Connecting the Database
connection = sqlite3.connect("AppData.db")
cursor = connection.cursor()


# The Register Function that checks that the username is Unique
def user_register(username, password):
    cursor.execute("SELECT * FROM AppData WHERE username=?", (username,))
    if cursor.fetchone() is None:
        username_header_register = f'{len(username):<{HEADER_LENGTH}}'.encode('utf-
8')
        username_text = username.encode('utf-8')
        client_socket.send(username_header_register + username_text)
        Password_header_Register = f'{len(password):<{HEADER_LENGTH}}'.encode('utf-
8')
        password_text = password.encode('utf-8')
        client_socket.send(Password_header_Register + password_text)
        client_socket.send('register'.encode('utf-8'))
        print("Welcome " + username + "")
        return True
    else:
        print(f'{Fore.RED} Username is not unique!')
```

```
        return False


# The login function that checks the username and the Password
def user_login(username, password):
    cursor.execute("SELECT password, salted_password FROM AppData WHERE username=?",
(username,))
    result = cursor.fetchone()
    if result:
        stored_hashed_password = result[0]
        stored_salt = result[1]
        entered_hashed_password = bcrypt.hashpw(password.encode('utf-8'),
stored_salt)
        if entered_hashed_password == stored_hashed_password:
            username_header_login = f'{len(username):<{HEADER_LENGTH}}'.encode('utf-
8')
            username_text = username.encode('utf-8')
            client_socket.send(username_header_login + username_text)
            Password_header_login = f'{len(password):<{HEADER_LENGTH}}'.encode('utf-
8')
            password_text = password.encode('utf-8')
            client_socket.send(Password_header_login + password_text)
            client_socket.send('login'.encode('utf-8'))
            print("Welcome " + username + "")
            return True
        else:
            print(f'{Fore.RED} invalid username or password!')
            return False
    else:
        print(f'{Fore.RED}invalid username or password!')
        return False


# Choosing the Login Or Register For the Client
choice = input("1.login\n2.register\n")

if choice == '1':
    # This While Loop makes enter the username and Password again if the data is not
correct
    LogingResult = False
    while not LogingResult:
        ClientUsername = input(f"{Fore.RESET}Enter Username: {Fore.LIGHTBLUE_EX}")
        ClientPassword = input(f"{Fore.RESET}Enter Password: {Fore.LIGHTBLUE_EX}")
        LogingResult = user_login(ClientUsername, ClientPassword)
else:
```

```
    RegisterResult = False
    while not RegisterResult:
        ClientUsername = input(f"{Fore.RESET}Enter Unique
Username:{Fore.LIGHTBLUE_EX} ")
        ClientPassword = input(f"{Fore.RESET}Create Password: {Fore.LIGHTBLUE_EX}")
        RegisterResult = user_register(ClientUsername, ClientPassword)

while True:
    # Handling the sending of the Messages in JSON format
    if msvcrt.kbhit():
        message = input(f'{Fore.RESET}{ClientUsername} ->{Fore.LIGHTMAGENTA_EX} ')
        if message:
            json_message = json.dumps({'username': ClientUsername, 'message':
message})
            message_header = f'{len(json_message):<{HEADER_LENGTH}}'.encode('utf-8')
            client_socket.send(message_header + json_message.encode('utf-8'))

    try:
        while True:
            message_header_received = client_socket.recv(HEADER_LENGTH)
            if not message_header_received:
                print('Connection closed by the server')
                sys.exit()

            message_length = int(message_header_received.decode('utf-8').strip())
            received_json_message = client_socket.recv(message_length).decode('utf-
8')
            received_data = json.loads(received_json_message)
            print(f'{Fore.RESET}{received_data["username"]} ->
{Fore.LIGHTBLUE_EX}{received_data["message"]}')

    # Handling Exceptions Like if the Server Closed
    except IOError as e:
        if e.errno != errno.EAGAIN and e.errno != errno.EWOULDBLOCK:
            print(f'{Fore.RED}Reading error:{str(e)} ')
            sys.exit()
        continue

    except Exception as e:
        print(f'{Fore.RED}Reading error:{str(e)} ')
        sys.exit()
```

ii. Server Code

```
import json
import socket
import sqlite3
import threading
import colorama
import bcrypt
from colorama import *

HEADER_LENGTH = 10
Flag = True
colorama.init(autoreset=True)

# Set up the Connection for the Server Using Sockets
IP = socket.gethostbyname(socket.gethostname())
PORT = 1234
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_socket.bind((IP, PORT))
server_socket.listen()

socket_list = [server_socket]
clients = {}
print(f'{Fore.RED}Listening for connection on {IP}:{PORT}{Fore.RESET}')


# Function Used to Receives Msgs From the Client
def receive_message(client_socket):
    try:
        message_header = client_socket.recv(HEADER_LENGTH)
        if not len(message_header):
            return False
        message_length = int(message_header.decode('utf-8').strip())
        return {'header': message_header, 'data': client_socket.recv(message_length)}
    except:
        return False


# Function To Add the user in the Database in  case of Register
def add_user(username, password):
    # Connecting the Database File
    connection = sqlite3.connect("AppData.db")
    cursor = connection.cursor()

    # Generating Salt For The salted Hashing using bcrypt
```

```
    salt = bcrypt.gensalt()
    cursor.execute("INSERT INTO AppData(username, password,salted_password) VALUES
(?, ? ,?)",
                   (username, bcrypt.hashpw(password.encode("utf-8"), salt), salt))
    connection.commit()
    connection.close()


# Functions used to handle single Client
def handle_client(client_socket, address):
    global Flag
    user = receive_message(client_socket)
    if user is False:
        return
    clients[client_socket] = user
    Flag = False
    print(f'{Fore.RESET}Accepted connection from {Fore.LIGHTBLUE_EX}{address}
{Fore.RESET}username: '
          f'{Fore.LIGHTBLUE_EX}{user["data"].decode("utf-8")}')

    # Accepting messages algorithm
    while True:
        if client_socket in clients and not Flag:
            password = receive_message(client_socket)
            user_object = clients[client_socket]
            result = client_socket.recv(1024).decode('utf-8')
            if result == 'register':
                add_user(user_object['data'].decode('utf-8'),
password['data'].decode('utf-8'))
                print('Registered for
user:{0}'.format(user_object['data'].decode('utf-8')))
            Flag = True
        else:
            message = receive_message(client_socket)
            # if no message then the connection is closed between this client  and
the server
            if message is False:
                print(f'{Fore.RED}Connection closed from: '
                      f'{clients[client_socket]["data"].decode("utf-8")}
{Fore.RESET}')
                try:
                    socket_list.remove(client_socket)
                except ValueError:
                    pass
                del clients[client_socket]
```

```
                break

            user_object = clients[client_socket]
            sender_username = user_object['data'].decode('utf-8')
            # Exchanging the messages in a JSON format
            received_json = message['data'].decode('utf-8')
            received_data = json.loads(received_json)
            received_username = received_data['username']
            received_message = received_data['message']

            print(f"{Fore.RESET}Received message from {received_username}:
{Fore.RED}{received_message} {Fore.RESET}")

            json_message = json.dumps({'username': sender_username, 'message':
received_message})

            for client_sock in clients:
                if client_sock != client_socket:
                    client_sock.send(
                        f'{len(json_message):<{HEADER_LENGTH}}'.encode('utf-8') +
json_message.encode('utf-8')
                    )

    client_socket.close()


# Handling the Multithreading in The file to make every client on a Separate thread
while True:
    client_socket, client_address = server_socket.accept()
    thread = threading.Thread(target=handle_client, args=(client_socket,
client_address))
    thread.start()
```

iii. Database Code

```
import sqlite3

# Creating The Database Table
connection = sqlite3.connect("AppData.db")
cursor = connection.cursor()

cursor.execute("""
CREATE TABLE IF NOT EXISTS AppData(
id INTEGER PRIMARY KEY,
```

```
username VARCHAR(255) NOT NULL,
password VARCHAR(255) NOT NULL,
salted_password VARCHAR(255) NOT NULL
)
""")

connection.commit()
connection.close()
```

III. PHASE 3

1. Registry

```
# CREATE CHAT ROOM
elif message[0] == "CREATE":
    room_name = message[1]  # Extract room name from the message
    room_exists = any(room['name'] == room_name for room in chat_rooms)
    if room_exists:
        response = "create-failed" + " "  # Room already exists
    else:
        response = "create-success " + self.username  # Room creation successful
        chat_rooms.append({"name": room_name, "participants": []})
    self.tcpClientSocket.send(response.encode())
```

Figure 26

In the code snippet above, if the request from the peer is created, we extract the room name from the received message. Afterwards, we check if the room name is unique. If it is unique, we send a success message to the peer. We then append this chat room in the list of chat rooms.

```
    # JOIN CHAT ROOM
    elif message[0] == "JOIN-CHAT":
        requested_room_name = message[1]
        room_exists = False
        ip_1, port_1 = get_peer_ip_port(self.username)
        for room in chat_rooms:
            if room["name"] == requested_room_name:
                participants_list = [f"{participant['ip']}:{participant['port']}"
                                     for participant in room["participants"]]

                room["participants"].append({"username": self.username, "ip": ip_1, "port": port_1})
                self.chatroom = requested_room_name
                # Collect all participants' IP and port in the room

                response = "join-chat-success " + " ".join(participants_list)
                self.tcpClientSocket.send(response.encode())
                room_exists = True
                break
        if not room_exists:
            response = "No-Room found"
            self.tcpClientSocket.send(response.encode())
```

Figure 27

In the code snippet above, we first extract the name of the chat room then we iterate through the chatroom list to see if the room we want actually exists or not. If the room exists, we append the list of room participants with the peer data (ip, port number, and username).

```
            self.tcpClientSocket.send(response.encode())

    # ONLINE USERS LIST
    elif message[0] == "users-list-request":
        if onlinePeers:
            users = ",".join([peer["username"] for peer in onlinePeers])
            response = f"users-list\n{users}"
            self.tcpClientSocket.send(response.encode())
        else:
            # If no users are online, send a message indicating that
            response = "users-list\nNo users online"
            self.tcpClientSocket.send(response.encode())
```

Figure 28

In the code snippet above, we first check if the online peers list isn't empty. If it isn't empty, we make a list of the users separated by a comma. We append in this list the username only. Afterwards, we send this list to the peer.

```
# CHAT ROOMS LIST
elif message[0] == "rooms-list-request":
    if chat_rooms:
        # Filter out non-string room names and join them
        chats = ",".join([room["name"] for room in chat_rooms if isinstance(room["name"], str)])
        response = f"chat-list\n{chats}"
        self.tcpClientSocket.send(response.encode())
    else:
        # If no rooms are available, send a message indicating that
        response = "chat-list\nNo rooms available"
        self.tcpClientSocket.send(response.encode())
```

Figure 29

In the code snippet above, we first check if the chat list isn't empty. If it isn't empty, we make a list of the chats separated by a comma. We append in this list the chatroom name only. Afterwards, we send this list to the peer.

```
# LEAVING the CHAT ROOM
elif message[0] == "chatroom-leave":
    self.leave_chat_room()
```

```
2 usages  boudyK
def leave_chat_room(self):
    for room in chat_rooms:
        for participant in room['participants']:
            if participant['username'] == self.username:
                room['participants'].remove(participant)
                self.chatroom = None
                print(f"{self.username} left the chat room")
                break
```

Figure 30

In the code snippet above, if the user chooses to leave the chatroom, we call the leave_chat_room function which iterates through all participants in the room and removes the specified participant from the list of participants in that room. We then print a message indicating that this user left the chat room.

2. Peer Client

```
# the tcp socket of the peer's server, enters here
if s is self.tcpServerSocket:
    # accepts the connection, and adds its connection socket to the input list
    # so that we can monitor that socket as well
    connected, addr = s.accept()
    connected.setblocking(0)
    # inputs.append(connected)
    self.connectedPeers.append(connected)
    # if the user is not chatting, then the ip and the socket of
    # this peer are assigned to server variables
# if the socket that receives the data is the one that
# is used to communicate with a connected peer, then enters here
else:
    message = s.recv(1024).decode().split("\n")
    if len(message) == 0:
        s.close()
        self.connectedPeers.remove(s)
    elif message[0] == "chatroom-join":
        print(message[1] + " has joined the chatroom.")
        s.send("welcome".encode())
    elif message[0] == "chatroom-leave":
        print(f"{Fore.RED}{message[1]}" + f"{Fore.RED}has left the chatroom.{Fore.RESET}")
        s.close()
        self.connectedPeers.remove(s)
    elif message[0] == "chat-message":
        username = message[1]
        content = "\n".join(message[2:])
        print(username + " -> " + f"{Fore.LIGHTBLUE_EX}{content}{Fore.RESET}")
```

Figure 31

In the code snippet above, in the peer server class, if the s is a TCP socket we add the connection socket to the connected peers list. If it isn't a TCP socket, we receive a message from the peer and we check it. If the message is empty, the socket will be closed and it'll be removed from the connected peers list. If the message is "chatroom-join", we print that the username has joined the chatroom. If it is "chatroom-leave", we print that the user with the specific username has left the chatroom. Finally if it is "chat-message", we extract the username and the content of the message and then print them.

```
if peersToConnect is not None:
    for peer in peersToConnect:
        peer_data = peer.split(":")
        if len(peer_data) >= 2:
            peerHost = peer_data[0]
            peerPort = int(peer_data[1])
            sock = socket(AF_INET, SOCK_STREAM)
            sock.connect((peerHost, peerPort))
            message = "chatroom-join\n{}".format(self.username)
            sock.send(message.encode())
            self.peerServer.connectedPeers.append(sock)
```

The code snippet above is some lines of code added to the peer client object initialization to extract the data of the client. The data extracted are the host's ip address and the port number.

Figure 32

```
boudyK *
def run(self):
    print("Peer client started...")

    print(f'{Fore.RED}Chatroom joined Successfully. \nStart typing to send a message'
          f'. Send ":exit" to leave the chatroom.{Fore.RESET}')

    while self.chatroom is not None:
        content = input()

        if content == ":exit":
            message = "chatroom-leave\n" + self.username
        else:
            message = "chat-message\n{}\n{}".format( *args: self.username, content)

        for sock in self.peerServer.connectedPeers:
            try:
                sock.send(message.encode())
            except:
                pass

        if content == ":exit":
            self.chatroom = None
            for sock in self.peerServer.connectedPeers:
                sock.close()
```

The function in the code snippet above is a function that keeps the user in the chatroom until the user types “exit”. When he does, he will leave the chatroom. The content of the chatroom will then be none for this specific peer. When he doesn’t write “exit”, we put the username and the content of the message in the message variable and it’ll be sent to all the sockets of the connected peers.

Figure 33

3. Peer Name

```
        # and peer variables are set, and server and client sockets are closed
    elif self.isOnline:

        choice = input(f"{Fore.RESET}Choose: \nLogout: 3\nSearch: "
                       f"4\nStart a Chat: 5\nCreate Chat Room: 6\nJoin Chat: 7\n"
                       f"List Users: 8\nList Chat Rooms: 9\n")
```

Figure 34

In the code snippet above, we first check if the user is online and already logged in to give him access to the features of the application.

```
# Creating chat room
elif choice == "6":
    chat_name = input(f"Enter the Chat room name to Create: {Fore.RED}")
    self.Create_Chat_Room(chat_name)
```

```
1 usage   boudyK
def Create_Chat_Room(self, chat_room_name):
    message = "CREATE " + chat_room_name
    self.tcpClientSocket.send(message.encode())
    response = self.tcpClientSocket.recv(1024).decode()
    if response.startswith("create-success"):
        creator = response[len("create-success "):]  # Extract the creator's username
        print(f"{Fore.GREEN}Chat room {chat_room_name} was created successfully by {creator}{Fore.RESET}")
        self.Join_Chat_Room(chat_room_name)
        self.peerClient = PeerClient(self.username, self.peerServer, chat_room_name)
        self.peerClient.start()
        self.peerClient.join()
    else:
        print(f"{Fore.RED}There is already a chat room with that name.{Fore.RESET}")
```

Figure 35

In the code snippet above, when we receive the response from the registry file, we check if the response starts with “create-success” which signifies that the chatroom creation was successful in the registry file. We then extract the username of the creator of the chatroom and we call the function that makes this client join the chatroom so that the client joins the chatroom. We then hold the thread till the client types “exit” so that the options do not appear again until the client leaves the chatroom.

```
2 usages  boudyK
def Join_Chat_Room(self, chat_room_name):
    message = "JOIN-CHAT " + chat_room_name + " "
    self.tcpClientSocket.send(message.encode())
    response = self.tcpClientSocket.recv(1024).decode()
    if response.startswith("join-chat-success"):
        participants_list = response.split()[1:]  # Extract the list of participants
        if participants_list:
            print("Joined " + chat_room_name + ". Participants: " + ", ".join(participants_list))
            self.peerClient = PeerClient(self.username, self.peerServer, chat_room_name, participants_list)
            self.peerClient.start()
            self.peerClient.join()
        else:
            # If participants_list is empty, initiate the PeerClient without the list
            print("Joined " + chat_room_name + ". No participants in the room yet.")
            self.peerClient = PeerClient(self.username, self.peerServer, chat_room_name)
            self.peerClient.start()
            self.peerClient.join()
        self.tcpClientSocket.send("chatroom-leave".encode())
    elif response == "No-Room found":
        print("No room found by the name: " + chat_room_name)
```

Figure 36

In the code snippet above, when we receive the response from the registry file, we check if the response is a success. We then extract the list of participants from the response and check if the list is empty. If it's empty, we will print that there are no participants in the chatroom yet. If it isn't empty, we will print that the username has joined the chatroom and we will print the list of participants in the chatroom. We then hold the thread till the client types "exit" so that the options do not appear again until the client leaves the chatroom. the "chatroom-leave" command will not be executed until the user either leaves or logs out.

```
1 usage   boudyK *
def userList(self):
    message = "users-list-request" + " "
    self.tcpClientSocket.send(message.encode())
    response = self.tcpClientSocket.recv(1024).decode()
    if response.startswith("users-list"):
        users = response[len("users-list"):].split('\n')
        print("Online Users:\n")
        if len(users) > 1:  # Check if there are users listed
            for user in users[1:]:
                print(f"{Fore.RED}{user}" + ",")
            print(f"{Fore.RESET}\n")
        else:
            print("\tNo users currently online")
    else:
        print("Unexpected response:", response)
```

Figure 37

In the code snippet above, we first extract the users from the response and then we iterate through the list and we print each username followed by a comma.

```
def roomsList(self):
    message = "rooms-list-request"
    self.tcpClientSocket.send(message.encode())
    response = self.tcpClientSocket.recv(1024).decode()
    if response.startswith("chat-list"):
        chats = response[len("chat-list\n"):].split(',')
        print("Available Rooms:")
        for chat in chats:
            chat_name = chat.strip()
            if chat_name:  # Ensure the room name is not an empty string
                print(chat_name)
        if len(chats) < 1:  # Check if only the header or no rooms exist
            print("\tNo rooms currently exist")
    else:
        print("Unexpected response:", response)
```

Figure 38

In the code snippet above, we extract the users from the response then we iterate through the list and print each username followed by a comma.

4. Outputs

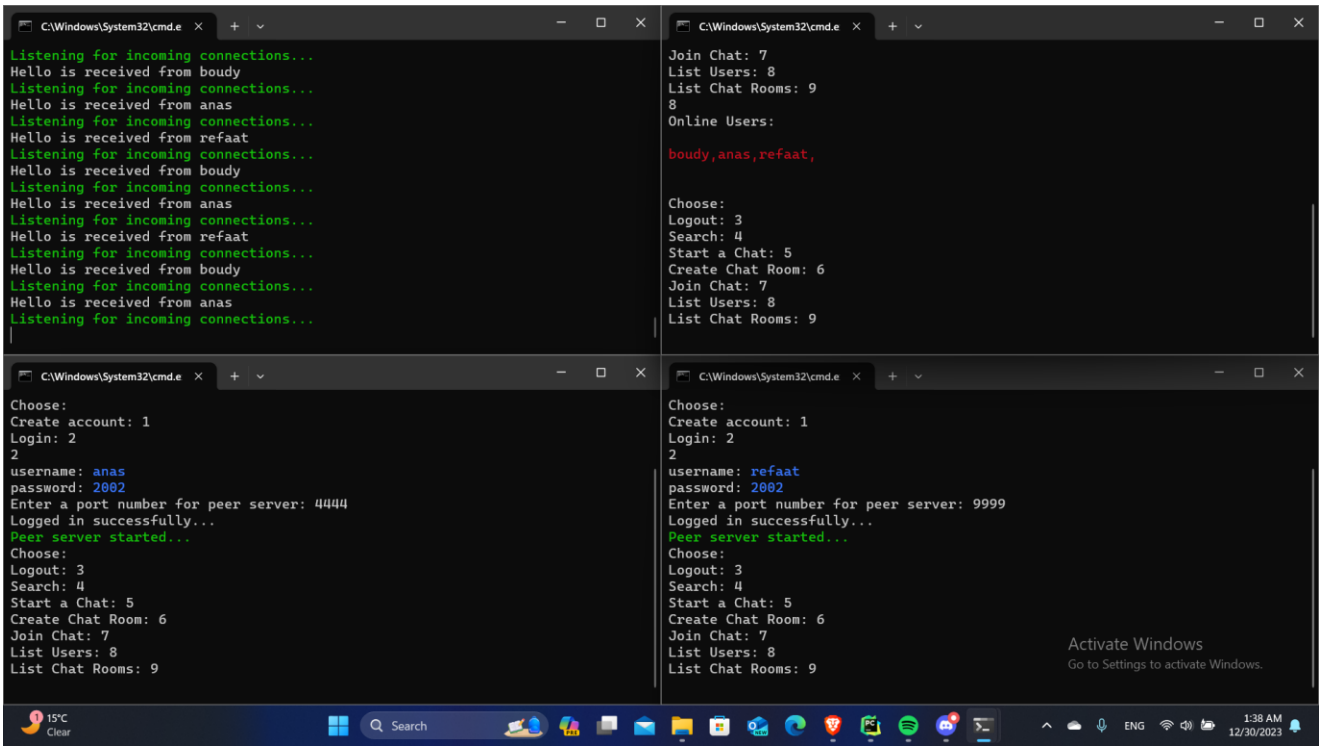


Figure 39

In this test case, we logged in with 3 users and we listed the online users. They were then listed successfully.

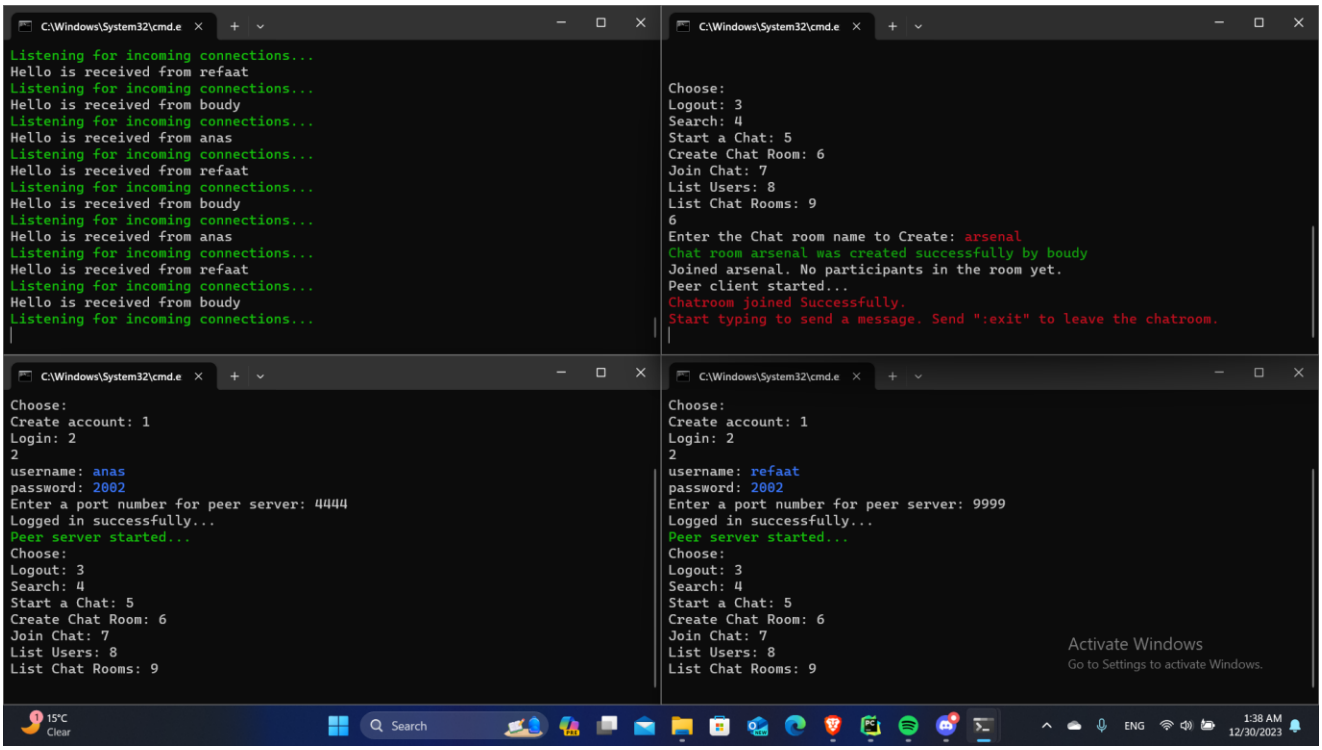


Figure 40

In this test case, we created a new chatroom called “Arsenal”

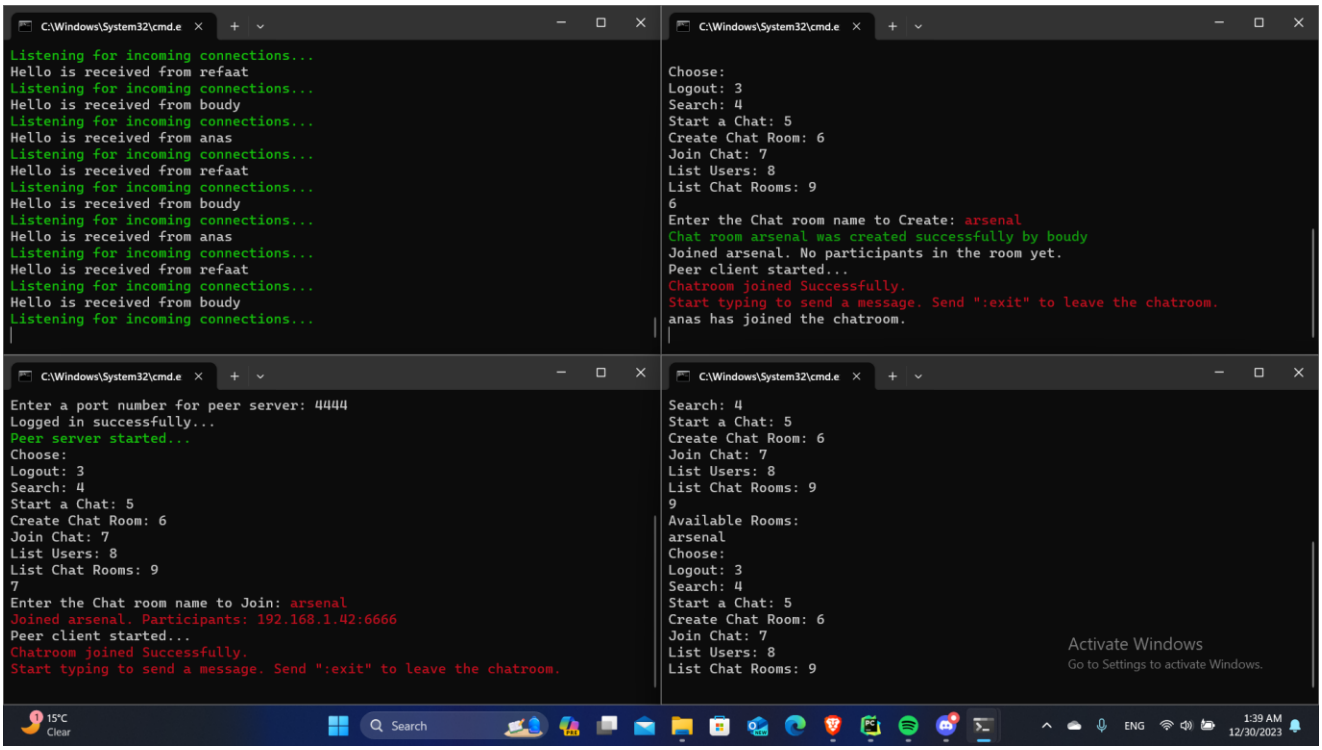


Figure 41

Here, we made the second client join the chatroom.

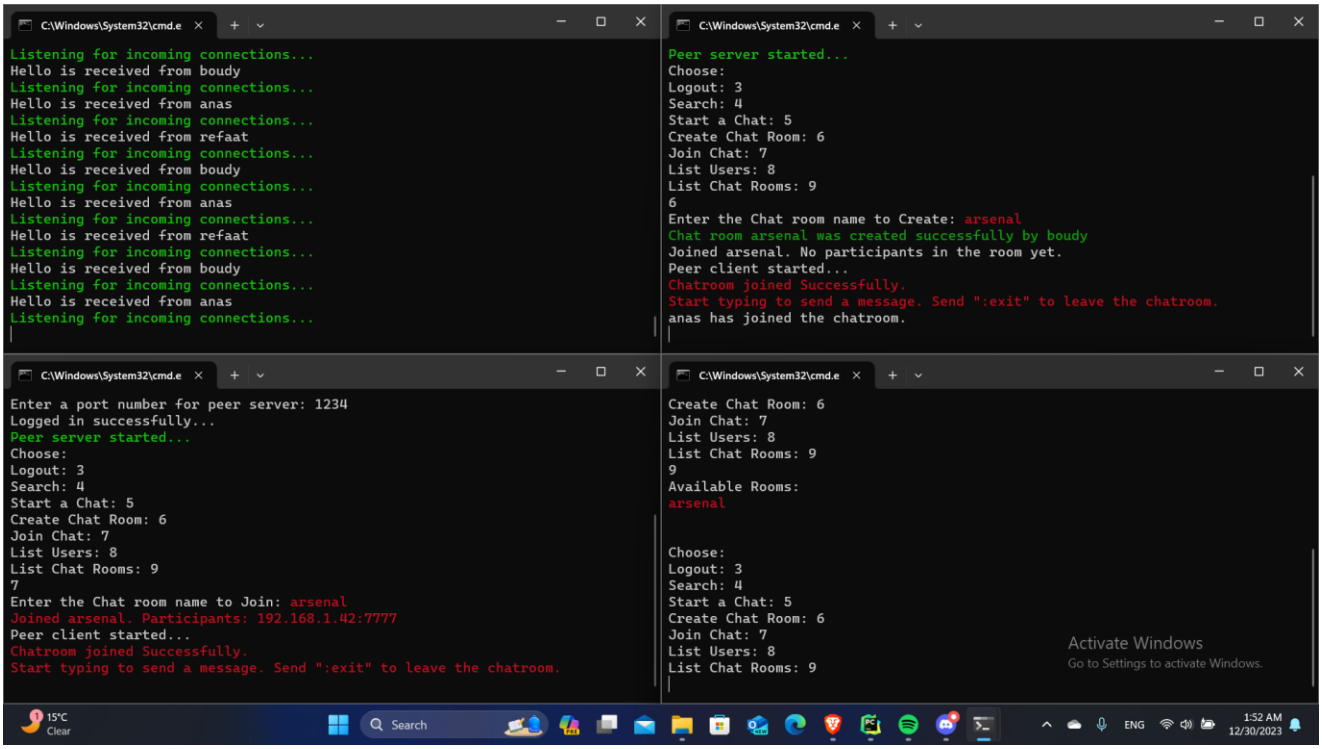


Figure 42

Here we created a chatroom then we left it. Another user also listed all the available chatrooms.

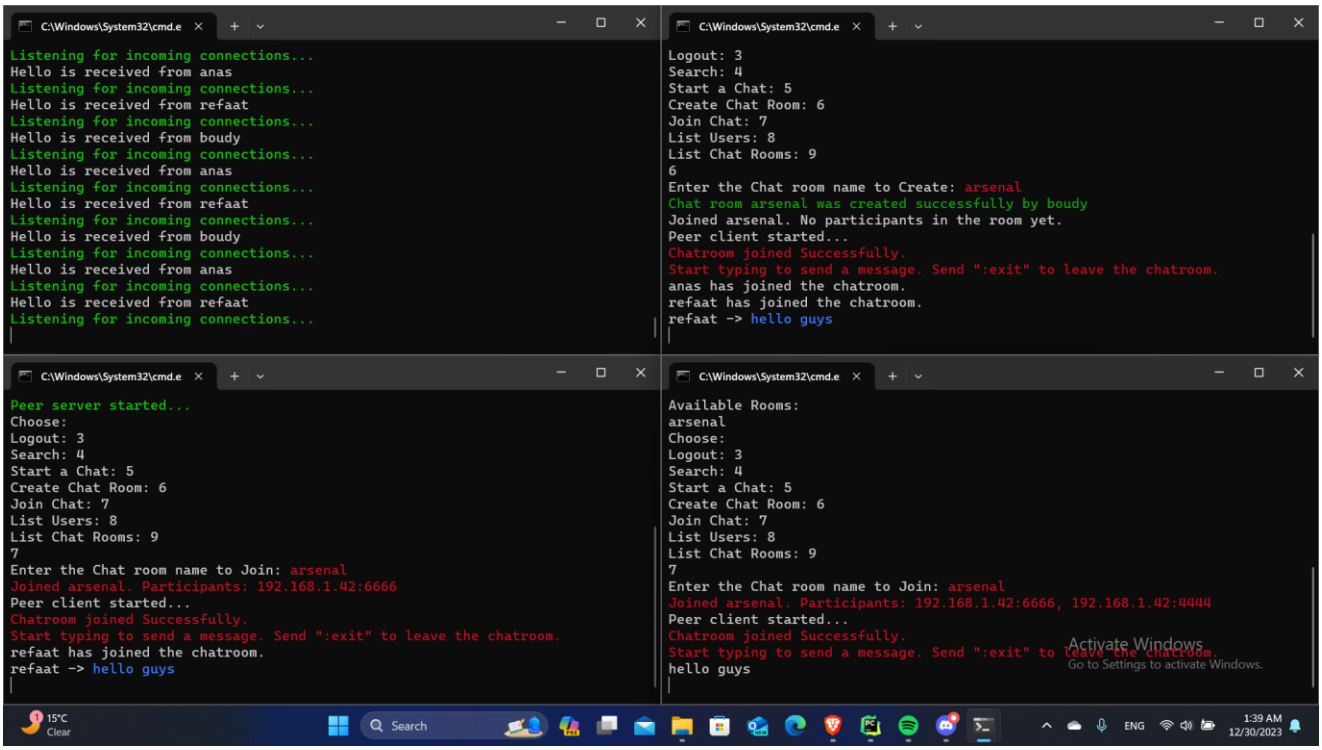


Figure 43

Here the user Refaat sent “hello guys” in the chatroom and the message was sent to all the participants in the chatroom.

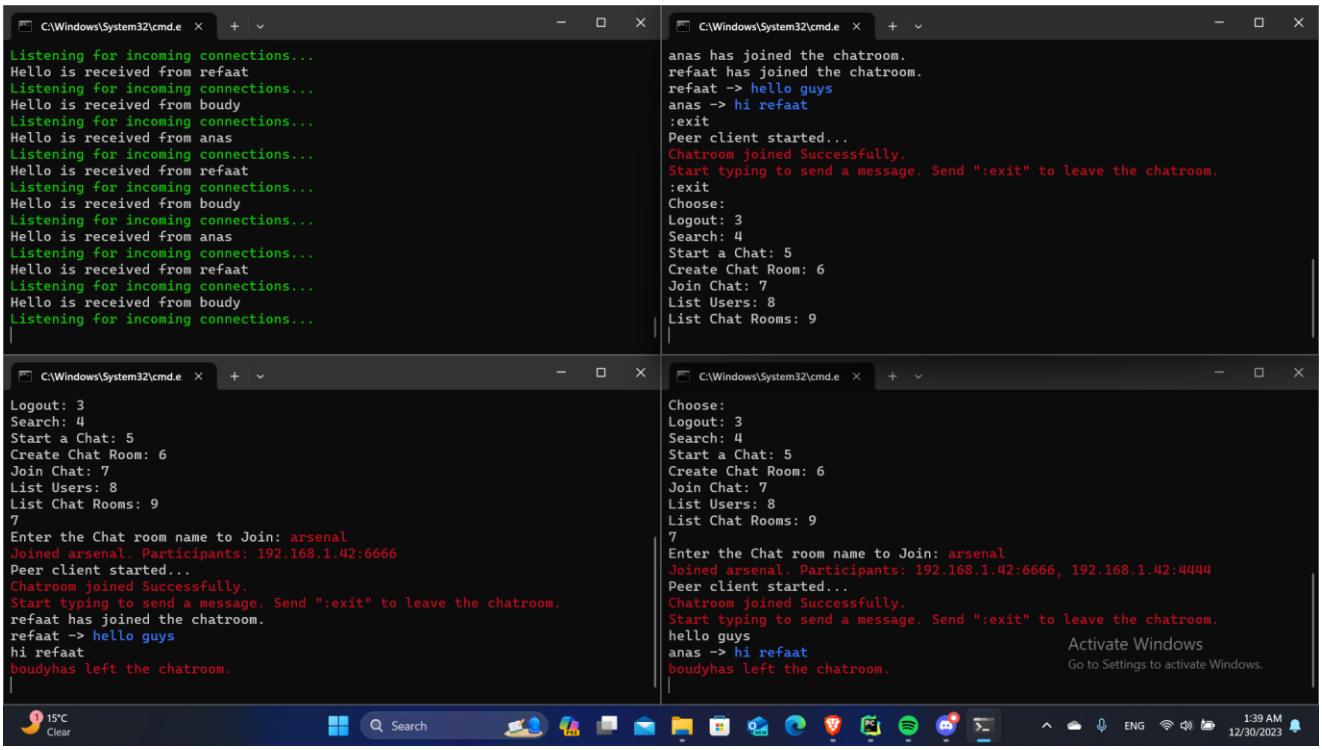


Figure 44

Here, when the user Boudy left the chatroom, the other clients got notified that the client with the username “Boudy” left the chat.

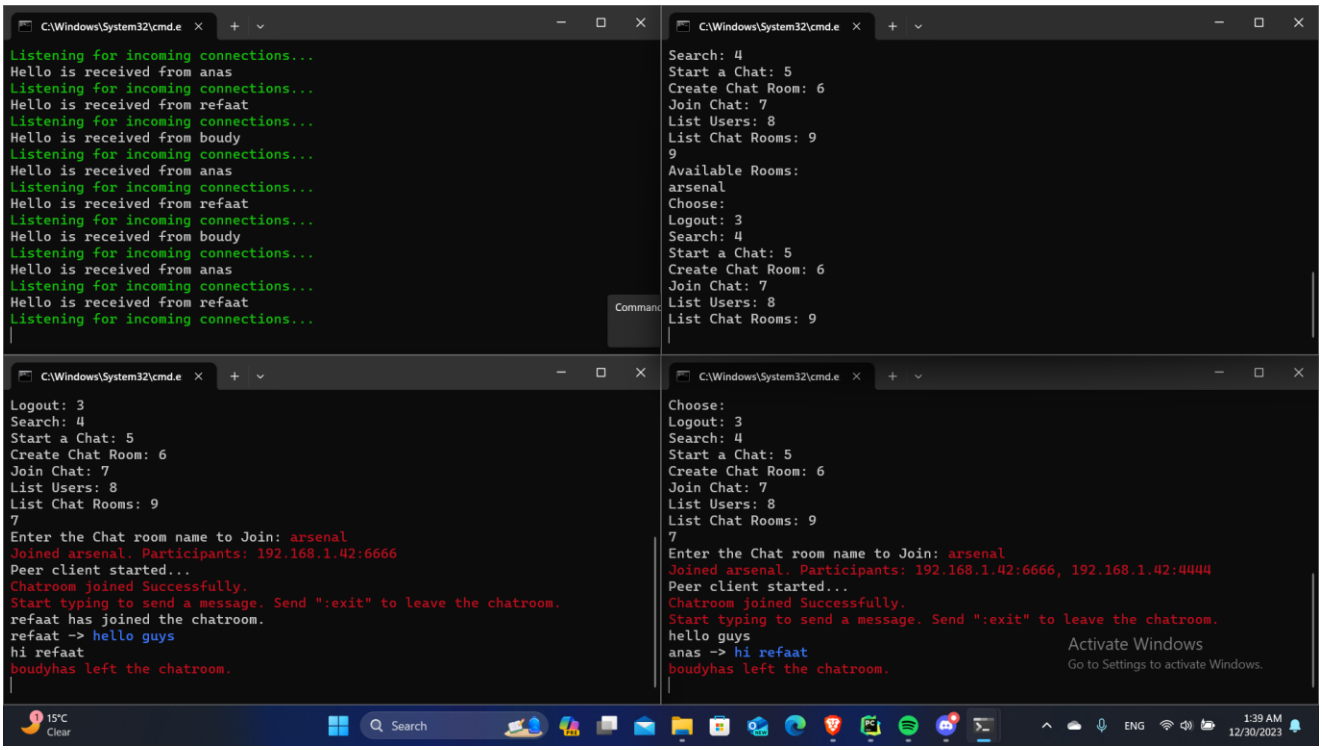


Figure 45

5. Repo Link

https://github.com/AbdelrahmanKhaled18/Peer_to_Peer_Multi_Chatting_App

Figure 46

6. Appendix 2

6.1 Registry

```
from socket import *
import threading
import select
import logging
import db
import bcrypt
import colorama
from colorama import *

colorama.init(autoreset=True)


# This class is used to process the peer messages sent to registry
# for each peer connected to registry, a new client thread is created

def is_account_online(username):
    for peer in onlinePeers:
        if peer["username"] == username:
            return True
    return False


def get_peer_ip_port(username):
    for peer in onlinePeers:
        if peer["username"] == username:
            return (peer["ip"], peer["port"])
    return (None, None)


class ClientThread(threading.Thread):
    # initializations for client thread
    def __init__(self, ip, port, tcpClientSocket):
        threading.Thread.__init__(self)
        # ip of the connected peer
        self.chatroom = None
        self.lock = None
        self.ip = ip
        # port number of the connected peer
        self.port = port
        # socket of the peer
        self.tcpClientSocket = tcpClientSocket
        # username, online status and udp server initializations
```

```
        self.username = None
        self.isOnline = True
        self.udpServer = None
        print("New thread started for " + ip + ":" + str(port))

    # main of the thread
    def run(self):
        # locks for thread which will be used for thread synchronization
        self.lock = threading.Lock()
        print("Connection from: " + self.ip + ":" + str(port))
        print("IP Connected: " + self.ip)

        while True:
            try:
                # waits for incoming messages from peers
                message = self.tcpClientSocket.recv(1024).decode().split()
                logging.info("Received from " + self.ip + ":" + str(self.port) + " ->
" + " ".join(message))
                #   JOIN #
                if message[0] == "JOIN":
                    # join-exist is sent to peer,
                    # if an account with this username already exists
                    if db.is_account_exist(message[1]):
                        response = "join-exist"
                        print("From-> " + self.ip + ":" + str(self.port) + " " +
response)
                        logging.info("Send to " + self.ip + ":" + str(self.port) + "
-> " + response)
                        self.tcpClientSocket.send(response.encode())
                    # join-success is sent to peer,
                    # if an account with this username does not exist, and the
account is created
                    else:
                        hashed_password = bcrypt.hashpw(message[2].encode('utf-8'),
bcrypt.gensalt())
                        db.register(message[1], hashed_password)
                        response = "join-success"
                        logging.info("Send to " + self.ip + ":" + str(self.port) + "
-> " + response)
                        self.tcpClientSocket.send(response.encode())
                #   LOGIN    #
                elif message[0] == "LOGIN":
                    # login-account-not-exist is sent to peer
                    # if an account with the username does not exist
                    if not db.is_account_exist(message[1]):
```

```
                        response = "login-account-not-exist"
                        logging.info("Send to " + self.ip + ":" + str(self.port) + "
-> " + response)
                        self.tcpClientSocket.send(response.encode())
                    # login-online is sent to peer
                    # if an account with the username already online
                    elif is_account_online(message[1]):
                        response = "login-online"
                        logging.info("Send to " + self.ip + ":" + str(self.port) + "
-> " + response)
                        self.tcpClientSocket.send(response.encode())
                        # onlinePeers.append(message[1])
                    # login-success is sent to peer
                    # if an account with the username exists and not online
                    else:
                        # retrieves the account's password, and checks if the one
entered by the user is correct
                        retrievedPass = db.get_password(message[1])
                        # if the password is correct, then peer's thread is added to
thread list
                        # peer is added to db with its username, port number, and ip
address
                        if bcrypt.checkpw(message[2].encode('utf-8'), retrievedPass):
                            self.username = message[1]
                            self.lock.acquire()
                            try:
                                tcpThreads[self.username] = self
                            finally:
                                self.lock.release()

                            db.user_login(message[1], self.ip, message[3])
                            # login-success is sent to peer,
                            # and an udp server thread is created for this peer, and
thread is started
                            # timer thread of the udp server is started
                            response = "login-success"
                            onlinePeers.append({"username": self.username, "ip":
self.ip, "port": message[3]})
                            logging.info("Send to " + self.ip + ":" + str(self.port)
+ " -> " + response)
                            self.tcpClientSocket.send(response.encode())
                            self.udpServer = UDPServer(self.username,
self.tcpClientSocket)
                            self.udpServer.start()
                            self.udpServer.timer.start()
```

```
                        # if password not matches and then login-wrong-password
response is sent
                        else:
                            response = "login-wrong-password"
                            logging.info("Send to " + self.ip + ":" + str(self.port)
+ " -> " + response)
                            self.tcpClientSocket.send(response.encode())
                #   LOGOUT  #
                elif message[0] == "LOGOUT":
                    # if a user is online,
                    # removes the user from onlinePeers list
                    # and removes the thread for this user from tcpThreads
                    # socket is closed, and the timer thread of the udp for this
                    # user is canceled
                    if self.chatroom is not None:
                        self.leave_chat_room()

                    if len(message) > 1 and message[1] is not None and
is_account_online(message[1]):
                        for peer in onlinePeers:
                            if peer["username"] == message[1]:
                                onlinePeers.remove(peer)
                                break
                        self.lock.acquire()
                        try:
                            if message[1] in tcpThreads:
                                del tcpThreads[message[1]]
                        finally:
                            self.lock.release()
                        print(self.ip + ":" + str(self.port) + " is logged out")
                        self.tcpClientSocket.close()
                        self.udpServer.timer.cancel()
                        break
                    else:
                        self.tcpClientSocket.close()
                        break
                #   SEARCH #
                elif message[0] == "SEARCH":
                    # checks if an account with the username exists
                    if db.is_account_exist(message[1]):
                        # checks if the account is online
                        # and sends the related response to peer
                        if is_account_online(message[1]):
                            peer_info_ip, peer_info_port =
get_peer_ip_port(message[1])
```

```
                            if peer_info_ip and peer_info_port:
                                response = f"search-success
{peer_info_ip}:{peer_info_port}"
                                logging.info(f"Send to {self.ip}:{str(self.port)} ->
{response}")
                                self.tcpClientSocket.send(response.encode())
                            else:
                                response = "search-user-not-found"  # Unable to fetch
IP and port for the username
                                logging.info(f"Send to {self.ip}:{str(self.port)} ->
{response}")
                                self.tcpClientSocket.send(response.encode())
                        else:
                            response = "search-user-not-online"
                            logging.info("Send to " + self.ip + ":" + str(self.port)
+ " -> " + response)
                            self.tcpClientSocket.send(response.encode())
                    # enters if username does not exist
                    else:
                        response = "search-user-not-found"
                        logging.info("Send to " + self.ip + ":" + str(self.port) + "
-> " + response)
                        self.tcpClientSocket.send(response.encode())

                # CREATE CHAT ROOM
                elif message[0] == "CREATE":
                    room_name = message[1]  # Extract room name from the message
                    room_exists = any(room['name'] == room_name for room in
chat_rooms)
                    if room_exists:
                        response = "create-failed" + " "  # Room already exists
                    else:
                        response = "create-success " + self.username  # Room creation
successful
                        chat_rooms.append({"name": room_name, "participants": []})
                    self.tcpClientSocket.send(response.encode())

                # JOIN CHAT ROOM
                elif message[0] == "JOIN-CHAT":
                    requested_room_name = message[1]
                    room_exists = False
                    ip_1, port_1 = get_peer_ip_port(self.username)
                    for room in chat_rooms:
                        if room["name"] == requested_room_name:
```

```
                            participants_list =
[f"{participant['ip']}:{participant['port']}"
                                                for participant in
room["participants"]]

                            room["participants"].append({"username": self.username,
"ip": ip_1, "port": port_1})
                            self.chatroom = requested_room_name
                            # Collect all participants' IP and port in the room

                            response = "join-chat-success " + "
".join(participants_list)
                            self.tcpClientSocket.send(response.encode())
                            room_exists = True
                            break
                    if not room_exists:
                        response = "No-Room found"
                        self.tcpClientSocket.send(response.encode())

                # ONLINE USERS LIST
                elif message[0] == "users-list-request":
                    if onlinePeers:
                        users = ",".join([peer["username"] for peer in onlinePeers])
                        response = f"users-list\n{users}"
                        self.tcpClientSocket.send(response.encode())
                    else:
                        # If no users are online, send a message indicating that
                        response = "users-list\nNo users online"
                        self.tcpClientSocket.send(response.encode())

                # CHAT ROOMS LIST
                elif message[0] == "rooms-list-request":
                    if chat_rooms:
                        # Filter out non-string room names and join them
                        chats = ",".join([room["name"] for room in chat_rooms if
isinstance(room["name"], str)])
                        response = f"chat-list\n{chats}"
                        self.tcpClientSocket.send(response.encode())
                    else:
                        # If no rooms are available, send a message indicating that
                        response = "chat-list\nNo rooms available"
                        self.tcpClientSocket.send(response.encode())

                # LEAVING the CHAT ROOM
                elif message[0] == "chatroom-leave":
```

```
                    self.leave_chat_room()

            except OSError as oErr:
                logging.error("OSError: {0}".format(oErr))

    # function for resetting the timeout for the udp timer thread
    def resetTimeout(self):
        self.udpServer.resetTimer()

    def leave_chat_room(self):
        for room in chat_rooms:
            for participant in room['participants']:
                if participant['username'] == self.username:
                    room['participants'].remove(participant)
                    self.chatroom = None
                    print(f"{self.username} left the chat room")
                    break


# implementation of the udp server thread for clients
class UDPServer(threading.Thread):

    # udp server thread initializations
    def __init__(self, username, clientSocket):
        threading.Thread.__init__(self)
        self.username = username
        # timer thread for the udp server is initialized
        self.timer = threading.Timer(3, self.waitHelloMessage)
        self.tcpClientSocket = clientSocket

    # if hello message is not received before timeout
    # then peer is disconnected
    def waitHelloMessage(self):
        for peer in onlinePeers:
            if peer["username"] is not None:
                onlinePeers.remove(peer)
                if self.username in tcpThreads:
                    del tcpThreads[self.username]
                    break
        self.tcpClientSocket.close()
        print("Removed " + self.username + " from online peers")

    # resets the timer for udp server
    def resetTimer(self):
        self.timer.cancel()
```

```
        self.timer = threading.Timer(3, self.waitHelloMessage)
        self.timer.start()


# tcp and udp server port initializations
print(f"{Fore.RED}Registry started...{Fore.RESET}")
port = 15400
portUDP = 15300

# db initialization
db = db.DB()

# gets the ip address of this peer
# first checks to get it for Windows devices
# if the device that runs this application is not windows
# it checks to get it for macOS devices
hostname = gethostname()
try:
    host = gethostbyname(hostname)
except gaierror:
    import netifaces as ni

    host = ni.ifaddresses('en0')[ni.AF_INET][0]['addr']

print(f"{Fore.RED}Registry IP address: {Fore.RESET}" + f"{Fore.LIGHTBLUE_EX}
{host}{Fore.RESET}")
print(f"{Fore.RED}Registry port number: {Fore.RESET}" +
f"{Fore.LIGHTBLUE_EX}{str(port)}{Fore.RESET}")

# onlinePeers list for an online account
onlinePeers = []
# chat rooms
chat_rooms = []
# accounts list for accounts
accounts = {}
# tcpThreads list for online client's thread
tcpThreads = {}

# tcp and udp socket initializations
tcpSocket = socket(AF_INET, SOCK_STREAM)
udpSocket = socket(AF_INET, SOCK_DGRAM)
tcpSocket.bind((host, port))
udpSocket.bind((host, portUDP))
tcpSocket.listen(5)
```

```
# input sockets that are listened to
inputs = [tcpSocket, udpSocket]

# log file initialization
logging.basicConfig(filename="registry.log", level=logging.INFO)

# as long as at least a socket exists to listen registry runs
while inputs:

    print(f"{Fore.LIGHTGREEN_EX}Listening for incoming connections... {Fore.RESET}")
    # monitors for the incoming connections
    readable, writable, exceptional = select.select(inputs, [], [])
    for s in readable:
        # if the message received comes to the tcp socket
        # the connection is accepted and a thread is created for it, and that thread
is started
        if s is tcpSocket:
            tcpClientSocket, addr = tcpSocket.accept()
            newThread = ClientThread(addr[0], addr[1], tcpClientSocket)
            newThread.start()
        # if the message received comes to the udp socket
        elif s is udpSocket:
            # received the incoming udp message and parses it
            message, clientAddress = s.recvfrom(1024)
            message = message.decode().split()
            # checks if it is a hello message
            if message[0] == "HELLO":
                # checks if the account that this hello message
                # is sent from is online
                if message[1] in tcpThreads:
                    # resets the timeout for that peer since the hello message is
received
                    tcpThreads[message[1]].resetTimeout()
                    print("Hello is received from " + message[1])
                    logging.info(
                        "Received from " + clientAddress[0] + ":" +
str(clientAddress[1]) + " -> " + " ".join(message))

# registry tcp socket is closed
tcpSocket.close()
```

6.2 Peer

```
from socket import *
```

```
import threading
import select
import logging
import colorama
from colorama import *

colorama.init(autoreset=True)


# Server side of peer
class PeerServer(threading.Thread):

    # Peer server initialization
    def __init__(self, username, peerServerPort):
        threading.Thread.__init__(self)
        # keeps the username of the peer
        self.peerServerHostname = None

        self.username = username
        # tcp socket for peer server
        self.tcpServerSocket = socket(AF_INET, SOCK_STREAM)
        # port number of the peer server
        self.peerServerPort = peerServerPort
        # if 1, then user is already chatting with someone
        # if 0, then user is not chatting with anyone
        self.isChatRequested = 0
        # keeps the socket for the peer that is connected to this peer
        self.connectedPeerSocket = None
        # keeps the ip of the peer connected to this peer's server
        self.connectedPeerIP = None
        # keeps the port number of the peer that is connected to this peer's server
        self.connectedPeerPort = None  # da elly el client byd5lo w2t el login
        # online status of the peer
        self.isOnline = True
        # keeps the username of the peer that this peer is chatting with
        self.chattingClientName = None
        # Available Chat Rooms
        self.chat_rooms = {}
        # connected peers
        self.connectedPeers = []

    # main method of the peer server thread
    def run(self):

        print(f"{Fore.GREEN}{Style.BRIGHT}Peer server started...{Fore.RESET}")
```

```
        # gets the ip address of this peer
        # first checks to get it for Windows devices
        # if the device that runs this application is not windows
        # it checks to get it for macOS devices
        hostname = gethostname()
        try:
            self.peerServerHostname = gethostbyname(hostname)
        except gaierror:
            import netifaces as ni
            self.peerServerHostname = ni.ifaddresses('en0')[ni.AF_INET][0]['addr']

        # ip address of this peer
        # self.peerServerHostname = 'localhost'
        # socket initializations for the server of the peer
        self.tcpServerSocket.bind((self.peerServerHostname, self.peerServerPort))
        self.tcpServerSocket.listen(4)
        # inputs sockets that should be listened to
        inputs = [self.tcpServerSocket]
        # the server listens as long as there is a socket to listen in the input list
and the user is online
        while inputs and self.isOnline:
            # monitors for the incoming connections
            try:
                readable, writable, exceptional = select.select(inputs +
self.connectedPeers, [], [], 1)
                # If a server waits to be connected enters here
                for s in readable:
                    # if the socket that is receiving the connection is
                    # the tcp socket of the peer's server, enters here
                    if s is self.tcpServerSocket:
                        # accepts the connection, and adds its connection socket to
the input list
                        # so that we can monitor that socket as well
                        connected, addr = s.accept()
                        connected.setblocking(0)
                        # inputs.append(connected)
                        self.connectedPeers.append(connected)
                        # if the user is not chatting, then the ip and the socket of
                        # this peer are assigned to server variables
                    # if the socket that receives the data is the one that
                    # is used to communicate with a connected peer, then enters here
                    else:
                        message = s.recv(1024).decode().split("\n")
                        if len(message) == 0:
```

```
                            s.close()
                            self.connectedPeers.remove(s)
                        elif message[0] == "chatroom-join":
                            print(message[1] + " has joined the chatroom.")
                            s.send("welcome".encode())
                        elif message[0] == "chatroom-leave":
                            print(f"{Fore.RED}{message[1]}" + f"{Fore.RED}has left
the chatroom.{Fore.RESET}")
                            s.close()
                            self.connectedPeers.remove(s)
                        elif message[0] == "chat-message":
                            username = message[1]
                            content = "\n".join(message[2:])
                            print(username + " -> " +
f"{Fore.LIGHTBLUE_EX}{content}{Fore.RESET}")
            # handles the exceptions, and logs them
            except OSError as oErr:
                logging.error("OSError: {0}".format(oErr))
            except ValueError as vErr:
                logging.error("ValueError: {0}".format(vErr))


# Client side of peer
class PeerClient(threading.Thread):
    # variable initializations for the client side of the peer
    def __init__(self, username, peerServer, chatroom, peersToConnect=None):
        threading.Thread.__init__(self)
        # keeps the ip address of the peer that this will connect

        # keeps the username of the peer
        self.chatroom = chatroom

        self.username = username
        # keeps the port number that this client should connect

        # client side tcp socket initialization
        self.tcpClientSocket = socket(AF_INET, SOCK_STREAM)
        # keeps the server of this client
        self.peerServer = peerServer
        # keeps the phrase used when creating the client
        # if the client is created with a phrase, it means this one received the
request
        # this phrase should be none if this is the client of the requester peer
        # self.responseReceived = responseReceived
        # keeps if this client is ending the chat or not
```

```
        self.isEndingChat = False

        if peersToConnect is not None:
            for peer in peersToConnect:
                peer_data = peer.split(":")
                if len(peer_data) >= 2:
                    peerHost = peer_data[0]
                    peerPort = int(peer_data[1])
                    sock = socket(AF_INET, SOCK_STREAM)
                    sock.connect((peerHost, peerPort))
                    message = "chatroom-join\n{}".format(self.username)
                    sock.send(message.encode())
                    self.peerServer.connectedPeers.append(sock)

    # main method of the peer client thread
    def run(self):
        print("Peer client started...")

        print(f'{Fore.RED}Chatroom joined Successfully. \nStart typing to send a
message'
              f'. Send ":exit" to leave the chatroom.{Fore.RESET}')

        while self.chatroom is not None:
            content = input()

            if content == ":exit":
                message = "chatroom-leave\n" + self.username
            else:
                message = "chat-message\n{}\n{}".format(self.username, content)

            for sock in self.peerServer.connectedPeers:
                try:
                    sock.send(message.encode())
                except:
                    pass

            if content == ":exit":
                self.chatroom = None
                for sock in self.peerServer.connectedPeers:
                    sock.close()


# main process of the peer
class peerMain:
```

```
    # peer initializations
    def __init__(self, username=None):
        # ip address of the registry
        self.registryName = input(f"{Fore.RED}Enter IP address of registry:
{Fore.LIGHTBLUE_EX}")
        # self.registryName = 'localhost'
        # port number of the registry
        self.registryPort = 15400
        # tcp socket connection to registry
        self.tcpClientSocket = socket(AF_INET, SOCK_STREAM)
        self.tcpClientSocket.connect((self.registryName, self.registryPort))
        # initializes udp socket which is used to send hello messages
        self.udpClientSocket = socket(AF_INET, SOCK_DGRAM)
        # udp port of the registry
        self.registryUDPPort = 15300
        # login info of the peer
        self.loginCredentials = (None, None)
        # online status of the peer
        self.isOnline = False
        # server port number of this peer
        self.peerServerPort = None
        # server of this peer
        self.peerServer = None
        # client of this peer
        self.peerClient = None
        # timer initialization
        self.timer = None
        # available chat rooms
        self.chat_rooms = []

        self.username = username

        choice = "0"
        # log file initialization
        logging.basicConfig(filename="peer.log", level=logging.INFO)
        # as long as the user is not logged out, asks to select an option in the menu
        while choice != "3":
            if not self.isOnline:
                # menu selection prompt
                choice = input(f"{Fore.RESET}Choose: \nCreate account: 1\nLogin:
2\n")

                # if choice is 1, creates an account with the username
                # and password entered by the user
                if choice == "1":
```

```
                    username = input(f"username: {Fore.LIGHTBLUE_EX}")
                    password = input(f"{Fore.RESET}password: {Fore.LIGHTBLUE_EX}")

                    self.createAccount(username, password)
                # if choice is 2 and user is not logged in, asks for the username
                # and the password to login
                elif choice == "2" and not self.isOnline:
                    username = input(f"{Fore.RESET}username: {Fore.LIGHTBLUE_EX}")
                    password = input(f"{Fore.RESET}password: {Fore.LIGHTBLUE_EX}")
                    # asks for the port number for server's tcp socket
                    peerServerPort = int(input(f"{Fore.RESET}Enter a port number for
peer server: "))

                    status = self.login(username, password, peerServerPort)
                    # is user logs in successfully, peer variables are set
                    if status == 1:
                        self.isOnline = True
                        self.loginCredentials = (username, password)
                        self.username = username
                        self.peerServerPort = peerServerPort
                        # creates the server thread for this peer, and runs it
                        self.peerServer = PeerServer(self.loginCredentials[0],
self.peerServerPort)
                        self.peerServer.start()
                        # hello message is sent to registry
                        self.sendHelloMessage()
                # if choice is 3 and user is logged in, then user is logged out
                # and peer variables are set, and server and client sockets are
closed
            elif self.isOnline:

                choice = input(f"{Fore.RESET}Choose: \nLogout: 3\nSearch: "
                               f"4\nStart a Chat: 5\nCreate Chat Room: 6\nJoin Chat:
7\n"
                               f"List Users: 8\nList Chat Rooms: 9\n")
                if choice == "3" and self.isOnline:
                    self.logout(1)
                    self.isOnline = False
                    self.loginCredentials = (None, None)
                    self.peerServer.isOnline = False
                    self.peerServer.tcpServerSocket.close()
                    if self.peerClient is not None:
                        self.peerClient.tcpClientSocket.close()
                    print("Logged out successfully")
                # is peer not logged in and exits the program?
```

```
                elif choice == "3":
                    self.logout(2)
                # if choice is 4 and user is online, then user is asked
                # for a username wanted to be searched
                elif choice == "4" and self.isOnline:
                    username = input("Username to be searched: ")
                    searchStatus = self.searchUser(username)
                    # if user is found its ip address is shown to the user
                    if searchStatus is not None and searchStatus != 0:
                        print("IP address of " + username + " is " + searchStatus)

                # if choice is 5 and user is online, then user is asked
                # to enter the username of the user that is wanted to be chatted
                elif choice == "5" and self.isOnline:
                    username = input("Enter the username of user to start chat: ")
                    searchStatus = self.searchUser(username)
                    # if searched user is found, then its ip address and port number
is retrieved,
                    # and a client thread is created
                    # main process waits for the client thread to finish its chat
                    if searchStatus is not None and searchStatus != 0:
                        searchStatus = searchStatus.split(":")
                        self.peerClient = PeerClient(searchStatus[0],
                                                     int(searchStatus[1]),
self.loginCredentials[0], self.peerServer,
                                                     None)
                        self.peerClient.start()
                        self.peerClient.join()
                # Creating chat room
                elif choice == "6":
                    chat_name = input(f"Enter the Chat room name to Create:
{Fore.RED}")
                    self.Create_Chat_Room(chat_name)
                # Joining Chat room
                elif choice == "7":
                    chat_name = input(f"Enter the Chat room name to Join:
{Fore.RED}")
                    self.Join_Chat_Room(chat_name)
                # listing online users
                elif choice == "8":
                    self.userList()
                # listing chat rooms
                elif choice == "9":
                    self.roomsList()
```

```
            # if this is the receiver side, then it will get the prompt to accept an
incoming request during the main
            # loop that's why response is evaluated in main process not the server
thread even though the prompt is
            # printed by server if the response is ok then a client is created for
this peer with the OK message and
            # that's why it will directly send an OK message to the requesting side
peer server and waits for the
            # user input main process waits for the client thread to finish its chat

            elif choice == "OK" and self.isOnline:
                okMessage = "OK " + self.loginCredentials[0]
                logging.info("Send to " + self.peerServer.connectedPeerIP + " -> " +
okMessage)
                self.peerServer.connectedPeerSocket.send(okMessage.encode())
                self.peerClient = PeerClient(self.username, self.peerServer,
                                             True, )
                self.peerClient.start()
                self.peerClient.join()
            # if user rejects the chat request then reject a message is sent to the
requester side
            elif choice == "REJECT" and self.isOnline:
                self.peerServer.connectedPeerSocket.send("REJECT".encode())
                self.peerServer.isChatRequested = 0
                logging.info("Send to " + self.peerServer.connectedPeerIP + " ->
REJECT")
            # if choice is cancel timer for hello message is canceled
            elif choice == "CANCEL":
                self.timer.cancel()
                break
        # if the main process is not ended with cancel selection
        # socket of the client is closed
        if choice != "CANCEL":
            self.tcpClientSocket.close()

    # account creation function
    def createAccount(self, username, password):
        # join a message to create an account is composed and sent to registry
        # if response is success then informs the user for account creation
        # if response exists then informs the user for account existence
```

```
        message = "JOIN " + username + " " + password
        logging.info("Send to " + self.registryName + ":" + str(self.registryPort) +
" -> " + message)
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode()
        logging.info("Received from " + self.registryName + " -> " + response)
        if response == "join-success":
            print("Account created...")
        elif response == "join-exist":
            print("choose another username or login...")

    def Create_Chat_Room(self, chat_room_name):
        message = "CREATE " + chat_room_name
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode()
        if response.startswith("create-success"):
            creator = response[len("create-success "):]  # Extract the creator's
username
            print(f"{Fore.GREEN}Chat room {chat_room_name} was created successfully
by {creator}{Fore.RESET}")
            self.Join_Chat_Room(chat_room_name)
            self.peerClient = PeerClient(self.username, self.peerServer,
chat_room_name)
            self.peerClient.start()
            self.peerClient.join()
        else:
            print(f"{Fore.RED}There is already a chat room with that
name.{Fore.RESET}")

    def userList(self):
        message = "users-list-request" + " "
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode()
        if response.startswith("users-list"):
            users = response[len("users-list"):].split('\n')
            print("Online Users:\n")
            if len(users) > 1:  # Check if there are users listed
                for user in users[1:]:
                    print(f"{Fore.RED}{user}" + ",")
                print(f"{Fore.RESET}\n")
            else:
                print("\tNo users currently online")
        else:
            print("Unexpected response:", response)
```

```
    def roomsList(self):
        message = "rooms-list-request"
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode()
        if response.startswith("chat-list"):
            chats = response[len("chat-list\n"):].split(',')
            print("Available Rooms:")
            for chat in chats:
                chat_name = chat.strip()
                if chat_name:  # Ensure the room name is not an empty string
                    print(f"{Fore.RED}{chat_name}{Fore.RESET}")
            print("\n")
            if len(chats) < 1:  # Check if only the header or no rooms exist
                print("\tNo rooms currently exist")
        else:
            print("Unexpected response:", response)

    def Join_Chat_Room(self, chat_room_name):
        message = "JOIN-CHAT " + chat_room_name + " "
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode()
        if response.startswith("join-chat-success"):
            participants_list = response.split()[1:]  # Extract the list of
participants
            if participants_list:
                print("Joined " + chat_room_name + ". Participants: " + ",
".join(participants_list))
                self.peerClient = PeerClient(self.username, self.peerServer,
chat_room_name, participants_list)
                self.peerClient.start()
                self.peerClient.join()
            else:
                # If participants_list is empty, initiate the PeerClient without the
list
                print("Joined " + chat_room_name + ". No participants in the room
yet.")
                self.peerClient = PeerClient(self.username, self.peerServer,
chat_room_name)
                self.peerClient.start()
                self.peerClient.join()
            self.tcpClientSocket.send("chatroom-leave".encode())
        elif response == "No-Room found":
            print("No room found by the name: " + chat_room_name)

    # login function
```

```
    def login(self, username, password, peerServerPort):
        # a login message is composed and sent to registry
        # and integer is returned according to each response
        message = "LOGIN " + username + " " + password + " " + str(peerServerPort)
        logging.info("Send to " + self.registryName + ":" + str(self.registryPort) +
" -> " + message)
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode()
        logging.info("Received from " + self.registryName + " -> " + response)
        if response == "login-success":
            print("Logged in successfully...")
            return 1
        elif response == "login-account-not-exist":
            print("Account does not exist...")
            return 0
        elif response == "login-online":
            print("Account is already online...")
            return 2
        elif response == "login-wrong-password":
            print("Wrong password...")
            return 3

    # logout function
    def logout(self, option):
        # a logout message is composed and sent to registry
        # timer is stopped
        if option == 1:
            message = "LOGOUT " + self.loginCredentials[0]
            self.timer.cancel()
        else:
            message = "LOGOUT"
        logging.info("Send to " + self.registryName + ":" + str(self.registryPort) +
" -> " + message)
        self.tcpClientSocket.send(message.encode())

    # function for searching an online user
    def searchUser(self, username):
        # a search message is composed and sent to registry
        # custom value is returned according to each response
        # to this search message
        message = "SEARCH " + username
        logging.info("Send to " + self.registryName + ":" + str(self.registryPort) +
" -> " + message)
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode().split()
```

```
        logging.info("Received from " + self.registryName + " -> " + "
".join(response))
        if response[0] == "search-success":
            print(username + " is found successfully...")
            return response[1]
        elif response[0] == "search-user-not-online":
            print(username + " is not online...")
            return 0
        elif response[0] == "search-user-not-found":
            print(username + " is not found")
            return None

    # function for sending hello message
    # a timer thread is used to send hello messages to udp socket of registry
    def sendHelloMessage(self):
        message = "HELLO " + self.loginCredentials[0]
        logging.info("Send to " + self.registryName + ":" + str(self.registryUDPPort)
+ " -> " + message)
        self.udpClientSocket.sendto(message.encode(), (self.registryName,
self.registryUDPPort))
        self.timer = threading.Timer(1, self.sendHelloMessage)
        self.timer.start()


# peer is started
main = peerMain()
```

6.3 Database

```
# Includes database operations
from pymongo import MongoClient


class DB:

    # db initializations
    def __init__(self):
        self.client = MongoClient('mongodb://localhost:27017/')
        self.db = self.client['p2p-chat']

    # checks if an account with the username exists
    def is_account_exist(self, username):
        return self.db.accounts.count_documents({'username': username}) > 0
```

```
    # registers a user
    def register(self, username, password):
        account = {
            "username": username,
            "password": password
        }
        self.db.accounts.insert_one(account)

    # retrieves the password for a given username
    def get_password(self, username):
        return self.db.accounts.find_one({"username": username})["password"]

    # checks if an account with the username online
    def is_account_online(self, username):
        return self.db.online_peers.count_documents({"username": username}) > 0

    # logs in the user
    def user_login(self, username, ip, port):
        online_peer = {
            "username": username,
            "ip": ip,
            "port": port
        }
        # self.db.online_peers.insert_one(online_peer)

    # logs out the user
    def user_logout(self, username):
        self.db.online_peers.remove_one({"username": username})

    # retrieves the ip address and the port number of the username
    def get_peer_ip_port(self, username):
        res = self.db.online_peers.find_one({"username": username})
        return (res["ip"], res["port"])
```

IV. PHASE 4

1. Code Updates in PeerServer Class

```
            else:
                try:
                    message = s.recv(1024).decode().split("\n")
                except:
                    s.close()
                    self.connectedPeers.remove(s)
                    if self.current_chat_room is None:
                        self.isChatRequested = False
                    continue

                if len(message) == 0:
                    s.close()
                    self.connectedPeers.remove(s)
                elif message[0] == "one-to-one-chat-request":
                    if self.isChatRequested:
                        if self.current_chat_room:
                            s.send(f"user-chatting\n{self.current_chat_room}".encode())
                        else:
                            s.send(f"user-chatting".encode())
                    else:
                        self.isChatRequested = 1
                        formatting_print(f"Receiving chat request from {message[1]}")
                        formatting_print("Would you like to accept (YES/NO)")
```

Figure 47

In the code snippet above, in the peer server class, we added an option that if message[0]== “one-to-one-chat-request”, then we’ll check if the user is chatting or not. If the user isn’t, we’ll send a message to the other user asking him if he wants to accept or decline messaging the first user.

```
elif message[0] == "one-to-one-chat-end":
    formatting_print(f"{message[1]} has ended the chat.")
    self.isChatRequested = 0
    s.close()
    self.connectedPeers.remove(s)
```

Figure 48

In the code snippet above, we added another option that allows the user to end or quit the chat. When the user chooses this option, we close the socket and set the chatting status of the user to 0 and we remove the socket from the connected peers.

2. Updates in the PeerClient Class

```
if chatRoom is not None:
    sock.send(f"chatroom-join\n{chatRoom}\n{self.peerServer.username}".encode())
    response = sock.recv(1024).decode()
    if response == "welcome":
        self.peerServer.connectedPeers.append(sock)

else:
    sock.send(f"one-to-one-chat-request\n{self.peerServer.username}".encode())
    response = sock.recv(1024).decode().split("\n")

    if response[0] == "user-chatting":
        self.peerServer.isChatRequested = 0
        if len(response) == 1:
            print("User is already in a private chat")
        else:
            print(f"User is in chatroom: {response[1]}")

    elif response[0] == "chat-request-reject":
        print("User has rejected the chat request")
        self.peerServer.isChatRequested = 0

    elif response[0] == "chat-request-accept":
        self.peerServer.connectedPeers.append(sock)
        print("User has accepted the chat.")
```

Figure 49

In the initialization, we check if the user is in a chatroom or not. If the user isn't a chatroom, we send a request to the peer server with the message "one-to-one-chat-request". We then receive and check the response. If the response == "user-chatting", then the user is already in a chat and isn't available. If the response == "chat-request-reject", we print that the user has rejected the chat request. Else, if the response is "chat-request-accept", we append the socket to the connected peers list.

```
while self.peerServer.isChatRequested:
    content = input("You -> ")

    if content == ":exit":
        if self.peerServer.current_chat_room is None:
            message = f"one-to-one-chat-end\n{self.peerServer.username}"
        else:
            message = f"chatroom-leave\n{self.peerServer.username}"
    else:
        message = f"chat-message\n{self.peerServer.username}\n{content}"

    for sock in self.peerServer.connectedPeers:
        sock.send(message.encode())

    if content == ":exit":
        self.peerServer.isChatRequested = 0
        self.peerServer.current_chat_room = None
        for sock in self.peerServer.connectedPeers:
            sock.close()
        self.peerServer.connectedPeers = []
```

Figure 50

In the code snippet above, we'll handle when the user decides to exit the chat. We first check if the user wants to exit a chatroom or a one-to-one chat.

3. Using the PeerMain Class

```
# to enter the username of the user that is wanted to be chatted
elif choice == "5" and self.isOnline:
    username = input("Enter the username of user to start chat: ")
    searchStatus = self.searchUser(username)
    # if searched user is found, then its ip address and port number is retrieved,
    # and a client thread is created
    # main process waits for the client thread to finish its chat
    if searchStatus is not None and searchStatus != 0:
        searchStatus = searchStatus.split(":")
        self.peerClient = PeerClient(self.peerServer, peersToConnect: [f"{searchStatus[0]}:{searchStatus[1]}"])
        self.peerClient.start()
        self.peerClient.join()

        self.peerClient = None
```

Figure 51

In the peerMain class, we added an option for the user to choose one of the functionalities of the application to enjoy. When the user chooses “5”, we started the “start-chat” option by letting the user type down the name of the user he wants to chat with. We then call searchUser to look for the user and if the user exists, we’ll then create an instance from the peer client class passing the peer server and the peer will connect too. We’ll start the peer client thread and then we’ll call the function “join” to hold the thread until the user exits the chat.

```
    elif choice == "YES" and self.isOnline:
        if self.peerServer.isChatRequested:
            sock = self.peerServer.connectedPeers[0]
            sock.send("chat-request-accept".encode())
            self.peerClient = PeerClient(self.peerServer)
            self.peerClient.start()
            self.peerClient.join()
        else:
            print("Invalid input. Please try again")
    # if user rejects the chat request then reject a message is sent to the requester side
    elif choice == "NO" and self.isOnline:
        if self.peerServer.isChatRequested:
            sock = self.peerServer.connectedPeers[0]
            sock.send("chat-request-reject".encode())
            self.peerServer.isChatRequested = 0
        else:
            print("Invalid input. Please try again")
```

Figure 52

In the code snippet above, the user chooses whether he wants to accept or reject the chat request. If he accepts the request we’ll send a message to the peerClient saying “chat-request-accept”. If the user rejects the request, a message will be sent to the peerClient saying “chat-request-reject”.

4. Testing

4.1. DBTEST

```
import time

from db import DB


# Includes database operations
def stress_register_users(user_count):
    start_time = time.time()
    db = DB()  # Instantiate DB class
    for i in range(user_count):
        username = f'test_user_{i}'
        password = f'test_password_{i}'
        db.register(username, password)  # Call register method in DB class
    total_time = time.time() - start_time
    return total_time


def stress_login_users(user_count):
    start_time = time.time()
    db = DB()  # Instantiate DB class
    for i in range(user_count):
        username = f'test_user_{i}'
        ip = '127.0.0.1'
        port = 12340 + i
        db.user_login(username, ip, port)  # Call user_login method in DB class
    total_time = time.time() - start_time
    return total_time


# Stress testing
register_users_time = stress_register_users(10000)
login_users_time = stress_login_users(10000)
# Call other stress testing methods here if uncommented and modified

# Print results
print(f"Register Users Time: {register_users_time} seconds")
print(f"Login Users Time: {login_users_time} seconds")
# Print other stress testing results if applicable
```

4.2. Stress_testing

```python
import time

from db import DB


# Includes database operations
def stress_register_users(user_count):
    start_time = time.time()
    db = DB()  # Instantiate DB class
    for i in range(user_count):
        username = f'test_user_{i}'
        password = f'test_password_{i}'
        db.register(username, password)  # Call register method in DB class
    total_time = time.time() - start_time
    return total_time


def stress_login_users(user_count):
    start_time = time.time()
    db = DB()  # Instantiate DB class
    for i in range(user_count):
        username = f'test_user_{i}'
        ip = '127.0.0.1'
        port = 12340 + i
        db.user_login(username, ip, port)  # Call user_login method in DB class
    total_time = time.time() - start_time
    return total_time


# Stress testing
register_users_time = stress_register_users(10000)
login_users_time = stress_login_users(10000)
# Call other stress testing methods here if uncommented and modified

# Print results
print(f"Register Users Time: {register_users_time} seconds")
print(f"Login Users Time: {login_users_time} seconds")
# Print other stress testing results if applicable
```

4.3. Test_peer

```
import unittest
from unittest.mock import patch, MagicMock
from io import StringIO
from contextlib import redirect_stdout
from colorama import Fore
from peer import peerMain

class TestPeerMain(unittest.TestCase):
    def setUp(self):
        # Initialize any resources needed for tests
        pass

    def tearDown(self):
        # Clean up any resources created during tests
        pass

    @patch('builtins.input', side_effect=["192.168.1.106","1", "testuser",
"testpassword"])
    def test_create_account(self, mock_input):
        with patch('sys.stdout', new_callable=StringIO) as mock_stdout:
            main = peerMain()
            main.createAccount("testuser", "testpassword")
            output = mock_stdout.getvalue().strip()
            self.assertEqual(output, "Account created...")

    @patch('builtins.input', side_effect=["192.168.1.106",'2', 'test_user',
'test_password', '1234'])
    @patch('sys.stdout', new_callable=StringIO)
    def test_login_success(self, mock_stdout, mock_input):
        with self.assertRaises(SystemExit):
            main = peerMain()
        output = mock_stdout.getvalue()
        self.assertIn("Logged in successfully", output)

    @patch('builtins.input', side_effect=['3'])
    @patch('sys.stdout', new_callable=StringIO)
    def test_logout(self, mock_stdout, mock_input):
        with self.assertRaises(SystemExit):
            main = peerMain()
        output = mock_stdout.getvalue()
        self.assertIn("Logged out successfully", output)
```

```
    @patch('builtins.input', side_effect=["192.168.1.106",'2', 'test_user',
'test_password', '1234','4', 'test_search_user'])
    @patch('sys.stdout', new_callable=StringIO)
    def test_search_user_online(self, mock_stdout, mock_input):
        with self.assertRaises(SystemExit):
            main = peerMain()
        output = mock_stdout.getvalue()
        self.assertIn("test_search_user is found successfully", output)

    @patch('builtins.input', side_effect=["192.168.1.106",'2', 'test_user',
'test_password', '1234','5', 'test_user_to_chat'])
    @patch('sys.stdout', new_callable=StringIO)
    def test_start_chat(self, mock_stdout, mock_input):
        with self.assertRaises(SystemExit):
            main = peerMain()
        output = mock_stdout.getvalue()
        self.assertIn("Peer client started", output)

    @patch('builtins.input', side_effect=["192.168.1.106",'2', 'test_user',
'test_password', '1234',"6", "testchatroom"])
    def test_create_chat_room(self, mock_input):
        with patch('sys.stdout', new_callable=StringIO) as mock_stdout:
            main = peerMain()
            main.Create_Chat_Room("testchatroom")
            output = mock_stdout.getvalue().strip()
            expected_output = f"{Fore.GREEN}Chat room testchatroom was created
successfully by None{Fore.RESET}"
            self.assertEqual(output, expected_output)

    @patch('builtins.input', side_effect=["192.168.1.106",'2', 'test_user',
'test_password', '1234',"7", "testchatroom"])
    def test_join_chat_room(self, mock_input):
        with patch('sys.stdout', new_callable=StringIO) as mock_stdout:
            main = peerMain()
            main.Join_Chat_Room("testchatroom")
            output = mock_stdout.getvalue().strip()
            expected_output = "No room found by the name: testchatroom"
            self.assertEqual(output, expected_output)

    @patch('builtins.input', side_effect=["192.168.1.106",'2', 'test_user',
'test_password', '1234',"8"])
    def test_user_list(self, mock_input):
        with patch('sys.stdout', new_callable=StringIO) as mock_stdout:
            main = peerMain()
```

```
            main.userList()
            output = mock_stdout.getvalue().strip()
            self.assertIn("No users currently online", output)

    @patch('builtins.input', side_effect=["9"])
    def test_rooms_list(self, mock_input):
        with patch('sys.stdout', new_callable=StringIO) as mock_stdout:
            main = peerMain()
            main.roomsList()
            output = mock_stdout.getvalue().strip()
            self.assertIn("No rooms currently exist", output)

    # Add more test methods for other functions...

if __name__ == "__main__":
    unittest.main()
```

4.4. Test_registry

```
import sys
import unittest
from unittest.mock import MagicMock, patch, Mock
from io import StringIO
import socket
import threading
import time

import peer
# Import the functions and classes you want to test
from registry import *


class TestClientThread(unittest.TestCase):
    def setUp(self):
        self.mock_socket = Mock()
        self.peer_instance = peer.peerMain()
        self.peer_instance.tcpClientSocket = MagicMock()
        self.peer_instance.registryName = 'testRegistry'
        self.peer_instance.registryPort = 15400

    def tearDown(self):
        self.peer_instance.tcpClientSocket.close()
```

```
    @patch('threading.Timer')
    def test_send_hello_message(self, mock_timer):
        # Arrange
        self.peer_instance.loginCredentials = ('testUser', 'testPassword')
        self.peer_instance.udpClientSocket.sendto.return_value = None
        mock_timer_instance = mock_timer.return_value
        mock_timer_instance.start.return_value = None

        # Act
        self.peer_instance.sendHelloMessage()

        # Assert
        self.peer_instance.udpClientSocket.sendto.assert_called_once()
        mock_timer.assert_called_once_with(1, self.peer_instance.sendHelloMessage)
        mock_timer_instance.start.assert_called_once()

    def test_create_account_success(self):
        # Arrange
        username = 'testUser'
        password = 'testPassword'
        self.peer_instance.tcpClientSocket.recv.return_value = b'join-success'
        sys.stdout = StringIO()

        # Act
        self.peer_instance.createAccount(username, password)

        # Assert
        self.peer_instance.tcpClientSocket.send.assert_called_once_with(b'JOIN
testUser testPassword')
        output = sys.stdout.getvalue().strip()
        self.assertEqual(output, Fore.LIGHTGREEN_EX + "Account created...")




    # def test_login_account_not_exist(self):
    #     # Mocking the database to return False for account existence
    #     with unittest.mock.patch('registry.db.is_account_exist',
return_value=False):
    #         client_thread = ClientThread('127.0.0.1', 1234, self.mock_socket)
    #         # Replace message[1], message[2], message[3] with appropriate values
    #         client_thread.message = ["", "username_not_existing", "password",
"port_number"]
    #         client_thread.login()
```

```
    #         self.mock_socket.send.assert_called_with("login-account-not-
exist".encode())

    # def test_signup(self):
    #     # Test JOIN with a nonexistent account
    #     client_thread = ClientThread('127.0.0.1', 1234, self.mock_socket)
    #     self.assertTrue(client_thread.join_account('new_user', 'password'))

    # Add more test cases for other functions in ClientThread


class TestUDPServer(unittest.TestCase):
    def setUp(self):
        # Set up any necessary objects or data for testing
        self.mock_tcp_socket = MagicMock()
        self.mock_tcp_socket.send.return_value = None

    def test_wait_hello_message(self):
        # Test waitHelloMessage method
        udp_server = UDPServer('test_user', self.mock_tcp_socket)
        udp_server.waitHelloMessage()  # This should not raise an exception

    # Add more test cases for other functions in UDPServer


if __name__ == '__main__':
    unittest.main()
```

5. Repo Link

https://github.com/AbdelrahmanKhaled18/Peer_to_Peer_Multi_Chatting_App

Figure 53

6. Presentation Link

https://drive.google.com/drive/folders/1xF3757HSf5gKEPWL11n4O7dTRPmlQFZy?usp=sharing

Figure 54: Presentation Link

7. Appendix 3

7.1. Registry

```
from socket import *
import threading
import select
import logging
import db
import bcrypt
import colorama
from colorama import *

colorama.init(autoreset=True)


# This class is used to process the peer messages sent to registry
# for each peer connected to registry, a new client thread is created

def is_account_online(username):
    for peer in onlinePeers:
        if peer["username"] == username:
            return True
    return False


def get_peer_ip_port(username):
```

```
    for peer in onlinePeers:
        if peer["username"] == username:
            return (peer["ip"], peer["port"])
    return (None, None)


class ClientThread(threading.Thread):
    # initializations for client thread
    def __init__(self, ip, port, tcpClientSocket):
        threading.Thread.__init__(self)
        # ip of the connected peer
        self.chatroom = None
        self.lock = None
        self.ip = ip
        # port number of the connected peer
        self.port = port
        # socket of the peer
        self.tcpClientSocket = tcpClientSocket
        # username, online status and udp server initializations
        self.username = None
        self.isOnline = True
        self.udpServer = None
        print("New thread started for " + ip + ":" + str(port))

    # main of the thread
    def run(self):
        # locks for thread which will be used for thread synchronization
        self.lock = threading.Lock()
        print("Connection from: " + self.ip + ":" + str(port))
        print("IP Connected: " + self.ip)

        while True:
            try:
                # waits for incoming messages from peers
                message = self.tcpClientSocket.recv(1024).decode().split()
                #   JOIN #
                if message[0] == "JOIN":
                    self.create_account(message[1], message[2])
                #   LOGIN    #
                elif message[0] == "LOGIN":
                    self.login(message[1], message[2], message[3])

                #   LOGOUT  #
                elif message[0] == "LOGOUT":
                    # if a user is online,
```

```
                    # removes the user from onlinePeers list
                    # and removes the thread for this user from tcpThreads
                    # socket is closed, and the timer thread of the udp for this
                    # user is canceled
                    if self.chatroom is not None:
                        self.leave_chat_room()

                    if len(message) > 1 and message[1] is not None and
is_account_online(message[1]):
                        for peer in onlinePeers:
                            if peer["username"] == message[1]:
                                onlinePeers.remove(peer)
                                break
                        self.lock.acquire()
                        try:
                            if message[1] in tcpThreads:
                                del tcpThreads[message[1]]
                        finally:
                            self.lock.release()
                        print(self.ip + ":" + str(self.port) + " is logged out")
                        self.tcpClientSocket.close()
                        self.udpServer.timer.cancel()
                        break
                    else:
                        self.tcpClientSocket.close()
                        break
                #   SEARCH #
                elif message[0] == "SEARCH":
                    # checks if an account with the username exists
                    if db.is_account_exist(message[1]):
                        # checks if the account is online
                        # and sends the related response to peer
                        if is_account_online(message[1]):
                            peer_info_ip, peer_info_port =
get_peer_ip_port(message[1])
                            if peer_info_ip and peer_info_port:
                                response = f"search-success
{peer_info_ip}:{peer_info_port}"

                                self.tcpClientSocket.send(response.encode())
                            else:
                                response = "search-user-not-found"  # Unable to fetch
IP and port for the username

                                self.tcpClientSocket.send(response.encode())
```

```
                        else:
                            response = "search-user-not-online"

                            self.tcpClientSocket.send(response.encode())
                    # enters if username does not exist
                    else:
                        response = "search-user-not-found"

                        self.tcpClientSocket.send(response.encode())

                # CREATE CHAT ROOM
                elif message[0] == "CREATE":
                    room_name = message[1]  # Extract room name from the message
                    room_exists = any(room['name'] == room_name for room in
chat_rooms)
                    if room_exists:
                        response = "create-failed" + " "  # Room already exists
                    else:
                        response = "create-success " + self.username  # Room creation
successful
                        chat_rooms.append({"name": room_name, "participants": []})
                    self.tcpClientSocket.send(response.encode())

                # JOIN CHAT ROOM
                elif message[0] == "JOIN-CHAT":
                    requested_room_name = message[1]
                    room_exists = False
                    ip_1, port_1 = get_peer_ip_port(self.username)
                    for room in chat_rooms:
                        if room["name"] == requested_room_name:
                            participants_list =
[f"{participant['ip']}:{participant['port']}"
                                                    for participant in
room["participants"]]

                            room["participants"].append({"username": self.username,
"ip": ip_1, "port": port_1})
                            self.chatroom = requested_room_name
                            # Collect all participants' IP and port in the room

                            response = "join-chat-success " + "
".join(participants_list)
                            self.tcpClientSocket.send(response.encode())
                            room_exists = True
                            break
```

```
                    if not room_exists:
                        response = "No-Room found"
                        self.tcpClientSocket.send(response.encode())

                # ONLINE USERS LIST
                elif message[0] == "users-list-request":
                    if onlinePeers:
                        users = ",".join([peer["username"] for peer in onlinePeers])
                        response = f"users-list\n{users}"
                        self.tcpClientSocket.send(response.encode())
                    else:
                        # If no users are online, send a message indicating that
                        response = "users-list\nNo users online"
                        self.tcpClientSocket.send(response.encode())

                # CHAT ROOMS LIST
                elif message[0] == "rooms-list-request":
                    if chat_rooms:
                        # Filter out non-string room names and join them
                        chats = ",".join([room["name"] for room in chat_rooms if
isinstance(room["name"], str)])
                        response = f"chat-list\n{chats}"
                        self.tcpClientSocket.send(response.encode())
                    else:
                        # If no rooms are available, send a message indicating that
                        response = "chat-list\nNo rooms available"
                        self.tcpClientSocket.send(response.encode())

                # LEAVING the CHAT ROOM
                elif message[0] == "chatroom-leave":
                    self.leave_chat_room()
            except OSError as oErr:
                pass

    # function for resetting the timeout for the udp timer thread
    def resetTimeout(self):
        self.udpServer.resetTimer()

    def login(self, username, password, port_login):
        # login-account-not-exist is sent to peer
        # if an account with the username does not exist
        if not db.is_account_exist(username):
            response = "login-account-not-exist"

            self.tcpClientSocket.send(response.encode())
```

```
        # login-online is sent to peer
        # if an account with the username already online
        elif is_account_online(username):
            response = "login-online"
            self.tcpClientSocket.send(response.encode())
            # onlinePeers.append(message[1])
        # login-success is sent to peer
        # if an account with the username exists and not online
        else:
            # retrieves the account's password, and checks if the one entered by the
user is correct
            retrievedPass = db.get_password(username)
            # if the password is correct, then peer's thread is added to thread list
            # peer is added to db with its username, port number, and ip address
            if bcrypt.checkpw(password.encode('utf-8'), retrievedPass):
                self.username = username
                self.lock.acquire()
                try:
                    tcpThreads[self.username] = self
                finally:
                    self.lock.release()

                db.user_login(username, self.ip, port_login)
                # login-success is sent to peer,
                # and an udp server thread is created for this peer, and thread is
started
                # timer thread of the udp server is started
                response = "login-success"
                onlinePeers.append({"username": self.username, "ip": self.ip, "port":
port_login})

                self.tcpClientSocket.send(response.encode())
                self.udpServer = UDPServer(self.username, self.tcpClientSocket)
                self.udpServer.start()
                self.udpServer.timer.start()
            # if password not matches and then login-wrong-password response is sent
            else:
                response = "login-wrong-password"
                self.tcpClientSocket.send(response.encode())

    def create_account(self, username, password):
        # join-exist is sent to peer,
        # if an account with this username already exists
        if db.is_account_exist(username):
            response = "join-exist"
```

```
            print("From-> " + self.ip + ":" + str(self.port) + " " + response)

            self.tcpClientSocket.send(response.encode())
        # join-success is sent to peer,
        # if an account with this username does not exist, and the account is created
        else:
            hashed_password = bcrypt.hashpw(password.encode('utf-8'),
bcrypt.gensalt())
            db.register(username, hashed_password)
            response = "join-success"
            self.tcpClientSocket.send(response.encode())

    def leave_chat_room(self):
        for room in chat_rooms:
            for participant in room['participants']:
                if participant['username'] == self.username:
                    room['participants'].remove(participant)
                    self.chatroom = None
                    print(f"{self.username} left the chat room")
                    break


# implementation of the udp server thread for clients
class UDPServer(threading.Thread):

    # udp server thread initializations
    def __init__(self, username, clientSocket):
        threading.Thread.__init__(self)
        self.username = username
        # timer thread for the udp server is initialized
        self.timer = threading.Timer(3, self.waitHelloMessage)
        self.tcpClientSocket = clientSocket

    # if hello message is not received before timeout
    # then peer is disconnected
    def waitHelloMessage(self):
        for peer in onlinePeers:
            if peer["username"] is not None:
                onlinePeers.remove(peer)
                if self.username in tcpThreads:
                    del tcpThreads[self.username]
                    break
        self.tcpClientSocket.close()
        print("Removed " + self.username + " from online peers")
```

```
    # resets the timer for udp server
    def resetTimer(self):
        self.timer.cancel()
        self.timer = threading.Timer(3, self.waitHelloMessage)
        self.timer.start()


# tcp and udp server port initializations
print(f"{Fore.RED}Registry started...{Fore.RESET}")
port = 15400
portUDP = 15300

# db initialization
db = db.DB()

# gets the ip address of this peer
# first checks to get it for Windows devices
# if the device that runs this application is not windows
# it checks to get it for macOS devices
hostname = gethostname()
try:
    host = gethostbyname(hostname)
except gaierror:
    import netifaces as ni

    host = ni.ifaddresses('en0')[ni.AF_INET][0]['addr']

print(f"{Fore.RED}Registry IP address: {Fore.RESET}" + f"{Fore.LIGHTBLUE_EX}
{host}{Fore.RESET}")
print(f"{Fore.RED}Registry port number: {Fore.RESET}" +
f"{Fore.LIGHTBLUE_EX}{str(port)}{Fore.RESET}")

# onlinePeers list for an online account
onlinePeers = []
# chat rooms
chat_rooms = []
# accounts list for accounts
accounts = {}
# tcpThreads list for online client's thread
tcpThreads = {}

# tcp and udp socket initializations
tcpSocket = socket(AF_INET, SOCK_STREAM)
udpSocket = socket(AF_INET, SOCK_DGRAM)
tcpSocket.bind((host, port))
```

```
udpSocket.bind((host, portUDP))
tcpSocket.listen(5)

# input sockets that are listened to
inputs = [tcpSocket, udpSocket]

# log file initialization


# as long as at least a socket exists to listen registry runs
while inputs:

    print(f"{Fore.LIGHTGREEN_EX}Listening for incoming connections... {Fore.RESET}")
    # monitors for the incoming connections
    readable, writable, exceptional = select.select(inputs, [], [])
    for s in readable:
        # if the message received comes to the tcp socket
        # the connection is accepted and a thread is created for it, and that thread
is started
        if s is tcpSocket:
            tcpClientSocket, addr = tcpSocket.accept()
            newThread = ClientThread(addr[0], addr[1], tcpClientSocket)
            newThread.start()
        # if the message received comes to the udp socket
        elif s is udpSocket:
            # received the incoming udp message and parses it
            message, clientAddress = s.recvfrom(1024)
            message = message.decode().split()
            # checks if it is a hello message
            if message[0] == "HELLO":
                # checks if the account that this hello message
                # is sent from is online
                if message[1] in tcpThreads:
                    # resets the timeout for that peer since the hello message is
received
                    tcpThreads[message[1]].resetTimeout()
                    print("Hello is received from " + message[1])

# registry tcp socket is closed
tcpSocket.close()
```

7.2. Peer

```
from socket import *
```

```
import threading
import select
import logging
import colorama
from colorama import *

colorama.init(autoreset=True)


def formatting_print(msg):
    print("\u001B[s", end="")
    print("\u001B[A", end="")
    print("\u001B[999D", end="")
    print("\u001B[S", end="")
    print("\u001B[L", end="")
    print(msg, end="")
    print("\u001B[u", end="", flush=True)


# Server side of peer
class PeerServer(threading.Thread):

    # Peer server initialization
    def __init__(self, username, peerServerPort):
        threading.Thread.__init__(self)
        # keeps the username of the peer
        self.peerServerHostname = None

        self.username = username
        # tcp socket for peer server
        self.tcpServerSocket = socket(AF_INET, SOCK_STREAM)
        # port number of the peer server
        self.peerServerPort = peerServerPort
        # if 1, then user is already chatting with someone
        # if 0, then user is not chatting with anyone
        self.isChatRequested = 0
        # keeps the socket for the peer that is connected to this peer
        self.connectedPeerSocket = None
        # keeps the ip of the peer connected to this peer's server
        self.connectedPeerIP = None
        # keeps the port number of the peer that is connected to this peer's server
        self.connectedPeerPort = None  # da elly el client byd5lo w2t el login
        # online status of the peer
        self.isOnline = True
        # keeps the username of the peer that this peer is chatting with
```

```
        self.chattingClientName = None
        # Available Chat Rooms
        self.current_chat_room = None
        # connected peers
        self.connectedPeers = []

    # main method of the peer server thread
    def run(self):

        print(f"{Fore.GREEN}{Style.BRIGHT}Peer server started...{Fore.RESET}")

        # gets the ip address of this peer
        # first checks to get it for Windows devices
        # if the device that runs this application is not windows
        # it checks to get it for macOS devices
        hostname = gethostname()
        try:
            self.peerServerHostname = gethostbyname(hostname)
        except gaierror:
            import netifaces as ni
            self.peerServerHostname = ni.ifaddresses('en0')[ni.AF_INET][0]['addr']

        # ip address of this peer
        # self.peerServerHostname = 'localhost'
        # socket initializations for the server of the peer
        self.tcpServerSocket.bind((self.peerServerHostname, self.peerServerPort))
        self.tcpServerSocket.listen(4)
        # inputs sockets that should be listened to
        inputs = [self.tcpServerSocket]
        # the server listens as long as there is a socket to listen in the input list
and the user is online
        while inputs and self.isOnline:
            # monitors for the incoming connections
            try:
                readable, writable, exceptional = select.select(inputs +
self.connectedPeers, [], [], 1)
                # If a server waits to be connected enters here
                for s in readable:
                    # if the socket that is receiving the connection is
                    # the tcp socket of the peer's server, enters here
                    if s is self.tcpServerSocket:
                        # accepts the connection, and adds its connection socket to
the input list
                        # so that we can monitor that socket as well
                        connected, addr = s.accept()
```

```
                        connected.setblocking(0)
                        # inputs.append(connected)
                        self.connectedPeers.append(connected)
                        # if the user is not chatting, then the ip and the socket of
                        # this peer are assigned to server variables
                    # if the socket that receives the data is the one that
                    # is used to communicate with a connected peer, then enters here
                    else:
                        try:
                            message = s.recv(1024).decode().split("\n")
                        except:
                            s.close()
                            self.connectedPeers.remove(s)
                            if self.current_chat_room is None:
                                self.isChatRequested = False
                            continue

                        if len(message) == 0:
                            s.close()
                            self.connectedPeers.remove(s)
                        elif message[0] == "one-to-one-chat-request":
                            if self.isChatRequested:
                                if self.current_chat_room:
                                    s.send(f"user-
chatting\n{self.current_chat_room}".encode())
                                else:
                                    s.send(f"user-chatting".encode())
                            else:
                                self.isChatRequested = 1
                                formatting_print(f"Receiving chat request from
{message[1]}")
                                formatting_print("Would you like to accept (YES/NO)")

                        elif message[0] == "chatroom-join":
                            if message[1] == self.current_chat_room:
                                formatting_print(f"{message[2]} has joined the
chatroom.")
                                s.send("welcome".encode())
                            else:
                                s.send("user-not-in-chatroom".encode())

                        elif message[0] == "chatroom-leave":
                            formatting_print(
                                f"{Fore.RED}{message[1]} " + f"{Fore.RED}has left the
chatroom.{Fore.RESET}")
```

```
                            s.close()
                            self.connectedPeers.remove(s)
                        elif message[0] == "chat-message":
                            username = message[1]
                            content = "\n".join(message[2:])
                            formatting_print(username + " -> " +
f"{Fore.LIGHTBLUE_EX}{content}{Fore.RESET}")

                        elif message[0] == "one-to-one-chat-end":
                            formatting_print(f"{message[1]} has ended the chat.")
                            self.isChatRequested = 0
                            s.close()
                            self.connectedPeers.remove(s)

            # handles the exceptions, and logs them
            except OSError as oErr:
                logging.error("OSError: {0}".format(oErr))
            except ValueError as vErr:
                logging.error("ValueError: {0}".format(vErr))


# Client side of peer
class PeerClient(threading.Thread):
    # variable initializations for the client side of the peer
    def __init__(self, peerServer, peersToConnect=None, chatRoom=None):
        threading.Thread.__init__(self)
        # keeps the ip address of the peer that this will connect
        self.peerServer = peerServer

        # keeps the username of the peer
        self.peerServer.current_chat_room = chatRoom

        self.peerServer.isChatRequested = 1
        # keeps the port number that this client should connect

        # client side tcp socket initialization
        self.tcpClientSocket = socket(AF_INET, SOCK_STREAM)
        # keeps the server of this client

        # keeps the phrase used when creating the client
        # if the client is created with a phrase, it means this one received the
request
        # this phrase should be none if this is the client of the requester peer
        # self.responseReceived = responseReceived
        # keeps if this client is ending the chat or not
```

```
        self.isEndingChat = False

        if peersToConnect is not None:
            for peer in peersToConnect:
                peer_data = peer.split(":")
                if len(peer_data) >= 2:
                    peerHost = peer_data[0]
                    peerPort = int(peer_data[1])
                    sock = socket(AF_INET, SOCK_STREAM)
                    sock.connect((peerHost, peerPort))
                    if chatRoom is not None:
                        sock.send(f"chatroom-
join\n{chatRoom}\n{self.peerServer.username}".encode())
                        response = sock.recv(1024).decode()
                        if response == "welcome":
                            self.peerServer.connectedPeers.append(sock)

                    else:
                        sock.send(f"one-to-one-chat-
request\n{self.peerServer.username}".encode())
                        response = sock.recv(1024).decode().split("\n")

                        if response[0] == "user-chatting":
                            self.peerServer.isChatRequested = 0
                            if len(response) == 1:
                                print("User is already in a private chat")
                            else:
                                print(f"User is in chatroom: {response[1]}")

                        elif response[0] == "chat-request-reject":
                            print("User has rejected the chat request")
                            self.peerServer.isChatRequested = 0

                        elif response[0] == "chat-request-accept":
                            self.peerServer.connectedPeers.append(sock)
                            print("User has accepted the chat.")

    # main method of the peer client thread
    def run(self):
        if self.peerServer.isChatRequested:
            if self.peerServer.current_chat_room:
                print(f"{Fore.RED}Chatroom joined Successfully.{Fore.RESET}")
            print(f"{Fore.RED}Start typing to send a message. Send ':exit' to leave
the chatroom.{Fore.RESET}")
```

```
        while self.peerServer.isChatRequested:
            content = input("-> ")

            if content == ":exit":
                if self.peerServer.current_chat_room is None:
                    message = f"one-to-one-chat-end\n{self.peerServer.username}"
                else:
                    message = f"chatroom-leave\n{self.peerServer.username}"
            else:
                message = f"chat-message\n{self.peerServer.username}\n{content}"

            for sock in self.peerServer.connectedPeers:
                sock.send(message.encode())

            if content == ":exit":
                self.peerServer.isChatRequested = 0
                self.peerServer.current_chat_room = None
                for sock in self.peerServer.connectedPeers:
                    sock.close()
                self.peerServer.connectedPeers = []


# main process of the peer
class peerMain:

    # peer initializations
    def __init__(self, username=None):
        # ip address of the registry
        self.registryName = input(f"{Fore.RED}Enter IP address of registry:
{Fore.LIGHTBLUE_EX}")
        # self.registryName = 'localhost'
        # port number of the registry
        self.registryPort = 15400
        # tcp socket connection to registry
        self.tcpClientSocket = socket(AF_INET, SOCK_STREAM)
        self.tcpClientSocket.connect((self.registryName, self.registryPort))
        # initializes udp socket which is used to send hello messages
        self.udpClientSocket = socket(AF_INET, SOCK_DGRAM)
        # udp port of the registry
        self.registryUDPPort = 15300
        # login info of the peer
        self.loginCredentials = (None, None)
        # online status of the peer
        self.isOnline = False
        # server port number of this peer
```

```
        self.peerServerPort = None
        # server of this peer
        self.peerServer = None
        # client of this peer
        self.peerClient = None
        # timer initialization
        self.timer = None
        # available chat rooms
        self.chat_rooms = []
        self.username = username

        choice = "0"
        # log file initialization
        logging.basicConfig(filename="peer.log", level=logging.INFO)
        # as long as the user is not logged out, asks to select an option in the menu
        while choice != "3":
            if not self.isOnline:
                # menu selection prompt
                choice = input(f"{Fore.RESET}Choose: \nCreate account: 1\nLogin:
2\n")

                # if choice is 1, creates an account with the username
                # and password entered by the user
                if choice == "1":
                    username = input(f"username: {Fore.LIGHTBLUE_EX}")
                    password = input(f"{Fore.RESET}password: {Fore.LIGHTBLUE_EX}")

                    self.createAccount(username, password)
                # if choice is 2 and user is not logged in, asks for the username
                # and the password to login
                elif choice == "2" and not self.isOnline:
                    username = input(f"{Fore.RESET}username: {Fore.LIGHTBLUE_EX}")
                    password = input(f"{Fore.RESET}password: {Fore.LIGHTBLUE_EX}")
                    # asks for the port number for server's tcp socket
                    peerServerPort = int(input(f"{Fore.RESET}Enter a port number for
peer server: "))

                    status = self.login(username, password, peerServerPort)
                    # is user logs in successfully, peer variables are set
                    if status == 1:
                        self.isOnline = True
                        self.loginCredentials = (username, password)
                        self.username = username
                        self.peerServerPort = peerServerPort
                        # creates the server thread for this peer, and runs it
```

```
                        self.peerServer = PeerServer(self.loginCredentials[0],
self.peerServerPort)
                        self.peerServer.start()
                        # hello message is sent to registry
                        self.sendHelloMessage()
                # if choice is 3 and user is logged in, then user is logged out
                # and peer variables are set, and server and client sockets are
closed
            elif self.isOnline:

                choice = input(f"{Fore.RESET}Choose: \nLogout: 3\nSearch: "
                               f"4\nStart a Chat: 5\nCreate Chat Room: 6\nJoin Chat:
7\n"
                               f"List Users: 8\nList Chat Rooms: 9\n")
                if choice == "3" and self.isOnline:
                    self.logout(1)
                    self.isOnline = False
                    self.loginCredentials = (None, None)
                    self.peerServer.isOnline = False
                    self.peerServer.tcpServerSocket.close()
                    if self.peerClient is not None:
                        self.peerClient.tcpClientSocket.close()
                    print("Logged out successfully")
                # is peer not logged in and exits the program?
                elif choice == "3":
                    self.logout(2)
                # if choice is 4 and user is online, then user is asked
                # for a username wanted to be searched
                elif choice == "4" and self.isOnline:
                    username = input("Username to be searched: ")
                    searchStatus = self.searchUser(username)
                    # if user is found its ip address is shown to the user
                    if searchStatus is not None and searchStatus != 0:
                        searchStatus = searchStatus.split(":")
                        print(f"{Fore.RED}IP address of " + username + " is " +
                              searchStatus[0] + " and the Port number is " +
searchStatus[1] + f"{Fore.RESET}")

                # if choice is 5 and user is online, then user is asked
                # to enter the username of the user that is wanted to be chatted
                elif choice == "5" and self.isOnline:
                    username = input("Enter the username of user to start chat: ")
                    searchStatus = self.searchUser(username)
                    # if searched user is found, then its ip address and port number
is retrieved,
```

```
                    # and a client thread is created
                    # main process waits for the client thread to finish its chat
                    if searchStatus is not None and searchStatus != 0:
                        searchStatus = searchStatus.split(":")
                        self.peerClient = PeerClient(self.peerServer,
[f"{searchStatus[0]}:{searchStatus[1]}"])
                        self.peerClient.start()
                        self.peerClient.join()
                        self.peerClient = None


                # Creating chat room
                elif choice == "6":
                    chat_name = input(f"Enter the Chat room name to Create:
{Fore.RED}")
                    self.Create_Chat_Room(chat_name)
                # Joining Chat room
                elif choice == "7":
                    chat_name = input(f"Enter the Chat room name to Join:
{Fore.RED}")
                    self.Join_Chat_Room(chat_name)
                # listing online users
                elif choice == "8":
                    self.userList()
                # listing chat rooms
                elif choice == "9":
                    self.roomsList()

                elif choice == "YES" and self.isOnline:
                    if self.peerServer.isChatRequested:
                        sock = self.peerServer.connectedPeers[0]
                        sock.send("chat-request-accept".encode())
                        self.peerClient = PeerClient(self.peerServer)
                        self.peerClient.start()
                        self.peerClient.join()
                    else:
                        print("Invalid input. Please try again")
                # if user rejects the chat request then reject a message is sent to
the requester side
                elif choice == "NO" and self.isOnline:
                    if self.peerServer.isChatRequested:
                        sock = self.peerServer.connectedPeers[0]
                        sock.send("chat-request-reject".encode())
                        self.peerServer.isChatRequested = 0
                    else:
```

```
                        print("Invalid input. Please try again")
            # if choice is cancel timer for hello message is canceled
            # elif choice == "CANCEL":
            #     self.timer.cancel()
            #     break
        # if the main process is not ended with cancel selection
        # socket of the client is closed
        # if choice != "CANCEL":
        #     self.tcpClientSocket.close()

    # account creation function
    def createAccount(self, username, password):
        # join a message to create an account is composed and sent to registry
        # if response is success then informs the user for account creation
        # if response exists then informs the user for account existence
        message = "JOIN " + username + " " + password
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode()
        if response == "join-success":
            print("Account created...")
        elif response == "join-exist":
            print("choose another username or login...")

    def Create_Chat_Room(self, chat_room_name):
        message = "CREATE " + chat_room_name
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode()
        if response.startswith("create-success"):
            creator = response[len("create-success "):]  # Extract the creator's
username
            print(f"{Fore.GREEN}Chat room {chat_room_name} was created successfully
by {creator}{Fore.RESET}")
            self.Join_Chat_Room(chat_room_name)
            # self.peerClient = PeerClient(self.username, self.peerServer,
chat_room_name)
            # self.peerClient.start()
            # self.peerClient.join()
        else:
            print(f"{Fore.RED}There is already a chat room with that
name.{Fore.RESET}")

    def userList(self):
        message = "users-list-request" + " "
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode()
```

```
        if response.startswith("users-list"):
            users = response[len("users-list"):].split('\n')
            print("Online Users:\n")
            if len(users) > 1:  # Check if there are users listed
                for user in users[1:]:
                    print(f"{Fore.RED}{user}" + ",")
                print(f"{Fore.RESET}\n")
            else:
                print("\tNo users currently online")
        else:
            print("Unexpected response:", response)

    def roomsList(self):
        message = "rooms-list-request"
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode()
        if response.startswith("chat-list"):
            chats = response[len("chat-list\n"):].split(',')
            print("Available Rooms:")
            for chat in chats:
                chat_name = chat.strip()
                if chat_name:  # Ensure the room name is not an empty string
                    print(f"{Fore.RED}{chat_name}{Fore.RESET}")
            print("\n")
            if len(chats) < 1:  # Check if only the header or no rooms exist
                print("\tNo rooms currently exist")
        else:
            print("Unexpected response:", response)

    def Join_Chat_Room(self, chat_room_name):
        message = "JOIN-CHAT " + chat_room_name + " "
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode()
        if response.startswith("join-chat-success"):
            participants_list = response.split()[1:]  # Extract the list of
participants
            if participants_list:
                print("Joined " + chat_room_name + "\n. Participants: " + ",
".join(participants_list))
                self.peerClient = PeerClient(self.peerServer, participants_list,
chat_room_name)
                self.peerClient.start()
                self.peerClient.join()
            else:
```

```
                # If participants_list is empty, initiate the PeerClient without the
list
                print("Joined " + chat_room_name + ". No participants in the room
yet.")
                self.peerClient = PeerClient(self.peerServer, None, chat_room_name)
                self.peerClient.start()
                self.peerClient.join()

            self.tcpClientSocket.send("chatroom-leave".encode())
            self.peerClient = None
        elif response == "No-Room found":
            print("No room found by the name: " + chat_room_name)

    # login function
    def login(self, username, password, peerServerPort):
        # a login message is composed and sent to registry
        # and integer is returned according to each response
        message = "LOGIN " + username + " " + password + " " + str(peerServerPort)
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode()
        if response == "login-success":
            print("Logged in successfully...")
            return 1
        elif response == "login-account-not-exist":
            print("Account does not exist...")
            return 0
        elif response == "login-online":
            print("Account is already online...")
            return 2
        elif response == "login-wrong-password":
            print("Wrong password...")
            return 3

    # logout function
    def logout(self, option):
        # a logout message is composed and sent to registry
        # timer is stopped
        if option == 1:
            message = "LOGOUT " + self.loginCredentials[0]
            self.timer.cancel()
        else:
            message = "LOGOUT"
        self.tcpClientSocket.send(message.encode())

    # function for searching an online user
```

```
    def searchUser(self, username):
        # a search message is composed and sent to registry
        # custom value is returned according to each response
        # to this search message
        message = "SEARCH " + username
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode().split()
        if response[0] == "search-success":
            print(f"{Fore.LIGHTGREEN_EX}" + username + f" is found
successfully...{Fore.RESET}")
            return response[1]
        elif response[0] == "search-user-not-online":
            print(username + " is not online...")
            return 0
        elif response[0] == "search-user-not-found":
            print(username + " is not found")
            return None

    # function for sending hello message
    # a timer thread is used to send hello messages to udp socket of registry
    def sendHelloMessage(self):
        message = "HELLO " + self.loginCredentials[0]
        self.udpClientSocket.sendto(message.encode(), (self.registryName,
self.registryUDPPort))
        self.timer = threading.Timer(1, self.sendHelloMessage)
        self.timer.start()


# peer is started
main = peerMain()
```

7.3. Database

```
# Includes database operations
from pymongo import MongoClient


class DB:

    # db initializations
    def __init__(self):
        self.client = MongoClient('mongodb://localhost:27017/')
        self.db = self.client['p2p-chat']
```

```
    # checks if an account with the username exists
    def is_account_exist(self, username):
        return self.db.accounts.count_documents({'username': username}) > 0

    # registers a user
    def register(self, username, password):
        account = {
            "username": username,
            "password": password
        }
        self.db.accounts.insert_one(account)

    # retrieves the password for a given username
    def get_password(self, username):
        return self.db.accounts.find_one({"username": username})["password"]

    # checks if an account with the username online
    def is_account_online(self, username):
        return self.db.online_peers.count_documents({"username": username}) > 0

    # logs in the user
    def user_login(self, username, ip, port):
        online_peer = {
            "username": username,
            "ip": ip,
            "port": port
        }
        # self.db.online_peers.insert_one(online_peer)

    # logs out the user
    def user_logout(self, username):
        self.db.online_peers.remove_one({"username": username})

    # retrieves the ip address and the port number of the username
    def get_peer_ip_port(self, username):
        res = self.db.online_peers.find_one({"username": username})
        return (res["ip"], res["port"])
```