Senior - 1

Computer Networks – CSE351
Project Report

SUBMITTED TO:

Dr. Ayman M. Bahaa-Eldin
Eng. Noha Wahdan

SUBMITTED BY:

Group **17** Members:

Ahmed Amin	20P2629
Ahmed Ibrahim Eldera	20P2701
Yassin Khaled Abd El Samie	20P2668
Amr Osama Mohammad Kamel	20P2493

Table of Contents

Table of Figures

Phase 1

1) Overall Project Scope

The goal of this project is to design and implement a robust Peer-to-Peer Multi-User Chatting Application. The application aims to emulate features similar to popular platforms like Clubhouse, but with a focus on text-based communication. The project is used to demonstrate proficiency in network application protocol design, client-server and peer-to-peer architecture, and the proper use of TCP and UDP protocols.

1.1 User Authentication:

Step 1: User Registration System

- Develop a registration system where users can input a desired username and password.
- Utilize the BCrypt algorithm for password hashing to enhance security.
- Implement server-side validation to ensure uniqueness of usernames.
- Store user credentials securely with BCrypt-hashed passwords.

Step 2: Unique Usernames and Secure Passwords

- Implement BCrypt to automatically generate a unique salt for each password.
- Leverage BCrypt's simplicity and available libraries for easy integration.

Step 3: Login System

- Develop a login system where users provide their registered username and password.
- Verify BCrypt-hashed credentials against stored information during the login process.
- Establish a session or token-based authentication mechanism for subsequent interactions.

Step 4: Non-Functional Requirements

- Database Integration: integrates a database to store user account information securely. Also, it utilizes the database to efficiently retrieve user details during authentication.

1.2 Basic Client-Server Setup:

Step 1: Server Setup

- Create a server using Python sockets to listen for incoming connections.
- Implement the necessary logic to accept and manage multiple client connections concurrently.

Step 2: Client Application

- Develop a client application that can initiate a connection to the server.
- Implement error handling for connection establishment and termination.

Step 3: Display Online Users

- Maintain a list of online users on the server.
- Implement functionality to broadcast the list of online users to all connected clients.

1.3 Chat Room Functionality:

Step 1: Create New Chat Rooms

- Allow users to create new chat rooms by specifying a unique name.
- Implement logic to manage the creation and registration of new chat rooms on the server.

Step 2: Join Existing Chat Rooms

- Enable users to join existing chat rooms by providing the room name.
- Implement the necessary server-side logic to manage users within specific chat rooms.

Step 3: Display Available Chat Rooms

- Implement a mechanism to display a list of available chat rooms to users.
- Broadcast the list of chat rooms to all connected clients.

1.4 Group Messaging in Chat Rooms:

Step 1: Send Messages in Chat Rooms

- Develop the ability for users to send messages to everyone in the chat room.
- Implement broadcasting of messages to all members of a specific chat room.

Step 2: Display Messages

- Implement the functionality to display messages from other users in the same chat room.
- Include timestamps and usernames for each message.

Step 3: Notifications for New Messages

- Implement a notification mechanism to alert users of new messages in the chat room.
- Ensure notifications are non-intrusive yet noticeable.

1.5 One-to-One Chat Functionality:

Step 1: Private Messaging

- Implement private messaging functionality between two users.
- Develop a system for initiating private conversations.

Step 2: Send and Receive Private Messages

- Allow users to send and receive private messages securely.
- Integrate private messages seamlessly with the overall chat application.

Step 3: Notifications for Private Messages

- Implement notifications to alert users of new private messages.
- Differentiate notifications for private messages from those for group messages.

1.6 Message Formatting and Features:

Step 1: Message Formatting

- Allow users to format their messages using simple formatting options (e.g., bold, italic).

Step 2: Share Hyperlinks

- Develop a feature allowing users to share hyperlinks within messages.
- Ensure the application recognizes and renders clickable hyperlinks.

Step 3: Hyperlink Functionality

- Implement functionality to open a web browser when users click on shared hyperlinks.

Step 4: Non-Functional Requirements

- Front-End Code (UI): Design and implement a responsive and user-friendly command-line interface. Also, utilize color-coding or other visual cues to enhance the user experience.

1.7 Error Handling and Resilience:

Step 1: Meaningful Error Messages

- Implement descriptive error messages for common issues (e.g., connection failures, authentication errors).
- Ensure error messages are helpful for users in troubleshooting.

Step 2: Automatic Reconnection

- Develop an automatic reconnection mechanism for users in case of network interruptions.
- Implement a timeout mechanism to handle inactive or disconnected users gracefully.

1.8 User Interface (UI) Enhancements:

Step 1: Clean and Intuitive UI

- Design a command-line interface that is visually appealing and easy to navigate.
- Consider using colors, symbols, or other visual cues for a better user experience.

Step 2: Message Type Identification

- Implement features that allow users to easily identify different types of messages (e.g., group messages, private messages).
- Use distinct visual elements or labels for each message type.

1.9 Documentation:

Step 1: Setup Guide

- Create a step-by-step guide explaining how to set up the application on both the server and client sides.
- Include any dependencies or configurations required for successful setup.

Step 2: User Guide

- Provide comprehensive documentation on how to use different features of the application.
- Include examples, tips, and troubleshooting information.

1.10 Testing:

Step 1: Unit Tests

- Develop and execute unit tests for each component to ensure individual functionality.
- Include tests for user authentication, message handling, and networking.

Step 2: Integration Tests

- Perform integration tests to validate the interaction between different components.
- Test scenarios involving multiple users, chat rooms, and messaging functionalities.

1.11 Scalability:

Step 1: Optimize Data Structures and Algorithms

- Evaluate and optimize data structures and algorithms to handle a larger number of users and messages efficiently.
- Consider the impact of scalability on both the server and client sides.

Step 2: Performance Testing

- Conduct performance testing to ensure the application performs well under increased load.
- Identify and address any bottlenecks or limitations related to scalability.

2) Authentication Mechanism

2.1 Hashing Algorithm Introduction:

A hashing algorithm is a mathematical function that converts input data into a fixed-length string of characters, commonly referred to as a hash code. It generates a unique output, known as the hash, for each unique input, making it convenient for verifying data integrity. In cybersecurity, password storage, and data integrity checks, hashing is commonly used. Strong hashing algorithms are resistant to both reverse engineering and collisions, which increases their worth in protecting sensitive data. BCrypt, SCrypt, and Argon2 are common examples, each having its own set of strengths and options for safe data protection. Using a strong hashing algorithm is essential to protect user's login data. We will compare the three popular password-hashing algorithms: BCrypt, SCrypt, and Argon2.

2.2 Why Choose BCrypt?

For our project, we have chosen to implement the **BCrypt** algorithm for password hashing. Here is why:

2.2.1 **Ease of Implementation**:

BCrypt is known for its simplicity and ease of integration into applications. It is a well-established algorithm with libraries available for various programming languages, making it straightforward to implement.

2.2.2 **Effectiveness Against Rainbow Tables**:

BCrypt effectively resists rainbow table attacks. By automatically generating a unique salt for each password and incorporating a cost factor, BCrypt ensures that attackers cannot use precomputed tables to crack passwords easily.

2.2.3 **Resistance to Brute Force**:

In addition to its resistance to rainbow tables, we plan to include a mechanism that limits the number of login trials within a specific period. This measure helps protect against brute force attacks, where attackers attempt multiple login combinations in rapid succession.

2.3 Comparing Algorithms:

All Three Resist Rainbow Tables:

It is important to note that all three algorithms (BCrypt, SCrypt, and Argon2) are designed to resist rainbow table attacks. They achieve this by using various techniques like salting, memory-hardness, and computational intensity.

Memory-Hardness and Why It Matters:

Both SCrypt and Argon2 are known for their memory-hardness, a property that makes it challenging for attackers to parallelize hash computations and crack passwords efficiently. This memory-hardness is crucial in resisting hardware-based attacks, such as GPU or ASIC acceleration.

Argon2's Advantages:

While SCrypt is also memory hard, Argon2 stands out due to several advantages:

1. **Configurability:** Argon2 offers more flexibility with parameters, including time cost, memory cost, and parallelism degree. This configurability allows fine-tuning to match specific security requirements.
2. **Standardization:** Argon2 has been standardized by the Internet Engineering Task Force (IETF), making it more widely recognized and accepted in the security community.
3. **Keeping Up with Hardware Evolution:** Argon2's parameter configurability makes it adaptable to evolving hardware capabilities. As hardware becomes more powerful, you can increase parameters to maintain strong security.

Conclusion:

While BCrypt is chosen for its simplicity and suitability for this project, it is essential to acknowledge the effectiveness and security of all three algorithms.

The choice of password-hashing algorithm depends on the project's specific needs and constraints. For projects with specific requirements, especially those that demand high security or need to adapt to changing hardware landscapes, Argon2 is an excellent choice. Its memory-hardness, configurability, and standardization make it a strong contender for the most security-conscious applications. SCrypt serves as a practical and secure option for many scenarios, while Argon2 offers greater configurability and adaptability for projects with evolving security demands. Combining these algorithms with login trial limitations enhances our resistance to both rainbow table and brute force attacks.

3) System Architecture

3.1 Overview of Project

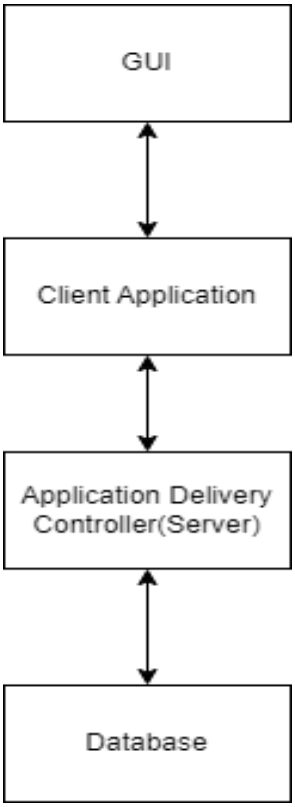


Figure 1: MVC diagram

Here is an architectural overview of the project showing all the different layers connected with each other and how they are communicating to achieve a specific action from the user. Starting from the GUI layer, taking an action from the user so it could inform the client application with the required command and the application will interact with the server to request something from the database. Thus, the server will interact with the database by requesting and the database will respond with a response that will travel all the way back to the client.

3.2 Peer-to-Peer Server Architecture

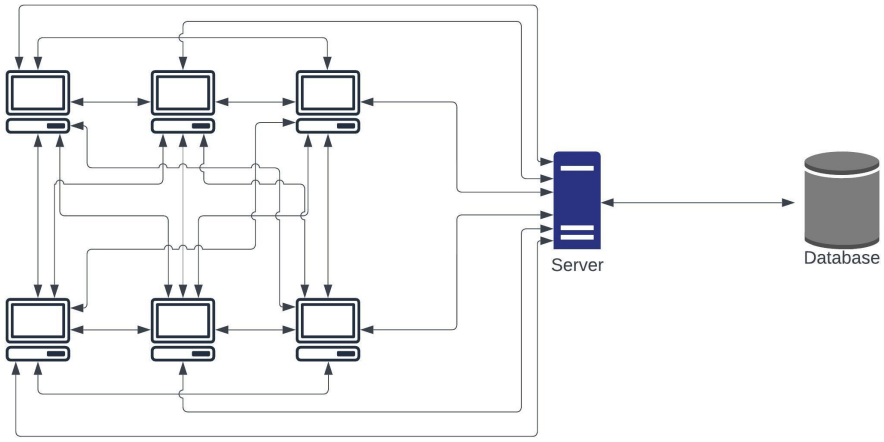


Figure 2: P2P architectural diagram

In peer-to-peer (p2p) architecture, all clients can communicate with each other and with servers. Each client can function as both a client where it's requesting files/data from other clients or server, and a server as other clients are requesting files/data from it unlike the client-server architecture where the server sends a copy to every single client as the clients can't communicate with each other directly. This interconnection between the clients eases the transmission of text-based messages and makes peer-to-peer communication more suitable for such a project.

3.3 Client-Server Architecture

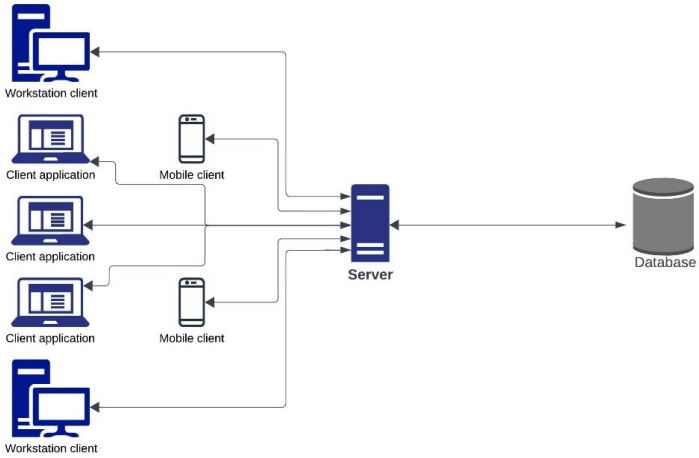


Figure 3: Client-server architectural diagram

Client server architecture is used for the process of creating new account and the continuous logins afterwards. Each client establishes a connection with the server to connect to the database as login credentials are saved there, thus a request of such info is needed to authenticate that the user is validated by comparing his input with the saved data.

3.4 Time Sequence Architecture of a possible scenario

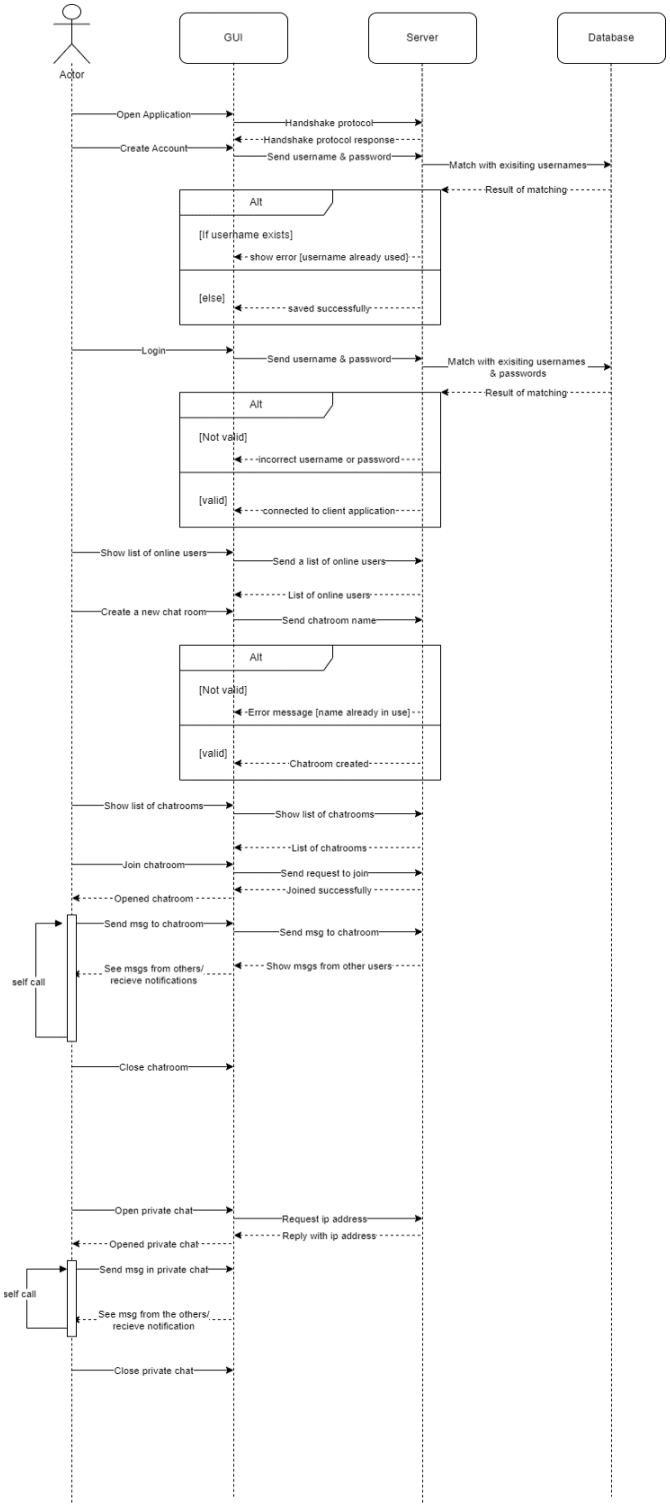


Figure 4: Time sequence diagram

4) Communication Protocol

4.1 Application Layer Protocol Design:

4.1.1 **Message Format**:

Define a standardized message format for both TCP and UDP interactions. This format may include fields such as `message_type`, `sender`, `recipient`, `content`, and `timestamp`.

Structure: We will use JSON and integrate it with SQL database.

Encoding: UTF-8.

4.1.2 **Semantic Meaning**:

Assign semantic meaning to different `message_type` values such as:

- `AUTH_REQUEST`: Initiate the authentication process.
- `CHAT_MESSAGE`: Transmit a message within a chat room.
- `JOIN_ROOM`: Request to join a chat room.
- `ERROR`: Indicate an error condition with an associated error code.

4.1.3 **Error Handling**:

Specify a consistent structure for error messages, including an `error_code` and `error_message`. This aids in meaningful error handling on both the client and server sides.

4.2 Usage of TCP and UDP:

4.2.1 **TCP (Transmission Control Protocol)**

- Utilize TCP for reliable and ordered communication during critical operations, such as user authentication.
- Implement an acknowledgment mechanism to confirm the receipt of critical messages, ensuring reliability.
- Take advantage of TCP's built-in flow control to prevent overwhelming the client or server with excessive data.

- ***Advantages:*** Guarantees delivery of data in the same order it was sent. Suitable for error- free transmission, like file transfers or detailed messages.
- ***Drawbacks:*** Slower due to acknowledgments and retransmission of lost packets.

4.2.2 UDP (User Datagram Protocol):

- Use UDP for real-time interactions, such as broadcasting messages within chat rooms, to achieve lower latency.
- Implement a loss-tolerant approach for UDP messages in non-critical scenarios like chat messages.
- Leverage UDP multicast groups for efficient broadcasting of messages to all users in a chat room.
- ***Advantages***: Faster due to no error-checking or recovery mechanisms. Ideal for real-time applications like voice or video chat where speed is more important than perfect accuracy.
- ***Drawbacks***: Does not guarantee delivery, order, or error-free communication.

4.3 Protocol Overview:

We've established a simple application layer protocol that governs the structure and meaning of messages exchanged between the client and server. This protocol is implemented over both TCP and UDP, each serving specific purposes. TCP ensures reliable and ordered communication for critical tasks, while UDP is employed for real-time interactions, providing lower-latency broadcasting within chat rooms. This approach offers a clear and organized foundation for the development of your Peer-to-Peer Multi-User Chatting Application, emphasizing consistency, security, and adaptability.

Phase 2

5) Basic Client Server Connection

Socket programming is a fundamental concept in computer networking that enables communication between computers over a network. It allows processes (programs or applications) to run on different devices to exchange data. Sockets provide a programming interface for network communication and are commonly used in both client-server and peer-to-peer communication scenarios.

Socket programming is a way of connecting between two nodes or sockets to communicate with each other. One socket (server socket) listens to a particular port at an IP address. While the other socket (client socket) reaches out to the first socket to establish a connection.

5.1 Code Overview Between Client & Server

The client-server connection is established when the client uses **socket.connect((host, port))** to connect to the server. In this case, the host is 'localhost', and the port is 12345. The server, on the other hand, uses **server.accept()** to accept incoming connections.

To run this system, you would typically start the server first and then run one or more clients. The server listens for incoming connections and spawns a new thread for each client. The client connects to the server, sends commands, and receives responses.

1. **Server Setup:**
    - Creates a server socket using **socket.socket()**.
    - Binds the server to a specific host and port using **bind()**.
    - Listens for incoming connections using **listen()**.
2. **Client Handling:**
    - For each incoming connection, it spawns a new thread (**handle_client**) to handle communication with that client.
3. **Client Setup:**
    - Establishes a connection to the server using socket.socket() and connect().
4. **User Interaction:**
    - Prompts the user to input commands ('register', 'login', 'logout', 'show online peers').
5. **Communication:**
    - Sends commands and user input to the server using send().
    - Receives and prints responses from the server using recv().
6. **Server Responses Handling:**
    - Processes server responses to handle login, registration, and other commands.

5.2 Part of server.py code:

```python
def start_server(port):
    try:
        # Set up the server socket
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server.bind(('', port))
        server.listen(5)
        print("Server listening on port", port)

        # Thread for the loop that prints active connections
        def print_active_connections():
            while not shutdown_flag:
                print(f"Active connections: {len(active_connections)} - {active_connections}")
                time.sleep(1)

        # Start the active connections printing loop in a separate thread
        print_thread = threading.Thread(target=print_active_connections)
        print_thread.start()

        # Main server loop for handling incoming client connections
        while not shutdown_flag:
            client_socket, client_address = server.accept()
            thread = threading.Thread(target=handle_client, args=(client_socket, client_address))
            thread.start()
            print(f"Active connections: {len(active_connections)} - {active_connections}")

    except Exception as e:
        print(f"Server error: {e}")
    finally:
        print("Server shutting down...")
        server.close()
```

Figure 5: Server part to start connection.

5.3 Peer.py Code

```python
# peer.py
import socket

1 usage
def connect_to_server(host, port):
    try:
        # Establish a connection to the server
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as client:
            client.connect((host, port))
            print("Connected to the server.")

            username = None  # Initialize username as None

            while True:
                # Prompt the user for input based on the current state
                if username:
                    command = input("Enter 'show online peers' or 'logout': ").lower().strip()
                else:
                    command = input("Enter 'register', 'login', or 'logout': ").lower().strip()

                # Process user commands
                if command == 'logout':
                    # Send logout command to the server
                    client.send(f"{command}".encode())
                    response = client.recv(4096)
                    print(f"Server response: {response.decode()}")
                    print("Logged out successfully.")
                    break
                elif command == 'show online peers':
                    # Send request to the server to show online peers
                    client.send(f"{command}".encode())
                    response = client.recv(4096)
                    print(f"Server response: {response.decode()}")
                elif command not in ['register', 'login']:
```

Figure 6: Client part1 of connection

```python
                    print("Invalid command. Please try again.")
                else:
                    # Process registration or login commands
                    if not username:
                        username = input("Enter username: ").strip()
                        password = input("Enter password: ").strip()

                    # Send registration or login command with username and password to the server
                    client.send(f"{command},{username},{password}".encode())
                    response = client.recv(4096)
                    print(f"Server response: {response.decode()}")

                    # Handle server responses for login and registration
                    if command == 'login':
                        if response.decode().endswith('successful'):
                            print(f"Login successful. You can now use 'show online peers' or 'logout'.")
                        elif response.decode().endswith('failed_username'):
                            print("Wrong Username")
                            username = None  # Reset the username if login failed
                        elif response.decode().endswith('failed_password'):
                            print("Wrong Password")
                            username = None  # Reset the username if login failed
                        elif response.decode().endswith('failed'):
                            print("Login failed. Please try again.")
                            username = None  # Reset the username if login failed
                    elif command == 'register':
                        if response.decode().endswith('successful'):
                            print(f"Registration successful. You can now use 'show online peers' or 'logout'.")
                        elif response.decode().endswith('failed'):
                            print("Already Registered Username")
                            username = None  # Reset the username if registration failed
    except Exception as e:
```

Figure 7: Client part2 of connection

```
        print(f"Connection error: {e}")

if __name__ == "__main__":
    server_host = 'localhost'  # Replace with server's IP address if needed
    server_port = 12345
    connect_to_server(server_host, server_port)
```

Figure 8: Client part3 of connection

6) User Authentication & Secured Login

The following functions work together to ensure secure registration and login processes, with passwords securely hashed and validated using BCrypt. If the provided username and password match, the login is considered successful. If not, appropriate error messages are returned to the client.

6.0 BCrypt Hashing

```
def hash_password(password):
    # Hash the password using bcrypt
    hashed_password = bcrypt.hashpw(password.encode(), bcrypt.gensalt())
    return hashed_password
```

Figure 9: Hashing Technique BCrypt

6.1 User Registration

When a user registers (**command == 'register'**), the server calls the **register_user** function. The user's password is hashed using the **hash_password** function, which uses BCrypt for secure password hashing. The username and hashed password are then inserted into the SQLite database.

```
def register_user(username, password):
    try:
        # Attempt to register a new user in the database
        hashed_password = hash_password(password)
        connection = sqlite3.connect('user_database.db')
        cursor = connection.cursor()
        cursor.execute( __sql: 'INSERT INTO users (username, password) VALUES (?, ?)',  __parameters: (username, hashed_password))
        connection.commit()
        return True
    except sqlite3.IntegrityError:
        print("Username already exists.")
        return False
    except sqlite3.Error as e:
        print("Database error:", e)
        return False
    finally:
        connection.close()
```

Figure 10: Register User Code

6.2 User Login Validation

When a user logs in (**command == 'login'**), the server calls the **validate_login** function. The stored hashed password for the provided username is retrieved from the database. BCrypt is used to check if the provided password matches the stored hash.

```python
def validate_login(username, password):
    try:
        # Validate user login credentials against the database
        connection = sqlite3.connect('user_database.db')
        cursor = connection.cursor()

        cursor.execute( __sql: 'SELECT password FROM users WHERE username = ?', __parameters: (username,))
        stored_password_hash = cursor.fetchone()
        connection.close()

        if stored_password_hash:
            stored_password_hash = stored_password_hash[0]
            # Check if the provided password matches the stored hash
            is_password_correct = bcrypt.checkpw(password.encode(), stored_password_hash)
            return is_password_correct
        else:
            return False
    except sqlite3.Error as e:
        print("Database error:", e)
        return False
```

Figure 11: Login Validation

6.3 Password Validation

The **validate_password** function is used to validate the password without logging the user in. Similar to **validate_login**, it checks if the provided password matches the stored hash.

```python
def validate_password(username, password):
    try:
        # Validate the password without logging the user in
        connection = sqlite3.connect('user_database.db')
        cursor = connection.cursor()

        cursor.execute( __sql: 'SELECT password FROM users WHERE username = ?', __parameters: (username,))
        stored_password_hash = cursor.fetchone()
        connection.close()

        if stored_password_hash:
            stored_password_hash = stored_password_hash[0]
            # Check if the provided password matches the stored hash
            is_password_correct = bcrypt.checkpw(password.encode(), stored_password_hash)
            return is_password_correct
        else:
            return False
    except sqlite3.Error as e:
        print("Database error:", e)
        return False
```

Figure 12: Password Validation

7) Multi-Threading

This multithreading setup enables the server to handle multiple clients simultaneously, improving the overall responsiveness of the system. Each connected client is handled in a separate thread, allowing multiple clients to be processed concurrently without blocking each other. The threads run independently, and the server can accept new connections while still handling existing ones.

'print_active_connections' Thread:

- There's a separate thread (**print_active_connections**) that runs in the background and prints information about active connections. This thread runs in parallel with the main server loop.
- New thread created using **threading.Thread(target=print_active_connections)** and calling **start()** on that thread.
- This thread continuously prints information about active connections in the background.

In the Main server Loop:

- The server listens for incoming connections using server.accept() in the main loop. For each incoming connection, a new thread is spawned using threading.Thread(target=handle_client, args= (client_socket, client_address)).
- The handle_client function is responsible for handling communication with the specific client. The new thread is started with thread.start().

Shutdown:

- The server listens for a shutdown signal (**KeyboardInterrupt**) and sets the **shutdown_flag** to **True** to exit the main server loop. Upon shutdown, the server closes the socket and stops all threads.

```
def start_server(port):
    try:
        # Set up the server socket
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server.bind(('', port))
        server.listen(5)
        print("Server listening on port", port)

        # Thread for the loop that prints active connections
        def print_active_connections():
            while not shutdown_flag:
                print(f"Active connections: {len(active_connections)} - {active_connections}")
                time.sleep(1)

        # Start the active connections printing loop in a separate thread
        print_thread = threading.Thread(target=print_active_connections)
        print_thread.start()

        # Main server loop for handling incoming client connections
        while not shutdown_flag:
            client_socket, client_address = server.accept()
            thread = threading.Thread(target=handle_client, args=(client_socket, client_address))
            thread.start()
            print(f"Active connections: {len(active_connections)} - {active_connections}")

    except Exception as e:
        print(f"Server error: {e}")
    finally:
        print("Server shutting down...")
        server.close()
```

Figure 13: Multithreading Handling

8) User Interface

8.1 Run Registry

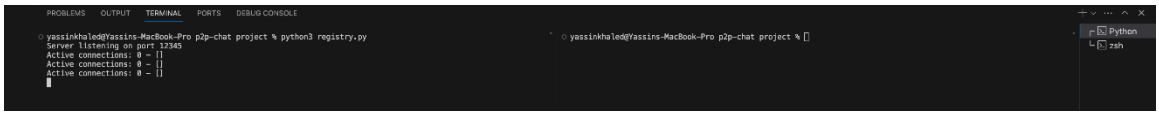


Figure 14: Running registry.py

8.2 User Registration & Logout Handling

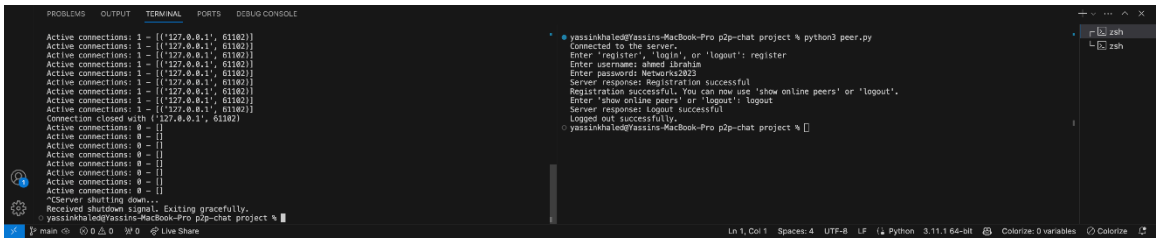


Figure 15: Successful registration

8.3 User Registration with already used Username

Figure 16: Wrong registration

8.4 Successful Login

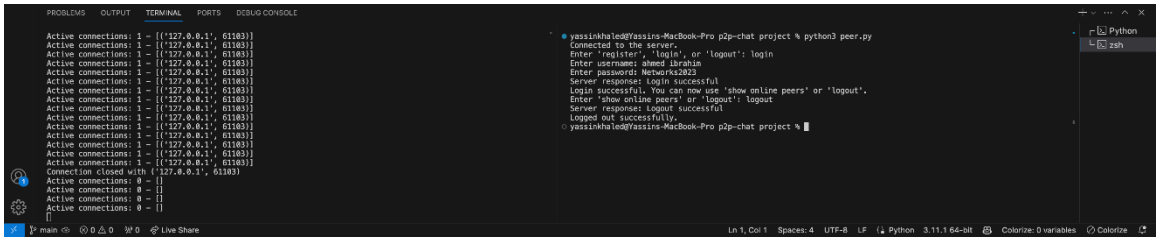


Figure 17: Successful login

8.5 Login with wrong Username

Figure 18: Unsuccessful login – user

8.6 Login with wrong Password

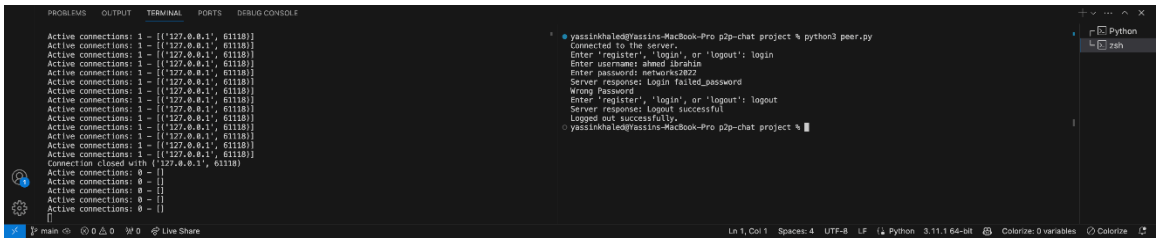


Figure 19: Unsuccessful login – password

8.7 Handling Multithreading & Showing Online Users

Figure 20: Running registry with multiple users.

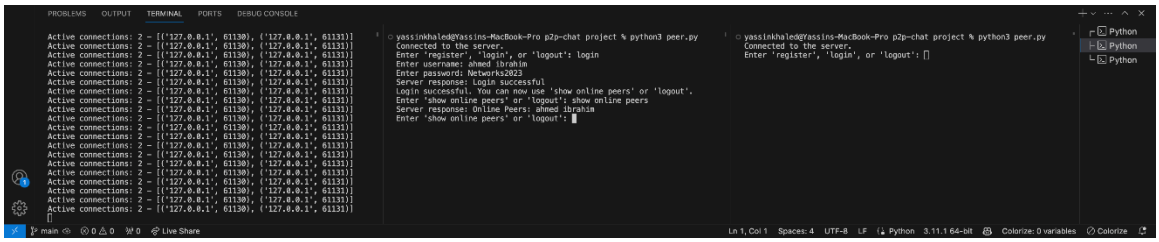


Figure 21: Showing online users.

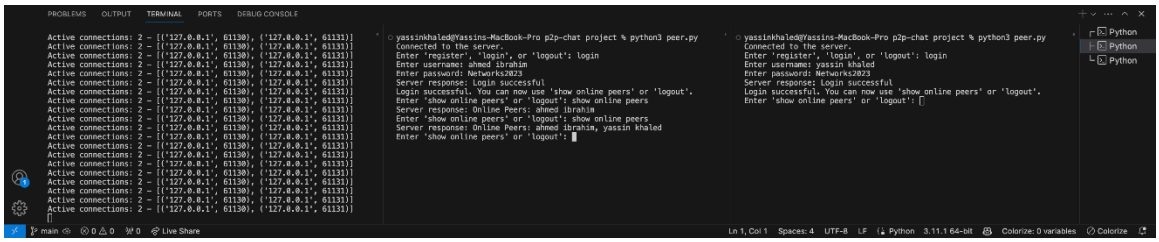


Figure 22: Showing online users part2.

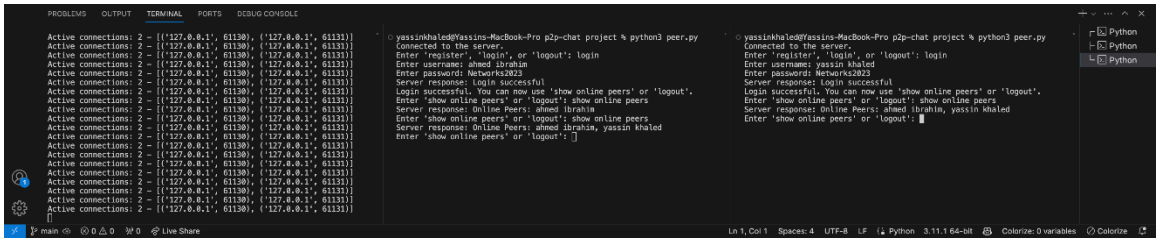


Figure 23: Showing online users part3.

Figure 24: Showing online users when a user leaves.

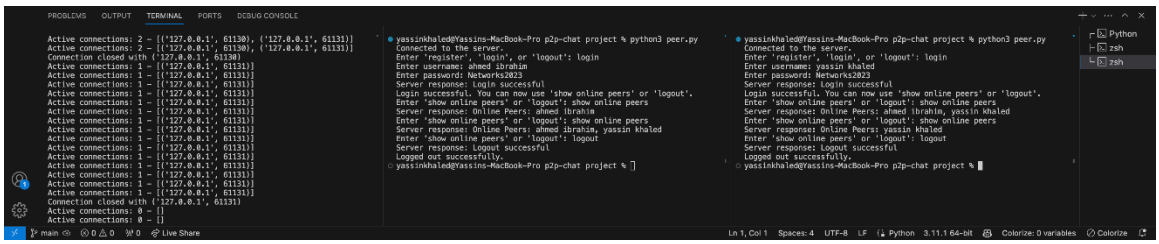


Figure 25: End connection to all users

Phase 3

9) Peer to Peer Connection

In the following part of the code, we are trying to make a peer-to-peer connection and try showing an output of a successful run and connection establishment.

9.1 P2P Connection

1. **get_user_ip(username, server_socket):**
    - Sends a request to the server to get the IP address of the specified username.
    - Returns the IP address obtained from the server.
2. **establish_peer_connection(target_username, target_ip, target_port):**
    - Establishes a peer-to-peer connection with the specified target.
    - Initiates a thread to handle messages from the peer.
    - Allows the user to send messages to the peer until 'exit' is entered.
3. **handle_peer_messages(peer_socket):**
    - Handles incoming messages from the connected peer.
    - Continuously receives and prints messages from the peer until the connection is closed.
4. **start_peer_server(host, port):**
    - Starts a peer server to accept incoming peer connections.
    - Listens for incoming connections and starts a thread to handle messages from each connected peer.
    - Runs indefinitely until an exception occurs or the server is closed.
5. **connect_to_server(host, port):**
    - Establishes a connection to the main server.
    - Handles user commands such as registration, login, showing online peers, peer connection, and logout.
    - Utilizes multithreading to allow concurrent user input and message handling.
6. **main:**
    - Configures the server's host and port.
    - Starts the peer server in a separate thread.
    - Connects to the main server using the **connect_to_server** function.

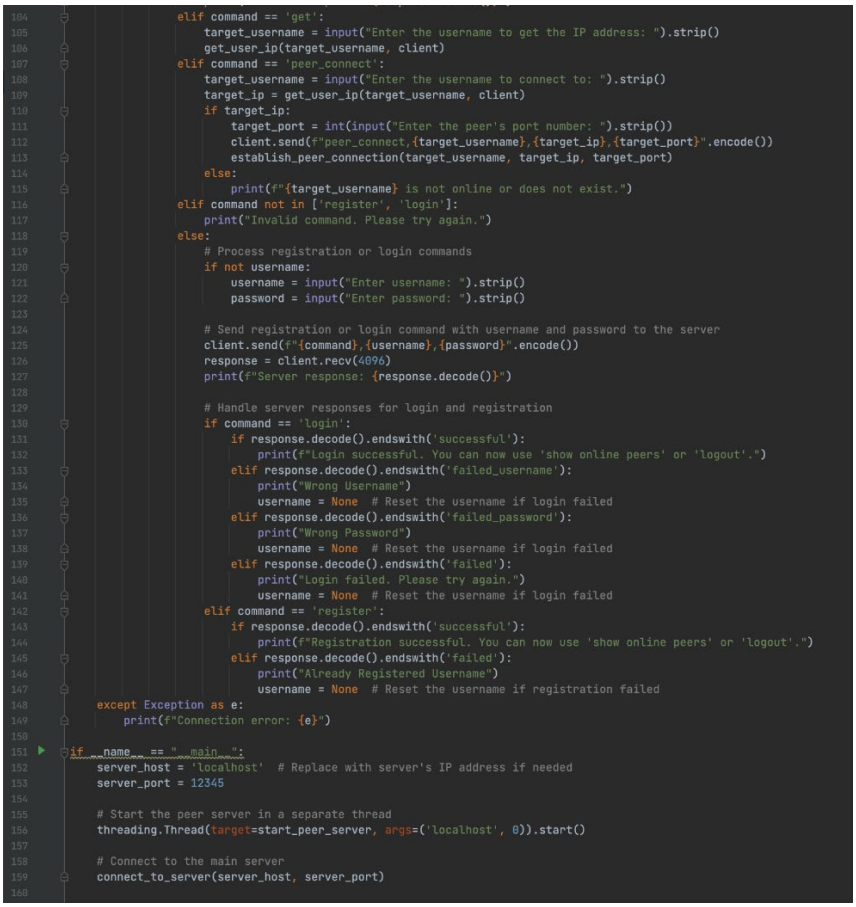
9.1.1 Connection Implementation

```python
import socket
import threading

# Global variables for peer server socket and username
peer_server_socket = None
username = None

def get_user_ip(username, server_socket):
    try:
        # Send a request to the server to get the IP address of the specified username
        server_socket.send(f"get,{username}".encode())
        response = server_socket.recv(4096)
        print(response.decode())
        return response.decode()  # Return the IP address obtained from the server
    except Exception as e:
        print(f"Error: {e}")
        return None

def establish_peer_connection(target_username, target_ip, target_port):
    try:
        # Establish a peer-to-peer connection
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as peer_socket:
            peer_socket.connect((target_ip, target_port))  # Fix: Use (target_ip, target_port)
            print("Peer-to-peer connection established.")

            # Start a thread to handle messages from the peer
            threading.Thread(target=handle_peer_messages, args=(peer_socket,)).start()

            while True:
                # Send messages to the peer
                message = input("Enter message to send (or 'exit' to close the connection): ")
                if message.lower() == 'exit':
                    break
                peer_socket.sendall(message.encode())
    except Exception as e:
        print(f"Peer connection error: {e}")
    finally:
        print("Peer-to-peer connection closed.")

def handle_peer_messages(peer_socket):
    try:
        while True:
            message = peer_socket.recv(1024).decode()
            if not message:
                break
            print(f"Received message from peer: {message}")
    except Exception as e:
        print(f"Peer message error: {e}")
    finally:
        peer_socket.close()
```

Figure 26: P2P connection part 1

```python
def start_peer_server(host, port):
    global peer_server_socket
    try:
        peer_server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        peer_server_socket.bind((host, port))
        peer_server_socket.listen(5)

        # Get the dynamically assigned port
        _, assigned_port = peer_server_socket.getsockname()
        print(f"Peer server listening on {host}:{assigned_port}")

        while True:
            # Accept incoming connections
            client_socket, client_address = peer_server_socket.accept()
            print(f"Accepted connection from {client_address}")

            # Start a thread to handle messages from the connected peer
            threading.Thread(target=handle_peer_messages, args=(client_socket,)).start()
    except Exception as e:
        print(f"Peer server error: {e}")
    finally:
        if peer_server_socket:
            peer_server_socket.close()
def connect_to_server(host, port):
    global username
    try:
        # Establish a connection to the server
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as client:
            client.connect((host, port))
            print("Connected to the server.")

            while True:
                # Prompt the user for input based on the current state
                if username:
                    command = input("Enter 'show online peers', 'peer_connect', 'get', or 'logout': ").lower().strip()
                else:
                    command = input("Enter 'register', 'login', or 'logout': ").lower().strip()

                # Process user commands
                if command == 'logout':
                    # Send logout command to the server
                    client.send(f"{command}".encode())
                    response = client.recv(4096)
                    print(f"Server response: {response.decode()}")
                    print("Logged out successfully.")
                    username = None

                elif command == 'show online peers':
                    # Send request to the server to show online peers
                    client.send(f"{command}".encode())
                    response = client.recv(4096)
                    print(f"Server response: {response.decode()}")
```

Figure 27: P2P connection part 2



```
            elif command == 'get':
                target_username = input("Enter the username to get the IP address: ").strip()
                get_user_ip(target_username, client)
            elif command == 'peer_connect':
                target_username = input("Enter the username to connect to: ").strip()
                target_ip = get_user_ip(target_username, client)
                if target_ip:
                    target_port = int(input("Enter the peer's port number: ").strip())
                    client.send(f"peer_connect,{target_username},{target_ip},{target_port}".encode())
                    establish_peer_connection(target_username, target_ip, target_port)
                else:
                    print(f"{target_username} is not online or does not exist.")
            elif command not in ['register', 'login']:
                print("Invalid command. Please try again.")
            else:
                # Process registration or login commands
                if not username:
                    username = input("Enter username: ").strip()
                    password = input("Enter password: ").strip()

                # Send registration or login command with username and password to the server
                client.send(f"{command},{username},{password}".encode())
                response = client.recv(4096)
                print(f"Server response: {response.decode()}")

                # Handle server responses for login and registration
                if command == 'login':
                    if response.decode().endswith('successful'):
                        print(f"Login successful. You can now use 'show online peers' or 'logout'.")
                    elif response.decode().endswith('failed_username'):
                        print("Wrong Username")
                        username = None  # Reset the username if login failed
                    elif response.decode().endswith('failed_password'):
                        print("Wrong Password")
                        username = None  # Reset the username if login failed
                    elif response.decode().endswith('failed'):
                        print("Login failed. Please try again.")
                        username = None  # Reset the username if login failed
                elif command == 'register':
                    if response.decode().endswith('successful'):
                        print(f"Registration successful. You can now use 'show online peers' or 'logout'.")
                    elif response.decode().endswith('failed'):
                        print("Already Registered Username")
                        username = None  # Reset the username if registration failed
    except Exception as e:
        print(f"Connection error: {e}")

if __name__ == "__main__":
    server_host = 'localhost'  # Replace with server's IP address if needed
    server_port = 12345

    # Start the peer server in a separate thread
    threading.Thread(target=start_peer_server, args=('localhost', 0)).start()

    # Connect to the main server
    connect_to_server(server_host, server_port)
```

Figure 28: P2P connection part 3

9.1.1 Connection Outputs

```
Peer server listening on localhost:55798
Connected to the server.
Enter 'register', 'login', or 'logout': login
Enter username: bal
Enter password: bal
Server response: Login successful
Login successful. You can now use 'show online peers' or 'l
ogout'.
```

```
Peer server listening on localhost:55802
Connected to the server.
Enter 'register', 'login', or 'logout': login
Enter username: bol
Enter password: bol
Server response: Login successful
Login successful. You can now use 'show online peers' or
'logout'.
```

Figure 29: Peers Connections

```
Enter 'show online peers', 'peer_connect', 'get', or 'log
out': peer_connect
Enter the username to connect to: bal
127.0.0.1
Enter the peer's port number: 55798
Peer-to-peer connection established.
Enter message to send (or 'exit' to close the connection)
: Received message from peer: hi
hi
```

Figure 30: Peer 1 connecting to Peer 2

```
Enter 'show online peers', 'peer_connect', 'get', or 'logou
t': peer_connect
Enter the username to connect to: bol
127.0.0.1
Enter the peer's port number: 55802
Peer-to-peer connection established.
Enter message to send (or 'exit' to close the connection):
Accepted connection from ('127.0.0.1', 55825)
hi
Enter message to send (or 'exit' to close the connection):
Received message from peer: hi
```

Figure 31: Peer 2 connecting to Peer 1

9.2 Chat-room Creation

Handling chat room creation and management in a peer-to-peer (P2P) network setting requires defining global variables for tracking connections, online peers, and chat rooms. There are functions for a user leaving a chat room (**left**), creating a new chat room (**create_chatroom**), joining a room (**join_room**), and broadcasting a message to all users in a chat room (**broadcast_chatRoom**). The **broadcast_chatRoom** function includes a mechanism to exit the chat based on a specific command.

1. **Global variables section:**
    - **chatRooms**: A dictionary to keep track of chat rooms.
    - **clients_sockets**: A list of client sockets.
    - **chatflag**: A flag used within the **broadcast_chatRoom** function.
2. **'left' function:**
    - Checks if the chat room exists and removes the client's socket from that room.
3. **'create_chatroom' function:**
    - Creates a new chat room by adding an entry to the **chatRooms** dictionary with the room name as the key and an empty list as the value.
4. **'join_room' function:**
    - Adds a user's socket to the list of sockets in the specified chat room.
5. **'broadcast_chatRoom' function:**
    - Uses a global flag **chatflag**.
    - Parses the message to check for an exit command. (shown in Figure 40)
    - If an exit command is detected, resets **chatflag** and sends an exit message to the client.
    - Otherwise, iterates over the sockets in the chat room and sends the message to all sockets except the one sending the message.

9.2.1 Chatroom implementation

```python
# Global variables
active_connections = []  # List to track active client addresses
online_peers = {}
online_peers_ports={}      # Dictionary to store online peers and their addresses
shutdown_flag = False    # Flag to signal server shutdown
connected_peers = {}  # Dictionary to store connected peers and their sockets
chatRooms={}
clients_sockets=[]
chatflag=0

def left(chatRoomName,client_socket):
    if chatRooms.__contains__(chatRoomName):
        chatRooms[chatRoomName].remove(client_socket)

def create_chatroom(name):
    chatRooms[name] =[]
    print(chatRooms)

def join_room(name,user_socket):
    chatRooms[name].append(user_socket)
    print(chatRooms)

def broadcast_chatRoom(name,client_socket,msg):
    global chatflag
    exit = msg.split(' : ')[1]
    if exit == "--exit--":
        chatflag = 0
        print(exit)
        client_socket.send(exit.encode())
    else:
        for i in chatRooms[name]:
            if i != client_socket:
                i.send(msg.encode())
```

Figure 32: Chatroom functions

```python
        elif command == 'show chat rooms':
            if not logged_in:
                response = f'Please login first'
            else:
                response = f'Available chat rooms: {", ".join(chatRooms.keys())}'
        else:
            if chatflag==1:

                broadcast_chatRoom(chatRoomName,client_socket,command)
                continue
            response = 'Invalid command'
```

Figure 33: Error-handling Chatroom

```python
        elif command == 'create chat room':
            if not logged_in:
                    response = f'Please login first'
            else:
                chatRoomName = parts[1]
                create_chatroom(chatRoomName)
            response = "created"
        elif command == 'join chat room':
            if not logged_in:
                    response = f'please login first'
            else:
                chatRoomName = parts[1]
                chatflag=1
                join_room(chatRoomName,client_socket)
                response = "Joined"
```

Figure 34: Creating & Joining Chatroom

9.2.2 Chatroom Creation & Joining Outputs

```
Connected to the server.
Enter 'register', 'login', or 'logout': login
Enter username: bol
Enter password: bol
Server response: Login successful
Login successful. You can now use 'show online peers' or 'logout'.
Enter 'show online peers','show chat rooms', 'peer_connect', 'create chat room','join chat room', or
 'logout': create chat room
enter chat room : t1
created
Enter 'show online peers','show chat rooms', 'peer_connect', 'create chat room','join chat room', or
 'logout': show chat rooms
Server response: Available chat rooms: t1
Enter 'show online peers','show chat rooms', 'peer_connect', 'create chat room','join chat room', or
 'logout': join chat room
enter chat room : t1
Enter message to send (or '--exit--' to close the connection):
 Joined
```

Figure 35: User1 Creating & Joining Chatroom

```
Connected to the server.
Enter 'register', 'login', or 'logout': login
Enter username: bal
Enter password: bal
Server response: Login successful
Login successful. You can now use 'show online peers' or 'logout'.
Enter 'show online peers','show chat rooms', 'peer_connect', 'create chat room','join chat room', or
 'logout': show chat rooms
Server response: Available chat rooms: t1
Enter 'show online peers','show chat rooms', 'peer_connect', 'create chat room','join chat room', or
 'logout': join chat room
enter chat room : t1
Enter message to send (or '--exit--' to close the connection):
 Joined
```

Figure 36: User2 Joining (1st way)

```
Connected to the server.
Enter 'register', 'login', or 'logout': login
Enter username: zxc
Enter password: zxc
Server response: Login successful
Login successful. You can now use 'show online peers' or 'logout'.
Enter 'show online peers','show chat rooms', 'peer_connect', 'create chat room','join chat room', or
 'logout': join chat room
enter chat room : t1
Enter message to send (or '--exit--' to close the connection):
 Joined
```

Figure 37: User3 Joining (2nd way)

9.3 Messaging System Outputs

```
Enter 'show online peers','show chat rooms', 'peer_connect', 'create chat room','join chat room', or 'logou
t': join chat room
enter chat room : t1
Enter message to send (or '--exit--' to close the connection):
 Joined
 bol : Hi
Hello
 bal : How are you guys doing?
I'm doing great
What about you?
 bal : I'm doing fine
 bol : I'm enjoying the weather here
 bal : Me too
It's cold here
 bol : Are we going out soon?
I am sick
 bal : No I'm studying
 bol : I have to leave bye
```

Figure 38: User3 Chatting

```
 Enter 'show online peers','show chat rooms', 'peer_connect', 'create chat room','join chat room', or 'logout
 ': join chat room
 enter chat room : t1
 Enter message to send (or '--exit--' to close the connection):
  Joined
  bol : Hi
  zxc : Hello
 How are you guys doing?
  zxc : I'm doing great
  zxc : What about you?
 I'm doing fine
  bol : I'm enjoying the weather here
 Me too
  zxc : It's cold here
  bol : Are we going out soon?
  zxc : I am sick
 No I'm studying
  bol : I have to leave bye
```

Figure 39: User2 Chatting

```
Enter 'show online peers','show chat rooms', 'peer_connect', 'create chat room','join chat room', or 'logout':
join chat room
enter chat room : t1
Enter message to send (or '--exit--' to close the connection):
 Joined
Hi
 zxc : Hello
 bal : How are you guys doing?
 zxc : I'm doing great
 zxc : What about you?
 bal : I'm doing fine
I'm enjoying the weather here
 bal : Me too
 zxc : It's cold here
Are we going out soon?
 zxc : I am sick
 bal : No I'm studying
I have to leave bye
--exit--
Enter 'show online peers','show chat rooms', 'peer_connect', 'create chat room','join chat room', or 'logout':
```

Figure 40: User1 Chatting & Leaving

Phase 4

10) One-to-One Functionality:

In this part we have handled one-to-one functionality between users. A function **start_peer_server** is defined, initializing a server socket, and listening for incoming connections. When a client connects, the server receives the client's listening port and starts a new thread to handle messages from this client. If a special flag (**flag02**) is not set, the server tries to establish a connection back to the client. The **establish_peer_connection** function attempts to connect to a peer using its IP address and port. Once connected, it sends the server's listening port to the peer and starts a thread to handle incoming messages. The user can then send messages to the peer, with the option to exit the chat by sending a special exit command. The **handle_peer_messages** function handles incoming messages from a connected peer. If the message is an exit command, it sets a flag (**flag02**) to signal that the other peer has left, prompts the user to type 'exit', and closes the connection. Otherwise, it prints the received message and continues to listen for new messages.

Also, a developed command-line interface for simplicity and ease of use we have added color-coded messages and user-friendly commands.

10.1 One-to-One Code Implementation

```
def start_peer_server(host, port):
    global flag02
    global peer_server_socket,x,z,my_listening_port
    try:

        peer_server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        peer_server_socket.bind((host, port))
        peer_server_socket.listen(5)
        _, assigned_port = peer_server_socket.getsockname()
        print(f"Peer server listening on {host}:{assigned_port}")
        my_listening_port=assigned_port
        while True:
            if flag02 == 0:
                client_socket, client_address = peer_server_socket.accept()
                client_ip, _ = client_address
                print(f"Accepted connection from {client_address}")
                peer_listening_port = int(client_socket.recv(1024).decode())
                print(f"Peer is listening on port {peer_listening_port}")
                threading.Thread(target=handle_peer_messages, args=(client_socket,)).start()
            try:
                # Read the listening port from the peer
                # Start a thread to handle messages from the connected peer
                # Automatically establish a connection back to the peer
                if flag02==0:
                    print(f"Connecting back to the peer at {client_ip}:{peer_listening_port}")
                    print('\n press Enter to procceed')
                    x=client_ip
                    z=peer_listening_port
                    flag02 = 1
            except ValueError:
                print("Invalid listening port received from peer.")
            except Exception as e:
                print(f"Error handling incoming connection: {e}")
    except Exception as e:
        print(f"Peer server error: {e}")
    finally:
        if peer_server_socket:
            peer_server_socket.close()
```

Figure 41: Start_peer Connection.

```python
def establish_peer_connection(target_username, target_ip, target_port):
    global flag02,my_listening_port
    flag02 = 1
    try:

        # Establish a peer-to-peer connection
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as peer_socket:
            peer_socket.connect((target_ip, target_port))  # Fix: Use (target_ip, target_port)
            print("Peer-to-peer connection established.")
            _, my_listening_port = peer_server_socket.getsockname()
            peer_socket.sendall(str(my_listening_port).encode())
            # Start a thread to handle messages from the peer
            threading.Thread(target=handle_peer_messages, args=(peer_socket,)).start()
            print("Enter message to send (or '--exit--' to close the connection): ")
            while True:
                # Send messages to the peer
                message = input()
                if message.lower() == '--exit--':
                    peer_socket.sendall(message.encode())
                    flag02 = 0
                    return
                if flag02==0:
                    return
                else:
                    peer_socket.sendall(f"{username} : {message}".encode())
    except Exception as e:
        print(f"Peer connection error: {e}")
        return
    finally:
        print("Peer-to-peer connection closed.")
        flag02=0
        #peer_socket.close()
```

Figure 42: Establish_peer Connection.

```python
def handle_peer_messages(peer_socket):
    global flag02
    x=0
    try:
        while True:
            message = peer_socket.recv(1024).decode()
            if not message:
                break
            if message == '--exit--':
                flag02=0
                print("other peer left please type 'exit' to go back")
                peer_socket.send("exit".encode())
                x=1
                return
            if message == 'exit':
                x=1
                return
            print(f" {message}")
    except Exception as e:
        if flag02==1:
            print(f"Peer message error: {e}")
        return
    finally:
        x=1
        peer_socket.close()
```

Figure 43: Error-Handling between peers

10.2 One-to-One Functionality Output

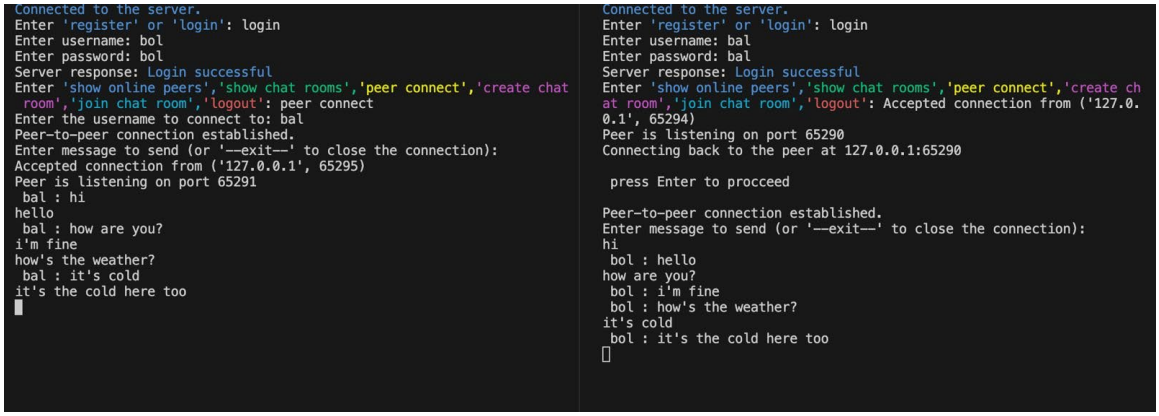


Figure 44: Private messaging between users.

Source-Code & Video Links

GitHub Link

Our source code can be found on the following GitHub link:

https://github.com/Yassin-ElSabagh/Group-17-Presentation-CSE351-UG2018--Computer-Networks.git

Video Link:

A video was recorded to show off the whole project's aspects, work, and progress. Here is a link for the video uploaded on google drive:

https://drive.google.com/drive/folders/1lfCYp3aLoc3kn7KnKnJpNZYmxVAuYBzG?usp=sharing