Ain Shams University Faculty of Engineering
CSE351: Computer Networks
Under the surveillance of Professor Ayman Bahaa.

Peer-to-Peer Multi-User Chatting Application

Group 7

Kareem Wael Hasan Ahmed	2001151
Malak ahmed yehia sherif	2001350
hussien ahmad abdelgelil mohammed khalifa	2000459
Mahmoud Talaat El-sayed Rezk	2001366

Abstract

In an era of digital connectivity, a novel peer-to-peer multi-user chat application emerges, built upon the foundation of Python and socket programming. It is a p2p chat application that uses centralized index approach. Through the creation and management of chat rooms, users can engage in both individual and group conversations. The application safeguards users with a secure authentication system, ensuring privacy and trust within the virtual environment. Beyond basic text-based messaging, the platform offers functionalities like text formatting and hyperlink sharing, enhancing the overall communication experience. By prioritizing user-friendliness through a simple command-line interface and visually distinct color-coded messages, the application caters to diverse user preferences. Moreover, robust error handling and automatic network reconnection ensure that user experience remains seamless and uninterrupted. This project promises a significant contribution to the realm of online communication, offering a secure, reliable, and engaging platform for users to connect and share their experiences.

Keywords: peer-to-peer, Python, socket programming, centralized index approach, authentication system , text-based messaging, network reconnection .

Table of Contents

Table Of Figures

Design Document

1. Security measurements

As a network application, implementing robust security measurement is crucial for the success of the software as a product, this is achieved using 3 measurements:

1. Hashing the password transferring over the network:

   Authentication is a crucial aspect of the peer-to-peer chatting application to ensure secure and authorized access. The system employs the **SHA-256 Cryptographic Hash Algorithm** authentication method:

   Client modules will communicate with the authentication utility module that is responsible for hashing the password to ensure secure transfer of sensitive information over the network.

   When the client registers, his password is hashed using SHA-256 algorithm and sent hashed over the network to the registry server, it then validates that the user have a unique username then create the user record in the database, when the user logs in, the password they type is yet again hashed using the same technique and sent hashed over the network, the registry server compares the hashed password sent using the login message with the hashed password stored in the database which should be identical in case of a correct password.

   The advantage of using a hashing technique over encryption is that hashing produces a result that cannot be decrypted back to the original message, making the transfer of hashed message secure even if the packet got sniffed.

   refer to the authentication sequence in Figure 6: User Registration (including authentication)

2. Storing the password hashed in the Database:

   This prevents DBA and developers from accessing users' sensitive information.

3. Using SSL/TLS to securely transfer messages over the network:

   SSL (Secure Sockets Layer) encryption, and its more modern and secure replacement, TLS (Transport Layer Security) encryption, protect data (not only passwords) sent over the internet or a computer network.

   It is implemented between the central server called Registry (which is responsible for the Database operations as will be discussed in the Documentation) and the peers to ensure secure data transfer over the network between the clients and the database at the other end. However, using SSL/TLS in the message exchange between the peers might not be the best idea since it imposes overhead and restricts system scalability.

2. System Components

The system architecture defines the structure and organization of the peer-to-peer chatting application. It includes components, their interactions, and the overall design of the system.

2.1 Components

2.1.1 Peer Client

- **Description:** Represents an individual user of the application.
- **Responsibilities:** Manages user interface, sending messages, and interactions within the application.
- **Components:**
    - User Interface
    - Communication Module
    - Peer Client Logic

2.1.2 Peer Server

- **Description:** Replaces the server in the traditional Client-Server architecture
- **Responsibilities:** Receiving messages and sending notifications.
- **Components:**
    - Client handler
    - Chat room hosting

2.1.3 Registry Server

- **Description:** Centralized server managing user accounts, online status, and user searches.
- **Responsibilities:** Handles user authentication, search operations, and online status management.
- **Components:**
    - Registry Logic
    - Database

2.1.4 Authentication utility

- **Description:** Provides the functionality of hashing passwords
- **Responsibilities:** Hashing Passwords.
- **Components:**
    - Hashing module

2.1.5 Database

- **Description:** Stores user data, and other relevant information.
- **Responsibilities:** Manages user accounts and chat-related data storage.
- **Components:**
    - Accounts Data
    - Chat rooms data

2.2 Interactions

- **Peer-to-Peer Communication:** Peers communicate directly for real-time chat.
- **Registry-Database Interaction:** Registry interacts with the database for user account management.
- **Peer-Registry Interaction:** Peers interact with the registry for authentication and user-related operations.
- **Peer-Chat Server Interaction:** Peers communicate with the chat server for chat room management and messaging.

2.3 Error Handling

Robust error handling mechanisms are implemented throughout the application to ensure graceful handling of unexpected scenarios. Each component incorporates error detection, reporting, and recovery mechanisms to enhance the application's reliability.

3. System Architecture and Design diagrams

This section presents visual representations of the system architecture through various diagrams, aiding in the understanding of the architecture and interactions.

4.1 Component Diagram

The UML component diagram provides a visual representation of the various software and hardware components in the system and their relations

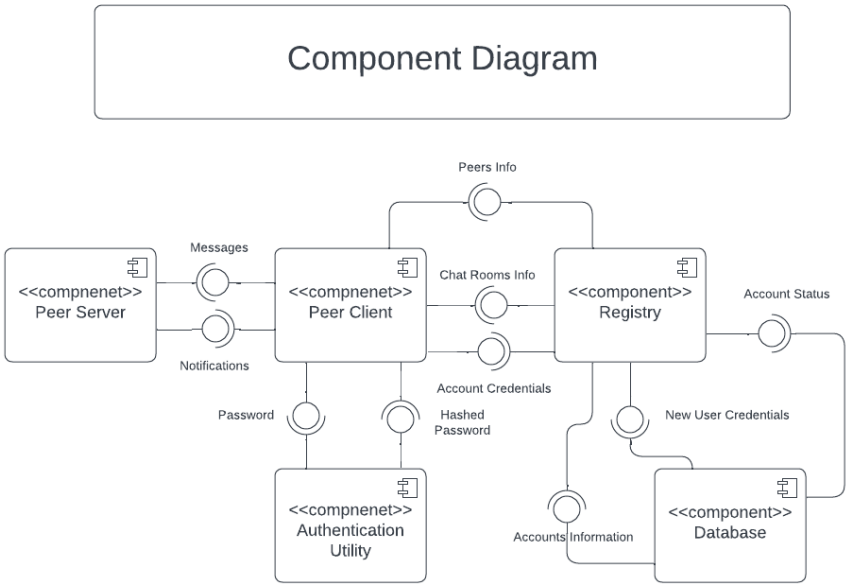

Figure 1: Component Diagram

4.2 Layered Architecture

The system is designed in layered architecture where every layer communicates only with the layer above/below it, this helps with the scalability of the application and makes it easier for modification and maintenance.

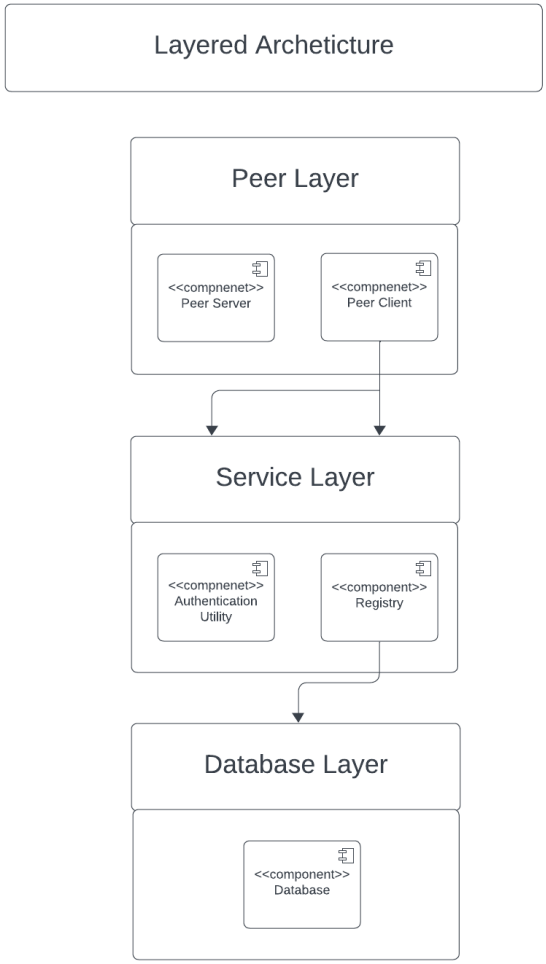

Figure 2: Layered architecture

The topmost layer represents the core components which the user interacts with as a peer in the application, the middle layer represents the service components which implements authentication at the client side and application organization at the server side, the bottom layer represents the interface to the database.

4.3 Sequence Diagrams

This sequence diagram details the sequence of interactions during the search operation, demonstrating how components collaborate to fulfill this specific functionality.

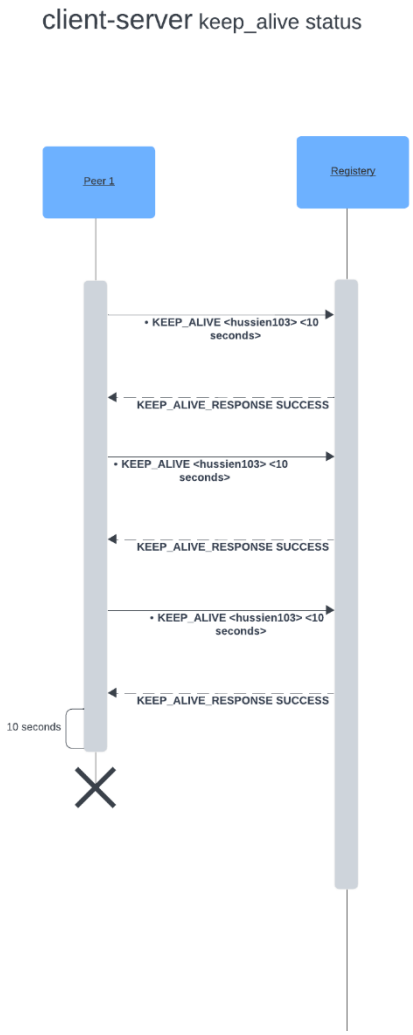

Figure 3: Client Server Keep Alive Status

With this mechanism the registry can detect when a user becomes offline for any reason, as the user have to send a message every x second to confirm that they are still there.

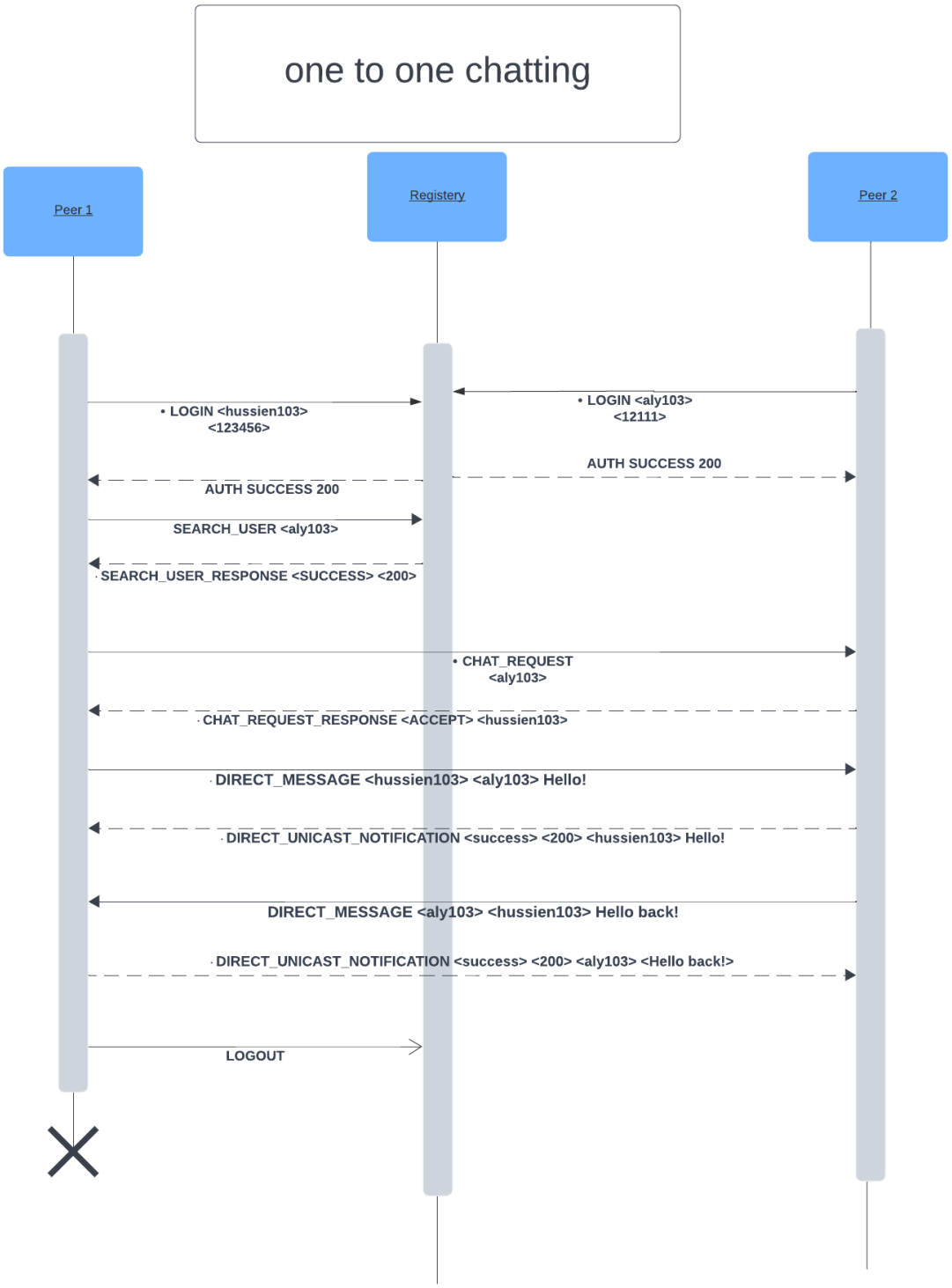

Figure 4:One to one Chatting

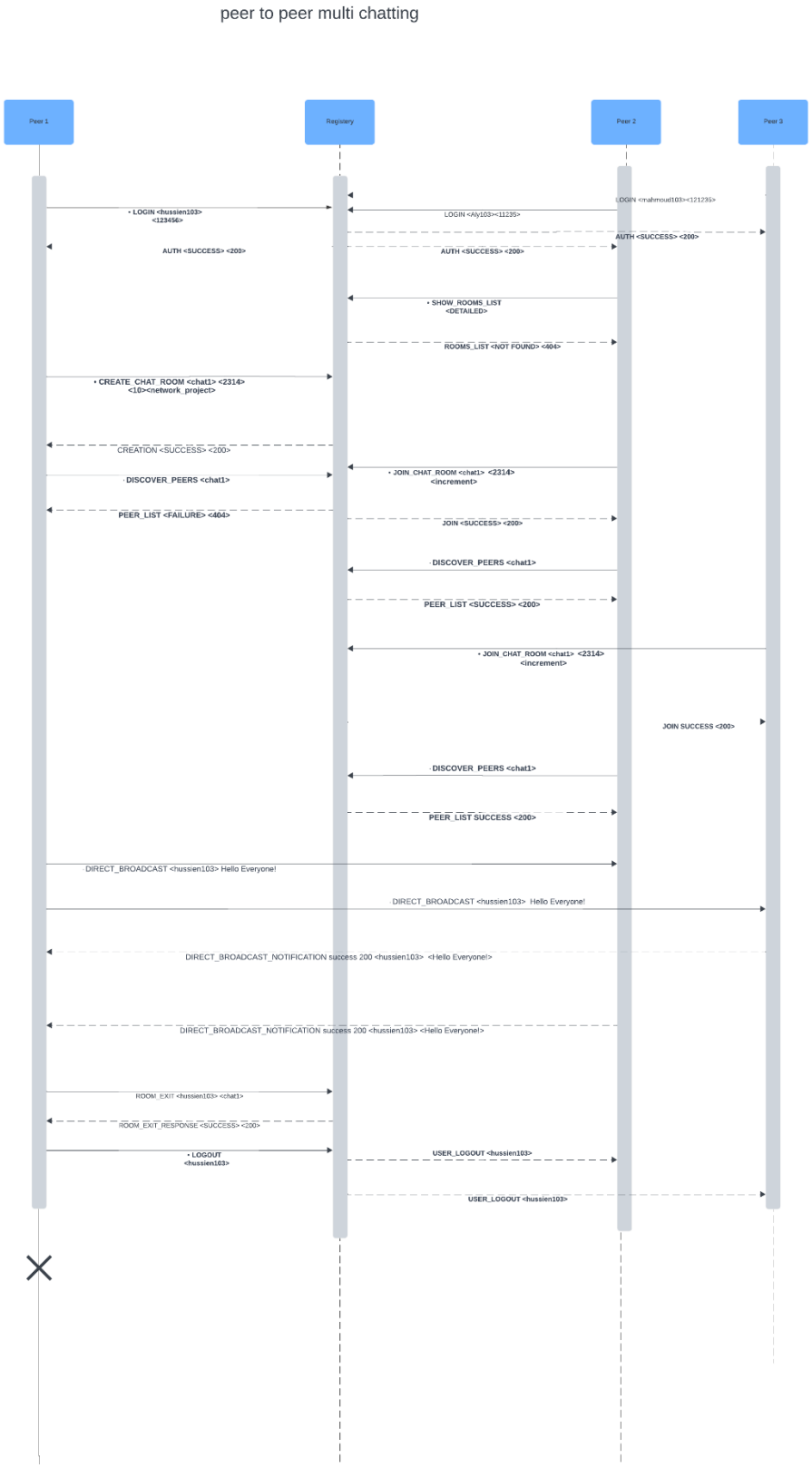

Figure 5:Multi Chatting

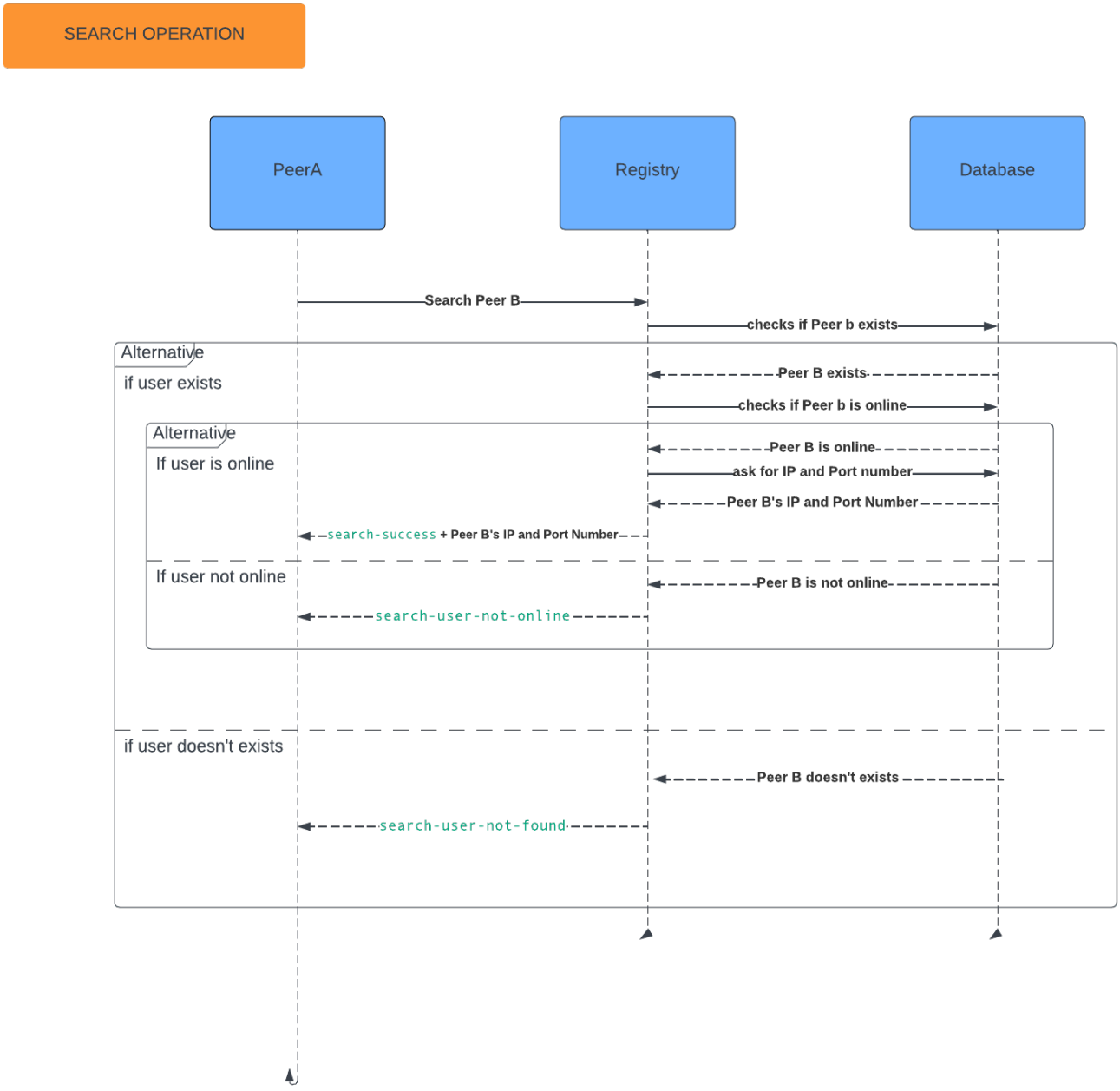

Figure 6: Search Operation

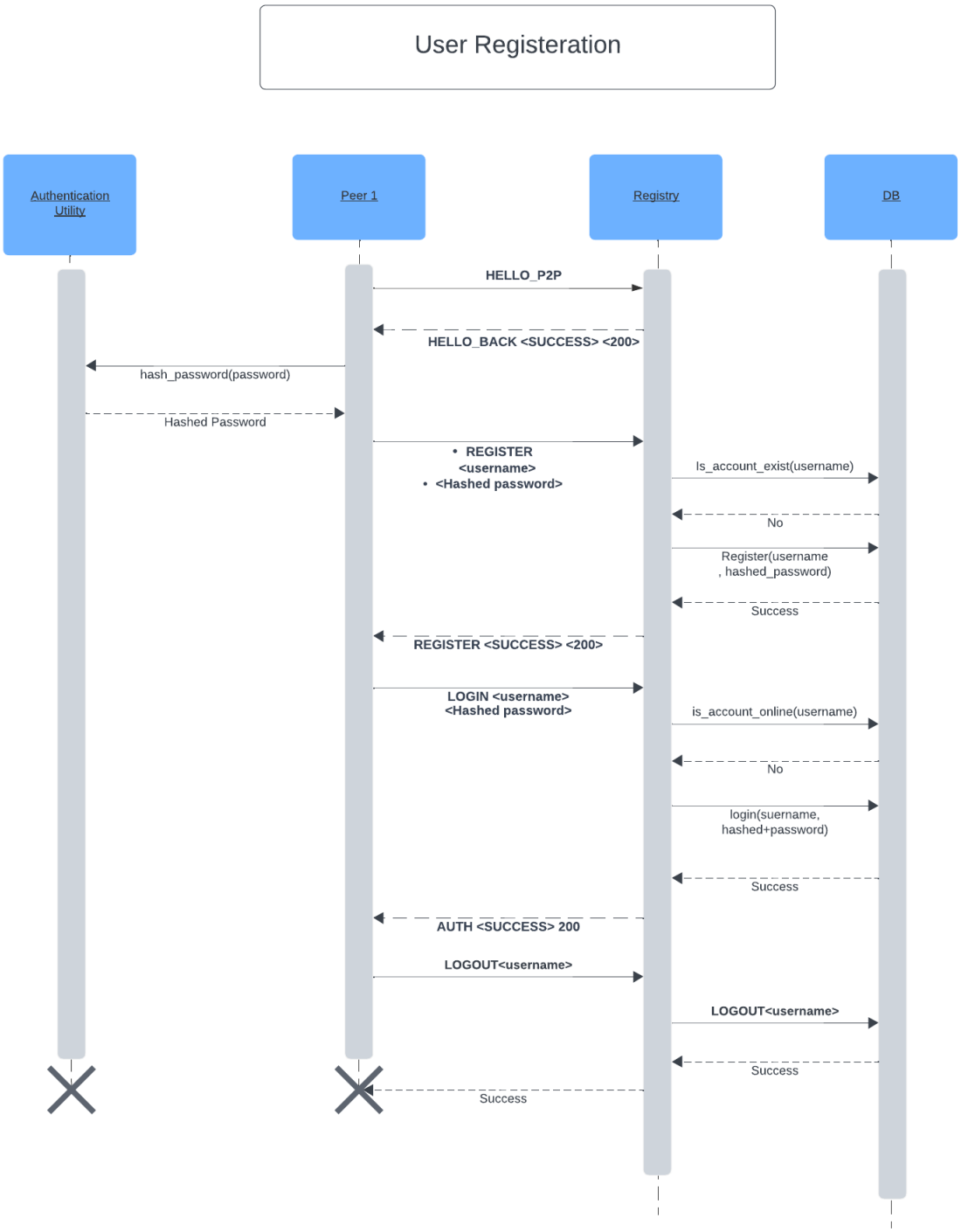

Figure 7: User Registration (including authentication)

4.4 Context Diagram

The context diagram describes the relation between the application and external sources/sinks, in the p2p chatting application the only external source/sink is the user using all functionality of the system.

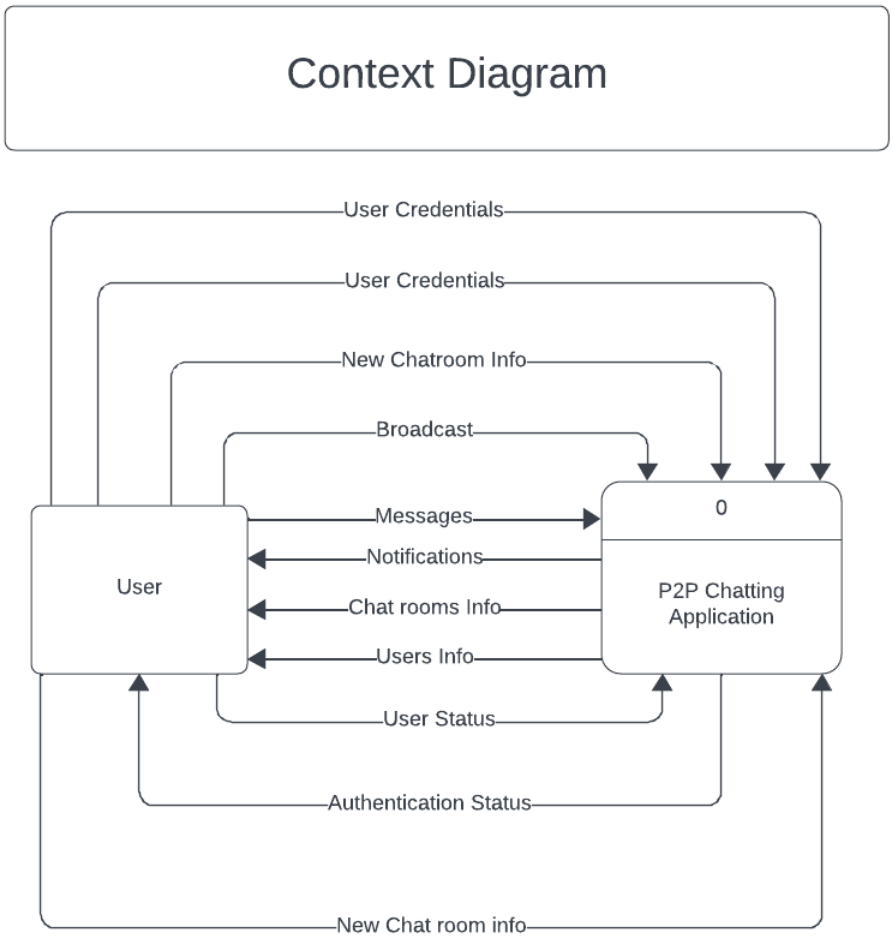

Figure 8: Context Diagram

4.5 Data Flow Diagram

The data flow diagram outlines the flow of information between different components of the system, providing a high-level view of how data moves through the application.

An overview of the main processes and data exchanged between them can be represented with Level 0 DFD:

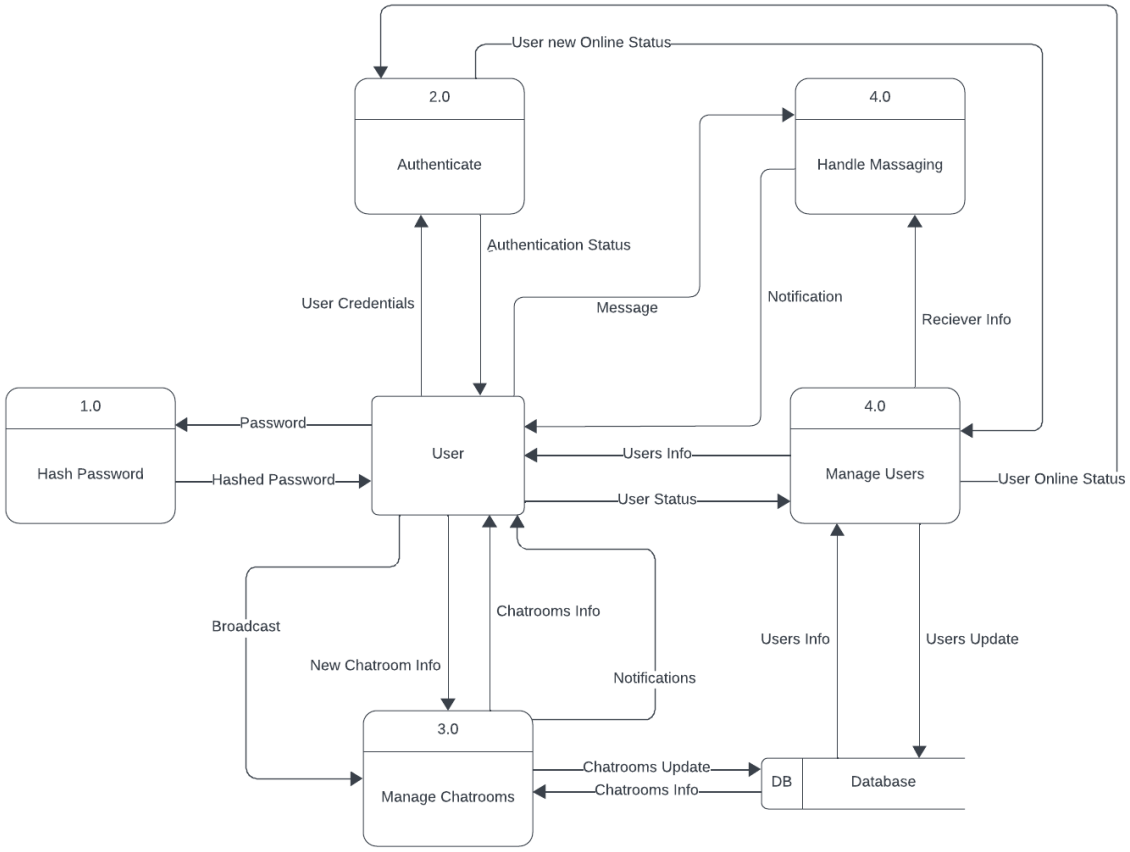

Figure 9: Level 0 DFD

4. Communication Protocols

I. Hello Message

- **Peer to Registry:**
  - TCP based Message: **HELLO_P2P**
  - Purpose: Notifies the registry about the peer's presence.
  - **Example request : HELLO_P2P**

[header lines]

command : HELLO_P2P

[data payload]

(empty)

- **Registry to Peer:**
  - TCP based Response: **HELLO_BACK <SUCCESS> <200> / HELLO <FAILURE> <404>**
  - Purpose: Confirms successful connection with the registry

.[header Lines]

Status line : HELLO_BACK

Status phrase : SUCCESS/FAILURE

Status code : 200 / 404

[data payload]

(empty)

II. User Registration

- **Client to Registry:**

- TCP based Message: **REGISTER <username> <password>**
- Purpose: Initiates the registration process with the registry.
- Example Request: **REGISTER <hussien> <123456>**

[Header Lines]

Command: REGISTER

Username: hussien

Password: 123456

[Payload]

(empty)

- **Registry to Client:**
  - TCP based Response: **REGISTER <SUCCESS> <200> or REGISTER <FAILURE> <404> or REGISTER <EXIST> <300>**
  - Purpose: Notifies the client about the success or failure of the registration.
  - Example Response:

[Header Lines]

Status Line: REGISTER

Status Phrase: SUCCESS/FAILURE/EXIST

Status Code: 200 / 404/300

[Payload]

(empty)

III. User Authentication:

- **Client to Registry:**
  - TCP based Message: **LOGIN <username> <hashed password>**
  - Purpose: Initiates the authentication process with the registry.
  - **Example request : login <hussien> <123456>**

[Header Lines]

Command: LOGIN

Username: hussien

Hashed Password: 123456

[Payload]

(empty)

- **Registry to Client:**
  - TCP based Response: **AUTH <SUCCESS> <200>** or **AUTH <FAILURE> <404> or AUTH <ONLINE> <300>**
  - Purpose: Notifies the client about the success or failure of the authentication.
  - **Example response:**

[Header Lines]

Status Line : AUTH

Status Phrase : SUCCESS/FAILURE/ONLINE

Status Code : 200 / 404 / 300

[Payload]

(empty)

IV. Creating Chat Room:

Client to Registry:

- TCP based Message: **CREATE_CHAT_ROOM <room_name> <room_password> <room-capacity><room_description>**
- Purpose: Requests the registry to create a new chat room with the specified name.
- **Example request :** CREATE_CHAT_ROOM <chat1> <123456> <10> <network_project>

[Header Lines]

command : CREATE_CHAT_ROOM

room_name : chat1

room_password:123456

room_capacity: 10

room_description : network_project

[Data Payload]

(empty)

- **Registry to Client:**
  - TCP based Response: CREATION <SUCCESS> or CREATION <FAILURE>
  - Purpose: Notifies the client about the success or failure of creating the chat room.
  - **Example response:** CREATION <SUCCESS> <200>

[Header Lines]

Status Line : CREATION

Status phrase : SUCCESS / FAILURE

Status code : 200/404

[Data payload]

(empty)

V. Joining Chat Room:

- **Client to Registry:**
  - TCP based Message: **JOIN_CHAT_ROOM <room_name> <room_password> <room_members_count>**
  - Purpose: Requests to join a specific chat room.
  - **Example request:** JOIN_CHAT_ROOM <chat1> <123455> <increment>

[Header Lines]

Command : JOIN_CHAT_ROOM

room_name : chat1

room_password : 123455

room_members_count : increment

[data payload]

(empty)

- **Registry to Client:**
  - TCP based Response: **JOIN <SUCCESS> <200>** or **JOIN <FAILURE> <404>**
  - Purpose: Notifies the client about the success or failure of joining the chat room.
  - Example Response: JOIN <FAILURE> <404>

[Header Lines]

Command : JOIN

Status phrase : FAILURE

Status code : 404

[data payload]

(empty)

VI. Leaving Chat Room:

- **Client to Registry:**
  - TCP based Message: **ROOM_EXIT <user_name><room_name>**
  - Purpose: Requests to leave a specific chat room.
  - Example Request: **ROOM_EXIT <hussien103> <chat1>**

[Header Lines]

Command : ROOM_EXIT

User_name : hussien103

Room_name : chat1

[data payload]

(empty)

- **Registry to Client:**
  - TCP based Response: **ROOM_EXIT_RESPONSE <SUCCESS> <200> or ROOM_EXIT_RESPONSE <FAILURE> <404>**
  - Purpose: Notifies the client about the success or failure of leaving the chat room.
  - Example Response: **ROOM_EXIT_RESPONSE <SUCCESS> <200>**

[Header Lines]

Status line: ROOM_EXIT_RESPONSE

Status Phrase: SUCCESS

Status Code: 200

[Payload]

(empty)

VII. Showing list of rooms :

- **Client to Registry:**
  - TCP based Message: **SHOW_ROOMS_LIST <type>**
  - Purpose: Show list of rooms with details or not.
  - **Example Request : SHOW_ROOMS_LIST <DETAILED>**

[Header Lines]

Command : SHOW_ROOMS_LIST

Type : DETAILED/PARTIAL

[data payload]

(empty)

- **Registry to Client:**
  - TCP based Response : **ROOMS_LIST <status_phrase> <status_code> + data**
  - Purpose: respond with list of rooms if found and not found response if not found.
  - Example response : ROOMS_LIST FOUND <200>

[Header Lines]

Status Line : ROOMS_LIST

Status Phrase : FOUND

Status Code: 200

[data payload]

<room1_name> <room1_description> <room1_members_count> <room1_capacity> ,

<room2_name> <room2_description> <room2_members_count> <room2_capacity> , …

VIII. Search User:

- **Client to Server:**
  - TCP based Message: **SEARCH_USER <username>**
  - Purpose : find a specified user
  - Example Request: **SEARCH_USER <aly103>**

[Header Lines]

Command: SEARCH_USER

Username: aly103

[data Payload]

(empty)

- **Server to Client:**
  - TCP based Response: **SEARCH_USER_RESPONSE <SUCCESS> <200> + data / SEARCH_USER_RESPONSE <NOT_ONLINE> <300> / SEARCH_USER_RESPONSE <NOT_FOUND ><404>**
  - Purpose : response with success or failure
  - Example Response: **SEARCH_USER_RESPONSE <SUCCESS> <200>**

[Header Lines]

Status Line : SEARCH_USER_RESPONSE

Status Phrase: SUCCESS/NOT_ONLINE/NOT_FOUND

Status Code : 200/300/404

[Data Payload]

<peer_username><peer_port><peer_IP>

IX. Start a Chat :

- **Peer to Peer(sender to reciever):**
  - TCP based Message: **CHAT_REQUEST < recipient_username>**
  - Purpose : requesting the reciever to start a chat
  - Example Request: **CHAT_REQUEST <aly103>**

[Header Lines]

Command: CHAT_REQUEST

recipient_Username: ahmed

[Payload]

(empty)

Peer to peer(recipient to sender):

- TCP based response: **CHAT_REQUEST_RESPONSE <ACCEPT> <200> < sender_username > / CHAT_REQUEST_RESPONSE <REJECT> <404>**
- Purpose : accepting or rejecting the chat request
- Example Request: **CHAT_REQUEST_RESPONSE <ACCEPT> <hussien103>**

[Header Lines]

Status line: CHAT_REQUEST_RESPONSE

Status phrase : ACCEPT / REJECT

Status code : 200 / 404

sender_username: hussien103

[data Payload]

(empty)

X. Online peer discovery

- **Peer to Registry:**
  - TCP based Message: **DISCOVER_PEERS <type>**
  - Purpose: Requests a list of online peers from the registry.
  - **Example request : DISCOVER_PEERS <DETAILED>**

[header lines]

Command : DISCOVER_PEERS

[data payload]

(empty)

- **Registry to Peer:**
  - TCP based Response: **PEER_LIST <SUCCESS>/<FAILURE> <200>/<404> + peer1, peer2 ...**
  - Purpose: Provides a list of online peers.

[header lines]

Status line : PEER_LIST

Status phrase : success

Status code : 404

[data payload]

<peer1_username> <peer1_roomname> <peer1_port><peer1_ip>,

<peer2_username> <peer2_roomname> <peer2_port><peer2_ip>, …

XI. Direct message in one to one chat

- **Peer to Peer (Direct Message):**
  - Direct TCP Message: **DIRECT_MESSAGE <sender_username> <recipient_username> + message**
  - Purpose: Allows users to send messages in one to one chat.
  - **Example Message: DIRECT_MESSAGE <hussien103> <aly103> Hello!**

[Header Lines]

Command: DIRECT_MESSAGE

sender_username: hussien103

recipient_username: aly103

[Data Payload]

Hello!

Peer to Sender (unicast notification) :

- TCP based Notification: DIRECT_UNICAST_NOTIFICATION <SUCCESS>/<FAILURE> <200>/<404> <sender_username> <message_content>
- Purpose: Notifies sender about receiving a unicast message or not.
- .**Example Notification: DIRECT_UNICAST_NOTIFICATION <SUCCESS> <200> <hussien103> <Hello!>**
- **[Header Lines]**

Status Line: DIRECT_BROADCAST_NOTIFICATION

Status phrase : SUCCESS/FAILURE

Status code : 200/404

Sender_username : hussien103

Message_content : Hello!

[Payload]

(empty)

XII. User Logout:

- **Client to Registry:**
  - TCP based Message: **LOGOUT**
  - Purpose: Informs the registry that the user is logging out.

[header links]

Command : LOGOUT

[data payload]

(empty)

- **Registry to peer (in the same room):**
  - TCP based Broadcast: **USER_LOGOUT <user_name>**
  - Purpose: Notifies other users in the same chat room about the user's logout.
  - **Example message : USER_LOUGOUT <hussien>**

[Header Links]

status line : USER_LOGOUT

user_name : hussien

[data payload]

(empty)

XIII. Keep-Alive Message:

- **Peer to Registry:**
  - UDP based Message: **KEEP_ALIVE <user_name> <timeout>**
  - Purpose: Periodically sent by peers to maintain their online status.

[header lines]

command : KEEP_ALIVE

user_name : hussien103

timeout : 10

[data payload]

(empty)

- **Registry to Peer:**
  - UDP based Response: **KEEP_ALIVE_RESPONSE <SUCCESS> /< FAILURE> <200>/<404>**
  - Purpose: Confirms the receipt of the keep-alive message.

[header lines]

Status line : KEEP_ALIVE_RESPONSE

Status phrase : SUCCESS / FAILURE

Status code : 200/404

[data payload]

(empty)

XIV. Timeout Notification:

- **Registry to Peer:**
  - UDP based Notification: **TIMEOUT <username>**
  - Purpose: Notifies the peer about a timeout (e.g., no keep-alive received).

[header lines]

notification line : TIMEOUT

user_name : hussien103

[data payload]

(empty)

XV. Room Peers Discovery Protocol:

Peer to Registry:

- TCP based Message: **DISCOVER_PEERS <current_room_name>**
- Purpose: Requests a list of online peers in a specific chat room.
- **Example Request : DISCOVER_PEERS <chat1>**

[Header Lines]

Command : DISCOVER_PEERS

Current_room_name : chat1

[data payload]

(empty)

- **Registry to Peer (Discovery Response):**
  - TCP based Response: **PEER_LIST <SUCCESS>/<FAILURE> <200>/<404> <room_name> + data**
  - Purpose: Provides a list of online peers in the specified chat room if it exists or has members.
  - **Example Response:**

[header lines]

Status line : PEER_LIST

Status phrase: SUCCESS/FAILURE

Status code : 200/404

[data payload]

<room_name> <peer1><peer2>

XVI. **Direct Peer Messaging:**

- **Peer to Peers:**
  - TCP based Message: DIRECT_BROADCAST <sender_username> + message
  - Purpose: Allows a peer to broadcast a message to all other peers in the same chat room.
  - **Example Request: DIRECT_BROADCAST <hussien103> Hello Everyone!**

[Header Lines]

Command: DIRECT_BROADCAST

Sender_username : hussien103

[data Payload]

Hello, everyone!

- **Peers to Sender (broadcast notification) :**
  - TCP based Notification: DIRECT_BROADCAST_NOTIFICATION <SUCCESS>/<FAILURE> <200>/<404><sender_username> <message_content>
  - Purpose: Notifies the sender about a broadcast message
  - .**Example Notification: DIRECT_BROADCAST_NOTIFICATION <SUCCESS> <200> <hussien103> <Hello Everyone!>**

[Header Lines]

Status Line: DIRECT_BROADCAST_NOTIFICATION

Status phrase : success/failure

Status code : 200/404

Sender_username : hussien103

Message_content : Hello Everyone!

[Payload]

(empty)

5. System Scalability

The application is designed with scalability in mind to accommodate a growing user base. The system architecture allows for the seamless addition of resources, ensuring optimal performance even as the number of users increases.

The layered system architecture discussed in the previous section facilitates system scalability as managing increasing load can be easily done by updating the bottlenecked layer without affecting the other layers or the interfaces between them.

Stress testing will be conducted after the implementation of the main features in the application to ensure the system scales up to reasonable peers chatting simultaneously.

The Peer to Peer architecture ensures that theoretically no limit on the number of users can chat at the same time. However, the bottleneck is that how many connections can the Registry central server handle simultaneously which will be evaluated at testing.

Ain Shams University Faculty of Engineering
CSE351: Computer Networks
Under the surveillance of Professor Ayman Bahaa.

Peer-to-Peer Multi-User Chatting Application

Group 7

Kareem Wael Hasan Ahmed	2001151
Malak ahmed yehia sherif	2001350
hussien ahmad abdelgelil mohammed khalifa	2000459
Mahmoud Talaat El-sayed Rezk	2001366

Phase 2

Github Repository: https://github.com/KareemWael1/P2P-ChatRoom

Table of Contents

Table of Figures

Abstract

The Phase 2 of the project involves the implementation of foundational elements for a peer-to-peer chat system. The primary objectives include creating basic client and server applications with a focus on establishing TCP communication for user authentication. The deliverables for this phase encompass functional server and client components that facilitate multiple client connections.

The server application is designed to handle concurrent connections from multiple clients, while the client application is responsible for connecting to the server. The communication between the client and server is established using TCP, providing a secure channel for user authentication.

The client application features a simple command-line interface, allowing users to interact with the system seamlessly. This phase lays the groundwork for more advanced features in subsequent stages of the project, emphasizing the fundamental client-server setup essential for a peer-to-peer chat application.

Outputs of the code

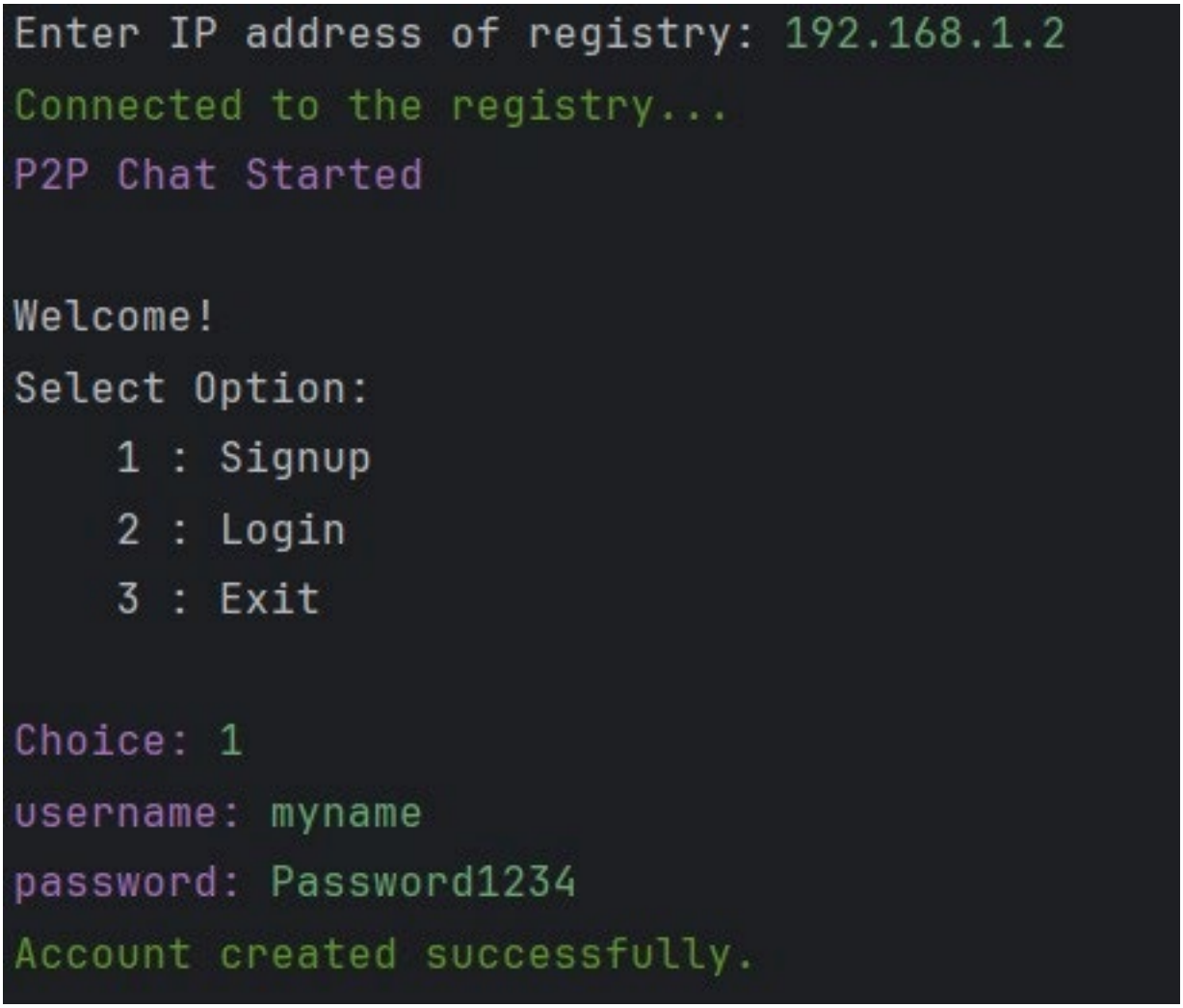

Figure 1: Sign up page

```
_id: ObjectId('65849adc6a2cfb989172213f')
username: "myname"
password: "a0f3285b07c26c0dcd2191447f391170d06035e8d57e31a048ba87074f3a9a15"
```

Figure 2: Account record in the DB (password is hashed)

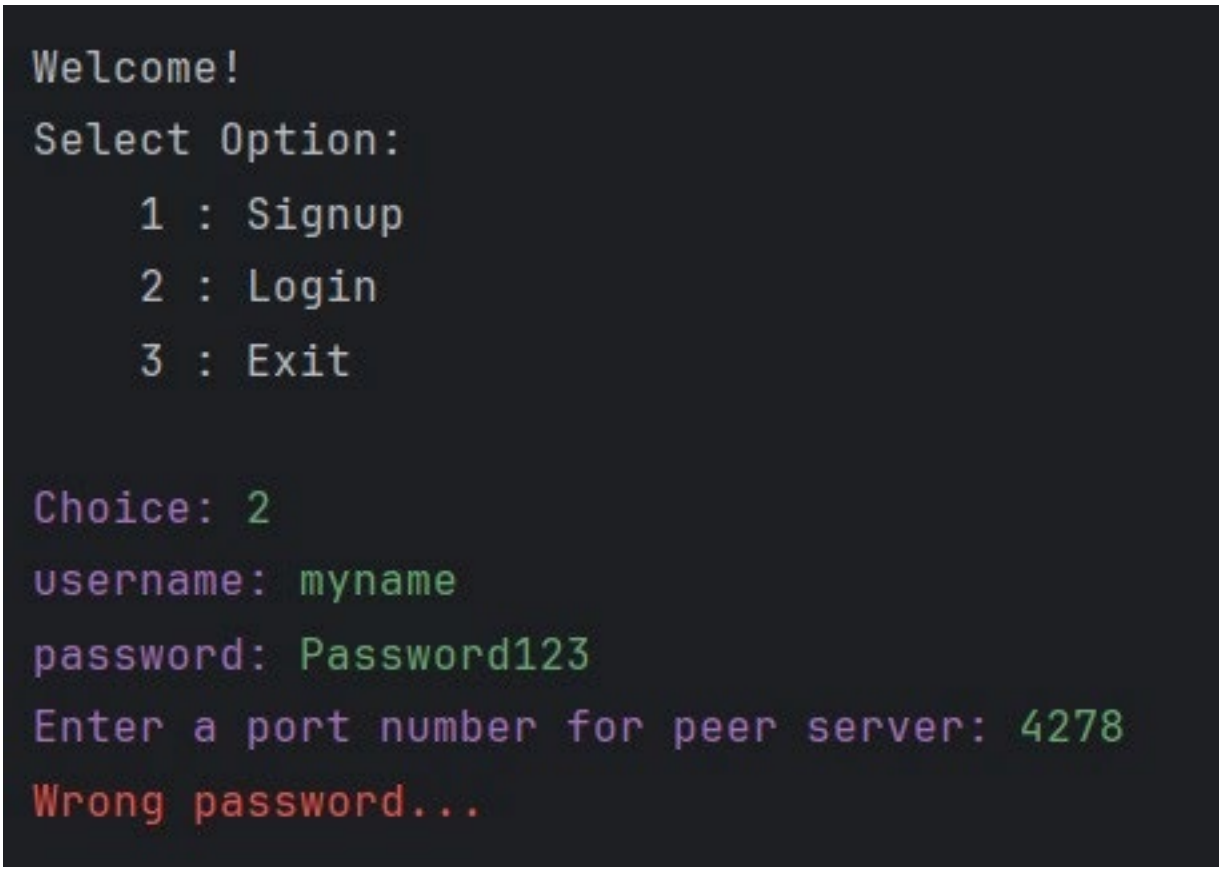

Figure 3: Login bad scenario

```
Welcome!
Select Option:
    1 : Signup
    2 : Login
    3 : Exit

Choice: 2
username: myname
password: Password1234
Enter a port number for peer server: 3798
Logged in successfully...

Main Menu
Select Option:
    1 : Find Online Users
    2 : Search User
    3 : Start a Chat
    4 : Logout

Choice:
```

Figure 4: Login Scenario

```
Main Menu
Select Option:
    1 : Find Online Users
    2 : Search User
    3 : Start a Chat
    4 : Logout

Choice: 1
Retrieve detailed list with users IP and Port numbers?(Choose y or n): y
#  Username       (IP:Port)
1  Kareem         (192.168.1.2:63142)
2  Ahmed          (192.168.1.2:63158)
```

Figure 5: Detailed Find online users

```
Main Menu
Select Option:
    1 : Find Online Users
    2 : Search User
    3 : Start a Chat
    4 : Logout

Choice: 1
Retrieve detailed list with users IP and Port numbers?(Choose y or n): N
Username
1  Kareem
2  Ahmed
```

Figure 6: Simple Find online user

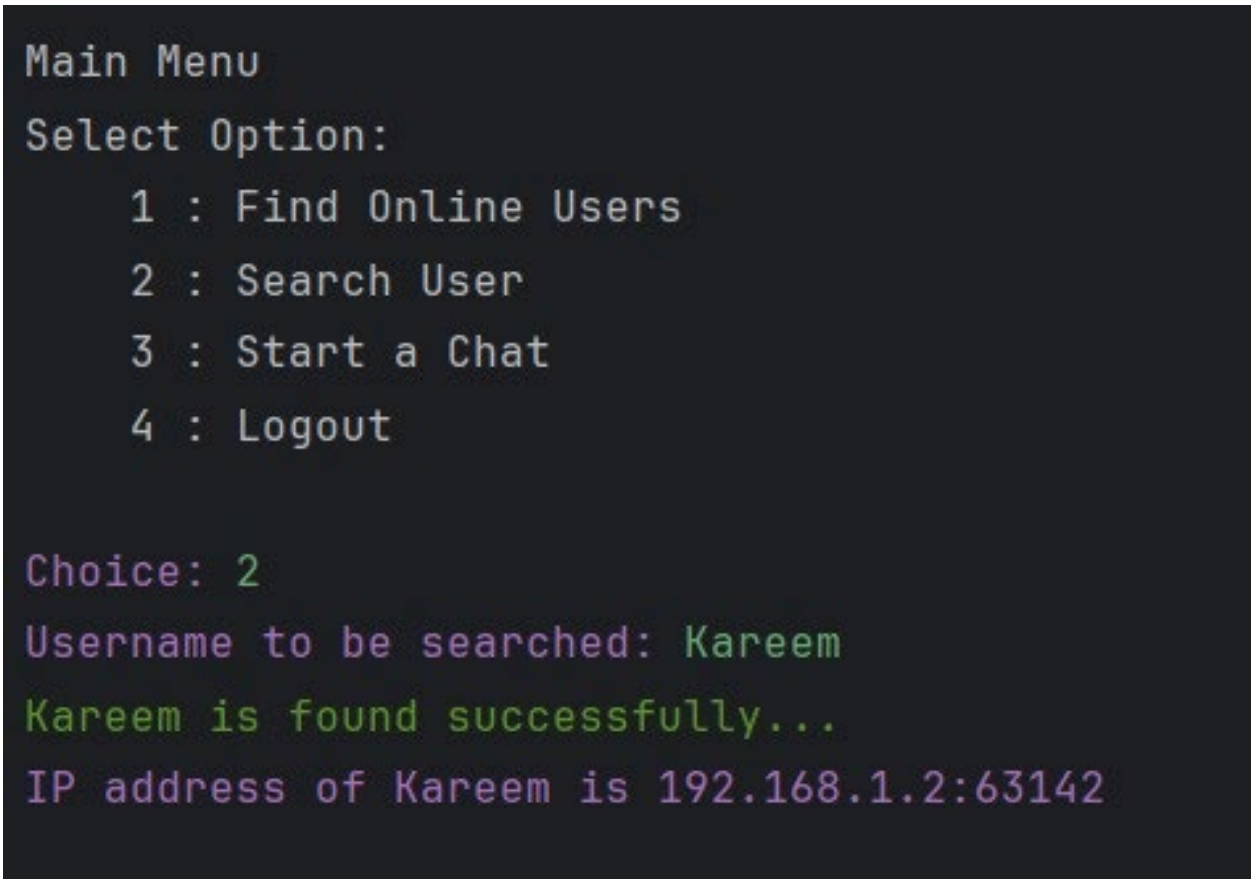

Figure 7: Search User (FOUND)

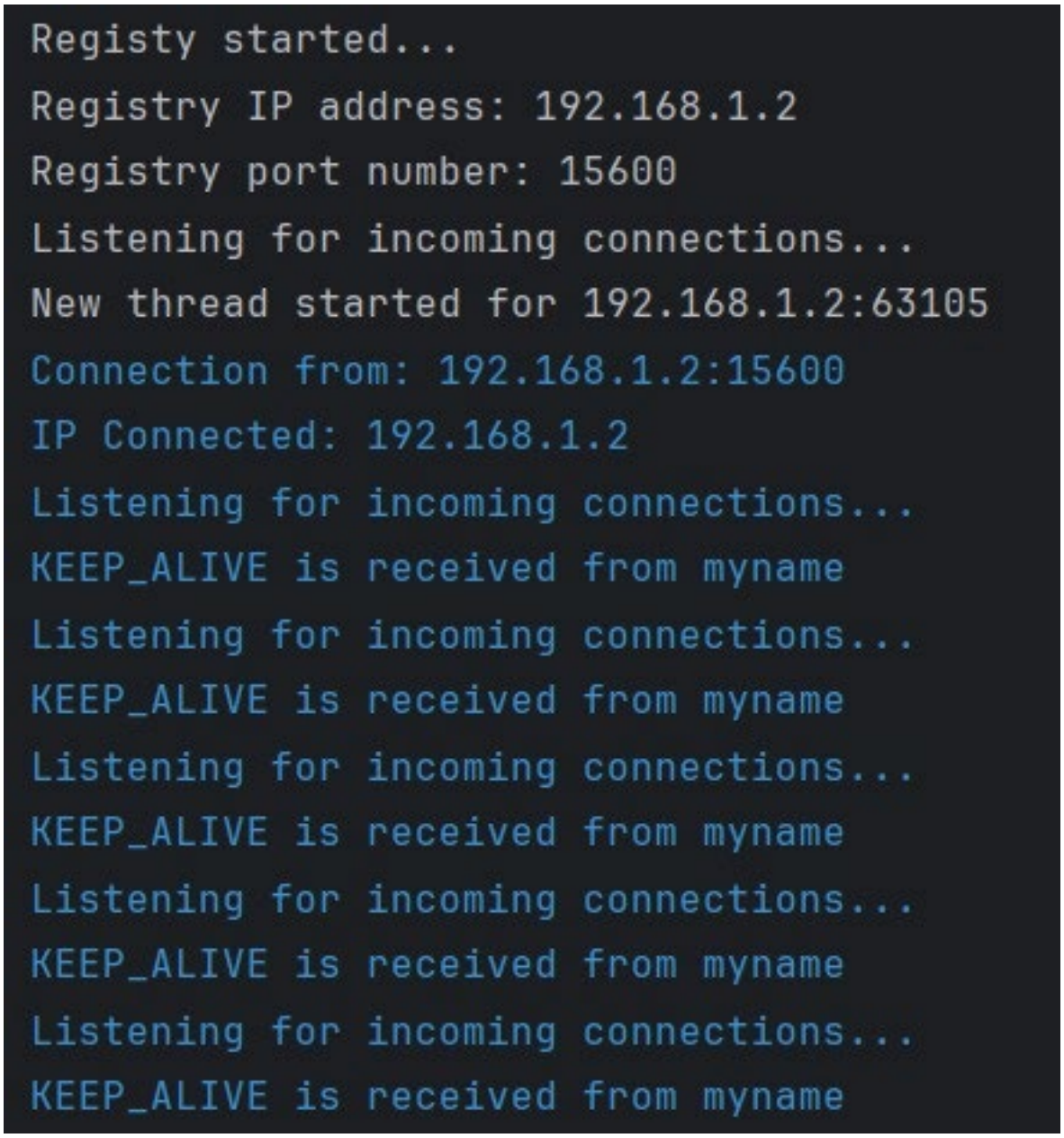

Figure 8: Output of registry

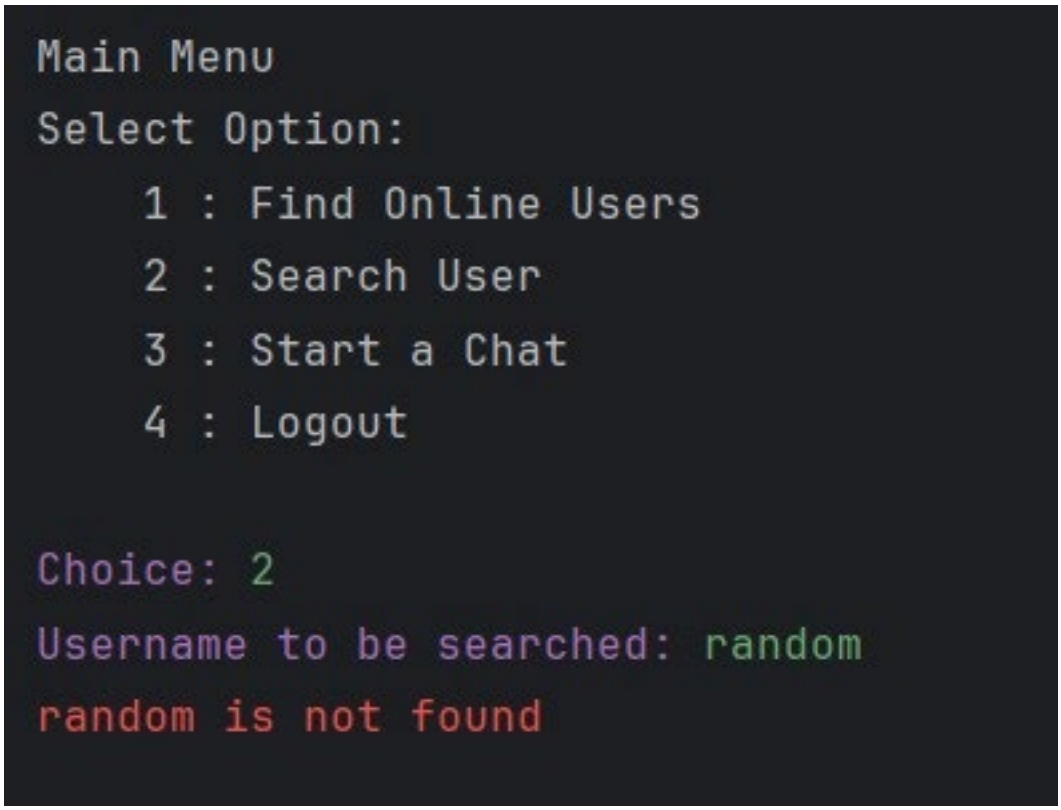

Figure 9: Search user (NOT EXIST)

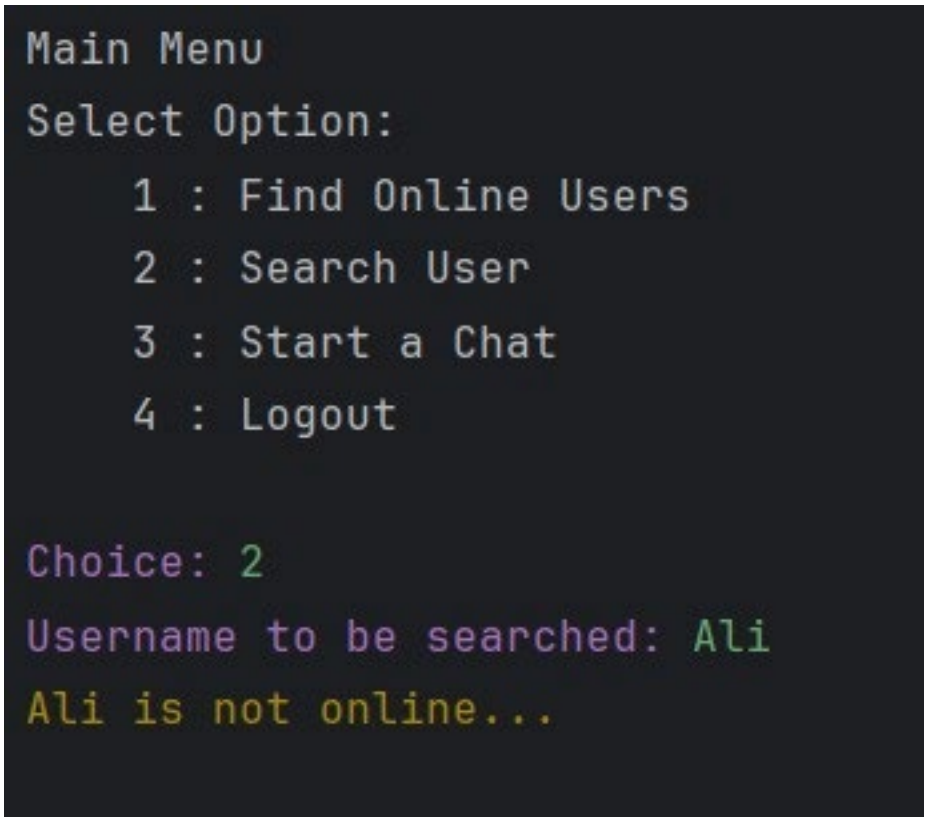

Figure 10: Search User (OFFLINE)

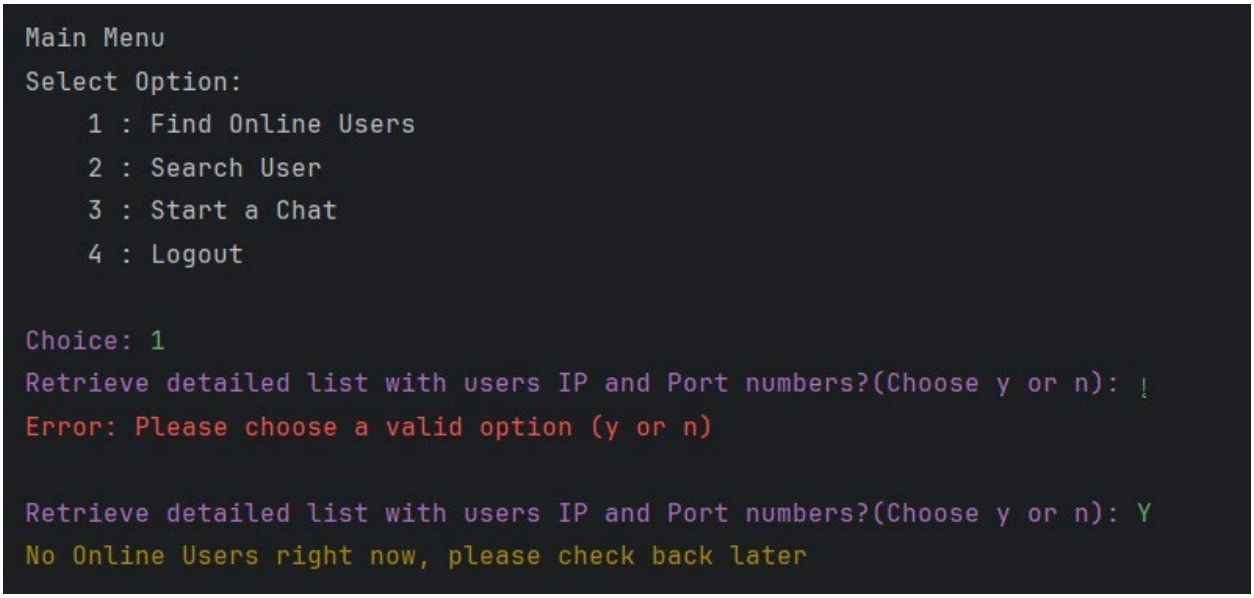

Figure 11: Find User (no one online +invalid option)

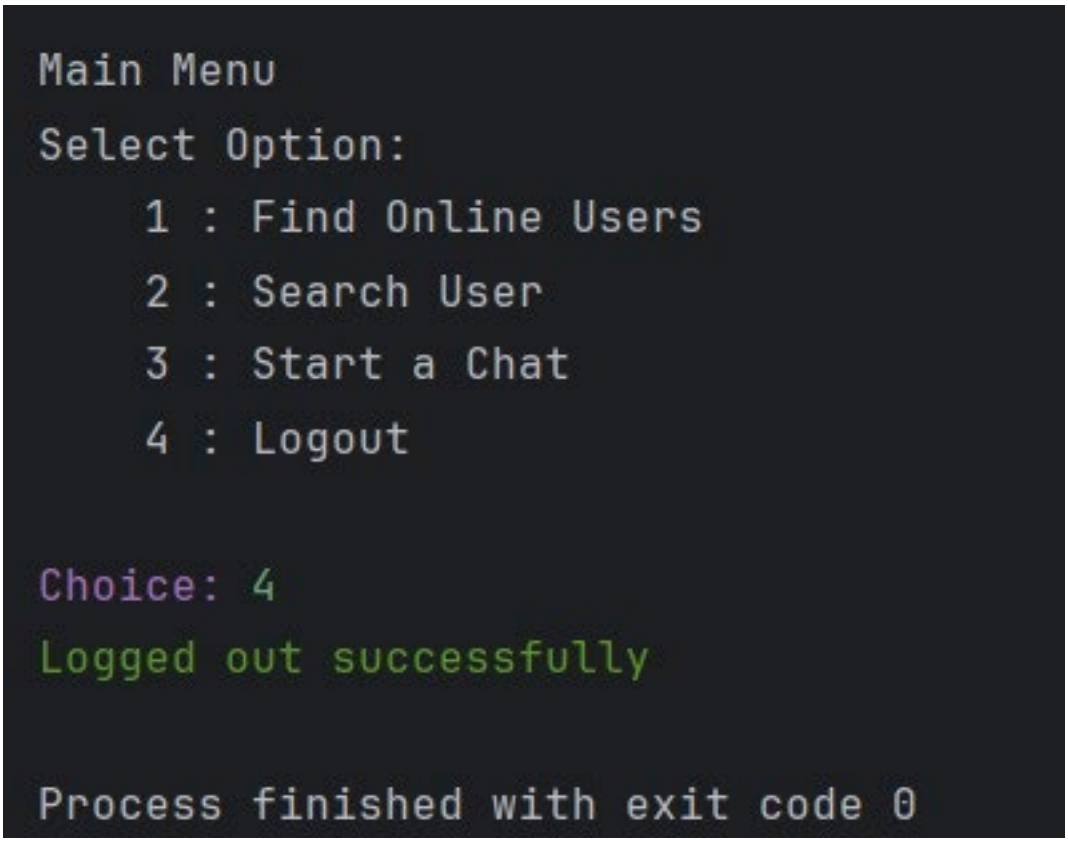

Figure 12: Logout

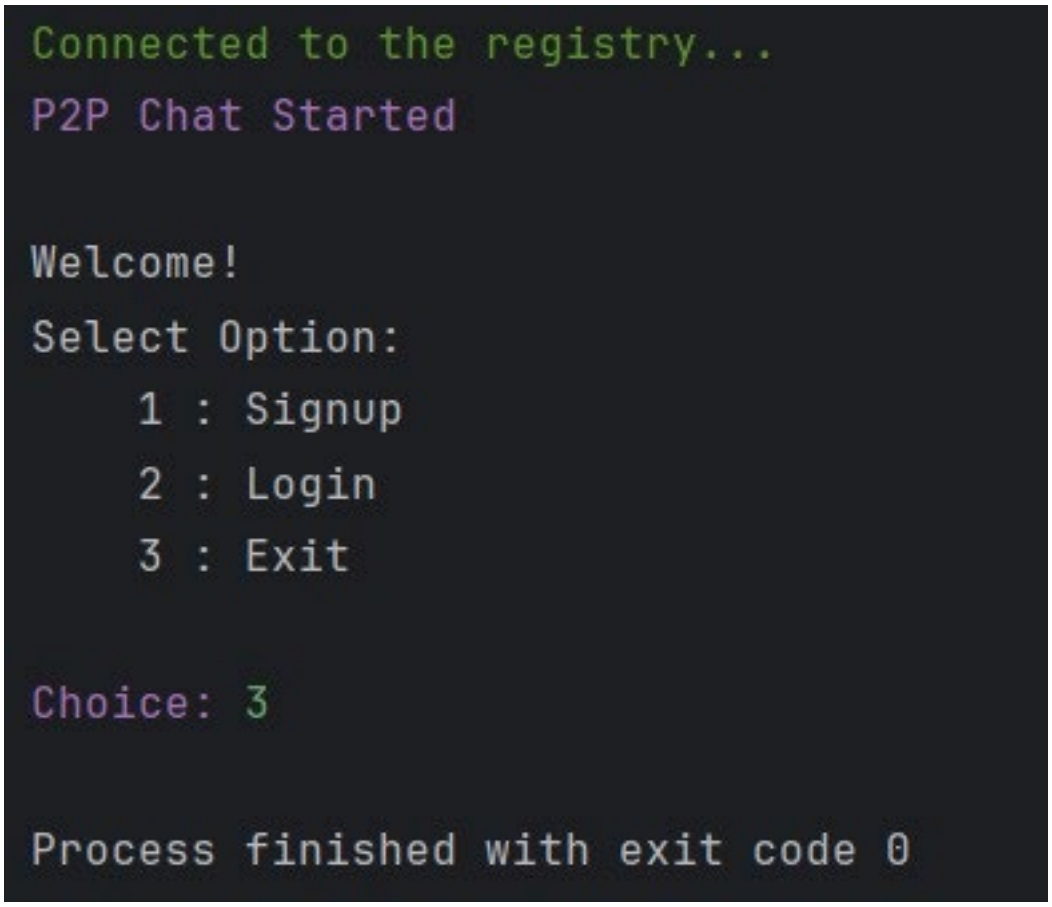

Figure 13: exit

CODE:

Code of database:

```python
from pymongo import MongoClient


# Includes database operations
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):
        cursor = self.db.accounts.find({'username': username})
        doc_count = 0

        for document in cursor:
            doc_count += 1

        if doc_count > 0:
            return True
        else:
            return False
    # 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):
        count = self.db.online_peers.count_documents({'username': username})
        return count > 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.delete_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"])

```

```
60.     def get_online_peer_list(self):
61.         online_peers_cursor = self.db.online_peers.find()
62.         online_peers_list = list(online_peers_cursor)
63.         return online_peers_list
64.
```

Description of Database Code:

Database Operations with MongoDB

This Python script contains a class called DB that encapsulates various operations for a peer-to-peer chat application using MongoDB as its database system.

Class: DB
This class initializes a connection to a MongoDB instance running locally on the default port 27017 and uses a database named 'p2p-chat'.

Methods:
init: Initializes the database connection when an instance of the DB class is created.

is_account_exist(username): Checks if an account with a specific username exists in the 'accounts' collection of the database. Returns True if the account exists, otherwise False.

register(username, password): Registers a new user by inserting a document into the 'accounts' collection with the provided username and password.

get_password(username): Retrieves the password associated with a given username from the 'accounts' collection.

is_account_online(username): Checks if an account with the provided username is currently online by counting documents in the 'online_peers' collection. Returns True if online, otherwise False.

user_login(username, ip, port): Logs in a user by inserting their details (username, ip, and port) into the 'online_peers' collection.

user_logout(username): Logs out a user by removing their details from the 'online_peers' collection based on the username.

get_peer_ip_port(username): Retrieves the IP address and port number associated with a specific username from the 'online_peers' collection.

get_online_peer_list(): Fetches a list of online peers by querying the 'online_peers' collection and returning a list of documents.

Usage:
This script demonstrates how to perform fundamental database operations such as user registration, login/logout, and user status checks within a peer-to-peer chat application using MongoDB.

Code of Peer:

```
 1. import logging
 2. import threading
 3. import time
 4. from socket import *
 5. import ssl
 6. from colorama import Fore
 7. import utility
 8.
 9.
10. # main process of the peer
11. class peerClient:
12.
13.     # peer initializations
14.     def __init__(self):
15.         # ip address of the registry
16.         self.registryName = input("Enter IP address of registry: ")
17.         # self.registryName = 'localhost'
18.         # port number of the registry
19.         self.registryPort = 15600
20.         # tcp socket connection to registry
21.         self.tcpClientSocket = socket(AF_INET, SOCK_STREAM)
22.         # Create an SSL context
23.         context = ssl.create_default_context()
24.         context.check_hostname = False
25.         context.verify_mode = ssl.CERT_NONE
26.         # Wrap the socket with SSL
27.         self.tcpClientSocket = context.wrap_socket(self.tcpClientSocket,
server_hostname=self.registryName)
28.         # Connect to the server
29.         self.tcpClientSocket.connect((self.registryName, self.registryPort))
30.         self.connectServer()
31.         # initializes udp socket which is used to send hello messages
32.         self.udpClientSocket = socket(AF_INET, SOCK_DGRAM)
33.         # udp port of the registry
34.         self.registryUDPPort = 15500
35.         # login info of the peer
36.         self.loginCredentials = (None, None)
37.         # online status of the peer
38.         self.isOnline = False
39.         self.timer = None
40.         # User Interface
41.         self.state = 0
42.         self.states = {1: "Welcome!", 2: "Main Menu"}
43.         self.options = {1: {1: "Signup", 2: "Login", 3: "Exit"},
44.                         2: {1: "Find Online Users", 2: "Search User", 3: "Start a Chat", 4:
"Logout"}}
45.
46.         # log file initialization
47.         logging.basicConfig(filename="logs/peer.log", level=logging.INFO)
48.         # as long as the user is not logged out, asks to select an option in the menu
49.         while True:
50.             # menu selection prompt
51.             if self.state == 0:
52.                 print(Fore.MAGENTA + "P2P Chat Started")
53.                 self.state = 1
54.
55.             print(Fore.RESET + '\n' + self.states[self.state] + '\nSelect Option:')
56.             for option_number, option_name in self.options[self.state].items():
57.                 print("\t" + str(option_number) + " : " + option_name)
58.             choice = input(Fore.MAGENTA + "\nChoice: ")
```

```
59.             self.handle_user_request(choice)
60.
61.     def handle_user_request(self, choice):
62.         selection = self.options[self.state][int(choice)]
63.
64.         if selection == "Signup":
65.             # Creates an account with the username and password entered by the user
66.             username = input("username: ")
67.             password = input("password: ")
68.             self.createAccount(username, password)
69.
70.         elif selection == "Login" and not self.isOnline:
71.             # Asks for the username and the password to login
72.             username = input("username: ")
73.             password = input("password: ")
74.             # asks for the port number for server's tcp socket, will be needed in late phases
75.             peer_server_port = int(input("Enter a port number for peer server: "))
76.
77.             status = self.login(username, password, peer_server_port)
78.             # is user logs in successfully, peer variables are set
79.             if status == 1:
80.                 self.isOnline = True
81.                 self.loginCredentials = (username, password)
82.                 # hello message is sent to registry
83.                 self.sendKeepAliveMessage(self.loginCredentials[0])
84.                 self.state = 2
85.
86.         elif selection == "Logout":
87.             # User is logged out and peer variables are set, and server and client sockets are
closed
88.             if self.isOnline:
89.                 self.logout(1)
90.                 self.isOnline = False
91.                 self.loginCredentials = (None, None)
92.                 print(Fore.GREEN + "Logged out successfully")
93.                 self.tcpClientSocket.close()
94.                 exit(0)
95.
96.         elif selection == "Exit":
97.             # Exits the program:
98.             self.logout(2)
99.             self.tcpClientSocket.close()
100.            exit(0)
101.
102.        elif selection == "Find Online Users":
103.            # Prompt user for the users list mode and return it
104.            while True:
105.                option = input(Fore.MAGENTA + "Retrieve detailed list with users IP and Port
numbers?(Choose y or n): ")
106.                if option == 'Y' or option == 'y':
107.                    self.find_online_user("DETAILED")
108.                    return
109.                elif option == 'N' or option == 'n':
110.                    self.find_online_user("SIMPLE")
111.                    return
112.                else:
113.                    print(Fore.RED + "Error: Please choose a valid option (y or n)\n")
114.
115.        elif selection == "Search User":
116.            # If user is online, then user is asked for a username that is wanted to be
searched
117.            if self.isOnline:
118.                username = input("Username to be searched: ")
119.                search_status = self.search_user(username)
```

```
120.                 # if user is found its ip address is shown to user
121.                 if search_status is not None and search_status != 0:
122.                     print(Fore.MAGENTA + "IP address of " + username + " is " + search_status)
123.                     time.sleep(1)
124.
125.         elif selection == "Start a Chat":
126.             print(Fore.RED + "Not available in this phase")
127.
128.         # if choice is cancel timer for hello message is cancelled
129.         elif choice == "CANCEL":
130.             self.timer.cancel()
131.         else:
132.             print(Fore.RED + "Invalid Option Selected, please try again.\n")
133.
134.     # account creation function
135.     def createAccount(self, username, password):
136.         # join message to create an account is composed and sent to registry
137.         # if response is "success" then informs the user for account creation
138.         # if response is "exist" then informs the user for account existence
139.         message = "REGISTER " + username + " " + utility.hash_password(password)
140.         response = self.send_credentials(message)
141.         # Process the response from the registry
142.
143.         if response[2] == "<200>":
144.             print(Fore.GREEN + "Account created successfully.")
145.             time.sleep(1)
146.         elif response[2] == "<300>":
147.             print(Fore.YELLOW + "Username already exists. Choose another username or login.")
148.             time.sleep(1)
149.         elif response[2] == "<404>":
150.             print(Fore.RED + "Failed to create an account. Please try again.")
151.             time.sleep(1)
152.
153.     def send_credentials(self, message):
154.         logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " +
message)
155.         self.tcpClientSocket.send(message.encode())
156.         response = self.tcpClientSocket.recv(1024).decode()
157.         logging.info("Received from " + self.registryName + " -> " + response)
158.         return response.split()
159.
160.     # login function
161.     def login(self, username, password, peerServerPort):
162.         # a login message is composed and sent to registry
163.         # an integer is returned according to each response
164.         message = "LOGIN " + username + " " + utility.hash_password(password) + " " +
str(peerServerPort)
165.         response = self.send_credentials(message)
166.         if response[2] == "<200>":
167.             print(Fore.GREEN + "Logged in successfully...")
168.             time.sleep(1)
169.             return 1
170.         elif response[2] == "<300>":
171.             print(Fore.YELLOW + "Account is already online...")
172.             time.sleep(1)
173.             return 2
174.         elif response[2] == "<404>":
175.             print(Fore.RED + "Wrong password...")
176.             time.sleep(1)
177.             return 3
178.
179.     # logout function
180.     def logout(self, option):
181.         # a logout message is composed and sent to registry
```

```
182.         # timer is stopped
183.         if option == 1:
184.             message = "LOGOUT " + self.loginCredentials[0]
185.             self.timer.cancel()
186.         else:
187.             message = "LOGOUT"
188.         logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " +
message)
189.         self.tcpClientSocket.send(message.encode())
190.
191.     # function for searching an online user
192.     def search_user(self, username):
193.         # a search message is composed and sent to registry
194.         # custom value is returned according to each response
195.         # to this search message
196.         message = "SEARCH_USER " + username
197.         logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " +
message)
198.         self.tcpClientSocket.send(message.encode())
199.         response = self.tcpClientSocket.recv(1024).decode().split()
200.         logging.info("Received from " + self.registryName + " -> " + " ".join(response))
201.         if response[2] == "<200>":
202.             print(Fore.GREEN + username + " is found successfully...")
203.             time.sleep(1)
204.             return response[3]
205.         elif response[2] == "<300>":
206.             print(Fore.YELLOW + username + " is not online...")
207.             time.sleep(1)
208.             return 0
209.         elif response[2] == "<404>":
210.             print(Fore.RED + username + " is not found")
211.             time.sleep(1)
212.             return None
213.
214.     def find_online_user(self, option):
215.         message = "DISCOVER_PEERS " + option + " " + self.loginCredentials[0]
216.         logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " +
message)
217.         self.tcpClientSocket.send(message.encode())
218.         response = self.tcpClientSocket.recv(1024).decode().split()
219.         logging.info("Received from " + self.registryName + " -> " + " ".join(response))
220.         if response[2] == "<200>":
221.             response = response[3:]
222.             if option == "DETAILED":
223.                 print(Fore.RESET + "#  Username".ljust(18) + "(IP:Port)")
224.                 for i in range(0, len(response), 2):
225.                     print(Fore.GREEN + f"{i+1}  {response[i]:15}{response[i+1]}")
226.             else:
227.                 print(Fore.RESET + "Username")
228.                 for username in response:
229.                     print(Fore.GREEN + username)
230.             time.sleep(1)
231.         elif response[2] == "<404>":
232.             print(Fore.YELLOW + "No Online Users right now, please check back later")
233.             time.sleep(1)
234.
235.     # function for sending hello message
236.     # a timer thread is used to send hello messages to udp socket of registry
237.     def sendKeepAliveMessage(self, username):
238.         message = "KEEP_ALIVE " + username
239.         logging.info("Send to " + self.registryName + ":" + str(self.registryUDPPort) + " -> "
+ message)
240.         self.udpClientSocket.sendto(message.encode(), (self.registryName,
self.registryUDPPort))
```

```
241.
242.         # Assuming you expect a response from the registry
243.
244.         # Schedule the next hello message
245.         self.timer = threading.Timer(1, self.sendKeepAliveMessage, args=[username])
246.         self.timer.start()
247.
248.     def connectServer(self):
249.         starting_message = "HELLO_P2P"
250.         logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " +
starting_message)
251.         self.tcpClientSocket.send(starting_message.encode())
252.         response = self.tcpClientSocket.recv(1024).decode().split()
253.         logging.info("Received from " + self.registryName + " -> " + " ".join(response))
254.         status_code = int(response[2])
255.         if status_code == 200:
256.             print(Fore.GREEN + "Connected to the registry...")
257.
258.
259. # peer is started
260. main = peerClient()
261.
```

P2P chatting Application

Description of Peer code:

The peer basic Client have the follwing functionalities:

User Registration/Login/Logout:

Users can register by creating an account with a username and password.
Registered users can log in using their credentials and a specified port number for the peer server.
Users can log out, which terminates the connection to the registry and closes sockets.
User Interaction:

The program prompts the user with various options like creating an account, logging in, searching for users, finding online users, etc.
Different options are handled within the handle_user_request method based on user input.

The handle_user_request method is a crucial part of the peerClient class. It manages the user's interaction with the peer-to-peer system by interpreting and acting upon the user's input based on the current state of the program. Here's a breakdown of its functionality:

Input Handling:

It takes the user's choice as input and identifies the corresponding action to be performed based on the current state of the system.
Menu Navigation:

The method interprets the user's choice and executes specific functionalities related to account creation, login, logout, searching users, and other available options.
Action Execution:

Based on the user's input, it triggers specific actions like creating an account, logging in, logging out, searching for users, finding online users, etc.
Here's a breakdown of how it handles various user choices:

Signup/Login/Logout/Exit:

If the user chooses to sign up, it initiates the account creation process.
For login, it prompts the user for credentials and attempts to log in.
If the user chooses to log out, it initiates the logout process.
Exiting the program involves logging out and closing connections.
Find Online Users/Search User/Start a Chat:

Searching for online users can be done in detailed or simple mode, providing information about users currently online.
Searching for a specific user prompts the user to input the username and attempts to find the user's IP address if they are online.
Starting a chat is not available in this phase, as indicated in the code.
Cancel/Invalid Choice:

The method handles cancellation of actions or invalid choices gracefully by informing the user of invalid selections and giving guidance on proper input.

Registry Communication:

Communication with the registry server happens via TCP sockets.
Different messages (e.g., registration, login, logout, search) are sent to the registry, and responses are received accordingly.
Keep-Alive Mechanism:

The sendKeepAliveMessage method seems to manage a timer to periodically send "hello" messages (KEEP_ALIVE) to the registry via UDP to maintain online status.
Logging:

Logs are maintained for various events, such as sending/receiving messages to/from the registry, errors, etc.
User Interface:

The program displays a menu-driven user interface using the colorama library for color-coded text in the terminal.

The color coding is mainly as follows:

Green : Successful operation
Red : Error
Yellow : Abnormal Scenario but not error
Magenta : Prompt user for input
Blue : Logs at the server
More details will be added after implementing the chatting functionality

Code of Registry:

```
'''
    ##  Implementation of registry
    ##  150114822 - Eren Ulaş
'''

from socket import *
import threading
import select
import logging

from colorama import Fore

import db
import ssl


# This class is used to process the peer messages sent to registry
# for each peer connected to registry, a new client thread is created
class ClientThread(threading.Thread):
    # initializations for client thread
    def __init__(self, ip, port, tcpClientSocket):
        threading.Thread.__init__(self)
        # ip of the connected peer

        self.ip = ip
        # port number of the connected peer
        self.port = port
        # socket of the peer
        self.tcpClientSocket = tcpClientSocket
        # Create SSL context
        self.context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
        self.context.load_cert_chain(certfile="security/server.crt",
keyfile="security/server.key")
        # 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(Fore.BLUE+"Connection from: " + self.ip + ":" + str(port))
        print(Fore.BLUE+"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] == "REGISTER":
                    # join-exist is sent to peer,
                    # if an account with this username already exists
                    if db.is_account_exist(message[1]):
                        response = "REGISTER <EXIST> <300>"
                        print("From-> " + self.ip + ":" + str(self.port) + " " + response)
                        logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
```

```
 59.                         self.tcpClientSocket.send(response.encode())
 60.                     # join-success is sent to peer,
 61.                     # if an account with this username is not exist, and the account is created
 62.                     else:
 63.                         db.register(message[1], message[2])
 64.                         response = "REGISTER <SUCCESS> <200>"
 65.                         logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
 66.                         self.tcpClientSocket.send(response.encode())
 67.                 #   LOGIN    #
 68.                 elif message[0] == "LOGIN":
 69.                     # login-account-not-exist is sent to peer,
 70.                     # if an account with the username does not exist
 71.                     if not db.is_account_exist(message[1]):
 72.                         response = "AUTH <FAILURE> <404>"
 73.                         logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
 74.                         self.tcpClientSocket.send(response.encode())
 75.                     # login-online is sent to peer,
 76.                     # if an account with the username already online
 77.                     elif db.is_account_online(message[1]):
 78.                         response = "AUTH <ONLINE> <300>"
 79.                         logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
 80.                         self.tcpClientSocket.send(response.encode())
 81.                     # login-success is sent to peer,
 82.                     # if an account with the username exists and not online
 83.                     else:
 84.                         # retrieves the account's password, and checks if the one entered by
the user is correct
 85.                         retrievedPass = db.get_password(message[1])
 86.                         # if password is correct, then peer's thread is added to threads list
 87.                         # peer is added to db with its username, port number, and ip address
 88.                         if retrievedPass == message[2]:
 89.                             self.username = message[1]
 90.                             self.lock.acquire()
 91.                             try:
 92.                                 tcpThreads[self.username] = self
 93.                             finally:
 94.                                 self.lock.release()
 95.
 96.                             db.user_login(message[1], self.ip, self.port)
 97.                             # login-success is sent to peer,
 98.                             # and a udp server thread is created for this peer, and thread is
started
 99.                             # timer thread of the udp server is started
100.                             response = "AUTH <SUCCESS> <200>"
101.                             logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
102.                             self.tcpClientSocket.send(response.encode())
103.                             self.udpServer = UDPServer(self.username, self.tcpClientSocket)
104.                             self.udpServer.start()
105.                             self.udpServer.timer.start()
106.                         # if password not matches and then login-wrong-password response is
sent
107.                         else:
108.                             response = "AUTH <FAILURE> <404>"
109.                             logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
110.                             self.tcpClientSocket.send(response.encode())
111.                 #   LOGOUT  #
112.                 elif message[0] == "LOGOUT":
113.                     # if user is online, removes the user from onlinePeers list and removes the
thread for this user
```

```
                    # from tcpThreads socket is closed and timer thread of the udp for this user is cancelled
                    if db.is_account_online(self.username):
                        db.user_logout(message[1])
                        self.lock.acquire()
                        try:
                            if self.username in tcpThreads:
                                del tcpThreads[self.username]
                        finally:
                            self.lock.release()
                        print(Fore.BLUE+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_USER":
                    # 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 db.is_account_online(message[1]):
                            peer_info = db.get_peer_ip_port(message[1])
                            response = "SEARCH_USER_RESPONSE <SUCCESS> <200> " + str(peer_info[0]) + ":" + str(
                                peer_info[1])
                            logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                            self.tcpClientSocket.send(response.encode())
                        else:
                            response = "SEARCH_USER_RESPONSE <NOT_ONLINE> <300>"
                            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_RESPONSE <NOT_FOUND> <404>"
                        logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                        self.tcpClientSocket.send(response.encode())
                # online peers discovery
                elif message[0] == "DISCOVER_PEERS":
                    peer_list = db.get_online_peer_list()
                    # remove the requesting user from the list
                    if peer_list:
                        for peer in peer_list:
                            if peer['username'] == message[2]:
                                peer_list.remove(peer)
                    if peer_list and len(peer_list) > 0:
                        # detailed list
                        if message[1] == "DETAILED":
                            response = "PEER_LIST <SUCCESS> <200> " + ' '.join(
                                f"{peer['username']} ({peer['ip']}:{peer['port']})" for peer in peer_list
                            )

                            logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                            self.tcpClientSocket.send(response.encode())
                        # partial list
                        else:
```

```
171.                             usernames = [peer['username'] for peer in peer_list]
172.                             response = "PEER_LIST <SUCCESS> <200> " + ' '.join(usernames)
173.                             logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
174.                             self.tcpClientSocket.send(response.encode())
175.                     # failure empty list
176.                     else:
177.                         response = "PEER_LIST <FAILURE> <404>"
178.                         logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
179.                         self.tcpClientSocket.send(response.encode())
180.
181.
182.             except OSError as oErr:
183.                 logging.error("OSError: {0}".format(oErr))
184.                 response = "HELLO_BACK " + "FAILURE " + "404"
185.                 logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
186.
187.                 db.user_logout(self.username)
188.
189.                 # function for resettin the timeout for the udp timer thread
190.
191.     def resetTimeout(self):
192.         self.udpServer.resetTimer()
193.
194.
195. # implementation of the udp server thread for clients
196. class UDPServer(threading.Thread):
197.
198.     # udp server thread initializations
199.     def __init__(self, username, clientSocket):
200.         threading.Thread.__init__(self)
201.         self.username = username
202.         self.default_timeout = 3
203.         # timer thread for the udp server is initialized
204.         self.timer = threading.Timer(self.default_timeout, self.waitKeepAliveMessage)
205.         self.tcpClientSocket = clientSocket
206.
207.     # if hello message is not received before timeout
208.     # then peer is disconnected
209.     def waitKeepAliveMessage(self):
210.
211.         if self.username is not None:
212.             notification = "TIMEOUT " + self.username
213.             self.tcpClientSocket.send(notification.encode())
214.             db.user_logout(self.username)
215.             if self.username in tcpThreads:
216.                 del tcpThreads[self.username]
217.         self.tcpClientSocket.close()
218.         print(Fore.BLUE +"Removed " + self.username + " from online peers")
219.
220.     # resets the timer for udp server
221.     def resetTimer(self):
222.         self.timer.cancel()
223.         self.timer = threading.Timer(self.default_timeout, self.waitKeepAliveMessage)
224.         self.timer.start()
225.
226.
227. # tcp and udp server port initializations
228. print("Registy started...")
229. port = 15600
230. portUDP = 15500
231.
232. # db initialization
```

```
233. db = db.DB()
234.
235. # gets the ip address of this peer
236. # first checks to get it for windows devices
237. # if the device that runs this application is not windows
238. # it checks to get it for macos devices
239. hostname = gethostname()
240. try:
241.     host = gethostbyname(hostname)
242. except gaierror:
243.     import netifaces as ni
244.
245.     host = ni.ifaddresses('en0')[ni.AF_INET][0]['addr']
246.
247. print("Registry IP address: " + host)
248. print("Registry port number: " + str(port))
249.
250. # onlinePeers list for online account
251. onlinePeers = {}
252. # accounts list for accounts
253. accounts = {}
254. # tcpThreads list for online client's thread
255. tcpThreads = {}
256.
257. # tcp and udp socket initializations
258. tcpSocket = socket(AF_INET, SOCK_STREAM)
259. udpSocket = socket(AF_INET, SOCK_DGRAM)
260. tcpSocket.bind((host, port))
261. udpSocket.bind((host, portUDP))
262. tcpSocket.listen(1000)
263.
264. # input sockets that are listened
265. inputs = [tcpSocket, udpSocket]
266.
267. # log file initialization
268. logging.basicConfig(filename="registry.log", level=logging.INFO)
269.
270. # as long as at least a socket exists to listen registry runs
271. while inputs:
272.
273.     print("Listening for incoming connections...")
274.     # monitors for the incoming connections
275.     readable, writable, exceptional = select.select(inputs, [], [])
276.     for s in readable:
277.         # if the message received comes to the tcp socket
278.         # the connection is accepted and a thread is created for it, and that thread is started
279.         if s is tcpSocket:
280.             tcpClientSocket, addr = tcpSocket.accept()
281.             newThread = ClientThread(addr[0], addr[1], tcpClientSocket)
282.             newThread.tcpClientSocket =
newThread.context.wrap_socket(newThread.tcpClientSocket, server_side=True)
283.             newThread.start()
284.             response = "HELLO_BACK " + "SUCCESS " + "200 "
285.             logging.info("Send to " + addr[0] + ":" + str(addr[1]) + " -> " + response)
286.             newThread.tcpClientSocket.send(response.encode())
287.         # if the message received comes to the udp socket
288.         elif s is udpSocket:
289.             # received the incoming udp message and parses it
290.             message, clientAddress = s.recvfrom(1024)
291.             message = message.decode().split()
292.             # checks if it is a hello message
293.             if message[0] == "KEEP_ALIVE":
294.                 # checks if the account that this hello message
295.                 # is sent from is online
```

```
296.                 if message[1] in tcpThreads:
297.                     # resets the timeout for that peer since the hello message is received
298.                     tcpThreads[message[1]].resetTimeout()
299.                     print("KEEP_ALIVE is received from " + message[1])
300.                     loggingmessage = "KEEP_ALIVE <SUCCESS> <200>"
301.
302.                     logging.info(
303.                         "Received from " + clientAddress[0] + ":" + str(clientAddress[1]) + " -
> " + " ".join(message))
304.                     # Send the response back to the UDP client
305.
306. # registry tcp socket is closed
307. tcpSocket.close()
308.
```

Description of Registry code:

ClientThread: Represents individual threads for each connected client. It handles communication, user registration, login, logout, peer search, and maintains a UDP server for each user for keeping them online.

UDPServer class manages the UDP server functionality for clients. Here's what each part of the class does:

Initialization:

init: Initializes the UDP server thread with the username and clientSocket parameters. It also sets a default timeout and initializes a timer to monitor incoming messages.
Timeout Handling:

waitKeepAliveMessage: Monitors for a specific message (a "hello" message or "keep alive" message) from the client. If this message isn't received before the timeout, it triggers a disconnection process. It notifies the client about the timeout, logs out the user from the system, and removes the user's thread from the active thread list (tcpThreads). Finally, it closes the socket and prints a message indicating the removal of the user from the online peers.
Timer Reset:

resetTimer: Resets the timer for the UDP server when necessary, canceling the existing timer and starting a new one.
This class is responsible for managing the keep-alive mechanism for the UDP connections to ensure that users stay connected within a specific time frame.

Main Section:

Initializes TCP and UDP sockets for communication.
Monitors incoming connections and creates a new ClientThread for TCP connections.

P2P chatting Application

Manages incoming UDP messages to keep clients online by sending keep-alive messages.
The code communicates with a MongoDB database (db module) for account management (registration, login/logout) and maintains lists of online peers and threads for each user.

The run() method within the ClientThread class handles the main functionality for each connected client, It mainly provides:

Receiving Messages: It continuously waits for incoming messages from connected peers via the tcpClientSocket.

Message Processing: It splits the received message and processes it based on its type.

REGISTER: Checks if an account exists with the username. If it does, sends a response indicating existence; otherwise, registers a new account.
LOGIN: Handles login operations, checking account existence, online status, and password verification. If successful, adds the user to the list of online peers.
LOGOUT: Logs the user out by removing them from online status and closing connections.
SEARCH_USER: Searches for a specific user and responds with their online status or presence details.
DISCOVER_PEERS: Retrieves a list of online peers, either detailed or partial, based on the request.
Exception Handling: Catches OSError and handles it by logging the error and sending a failure response.

Timeout Reset: Contains a method resetTimeout to reset the timeout for the UDP timer thread (udpServer).

The implementation uses SSL for secure communication (ssl module) and logs events into a file (registry.log) for debugging and monitoring purposes.

Code of Utility :

```python
import hashlib


def hash_password(password):
    # Encode the password string to bytes before hashing
    password_bytes = password.encode('utf-8')
    # Create an SHA-256 hash object
    sha256 = hashlib.sha256()
    # Update the hash object with the password bytes
    sha256.update(password_bytes)
    # Get the hexadecimal representation of the hashed password
    hashed_password = sha256.hexdigest()
    return hashed_password

```

Description of Utility code:

The Hashing technique for the application is described as follows:

Input:
The function takes a password string as input.

Encoding to Bytes:
The password string is converted into bytes using UTF-8 encoding. Hashing functions typically operate on bytes, hence the conversion from string to bytes.

Hashing:
It creates an SHA-256 hash object sha256.
The update() method updates the hash object with the encoded password bytes.
This effectively hashes the password using the SHA-256 algorithm.

Hexadecimal Representation:
The hexdigest() method generates the hexadecimal representation of the hashed password.
Hexadecimal representation is commonly used to present hash values in a human-readable format.

Return:
The function returns the hashed password as a hexadecimal string.

This utility function is used for securely storing passwords by converting them into irreversible hashed values, making it computationally infeasible to reverse the hash and obtain the original password. It's employed to enhance security by avoiding storing plain text passwords in the DB.