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 3

Peer-to-Peer Architecture and Chat Rooms

GitHub Repo: https://github.com/KareemWael1/P2P-ChatRoom

Table of Contents

Table of Figures

P2P chatting Application

Abstract

Phase 3 of the project focuses on the integration of a peer-to-peer architecture and the implementation of chat rooms within the system. The primary objectives include modifying the server to support peer-to-peer connections and incorporating features for creating and joining chat rooms. The implementation encompasses the establishment of basic text-based communication within these chat rooms using TCP. Specific tasks involve adapting the server to handle dynamic changes within chat rooms, such as user joins and exits. Overall, this phase aims to enhance the system's architecture, allowing users to seamlessly interact through peer-to-peer connections, create and join chat rooms, and engage in real-time text communication while efficiently managing dynamic changes within these virtual spaces.

1 Chat Rooms Functions:

1.1 Find Chat Rooms:

1.1.1 Code:

PEER MAIN CLASS:

```
    def findChatRooms(self):
        chatrooms_list = []
        message = "SHOW-ROOM-LIST"
        logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode()
        logging.info("Received from " + self.registryName + " -> " + " ".join(response))
        status_code = response.split()[2]
        if status_code == "<200>":
            # Extract the list part from the received message
            list_start_index = response.find("<200>") + len("<200>")
            chatrooms_list_str = response[list_start_index:].strip()

            # Split the string into a list
            chatrooms_list = list(chatrooms_list_str.split('.'))[:-1]
            for chatroom in chatrooms_list:
                chatroom = chatroom.split()
            return chatrooms_list
        return chatrooms_list

```

1.1.2 Description:

The findChatRooms responsible for querying a chat room registry, specified by self.registryName and self.registryPort, to retrieve a list of available chat rooms. It begins by constructing a message, "SHOW-ROOM-LIST," and logs the intention to send this message to the specified registry. The function then sends the encoded message using a TCP client socket (self.tcpClientSocket) and waits for a response. Upon receiving the response, it logs the received message and extracts the status code from it. If the status code is "<200>" (indicating a successful response), the function proceeds to extract and process the list of chat rooms from the response. It retrieves the substring starting from the position immediately following "<200>" and splits it into a list of chat rooms. The resulting list is then returned as the output of the function. If the status code is not "<200>" or if an error occurs during the process, an empty list is returned. The function provides a way to dynamically discover and retrieve the available chat rooms from the registry, facilitating the functionality of the broader chat application.

1.2 Create Chat Room:

1.2.1 Code:

```
    def createChatroom(self, name):
        message = "CREATE-CHAT-ROOM " + name + " " + self.loginCredentials[0]
        logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode().split()
        logging.info("Received from " + self.registryName + " -> " + " ".join(response))
        status_code = response[2]
        if status_code == "<200>":
            self.chatroom = name
            print(Fore.GREEN + "A chatroom with name " + name + " has been created...\n")
            time.sleep(1)
            self.connect_to_chatroom(self.loginCredentials[0])
            return True
        else:
            return False

```

1.2.2 Description:

The **createChatroom** takes the desired name for the chat room as a parameter and constructs a message in the format "CREATE-CHAT-ROOM <name> <creator_username>." The function logs the intention to send this message to the specified registry (**self.registryName** and **self.registryPort**). Using a TCP client socket (**self.tcpClientSocket**), the message is sent to the registry, and the function waits for a response. The received response is then logged, and the status code is extracted from it. If the status code is "<200>" (indicating a successful creation), the function sets the current chat room (**self.chatroom**) to the newly created room, prints a success message, introduces a brief delay using **time.sleep(1)**, and then connects the user to the created chat room. Finally, the function returns **True**. If the creation is unsuccessful (status code is not "<200>"), the function returns **False**. Overall, this function encapsulates the process of creating a chat room, updating the application state, and providing feedback to the user.

1.3 Join Chat Room:

1.3.1 Code:

```
def joinChatroom(self, name):
        message = "JOIN-CHAT-ROOM " + name + " " + self.loginCredentials[0]
        logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode().split()
        logging.info("Received from " + self.registryName + " -> " + " ".join(response))
        status_code = response[2]
        if status_code == "<200>":
            print(Fore.GREEN + "You have joined the room " + name + " successfully...\n")
            time.sleep(0.5)
            self.chatroom = name
            self.connect_to_chatroom(response[3])
            return True
        return False
```

1.3.2 Description:

The **joinChatroom** takes the name of the desired chat room as a parameter and constructs a message in the format "JOIN-CHAT-ROOM <name> <username>." The function logs the intention to send this message to the specified registry (**self.registryName** and **self.registryPort**). Using a TCP client socket (**self.tcpClientSocket**), the message is sent to the registry, and the function waits for a response. The received response is then logged, and the status code is extracted from it. If the status code is "<200>" (indicating successful entry), the function prints a success message, introduces a brief delay using **time.sleep(0.5)**, updates the current chat room (**self.chatroom**), and connects the user to the chat room. The function then returns **True**. If the entry is unsuccessful (status code is not "<200>"), the function returns **False**. Overall, this function encapsulates the process of joining a chat room, providing feedback to the user, and updating the application state accordingly.

1.4 Exit Chat Room:

1.4.1 Code:

```
    def exitChatroom(self, username):
        message = "ROOM-EXIT " + username + " " + self.chatroom
        logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode().split()
        logging.info("Received from " + self.registryName + " -> " + " ".join(response))
        status_code = response[2]
        if status_code == "<200>":
            return True
        return False

```

1.4.2 Description:

The **exitChatroom** takes the username of the user who wants to exit as a parameter and constructs a message in the format "ROOM-EXIT <username> <chatroom>." The function logs the intention to send this message to the specified registry (**self.registryName** and **self.registryPort**). Using a TCP client socket (**self.tcpClientSocket**), the message is sent to the registry, and the function waits for a response. The received response is then logged, and the status code is extracted from it. If the status code is "<200>" (indicating a successful exit), the function returns **True**. Otherwise, it returns **False**. This function encapsulates the process of notifying the chat room registry that a user is leaving a particular chat room, providing a feedback mechanism, and potentially allowing the application to perform any necessary cleanup or update actions associated with a user's departure from the room.

1.5 Connect To Chat Room:

1.5.1 Code:

```
   def connect_to_chatroom(self, host):
        search_status = self.search_user(host, False)
        # if searched user is found, then its port number is retrieved and a client thread is created
        if search_status and search_status != 0:
            search_status = search_status.split(":")
            # creates the server thread for this peer, and runs it
            self.peerServer = PeerServer(self.loginCredentials[0], int(search_status[1]))
            self.peerServer.start()
            self.peerClient = PeerClient(int(search_status[1]), self.loginCredentials[0], self.peerServer,
                                         self.chatroom)
            self.peerClient.start()
            self.peerClient.join()
            # Loop in the chatting until user exits
            self.peerClient.group_chat()
            # Exit from chatroom
            self.exitChatroom(self.loginCredentials[0])

```

1.5.2 Description:

The **connect_to_chatroom** searches for a user associated with a given host, and if successful, it sets up a server thread (**self.peerServer**) and a client thread (**self.peerClient**). These threads facilitate communication with the chat room hosted on the specified **host**. The method starts both threads, initiates a group chat loop, and ensures the client thread runs in the background. After the user exits the group chat loop, the method signals the user's departure from the chat room using the **exitChatroom** method. In summary, this method manages the connection to a chat room, enabling the user to participate in group chat and exit when desired.

1.6 Get Room Peers:

1.6.1 Code:

```
def getRoomPeers(self):
        room_peers = []
        message = "DISCOVER-ROOM-PEERS " + self.chatroom
        logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode()
        logging.info("Received from " + self.registryName + " -> " + " ".join(response))
        status_code = response.split()[2]
        if status_code == "<200>":
            # Assuming peers are present in the response starting from index 3
            list_start_index = response.find("<200>") + len("<200>")
            peerlist_list_str = response[list_start_index:].strip()

            # Split the string into a list
            room_peers = peerlist_list_str.split()

            # Print the chatrooms list

            print(Fore.CYAN, str(room_peers))
            return list(room_peers)
        return room_peers

```

PEER CLIENT CLASS:

1.7 Group Chat :

1.7.1 Code:

```
    def group_chat(self):
        print(Fore.GREEN + "Welcome to Chatroom " + self.chatroom_name)
        print("Enter a message to send, enter 'q' to leave the room\n")
        while True:
            try:
                message = input()
                if message == 'q':
                    message = "System: User " + self.username + " left."
                    self.peerServer.udp_socket.close()
                    time.sleep(0.1)
                    self.udp_socket.sendto(message.encode(), (self.multicast_group, self.multicast_port))
                    self.udp_socket.close()
                    return
                message = self.username + ": " + message
                self.udp_socket.sendto(message.encode(), (self.multicast_group, self.multicast_port))
            # handles the exceptions, and logs them
            except OSError as oErr:
                logging.error("OSError: {0}".format(oErr))
            except ValueError as vErr:
                logging.error("ValueError: {0}".format(vErr))

```

1.7.2 Description:

The **group_chat** manages the user experience during a group chat session. It prints a welcome message for the specified chat room, prompts the user for messages, and enters a loop to continuously handle user input. If the user types 'q,' indicating a desire to leave the chat room, the method constructs and sends an exit message, closes relevant sockets, and exits the loop. Regular messages from the user are formatted and sent to the chat room's multicast group. Exception handling is included to log any potential errors during the chat, such as **OSError** or **ValueError**. In essence, this method facilitates user interaction in a group chat setting, including the option to exit gracefully.

2 Outputs

```
Main Menu
Select Option:
    1 : Find Online Users
    2 : Search User
    3 : Create a Chat Room
    4 : Find Chat Rooms
    5 : Join a Chat Room
    6 : Logout

Choice: 4
No available Chat Rooms
```

Figure 1:Find chatrooms (no rooms)

```
Main Menu
Select Option:
    1 : Find Online Users
    2 : Search User
    3 : Create a Chat Room
    4 : Find Chat Rooms
    5 : Join a Chat Room
    6 : Logout

Choice: 3
Chat room name: Networks
A chatroom with name Networks has been created...

Welcome to Chatroom Networks
Enter a message to send, enter 'q' to leave the room
```

Figure 2:Create chatroom

```
Main Menu
Select Option:
    1 : Find Online Users
    2 : Search User
    3 : Create a Chat Room
    4 : Find Chat Rooms
    5 : Join a Chat Room
    6 : Logout

Choice: 4
#  Name          Host          group
1  Networks      Kareem        ['Kareem']
```

Figure 3: Find Chatroom (found one)

```
Main Menu
Select Option:
    1 : Find Online Users
    2 : Search User
    3 : Create a Chat Room
    4 : Find Chat Rooms
    5 : Join a Chat Room
    6 : Logout

Choice: 5
Chat room name: Random
No chatroom with the name Random!
Hint: enter quit to return to main menu
Chat room name: quit

Main Menu
Select Option:
    1 : Find Online Users
    2 : Search User
    3 : Create a Chat Room
    4 : Find Chat Rooms
    5 : Join a Chat Room
    6 : Logout
```

Figure 4: Join room (invalid name and quitting to main menu)

```
Main Menu
Select Option:
    1 : Find Online Users
    2 : Search User
    3 : Create a Chat Room
    4 : Find Chat Rooms
    5 : Join a Chat Room
    6 : Logout

Choice: 5
Chat room name: Networks
You have joined the room Networks successfully...

Welcome to Chatroom Networks
Enter a message to send, enter 'q' to leave the room
```

Figure 5: Join Room successfully.

```
Choice: 3
Chat room name: Networks
A chatroom with name Networks has been created...

Welcome to Chatroom Networks
Enter a message to send, enter 'q' to leave the room

Hi
Ahmed: Hello
How are you?
Ahmed: I am fine, thanks
Did you enjoy Comuter Networks course?
Ahmed: Yes! I just hope we get good grades at the end too
me too...
Ahmed: Ok, I'll be leaving then
Ok bye
System: User Ahmed left.
```

```
Choice: 5
Chat room name: Networks
You have joined the room Networks successfully...

Welcome to Chatroom Networks
Enter a message to send, enter 'q' to leave the room

Kareem: Hi
Hello
Kareem: How are you?
I am fine, thanks
Kareem: Did you enjoy Comuter Networks course?
Yes! I just hope we get good grades at the end too
Kareem: me too...
Ok, I'll be leaving then
Kareem: Ok bye
q
```

Figure 6: When chatting, the System notifies the peers in the room when someone leaves.

```
Main Menu
Select Option:
    1 : Find Online Users
    2 : Search User
    3 : Create a Chat Room
    4 : Find Chat Rooms
    5 : Join a Chat Room
    6 : Logout

Choice: 4
#  Name           Host           group
1  Networks       Kareem         ['Kareem']
```

Figure 7:The room still exists for the user who left to rejoin.

```
Hi again
Ahmed: Hi
Sorry, leaving for an urgent matter
Ahmed: ok
q

Main Menu
Select Option:
    1 : Find Online Users
    2 : Search User
    3 : Create a Chat Room
    4 : Find Chat Rooms
    5 : Join a Chat Room
    6 : Logout

Choice: 4
#  Name          Host          group
1  Networks      Ahmed         ['Ahmed']
```

Figure 8: When host (Kareem) leaves. When he sees the room, he finds that the host is transferred to someone else in the room (Ahmed).

```
Choice: 4
No available Chat Rooms

Main Menu
Select Option:
    1 : Find Online Users
    2 : Search User
    3 : Create a Chat Room
    4 : Find Chat Rooms
    5 : Join a Chat Room
    6 : Logout

Choice: |
```

```
ok
System: User Kareem left.
q

Main Menu
Select Option:
    1 : Find Online Users
    2 : Search User
    3 : Create a Chat Room
    4 : Find Chat Rooms
    5 : Join a Chat Room
    6 : Logout

Choice:
```

Figure 9: When no one in the room now (as Ahmed left) the room is deleted.

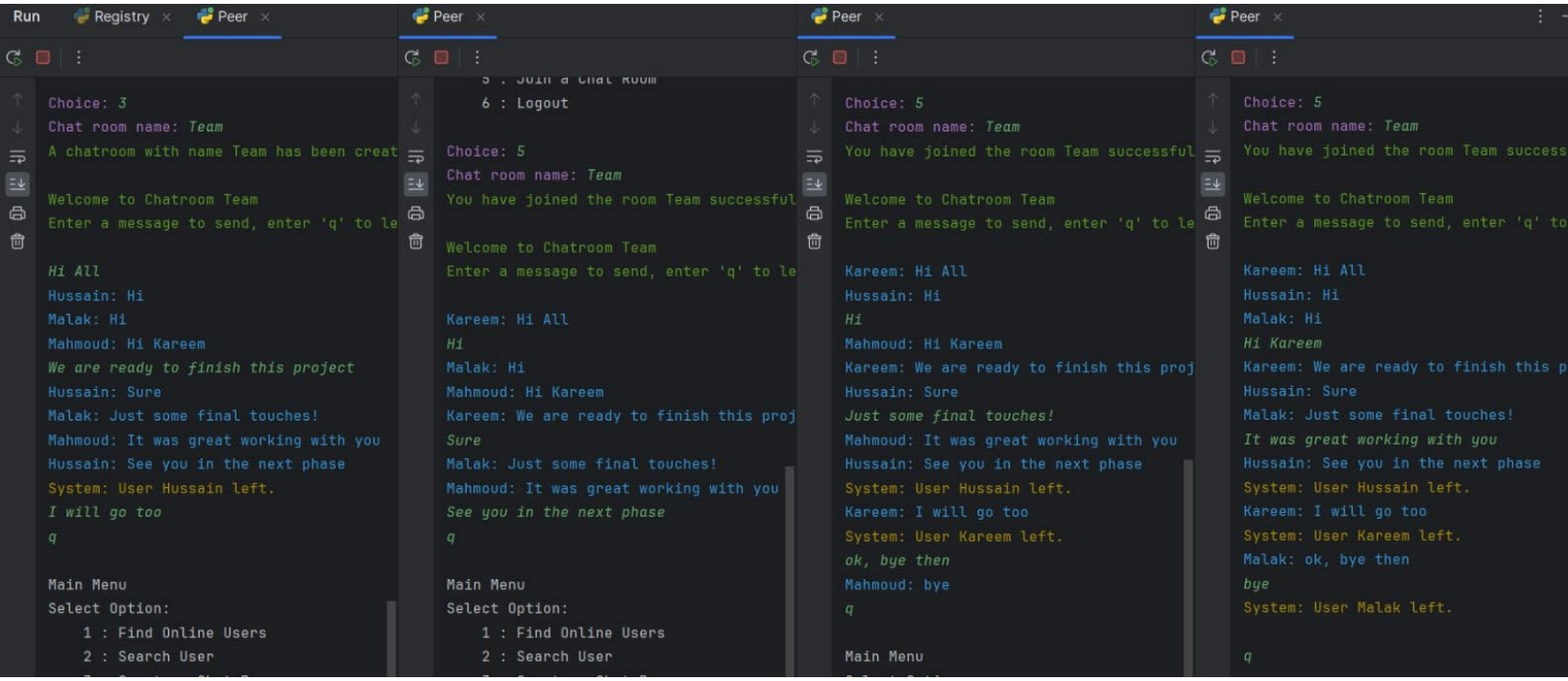

Figure 10: Chatroom with multiple users

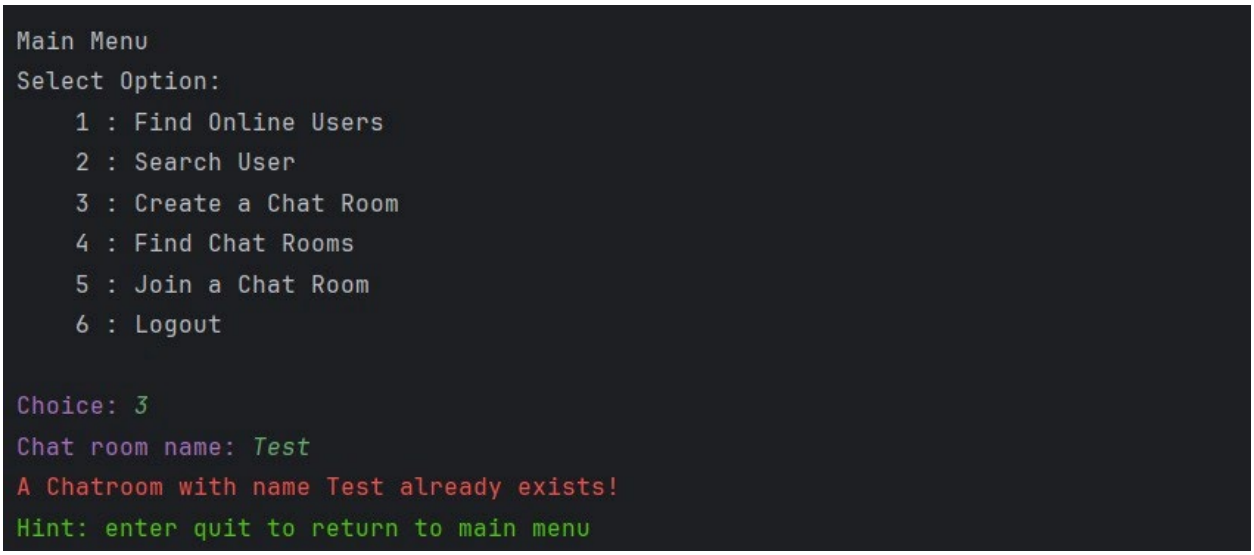

Figure 11: Trying to create a room with a name that already exists.

```
Choice: 3
Chat room name: Hi
A chatroom with name Hi has been created...

Welcome to Chatroom Hi
Enter a message to send, enter 'q' to leave the room

I am in the room
Malak: I am in the room
```

```
Choice: 5
Chat room name: Hi
You have joined the room Hi successfully...

Welcome to Chatroom Hi
Enter a message to send, enter 'q' to leave the room

Kareem: I am in the room
I am in the room
```

```
Choice: 4
#  Name          Host          Users in Chatroom
1  Hi            Kareem        Malak, Kareem

Main Menu
Select Option:
    1 : Find Online Users
    2 : Search User
    3 : Create a Chat Room
    4 : Find Chat Rooms
    5 : Join a Chat Room
```

Figure 12: List group chat with multiple users.

```
Choice: 3
Chat room name: room
A chatroom with name room has been created...

Welcome to Chatroom room
Enter a message to send, enter 'q' to leave the room

I am waiting for someone...
System: User Malak joined!
Hi
Malak: Hi
```

```
    6 : Logout

Choice: 5
Chat room name: room
You have joined the room room successfully...

Welcome to Chatroom room
Enter a message to send, enter 'q' to leave the room

Kareem: Hi
Hi
```

Figure 13: Notification when user joins a room.

3 Whole Code

3.1 DB class:

```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,
            "chatroom": None
        }
        self.db.online_peers.insert_one(online_peer)

    # logs out the user
    def user_logout(self, username):
        user = self.db.online_peers.find_one({"username": username})
        if user:
            chatroom = self.db.online_peers.find_one({"username": username})["chatroom"]
            if chatroom:
                self.remove_peer_from_chatroom(username, chatroom)
```

```
            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"]

    def get_peer_chatroom(self, username):
        res = self.db.online_peers.find_one({"username": username})
        return res["chatroom"]

    def get_online_peer_list(self):
        online_peers_cursor = self.db.online_peers.find()
        online_peers_list = list(online_peers_cursor)
        return online_peers_list

    def add_online_peer_chatroom(self, username, room_name):
        res = self.db.online_peers.find_one({"username": username})
        self.db.online_peers.update_one(
            {"username": res["username"]},
            {
                "$set": {
                    "ip": res["ip"],
                    "port": res["port"],
                    "chatroom": room_name
                }
            }
        )

    def add_chat_room(self, name, host):
        chat_room = {
            "name": name,
            "peers": [host],
            "host": host
        }
        self.db.Chatrooms.insert_one(chat_room)

    def get_chat_rooms_list(self):
        return list(self.db.Chatrooms.find())

    def is_room_exist(self, room_name):
        count = self.db.Chatrooms.count_documents({'name': room_name})

        return count > 0

    def remove_peer_from_chatroom(self, username, room_name):
        # Find the chat room
        chat_room = self.db.Chatrooms.find_one({"name": room_name})

        # Check if the chat room exists
        if chat_room:
            # Remove the username from the list of peers
            chat_room["peers"].remove(username)

            if len(chat_room["peers"]) == 0:
                self.delete_chatroom(room_name)
                return
            if chat_room["host"] == username:
                chat_room["host"] = chat_room["peers"][0]

            # Update the database with the modified chat room
            self.db.Chatrooms.update_one(
                {"name": room_name},
                {
```

```
                    "$set": {
                        "peers": chat_room["peers"],
                        "host": chat_room["host"]
                    }
                }
            )

    def get_chatroom_peers(self, room_name):

        chat_room = self.db.Chatrooms.find_one({"name": room_name})

        if chat_room:
            return chat_room.get("peers", [])
        else:
            return []

    def get_chatroom_host(self, room_name):

        chat_room = self.db.Chatrooms.find_one({"name": room_name})

        if chat_room:
            return chat_room.get("host")
        else:
            return None

    def update_chatroom(self, room_name, peers, host):
        self.db.Chatrooms.update_one(
            {"name": room_name},
            {
                "$set": {
                    "peers": peers,
                    "host": host
                }
            }
        )

    # Deletes the chatroom
    def delete_chatroom(self, name):
        self.db.Chatrooms.delete_one({"name": name})

```

3.2 Peer Class:

```
import logging
import struct
import threading
import time
from socket import *
import ssl
from colorama import Fore
import utility


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

    # Peer server initialization
    def __init__(self, username, peerServerPort):
        threading.Thread.__init__(self)
        # keeps the username of the peer
        self.username = username
```

```
        # Create a UDP socket for receiving multicast data
        self.udp_socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)
        # Allow multiple sockets to use the same port
        self.udp_socket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
        # The multicast address and port
        self.multicast_group = '224.1.1.1'
        self.multicast_port = peerServerPort

    # main method of the peer server thread
    def run(self):
        # Bind the socket to the multicast port
        self.udp_socket.bind(('', self.multicast_port))

        # Join the multicast group
        self.udp_socket.setsockopt(IPPROTO_IP, IP_ADD_MEMBERSHIP, inet_aton(self.multicast_group)
                                   + inet_aton('0.0.0.0'))
        # Initial receive that you have joined
        self.udp_socket.recvfrom(1024)
        try:
            # Receive data
            while True:
                data, address = self.udp_socket.recvfrom(1024)
                data = data.decode()
                sender = data.split(':')[0]
                if sender == "System" and data[-1] == '.':
                    print(Fore.YELLOW + data)
                elif sender == "System" and data[-1] == '!':
                    print(Fore.GREEN + data)
                elif sender != self.username:
                    print(Fore.BLUE + data)
        # handles the exceptions, and logs them
        except OSError as oErr:
            logging.error("OSError: {0}".format(oErr))
        except ValueError as vErr:
            logging.error("ValueError: {0}".format(vErr))
        finally:
            # Close the socket when done
            self.udp_socket.close()


# Client side of peer
class PeerClient(threading.Thread):
    # variable initializations for the client side of the peer
    def __init__(self, port, username, peerServer, chatroom_name):
        threading.Thread.__init__(self)
        # keeps the username of the peer
        self.username = username
        # keeps the server of this client
        self.peerServer = peerServer
        # keeps the username of the peer
        self.chatroom_name = chatroom_name
        # Create a UDP socket for multicast
        self.udp_socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)
        # Allow multiple sockets to use the same port
        self.udp_socket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
        # Multicast address and port
        self.multicast_group = '224.1.1.1'
        self.multicast_port = port

    # main method of the peer client thread
    def run(self):
        # Bind the socket to the multicast address and port
        self.udp_socket.bind(('', self.multicast_port))
```

```python

        # Set the IP_MULTICAST_TTL option (time-to-live for packets)
        self.udp_socket.setsockopt(IPPROTO_IP, IP_MULTICAST_TTL, struct.pack('b', 1))

    def group_chat(self):
        message = "System: User " + self.username + " joined!"
        self.udp_socket.sendto(message.encode(), (self.multicast_group, self.multicast_port))
        print(Fore.GREEN + "Welcome to Chatroom " + self.chatroom_name)
        print("Enter a message to send, enter 'q' to leave the room\n")
        while True:
            try:
                message = input()
                if message == 'q':
                    message = "System: User " + self.username + " left."
                    self.peerServer.udp_socket.close()
                    time.sleep(0.1)
                    self.udp_socket.sendto(message.encode(), (self.multicast_group, self.multicast_port))
                    self.udp_socket.close()
                    return
                message = self.username + ": " + message
                self.udp_socket.sendto(message.encode(), (self.multicast_group, self.multicast_port))
            # handles the exceptions, and logs them
            except OSError as oErr:
                logging.error("OSError: {0}".format(oErr))
            except ValueError as vErr:
                logging.error("ValueError: {0}".format(vErr))


# main process of the peer
class peerMain:

    # peer initializations
    def __init__(self):
        # ip address of the registry
        self.registryName = input("Enter IP address of registry: ")
        # self.registryName = 'localhost'
        # port number of the registry
        self.registryPort = 15600
        # tcp socket connection to registry
        self.tcpClientSocket = socket(AF_INET, SOCK_STREAM)
        # Create an SSL context
        context = ssl.create_default_context()
        context.check_hostname = False
        context.verify_mode = ssl.CERT_NONE
        # Wrap the socket with SSL
        self.tcpClientSocket = context.wrap_socket(self.tcpClientSocket, server_hostname=self.registryName)
        # Connect to the server
        self.tcpClientSocket.connect((self.registryName, self.registryPort))
        self.connectServer()
        # initializes udp socket which is used to send hello messages
        self.udpClientSocket = socket(AF_INET, SOCK_DGRAM)
        # udp port of the registry
        self.registryUDPPort = 15500
        # login info of the peer
        self.loginCredentials = (None, None)
        # online status of the peer
        self.isOnline = False
        # server port number of this peer
        self.peerServerPort = None
        # server of this peer
        self.peerServer = None
```

```
        # client of this peer
        self.peerClient = None
        # timer initialization
        self.timer = None
        self.chatroom = None
        # User Interface
        self.state = 0
        self.states = {1: "Welcome!", 2: "Main Menu"}
        self.options = {1: {1: "Signup", 2: "Login", 3: "Exit"},
                        2: {1: "Find Online Users", 2: "Search User", 3: "Create a Chat Room",
                            4: "Find Chat Rooms",5: "Join a Chat Room", 6: "Logout"}}
        # as long as the user is not logged out, asks to select an option in the menu
        while True:
            # menu selection prompt
            if self.state == 0:
                print(Fore.MAGENTA + "P2P Chat Started")
                self.state = 1

            print(Fore.RESET + '\n' + self.states[self.state] + '\nSelect Option:')
            for option_number, option_name in self.options[self.state].items():
                print("\t" + str(option_number) + " : " + option_name)
            choice = input(Fore.MAGENTA + "\nChoice: ")
            self.handle_user_request(choice)

    def handle_user_request(self, choice):
        selection = self.options[self.state][int(choice)]

        if selection == "Signup":
            # Creates an account with the username and password entered by the user
            username = input(Fore.MAGENTA + "username: ")
            password = input(Fore.MAGENTA + "password: ")
            self.createAccount(username, password)

        elif selection == "Login" and not self.isOnline:
            # Asks for the username and the password to login
            username = input(Fore.MAGENTA + "username: ")
            password = input(Fore.MAGENTA + "password: ")
            # asks for the port number for server's tcp socket
            peer_server_port = int(input(Fore.MAGENTA + "Enter a port number for peer server: "))

            status = self.login(username, password, peer_server_port)
            # is user logs in successfully, peer variables are set
            if status == 1:
                self.isOnline = True
                self.loginCredentials = (username, password)
                self.peerServerPort = peer_server_port
                # hello message is sent to registry
                self.sendKeepAliveMessage(self.loginCredentials[0])
                self.state = 2

        elif selection == "Logout":
            # User is logged out and peer variables are set, and server and client sockets are closed
            if self.isOnline:
                self.logout(1)
                self.isOnline = False
                self.loginCredentials = (None, None)
                if self.peerServer is not None:
                    self.peerServer.isOnline = False
                    self.peerServer.udp_socket.close()
                if self.peerClient is not None:
                    self.peerClient.udp_socket.close()
                print(Fore.GREEN + "Logged out successfully")
```

```
                self.tcpClientSocket.close()
                exit(0)

        elif selection == "Exit":
            # Exits the program:
            self.logout(2)
            self.tcpClientSocket.close()
            exit(0)

        elif selection == "Find Online Users":
            # Prompt user for the users list mode and return it
            while True:
                option = input(Fore.MAGENTA + "Retrieve detailed list with users IP and Port numbers?(Choose y or n): ")
                if option == 'Y' or option == 'y':
                    self.find_online_user("DETAILED")
                    return
                elif option == 'N' or option == 'n':
                    self.find_online_user("SIMPLE")
                    return
                else:
                    print(Fore.RED + "Error: Please choose a valid option (y or n)\n")

        elif selection == "Search User":
            # If user is online, then user is asked for a username that is wanted to be searched
            if self.isOnline:
                username = input(Fore.MAGENTA + "Username to be searched: ")
                search_status = self.search_user(username)
                # if user is found its ip address is shown to user
                if search_status is not None and search_status != 0:
                    print(Fore.MAGENTA + "IP address of " + username + " is " + search_status)
                    time.sleep(1)

        elif selection == "Create a Chat Room":
            while True:
                name = input(Fore.MAGENTA + "Chat room name: ")
                if name == 'quit':
                    break
                elif self.createChatroom(name):
                    break
                else:
                    print(Fore.RED + "A Chatroom with name " + name + " already exists!")
                    print(Fore.LIGHTGREEN_EX + "Hint: enter quit to return to main menu")
                    time.sleep(1)

        elif selection == "Find Chat Rooms":
            chat_rooms = self.findChatRooms()
            if len(chat_rooms) > 0:
                number = 1
                print(Fore.RESET + "#  Name".ljust(18) + "Host".ljust(15) + "Users in Chatroom")
                for chat_room in chat_rooms:
                    chat_room = str(chat_room).strip().split()
                    users = (str(chat_room[1:-1]).replace('[', '').replace(']', '')
                             .replace('\"', '').replace("\'", '')).replace(',,', ',')
                    print(Fore.GREEN + f"{number}  {chat_room[0]:15}{chat_room[-1]:15}{users}")
                    number += 1
            else:
                print(Fore.YELLOW + "No available Chat Rooms")
                time.sleep(1)

        elif selection == "Join a Chat Room":
            while True:
```

```
                name = input(Fore.MAGENTA + "Chat room name: ")
                if name == 'quit':
                    break
                elif self.joinChatroom(name):
                    break
                else:
                    print(Fore.RED + "No chatroom with the name " + name + "!")
                    print(Fore.LIGHTGREEN_EX + "Hint: enter quit to return to main menu")
                    time.sleep(1)

        elif selection == "show room peers":
            self.getRoomPeers()
        # if choice is cancel timer for hello message is cancelled
        elif choice == "CANCEL":
            self.timer.cancel()
        else:
            print(Fore.RED + "Invalid Option Selected, please try again.\n")

    # account creation function
    def createAccount(self, username, password):
        # join message to create an account is composed and sent to registry
        # if response is "success" then informs the user for account creation
        # if response is "exist" then informs the user for account existence
        message = "REGISTER " + username + " " + utility.hash_password(password)
        response = self.send_credentials(message)
        # Process the response from the registry

        if response[2] == "<200>":
            print(Fore.GREEN + "Account created successfully.")
            time.sleep(1)
        elif response[2] == "<300>":
            print(Fore.YELLOW + "Username already exists. Choose another username or login.")
            time.sleep(1)
        elif response[2] == "<404>":
            print(Fore.RED + "Failed to create an account. Please try again.")
            time.sleep(1)

    def send_credentials(self, message):
        logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode()
        logging.info("Received from " + self.registryName + " -> " + response)
        return response.split()

    # login function
    def login(self, username, password, peerServerPort):
        # a login message is composed and sent to registry
        # an integer is returned according to each response
        message = "LOGIN " + username + " " + utility.hash_password(password) + " " + str(peerServerPort)
        response = self.send_credentials(message)
        if response[2] == "<200>":
            print(Fore.GREEN + "Logged in successfully...")
            time.sleep(1)
            return 1
        elif response[2] == "<300>":
            print(Fore.YELLOW + "Account is already online...")
            time.sleep(1)
            return 2
        elif response[2] == "<404>":
            print(Fore.RED + "Wrong password...")
            time.sleep(1)
            return 3
```

```python

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

    # function for searching an online user
    def search_user(self, username, output = True):
        # a search message is composed and sent to registry
        # custom value is returned according to each response
        # to this search message
        message = "SEARCH_USER " + username
        logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode().split()
        logging.info("Received from " + self.registryName + " -> " + " ".join(response))
        if response[2] == "<200>":
            if output:
                print(Fore.GREEN + username + " is found successfully...")
                time.sleep(1)
            return response[3]
        elif response[2] == "<300>":
            if output:
                print(Fore.YELLOW + username + " is not online...")
                time.sleep(1)
            return 0
        elif response[2] == "<404>":
            if output:
                print(Fore.RED + username + " is not found")
                time.sleep(1)
            return None

    def find_online_user(self, option):
        message = "DISCOVER_PEERS " + option + " " + self.loginCredentials[0]
        logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode().split()
        logging.info("Received from " + self.registryName + " -> " + " ".join(response))
        if response[2] == "<200>":
            response = response[3:]
            number = 1
            if option == "DETAILED":
                print(Fore.RESET + "#  Username".ljust(18) + "(IP:Port)")
                for i in range(0, len(response), 2):
                    print(Fore.GREEN + f"{number}  {response[i]:15}{response[i + 1]}")
                    number += 1
            else:
                print(Fore.RESET + "Username")
                for username in response:
                    print(Fore.GREEN + str(number) + "  " + username)
                    number += 1
            time.sleep(1)
        elif response[2] == "<404>":
            print(Fore.YELLOW + "No Online Users right now, please check back later")
```

```
            time.sleep(1)

    # function for sending hello message
    # a timer thread is used to send hello messages to udp socket of registry
    def sendKeepAliveMessage(self, username):
        message = "KEEP_ALIVE " + username
        logging.info("Send to " + self.registryName + ":" + str(self.registryUDPPort) + " -> " + message)
        self.udpClientSocket.sendto(message.encode(), (self.registryName, self.registryUDPPort))

        # Assuming you expect a response from the registry

        # Schedule the next hello message
        self.timer = threading.Timer(1, self.sendKeepAliveMessage, args=[username])
        self.timer.start()

    def connectServer(self):
        starting_message = "HELLO_P2P"
        logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + starting_message)
        self.tcpClientSocket.send(starting_message.encode())
        response = self.tcpClientSocket.recv(1024).decode().split()
        logging.info("Received from " + self.registryName + " -> " + " ".join(response))
        status_code = int(response[2])
        if status_code == "<200>":
            print(Fore.GREEN + "Connected to the registry...")

    def createChatroom(self, name):
        message = "CREATE-CHAT-ROOM " + name + " " + self.loginCredentials[0]
        logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode().split()
        logging.info("Received from " + self.registryName + " -> " + " ".join(response))
        status_code = response[2]
        if status_code == "<200>":
            self.chatroom = name
            print(Fore.GREEN + "A chatroom with name " + name + " has been created...\n")
            time.sleep(1)
            self.connect_to_chatroom(self.loginCredentials[0])
            return True
        else:
            return False

    def joinChatroom(self, name):
        message = "JOIN-CHAT-ROOM " + name + " " + self.loginCredentials[0]
        logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode().split()
        logging.info("Received from " + self.registryName + " -> " + " ".join(response))
        status_code = response[2]
        if status_code == "<200>":
            print(Fore.GREEN + "You have joined the room " + name + " successfully...\n")
            time.sleep(0.5)
            self.chatroom = name
            self.connect_to_chatroom(response[3])
            return True
        return False

    def findChatRooms(self):
        chatrooms_list = []
        message = "SHOW-ROOM-LIST"
```

```
        logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode()
        logging.info("Received from " + self.registryName + " -> " + " ".join(response))
        status_code = response.split()[2]
        if status_code == "<200>":
            # Extract the list part from the received message
            list_start_index = response.find("<200>") + len("<200>")
            chatrooms_list_str = response[list_start_index:].strip()

            # Split the string into a list
            chatrooms_list = list(chatrooms_list_str.split('.'))[:-1]
            return chatrooms_list
        return chatrooms_list

    def exitChatroom(self, username):
        message = "ROOM-EXIT " + username + " " + self.chatroom
        logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode().split()
        logging.info("Received from " + self.registryName + " -> " + " ".join(response))
        status_code = response[2]
        if status_code == "<200>":
            return True
        return False

    def connect_to_chatroom(self, host):
        search_status = self.search_user(host, False)
        # if searched user is found, then its port number is retrieved and a client thread is created
        if search_status and search_status != 0:
            search_status = search_status.split(":")
            # creates the server thread for this peer, and runs it
            self.peerServer = PeerServer(self.loginCredentials[0], int(search_status[1]))
            self.peerServer.start()
            self.peerClient = PeerClient(int(search_status[1]), self.loginCredentials[0], self.peerServer,
                                         self.chatroom)
            self.peerClient.start()
            self.peerClient.join()
            # Loop in the chatting until user exits
            self.peerClient.group_chat()
            # Exit from chatroom
            self.exitChatroom(self.loginCredentials[0])

    def getRoomPeers(self):
        room_peers = []
        message = "DISCOVER-ROOM-PEERS " + self.chatroom
        logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode()
        logging.info("Received from " + self.registryName + " -> " + " ".join(response))
        status_code = response.split()[2]
        if status_code == "<200>":
            # Assuming peers are present in the response starting from index 3
            list_start_index = response.find("<200>") + len("<200>")
            peerlist_list_str = response[list_start_index:].strip()

            # Split the string into a list
            room_peers = peerlist_list_str.split()

```

```
            # Print the chatrooms list

            print(Fore.CYAN, str(room_peers))
            return list(room_peers)
        return room_peers


# log file initialization
logging.basicConfig(filename="logs/peer.log", level=logging.INFO)
# peer is started
main = peerMain()

```

3.3 Registry Class:

```
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)
                        self.tcpClientSocket.send(response.encode())
                    # join-success is sent to peer,
                    # if an account with this username is not exist, and the account is created
                    else:
                        db.register(message[1], message[2])
                        response = "REGISTER <SUCCESS> <200>"
                        logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
                        self.tcpClientSocket.send(response.encode())
                #   LOGIN    #
                elif message[0] == "LOGIN":
                    # login-account-not-exist is sent to peer,
                    # if an account with the username does not exist
                    if not db.is_account_exist(message[1]):
                        response = "AUTH <FAILURE> <404>"
                        logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
                        self.tcpClientSocket.send(response.encode())
                    # login-online is sent to peer,
                    # if an account with the username already online
                    elif db.is_account_online(message[1]):
                        response = "AUTH <ONLINE> <300>"
                        logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
                        self.tcpClientSocket.send(response.encode())
                    # login-success is sent to peer,
                    # if an account with the username exists and not online
                    else:
                        # retrieves the account's password, and checks if the one entered by
the user is correct
                        retrieved_pass = db.get_password(message[1])
                        # if password is correct, then peer's thread is added to threads list
                        # peer is added to db with its username, port number, and ip address
                        if retrieved_pass == message[2]:
                            self.username = message[1]
                            self.lock.acquire()
                            try:
                                tcpThreads[self.username] = self
                            finally:
                                self.lock.release()

                            db.user_login(message[1], self.ip, self.port)
                            # login-success is sent to peer,
                            # and a UDP server thread is created for this peer, and thread is
started
                            # timer thread of the udp server is started
                            response = "AUTH <SUCCESS> <200>"
                            logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
                            self.tcpClientSocket.send(response.encode())
                            self.udpServer = UDPServer(self.username, self.tcpClientSocket)
                            self.udpServer.start()
                            self.udpServer.timer.start()
```

```
                        # if password not matches and then login-wrong-password response is sent
                        else:
                            response = "AUTH <FAILURE> <404>"
                            logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                            self.tcpClientSocket.send(response.encode())
                #   LOGOUT  #
                elif message[0] == "LOGOUT":
                    # 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:
                            usernames = [peer['username'] for peer in peer_list]
                            response = "PEER_LIST <SUCCESS> <200> " + ' '.join(usernames)
                            logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                            self.tcpClientSocket.send(response.encode())
                    # failure empty list
                    else:
                        response = "PEER_LIST <FAILURE> <404>"
                        logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                        self.tcpClientSocket.send(response.encode())
                elif message[0] == "CREATE-CHAT-ROOM":
                    # CREATE-exist is sent to peer,
                    # if a room with this username already exists
                    if db.is_room_exist(message[1:-1]):
                        response = "CREATION <FAILURE> <404>"
                        print("From-> " + self.ip + ":" + str(self.port) + " " + response)
                        logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                        self.tcpClientSocket.send(response.encode())
                    else:
                        db.add_chat_room(message[1], message[2])
                        db.add_online_peer_chatroom(message[2], message[1])
                        response = "CREATION <SUCCESS> <200>"
                        logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                        self.tcpClientSocket.send(response.encode())

                elif message[0] == "ROOM-EXIT":
                    if db.is_room_exist(message[2]):
                        db.remove_peer_from_chatroom(message[1], message[2])
                        response = "ROOM-EXIT-RESPONSE <SUCCESS> <200>"
                        logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)

                        self.tcpClientSocket.send(response.encode())
                    else:
                        response = "ROOM-EXIT-RESPONSE <FAILURE> <404>"
                        logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)

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

                elif message[0] == "JOIN-CHAT-ROOM":
                    # checks if an account with the username exists
                    if db.is_room_exist(message[1]):
                        # checks if the room exists
                        # and sends the related response to peer
                        peers = db.get_chatroom_peers(message[1])
                        peers.append(message[2])
                        peers = list(set(peers))
                        room_host = db.get_chatroom_host(message[1])
                        db.update_chatroom(message[1], peers, room_host)
```

```
                        db.add_online_peer_chatroom(message[2], message[1])
                        response = "JOIN <SUCCESS> <200> " + room_host

                        logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                        self.tcpClientSocket.send(response.encode())
                    else:
                        response = "JOIN <FAILURE> <404>"
                        logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                        self.tcpClientSocket.send(response.encode())

                    # enters if username does not exist


                elif message[0] == "SHOW-ROOM-LIST":
                    chat_rooms_list = db.get_chat_rooms_list()
                    if chat_rooms_list is not None:
                        response = "ROOMS-LIST <SUCCESS> <200> " + ' '.join(
                            f"{chatroom['name']} {chatroom['peers']} {chatroom['host']}."
                            for chatroom in chat_rooms_list
                        )

                        logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                        self.tcpClientSocket.send(response.encode())
                    else:
                        response = "ROOM-LIST <FAILURE> <404>"
                        logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                        self.tcpClientSocket.send(response.encode())

                elif message[0] == "DISCOVER-ROOM-PEERS":
                    peer_room_list = db.get_chatroom_peers(message[1])
                    if peer_room_list is not None:
                        response = "PEER-LIST <SUCCESS> <200> " + ' '.join(
                            f"{peer}" for peer in peer_room_list
                        )
                        logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                        self.tcpClientSocket.send(response.encode())
                    else:
                        response = "PEER-LIST <FAILURE> <404>"
                        logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                        self.tcpClientSocket.send(response.encode())

            except OSError as oErr:
                logging.error("OSError: {0}".format(oErr))
                response = "HELLO_BACK " + "FAILURE " + "404"
                logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                db.user_logout(self.username)
            except ConnectionResetError as cErr:
                logging.error("ConnectionResetError: {0}".format(cErr))
                db.user_logout(self.username)
            except IndexError as iErr:
                logging.error("IndexError: {0}".format(iErr))

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


# implementation of the udp server thread for clients
```

```
class UDPServer(threading.Thread):

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

    # if hello message is not received before timeout
    # then peer is disconnected
    def waitKeepAliveMessage(self):

        if self.username is not None:
            db.user_logout(self.username)
            if self.username in tcpThreads:
                del tcpThreads[self.username]
        self.tcpClientSocket.close()
        print(Fore.BLUE + "Removed " + self.username + " from online peers")

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


# tcp and udp server port initializations
print("Registry started...")
port = 15600
portUDP = 15500

# db initialization
db = db.DB()

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

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

print("Registry IP address: " + host)
print("Registry port number: " + str(port))

# onlinePeers list for online account
onlinePeers = {}
# accounts list for accounts
accounts = {}
# tcpThreads list for online client's thread
tcpThreads = {}

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

```
tcpSocket.listen(1000)
print("Listening for incoming connections...")

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

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

# as long as at least a socket exists to listen registry runs
while inputs:
    # monitors for the incoming connections
    readable, writable, exceptional = select.select(inputs, [], [])
    for s in readable:
        # if the message received comes to the tcp socket
        # the connection is accepted and a thread is created for it, and that thread is started
        if s is tcpSocket:
            tcpClientSocket, addr = tcpSocket.accept()
            newThread = ClientThread(addr[0], addr[1], tcpClientSocket)
            newThread.tcpClientSocket = newThread.context.wrap_socket(newThread.tcpClientSocket, server_side=True)
            newThread.start()
            response = "HELLO_BACK " + "SUCCESS " + "200 "
            logging.info("Send to " + addr[0] + ":" + str(addr[1]) + " -> " + response)
            newThread.tcpClientSocket.send(response.encode())
        # if the message received comes to the udp socket
        elif s is udpSocket:
            # received the incoming udp message and parses it
            message, clientAddress = s.recvfrom(1024)
            message = message.decode().split()
            # checks if it is a hello message
            if message[0] == "KEEP_ALIVE":
                # checks if the account that this hello message
                # is sent from is online
                if message[1] in tcpThreads:
                    # resets the timeout for that peer since the hello message is received
                    tcpThreads[message[1]].resetTimeout()
                    logging_message = "KEEP_ALIVE <SUCCESS> <200>"
                     # Send the response back to the UDP client

# registry tcp socket is closed
tcpSocket.close()

```