



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

<b>1</b>	<b>CHAT ROOMS FUNCTIONS:</b>	<b>6</b>
1.1	FIND CHAT ROOMS:	6
1.1.1	Code:	6
1.1.2	Description:	6
1.2	CREATE CHAT ROOM:	7
1.2.1	Code:	7
1.2.2	Description:	7
1.3	JOIN CHAT ROOM:	8
1.3.1	Code:	8
1.3.2	Description:	8
1.4	EXIT CHAT ROOM:	9
1.4.1	Code:	9
1.4.2	Description:	9
1.5	CONNECT TO CHAT ROOM:	9
1.5.1	Code:	9
1.5.2	Description:	10
1.6	GET ROOM PEERS:	10
1.6.1	Code:	10
1.7	GROUP CHAT :	11
1.7.1	Code:	11
1.7.2	Description:	11
<b>2</b>	<b>OUTPUTS</b>	<b>12</b>
<b>3</b>	<b>WHOLE CODE</b>	<b>19</b>

3.1	DB CLASS: .....	19
3.2	PEER CLASS:.....	21
3.3	REGISTRY CLASS:.....	30

---

*Table of Figures*

---

Figure 1:Find chatrooms (no rooms) .....	12
Figure 2:Create chatroom .....	12
Figure 3: Find Chatroom (found one) .....	13
Figure 4: Join room (invalid name and quitting to main menu) .....	13
Figure 5: Join Room successfully.....	14
Figure 6: When chatting, the System notifies the peers in the room when someone leaves.....	14
Figure 7:The room still exists for the user who left to rejoin. ....	15
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).....	16
Figure 9: When no one in the room now (as Ahmed left) the room is deleted. ....	16
Figure 10: Chatroom with multiple users .....	17
Figure 11: Trying to create a room with a name that already exists. ....	17
Figure 12: List group chat with multiple users.....	18
Figure 13: Notification when user joins a room. ....	18

### **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:

```

1.     def findChatRooms(self):
2.         chatrooms_list = []
3.         message = "SHOW-ROOM-LIST"
4.         logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " +
message)
5.         self.tcpClientSocket.send(message.encode())
6.         response = self.tcpClientSocket.recv(1024).decode()
7.         logging.info("Received from " + self.registryName + " -> " + " ".join(response))
8.         status_code = response.split()[2]
9.         if status_code == "<200>":
10.            # Extract the list part from the received message
11.            list_start_index = response.find("<200>") + len("<200>")
12.            chatrooms_list_str = response[list_start_index:].strip()
13.
14.            # Split the string into a list
15.            chatrooms_list = list(chatrooms_list_str.split(' '))[:-1]
16.            for chatroom in chatrooms_list:
17.                chatroom = chatroom.split()
18.            return chatrooms_list
19.        return chatrooms_list
20.

```

#### 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:

```
1.     def createChatroom(self, name):
2.         message = "CREATE-CHAT-ROOM " + name + " " + self.loginCredentials[0]
3.         logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " +
message)
4.         self.tcpClientSocket.send(message.encode())
5.         response = self.tcpClientSocket.recv(1024).decode().split()
6.         logging.info("Received from " + self.registryName + " -> " + " ".join(response))
7.         status_code = response[2]
8.         if status_code == "<200>":
9.             self.chatroom = name
10.            print(Fore.GREEN + "A chatroom with name " + name + " has been created...\n")
11.            time.sleep(1)
12.            self.connect_to_chatroom(self.loginCredentials[0])
13.            return True
14.        else:
15.            return False
16.
```

#### 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:

```
1. def joinChatroom(self, name):
2.     message = "JOIN-CHAT-ROOM " + name + " " + self.loginCredentials[0]
3.     logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " +
message)
4.     self.tcpClientSocket.send(message.encode())
5.     response = self.tcpClientSocket.recv(1024).decode().split()
6.     logging.info("Received from " + self.registryName + " -> " + " ".join(response))
7.     status_code = response[2]
8.     if status_code == "<200>":
9.         print(Fore.GREEN + "You have joined the room " + name + " successfully...\n")
10.        time.sleep(0.5)
11.        self.chatroom = name
12.        self.connect_to_chatroom(response[3])
13.        return True
14.    return False
15.
```

#### 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:

```
1.     def exitChatroom(self, username):
2.         message = "ROOM-EXIT " + username + " " + self.chatroom
3.         logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " +
message)
4.         self.tcpClientSocket.send(message.encode())
5.         response = self.tcpClientSocket.recv(1024).decode().split()
6.         logging.info("Received from " + self.registryName + " -> " + " ".join(response))
7.         status_code = response[2]
8.         if status_code == "<200>":
9.             return True
10.        return False
11.
```

#### 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:

```
1.     def connect_to_chatroom(self, host):
4.         search_status = self.search_user(host, False)
5.         # if searched user is found, then its port number is retrieved and a client thread is
created
6.         if search_status and search_status != 0:
7.             search_status = search_status.split(":")
8.             # creates the server thread for this peer, and runs it
9.             self.peerServer = PeerServer(self.loginCredentials[0], int(search_status[1]))
10.            self.peerServer.start()
11.            self.peerClient = PeerClient(int(search_status[1]), self.loginCredentials[0],
self.peerServer,
12.                                       self.chatroom)
13.            self.peerClient.start()
14.            self.peerClient.join()
15.            # Loop in the chatting until user exits
16.            self.peerClient.group_chat()
17.            # Exit from chatroom
18.            self.exitChatroom(self.loginCredentials[0])
19.
```

### 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:

```
1. def getRoomPeers(self):
2.     room_peers = []
3.     message = "DISCOVER-ROOM-PEERS " + self.chatroom
4.     logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " +
message)
5.     self.tcpClientSocket.send(message.encode())
6.     response = self.tcpClientSocket.recv(1024).decode()
7.     logging.info("Received from " + self.registryName + " -> " + " ".join(response))
8.     status_code = response.split()[2]
9.     if status_code == "<200>":
10.         # Assuming peers are present in the response starting from index 3
11.         list_start_index = response.find("<200>") + len("<200>")
12.         peerlist_list_str = response[list_start_index:].strip()
13.
14.         # Split the string into a list
15.         room_peers = peerlist_list_str.split()
16.
17.         # Print the chatrooms list
18.
19.         print(Fore.CYAN, str(room_peers))
20.         return list(room_peers)
21.     return room_peers
22.
```

## PEER CLIENT CLASS:

## 1.7 Group Chat :

## 1.7.1 Code:

```
1.     def group_chat(self):
2.         print(Fore.GREEN + "Welcome to Chatroom " + self.chatroom_name)
3.         print("Enter a message to send, enter 'q' to leave the room\n")
4.         while True:
5.             try:
6.                 message = input()
7.                 if message == 'q':
8.                     message = "System: User " + self.username + " left."
9.                     self.peerServer.udp_socket.close()
10.                    time.sleep(0.1)
11.                    self.udp_socket.sendto(message.encode(), (self.multicast_group,
self.multicast_port))
12.                    self.udp_socket.close()
13.                    return
14.                    message = self.username + ": " + message
15.                    self.udp_socket.sendto(message.encode(), (self.multicast_group,
self.multicast_port))
16.                    # handles the exceptions, and logs them
17.                    except OSError as oErr:
18.                        logging.error("OSError: {}".format(oErr))
19.                    except ValueError as vErr:
20.                        logging.error("ValueError: {}".format(vErr))
21.
```

## 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

## P2P chatting Application

```
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)

## P2P chatting Application

```
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.

## P2P chatting Application

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.

## P2P chatting Application

```
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

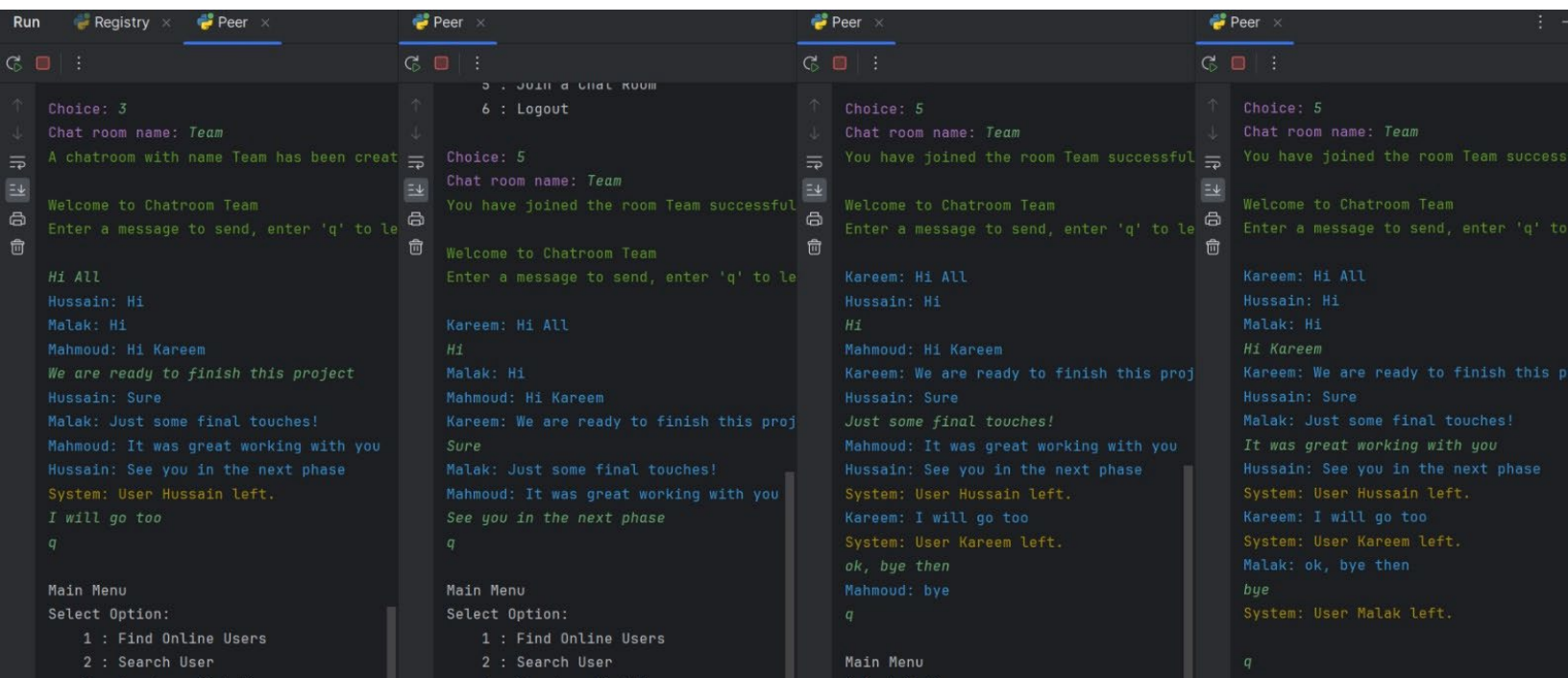
Main Menu
Select Option:
    1 : Find Online Users
    2 : Search User
    3 : Create a Chat Room
    4 : Find Chat Rooms
    5 : Join a Chat Room
    6 : Logout

Choice:
```

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



## P2P chatting Application

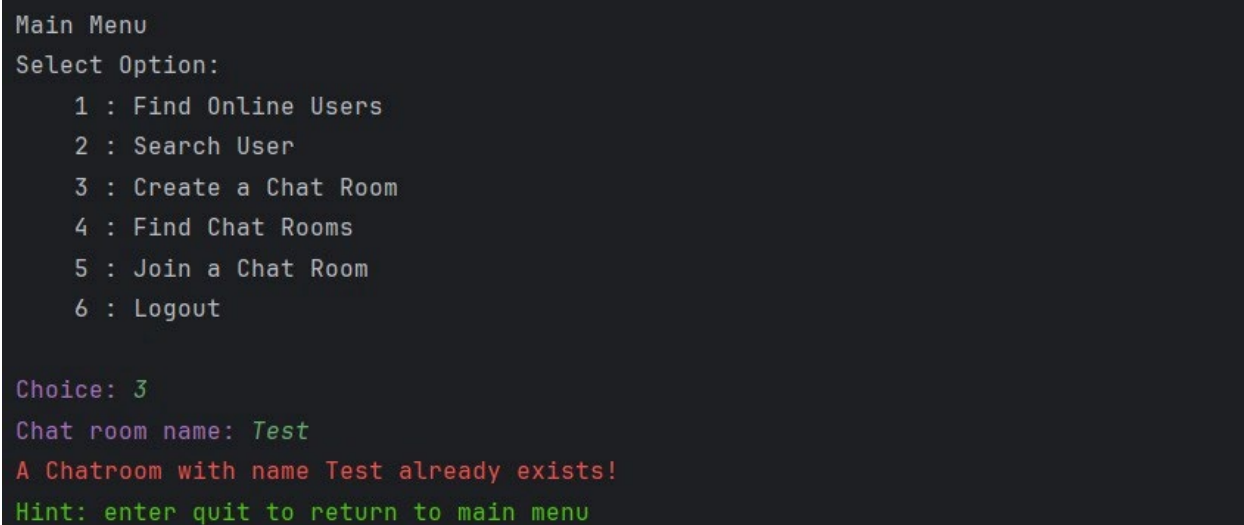


```
Run Registry x Peer x Peer x Peer x
Choice: 3
Chat room name: Team
A chatroom with name Team has been created
Welcome to Chatroom Team
Enter a message to send, enter 'q' to leave
Hi All
Hussain: Hi
Malak: Hi
Mahmoud: Hi Kareem
We are ready to finish this project
Hussain: Sure
Malak: Just some final touches!
Mahmoud: It was great working with you
Hussain: See you in the next phase
System: User Hussain left.
I will go too
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: 5
Chat room name: Team
You have joined the room Team successfully
Welcome to Chatroom Team
Enter a message to send, enter 'q' to leave
Kareem: Hi All
Hussain: Hi
Hi
Mahmoud: Hi Kareem
Kareem: We are ready to finish this project
Hussain: Sure
Just some final touches!
Mahmoud: It was great working with you
Hussain: See you in the next phase
System: User Hussain left.
Kareem: I will go too
System: User Kareem left.
ok, bye then
Mahmoud: bye
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: 5
Chat room name: Team
You have joined the room Team successfully
Welcome to Chatroom Team
Enter a message to send, enter 'q' to leave
Kareem: Hi All
Hussain: Hi
Malak: Hi
Hi Kareem
Kareem: We are ready to finish this project
Hussain: Sure
Malak: Just some final touches!
It was great working with you
Hussain: See you in the next phase
System: User Hussain left.
Kareem: I will go too
System: User Kareem left.
Malak: ok, bye then
bye
System: User Malak 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
```

Figure 10: Chatroom with multiple users

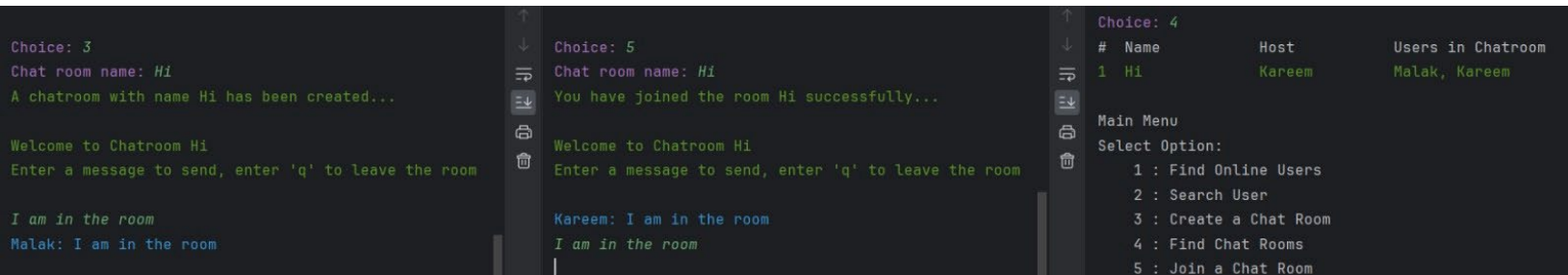


```
Main Menu
Select Option:
1 : Find Online Users
2 : Search User
3 : Create a Chat Room
4 : Find Chat Rooms
5 : Join a Chat Room
6 : Logout

Choice: 3
Chat room name: Test
A Chatroom with name Test already exists!
Hint: enter quit to return to main menu
```

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

## P2P chatting Application



```
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...

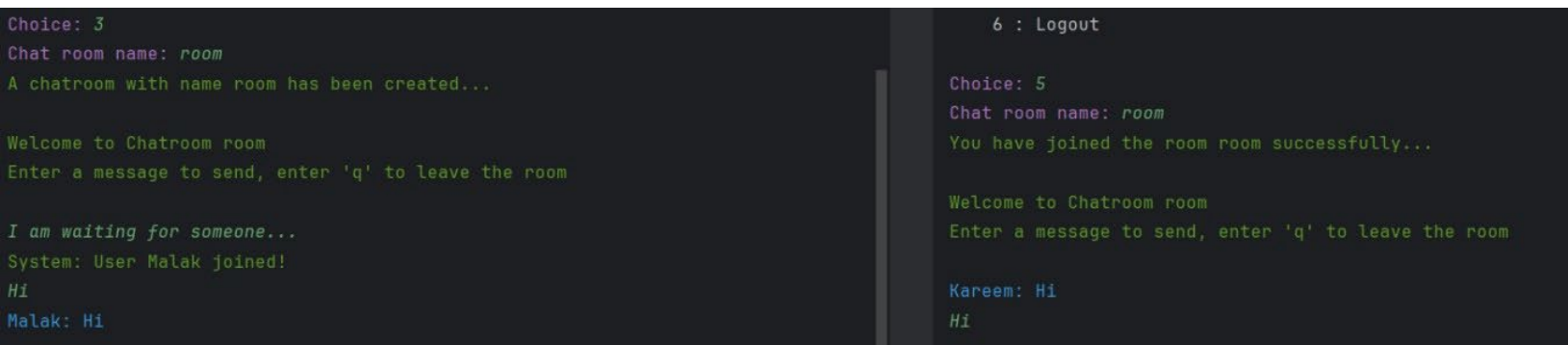
Welcome to Chatroom Hi
Enter a message to send, enter 'q' to leave the room

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:

```
1. from pymongo import MongoClient
2.
3.
4. # Includes database operations
5. class DB:
6.
7.     # db initializations
8.     def __init__(self):
9.         self.client = MongoClient('mongodb://localhost:27017/')
10.        self.db = self.client['p2p-chat']
11.
12.    # checks if an account with the username exists
13.    def is_account_exist(self, username):
14.        cursor = self.db.accounts.find({'username': username})
15.        doc_count = 0
16.
17.        for document in cursor:
18.            doc_count += 1
19.
20.        if doc_count > 0:
21.            return True
22.        else:
23.            return False
24.
25.    # registers a user
26.    def register(self, username, password):
27.        account = {
28.            "username": username,
29.            "password": password
30.        }
31.        self.db.accounts.insert_one(account)
32.
33.    # retrieves the password for a given username
34.    def get_password(self, username):
35.        return self.db.accounts.find_one({"username": username})["password"]
36.
37.    # checks if an account with the username online
38.    def is_account_online(self, username):
39.        count = self.db.online_peers.count_documents({'username': username})
40.        return count > 0
41.
42.    # logs in the user
43.    def user_login(self, username, ip, port):
44.        online_peer = {
45.            "username": username,
46.            "ip": ip,
47.            "port": port,
48.            "chatroom": None
49.        }
50.        self.db.online_peers.insert_one(online_peer)
51.
52.    # logs out the user
53.    def user_logout(self, username):
54.        user = self.db.online_peers.find_one({"username": username})
55.        if user:
56.            chatroom = self.db.online_peers.find_one({"username": username})["chatroom"]
57.            if chatroom:
58.                self.remove_peer_from_chatroom(username, chatroom)
```

## P2P chatting Application

```
59.         self.db.online_peers.delete_one({"username": username})
60.
61.     # retrieves the ip address and the port number of the username
62.     def get_peer_ip_port(self, username):
63.         res = self.db.online_peers.find_one({"username": username})
64.         return res["ip"], res["port"]
65.
66.     def get_peer_chatroom(self, username):
67.         res = self.db.online_peers.find_one({"username": username})
68.         return res["chatroom"]
69.
70.     def get_online_peer_list(self):
71.         online_peers_cursor = self.db.online_peers.find()
72.         online_peers_list = list(online_peers_cursor)
73.         return online_peers_list
74.
75.     def add_online_peer_chatroom(self, username, room_name):
76.         res = self.db.online_peers.find_one({"username": username})
77.         self.db.online_peers.update_one(
78.             {"username": res["username"]},
79.             {
80.                 "$set": {
81.                     "ip": res["ip"],
82.                     "port": res["port"],
83.                     "chatroom": room_name
84.                 }
85.             }
86.         )
87.
88.     def add_chat_room(self, name, host):
89.         chat_room = {
90.             "name": name,
91.             "peers": [host],
92.             "host": host
93.         }
94.         self.db.Chatrooms.insert_one(chat_room)
95.
96.     def get_chat_rooms_list(self):
97.         return list(self.db.Chatrooms.find())
98.
99.     def is_room_exist(self, room_name):
100.        count = self.db.Chatrooms.count_documents({'name': room_name})
101.
102.        return count > 0
103.
104.     def remove_peer_from_chatroom(self, username, room_name):
105.         # Find the chat room
106.         chat_room = self.db.Chatrooms.find_one({"name": room_name})
107.
108.         # Check if the chat room exists
109.         if chat_room:
110.             # Remove the username from the list of peers
111.             chat_room["peers"].remove(username)
112.
113.             if len(chat_room["peers"]) == 0:
114.                 self.delete_chatroom(room_name)
115.                 return
116.             if chat_room["host"] == username:
117.                 chat_room["host"] = chat_room["peers"][0]
118.
119.             # Update the database with the modified chat room
120.             self.db.Chatrooms.update_one(
121.                 {"name": room_name},
122.                 {
```

## P2P chatting Application

```
123.         "$set": {
124.             "peers": chat_room["peers"],
125.             "host": chat_room["host"]
126.         }
127.     }
128. )
129.
130. def get_chatroom_peers(self, room_name):
131.
132.     chat_room = self.db.Chatrooms.find_one({"name": room_name})
133.
134.     if chat_room:
135.         return chat_room.get("peers", [])
136.     else:
137.         return []
138.
139. def get_chatroom_host(self, room_name):
140.
141.     chat_room = self.db.Chatrooms.find_one({"name": room_name})
142.
143.     if chat_room:
144.         return chat_room.get("host")
145.     else:
146.         return None
147.
148. def update_chatroom(self, room_name, peers, host):
149.     self.db.Chatrooms.update_one(
150.         {"name": room_name},
151.         {
152.             "$set": {
153.                 "peers": peers,
154.                 "host": host
155.             }
156.         }
157.     )
158.
159. # Deletes the chatroom
160. def delete_chatroom(self, name):
161.     self.db.Chatrooms.delete_one({"name": name})
162.
```

### 3.2 Peer Class:

```
1. import logging
2. import struct
3. import threading
4. import time
5. from socket import *
6. import ssl
7. from colorama import Fore
8. import utility
9.
10.
11. # Server side of peer
12. class PeerServer(threading.Thread):
13.
14.     # Peer server initialization
15.     def __init__(self, username, peerServerPort):
16.         threading.Thread.__init__(self)
17.         # keeps the username of the peer
18.         self.username = username
```

## P2P chatting Application

```
19.         # Create a UDP socket for receiving multicast data
20.         self.udp_socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)
21.         # Allow multiple sockets to use the same port
22.         self.udp_socket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
23.         # The multicast address and port
24.         self.multicast_group = '224.1.1.1'
25.         self.multicast_port = peerServerPort
26.
27.     # main method of the peer server thread
28.     def run(self):
29.         # Bind the socket to the multicast port
30.         self.udp_socket.bind(('', self.multicast_port))
31.
32.         # Join the multicast group
33.         self.udp_socket.setsockopt(IPPROTO_IP, IP_ADD_MEMBERSHIP,
inet_aton(self.multicast_group)
34.                                     + inet_aton('0.0.0.0'))
35.         # Initial receive that you have joined
36.         self.udp_socket.recvfrom(1024)
37.         try:
38.             # Receive data
39.             while True:
40.                 data, address = self.udp_socket.recvfrom(1024)
41.                 data = data.decode()
42.                 sender = data.split(':')[0]
43.                 if sender == "System" and data[-1] == '.':
44.                     print(Fore.YELLOW + data)
45.                 elif sender == "System" and data[-1] == '!':
46.                     print(Fore.GREEN + data)
47.                 elif sender != self.username:
48.                     print(Fore.BLUE + data)
49.             # handles the exceptions, and logs them
50.             except OSError as oErr:
51.                 logging.error("OSError: {}".format(oErr))
52.             except ValueError as vErr:
53.                 logging.error("ValueError: {}".format(vErr))
54.             finally:
55.                 # Close the socket when done
56.                 self.udp_socket.close()
57.
58.
59. # Client side of peer
60. class PeerClient(threading.Thread):
61.     # variable initializations for the client side of the peer
62.     def __init__(self, port, username, peerServer, chatroom_name):
63.         threading.Thread.__init__(self)
64.         # keeps the username of the peer
65.         self.username = username
66.         # keeps the server of this client
67.         self.peerServer = peerServer
68.         # keeps the username of the peer
69.         self.chatroom_name = chatroom_name
70.         # Create a UDP socket for multicast
71.         self.udp_socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)
72.         # Allow multiple sockets to use the same port
73.         self.udp_socket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
74.         # Multicast address and port
75.         self.multicast_group = '224.1.1.1'
76.         self.multicast_port = port
77.
78.     # main method of the peer client thread
79.     def run(self):
80.         # Bind the socket to the multicast address and port
81.         self.udp_socket.bind(('', self.multicast_port))
```

## P2P chatting Application

```
82.
83.     # Set the IP_MULTICAST_TTL option (time-to-live for packets)
84.     self.udp_socket.setsockopt(IPPROTO_IP, IP_MULTICAST_TTL, struct.pack('b', 1))
85.
86.     def group_chat(self):
87.         message = "System: User " + self.username + " joined!"
88.         self.udp_socket.sendto(message.encode(), (self.multicast_group, self.multicast_port))
89.         print(Fore.GREEN + "Welcome to Chatroom " + self.chatroom_name)
90.         print("Enter a message to send, enter 'q' to leave the room\n")
91.         while True:
92.             try:
93.                 message = input()
94.                 if message == 'q':
95.                     message = "System: User " + self.username + " left."
96.                     self.peerServer.udp_socket.close()
97.                     time.sleep(0.1)
98.                     self.udp_socket.sendto(message.encode(), (self.multicast_group,
self.multicast_port))
99.                     self.udp_socket.close()
100.                    return
101.                    message = self.username + ": " + message
102.                    self.udp_socket.sendto(message.encode(), (self.multicast_group,
self.multicast_port))
103.                    # handles the exceptions, and logs them
104.                    except OSError as oErr:
105.                        logging.error("OSError: {}".format(oErr))
106.                    except ValueError as vErr:
107.                        logging.error("ValueError: {}".format(vErr))
108.
109.
110. # main process of the peer
111. class peerMain:
112.
113.     # peer initializations
114.     def __init__(self):
115.         # ip address of the registry
116.         self.registryName = input("Enter IP address of registry: ")
117.         # self.registryName = 'localhost'
118.         # port number of the registry
119.         self.registryPort = 15600
120.         # tcp socket connection to registry
121.         self.tcpClientSocket = socket(AF_INET, SOCK_STREAM)
122.         # Create an SSL context
123.         context = ssl.create_default_context()
124.         context.check_hostname = False
125.         context.verify_mode = ssl.CERT_NONE
126.         # Wrap the socket with SSL
127.         self.tcpClientSocket = context.wrap_socket(self.tcpClientSocket,
server_hostname=self.registryName)
128.         # Connect to the server
129.         self.tcpClientSocket.connect((self.registryName, self.registryPort))
130.         self.connectServer()
131.         # initializes udp socket which is used to send hello messages
132.         self.udpClientSocket = socket(AF_INET, SOCK_DGRAM)
133.         # udp port of the registry
134.         self.registryUDPPort = 15500
135.         # login info of the peer
136.         self.loginCredentials = (None, None)
137.         # online status of the peer
138.         self.isOnline = False
139.         # server port number of this peer
140.         self.peerServerPort = None
141.         # server of this peer
142.         self.peerServer = None
```

## P2P chatting Application

```
143.         # client of this peer
144.         self.peerClient = None
145.         # timer initialization
146.         self.timer = None
147.         self.chatroom = None
148.         # User Interface
149.         self.state = 0
150.         self.states = {1: "Welcome!", 2: "Main Menu"}
151.         self.options = {1: {1: "Signup", 2: "Login", 3: "Exit"},
152.                          2: {1: "Find Online Users", 2: "Search User", 3: "Create a Chat Room",
153.                              4: "Find Chat Rooms", 5: "Join a Chat Room", 6: "Logout"}}
154.         # as long as the user is not logged out, asks to select an option in the menu
155.         while True:
156.             # menu selection prompt
157.             if self.state == 0:
158.                 print(Fore.MAGENTA + "P2P Chat Started")
159.                 self.state = 1
160.
161.                 print(Fore.RESET + '\n' + self.states[self.state] + '\nSelect Option:')
162.                 for option_number, option_name in self.options[self.state].items():
163.                     print("\t" + str(option_number) + " : " + option_name)
164.                 choice = input(Fore.MAGENTA + "\nChoice: ")
165.                 self.handle_user_request(choice)
166.
167.         def handle_user_request(self, choice):
168.             selection = self.options[self.state][int(choice)]
169.
170.             if selection == "Signup":
171.                 # Creates an account with the username and password entered by the user
172.                 username = input(Fore.MAGENTA + "username: ")
173.                 password = input(Fore.MAGENTA + "password: ")
174.                 self.createAccount(username, password)
175.
176.             elif selection == "Login" and not self.isOnline:
177.                 # Asks for the username and the password to login
178.                 username = input(Fore.MAGENTA + "username: ")
179.                 password = input(Fore.MAGENTA + "password: ")
180.                 # asks for the port number for server's tcp socket
181.                 peer_server_port = int(input(Fore.MAGENTA + "Enter a port number for peer server:
182. "))
183.                 status = self.login(username, password, peer_server_port)
184.                 # is user logs in successfully, peer variables are set
185.                 if status == 1:
186.                     self.isOnline = True
187.                     self.loginCredentials = (username, password)
188.                     self.peerServerPort = peer_server_port
189.                     # hello message is sent to registry
190.                     self.sendKeepAliveMessage(self.loginCredentials[0])
191.                     self.state = 2
192.
193.             elif selection == "Logout":
194.                 # User is logged out and peer variables are set, and server and client sockets are
195.                 closed
196.                 if self.isOnline:
197.                     self.logout(1)
198.                     self.isOnline = False
199.                     self.loginCredentials = (None, None)
200.                     if self.peerServer is not None:
201.                         self.peerServer.isOnline = False
202.                         self.peerServer.udp_socket.close()
203.                     if self.peerClient is not None:
204.                         self.peerClient.udp_socket.close()
205.                     print(Fore.GREEN + "Logged out successfully")
```



## P2P chatting Application

```
205.         self.tcpClientSocket.close()
206.         exit(0)
207.
208.     elif selection == "Exit":
209.         # Exits the program:
210.         self.logout(2)
211.         self.tcpClientSocket.close()
212.         exit(0)
213.
214.     elif selection == "Find Online Users":
215.         # Prompt user for the users list mode and return it
216.         while True:
217.             option = input(Fore.MAGENTA + "Retrieve detailed list with users IP and Port
218. numbers?(Choose y or n): ")
219.             if option == 'Y' or option == 'y':
220.                 self.find_online_user("DETAILED")
221.                 return
222.             elif option == 'N' or option == 'n':
223.                 self.find_online_user("SIMPLE")
224.                 return
225.             else:
226.                 print(Fore.RED + "Error: Please choose a valid option (y or n)\n")
227.
228.     elif selection == "Search User":
229.         # If user is online, then user is asked for a username that is wanted to be
230.         # searched
231.         if self.isOnline:
232.             username = input(Fore.MAGENTA + "Username to be searched: ")
233.             search_status = self.search_user(username)
234.             # if user is found its ip address is shown to user
235.             if search_status is not None and search_status != 0:
236.                 print(Fore.MAGENTA + "IP address of " + username + " is " + search_status)
237.                 time.sleep(1)
238.
239.     elif selection == "Create a Chat Room":
240.         while True:
241.             name = input(Fore.MAGENTA + "Chat room name: ")
242.             if name == 'quit':
243.                 break
244.             elif self.createChatroom(name):
245.                 break
246.             else:
247.                 print(Fore.RED + "A Chatroom with name " + name + " already exists!")
248.                 print(Fore.LIGHTGREEN_EX + "Hint: enter quit to return to main menu")
249.                 time.sleep(1)
250.
251.     elif selection == "Find Chat Rooms":
252.         chat_rooms = self.findChatRooms()
253.         if len(chat_rooms) > 0:
254.             number = 1
255.             print(Fore.RESET + "#  Name".ljust(18) + "Host".ljust(15) + "Users in
256. Chatroom")
257.             for chat_room in chat_rooms:
258.                 chat_room = str(chat_room).strip().split()
259.                 users = (str(chat_room[1:-1]).replace('[', '').replace(']', ''))
260.                 .replace('\n', '').replace("\'", '').replace(',', ', ').replace(' ', ' ')
261.                 print(Fore.GREEN + f"{number}  {chat_room[0]:15}{chat_room[-1]:15}{users}")
262.                 number += 1
263.             else:
264.                 print(Fore.YELLOW + "No available Chat Rooms")
265.                 time.sleep(1)
266.
267.     elif selection == "Join a Chat Room":
268.         while True:
```

## P2P chatting Application

```
266.         name = input(Fore.MAGENTA + "Chat room name: ")
267.         if name == 'quit':
268.             break
269.         elif self.joinChatroom(name):
270.             break
271.         else:
272.             print(Fore.RED + "No chatroom with the name " + name + "!")
273.             print(Fore.LIGHTGREEN_EX + "Hint: enter quit to return to main menu")
274.             time.sleep(1)
275.
276.         elif selection == "show room peers":
277.             self.getRoomPeers()
278.         # if choice is cancel timer for hello message is cancelled
279.         elif choice == "CANCEL":
280.             self.timer.cancel()
281.         else:
282.             print(Fore.RED + "Invalid Option Selected, please try again.\n")
283.
284.     # account creation function
285.     def createAccount(self, username, password):
286.         # join message to create an account is composed and sent to registry
287.         # if response is "success" then informs the user for account creation
288.         # if response is "exist" then informs the user for account existence
289.         message = "REGISTER " + username + " " + utility.hash_password(password)
290.         response = self.send_credentials(message)
291.         # Process the response from the registry
292.
293.         if response[2] == "<200>":
294.             print(Fore.GREEN + "Account created successfully.")
295.             time.sleep(1)
296.         elif response[2] == "<300>":
297.             print(Fore.YELLOW + "Username already exists. Choose another username or login.")
298.             time.sleep(1)
299.         elif response[2] == "<404>":
300.             print(Fore.RED + "Failed to create an account. Please try again.")
301.             time.sleep(1)
302.
303.     def send_credentials(self, message):
304.         logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " +
message)
305.         self.tcpClientSocket.send(message.encode())
306.         response = self.tcpClientSocket.recv(1024).decode()
307.         logging.info("Received from " + self.registryName + " -> " + response)
308.         return response.split()
309.
310.     # login function
311.     def login(self, username, password, peerServerPort):
312.         # a login message is composed and sent to registry
313.         # an integer is returned according to each response
314.         message = "LOGIN " + username + " " + utility.hash_password(password) + " " +
str(peerServerPort)
315.         response = self.send_credentials(message)
316.         if response[2] == "<200>":
317.             print(Fore.GREEN + "Logged in successfully...")
318.             time.sleep(1)
319.             return 1
320.         elif response[2] == "<300>":
321.             print(Fore.YELLOW + "Account is already online...")
322.             time.sleep(1)
323.             return 2
324.         elif response[2] == "<404>":
325.             print(Fore.RED + "Wrong password...")
326.             time.sleep(1)
327.             return 3
```

## P2P chatting Application

```
328.
329.     # logout function
330.     def logout(self, option):
331.         # a logout message is composed and sent to registry
332.         # timer is stopped
333.         if option == 1:
334.             message = "LOGOUT " + self.loginCredentials[0]
335.             self.timer.cancel()
336.         else:
337.             message = "LOGOUT"
338.             logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " +
message)
339.             self.tcpClientSocket.send(message.encode())
340.
341.     # function for searching an online user
342.     def search_user(self, username, output = True):
343.         # a search message is composed and sent to registry
344.         # custom value is returned according to each response
345.         # to this search message
346.         message = "SEARCH_USER " + username
347.         logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " +
message)
348.         self.tcpClientSocket.send(message.encode())
349.         response = self.tcpClientSocket.recv(1024).decode().split()
350.         logging.info("Received from " + self.registryName + " -> " + " ".join(response))
351.         if response[2] == "<200>":
352.             if output:
353.                 print(Fore.GREEN + username + " is found successfully...")
354.                 time.sleep(1)
355.             return response[3]
356.         elif response[2] == "<300>":
357.             if output:
358.                 print(Fore.YELLOW + username + " is not online...")
359.                 time.sleep(1)
360.             return 0
361.         elif response[2] == "<404>":
362.             if output:
363.                 print(Fore.RED + username + " is not found")
364.                 time.sleep(1)
365.             return None
366.
367.     def find_online_user(self, option):
368.         message = "DISCOVER_PEERS " + option + " " + self.loginCredentials[0]
369.         logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " +
message)
370.         self.tcpClientSocket.send(message.encode())
371.         response = self.tcpClientSocket.recv(1024).decode().split()
372.         logging.info("Received from " + self.registryName + " -> " + " ".join(response))
373.         if response[2] == "<200>":
374.             response = response[3:]
375.             number = 1
376.             if option == "DETAILED":
377.                 print(Fore.RESET + "# Username".ljust(18) + "(IP:Port)")
378.                 for i in range(0, len(response), 2):
379.                     print(Fore.GREEN + f"{number} {response[i]:15}{response[i + 1]}")
380.                     number += 1
381.             else:
382.                 print(Fore.RESET + "Username")
383.                 for username in response:
384.                     print(Fore.GREEN + str(number) + " " + username)
385.                     number += 1
386.                 time.sleep(1)
387.         elif response[2] == "<404>":
388.             print(Fore.YELLOW + "No Online Users right now, please check back later")
```

## P2P chatting Application

```
389.         time.sleep(1)
390.
391.     # function for sending hello message
392.     # a timer thread is used to send hello messages to udp socket of registry
393.     def sendKeepAliveMessage(self, username):
394.         message = "KEEP_ALIVE " + username
395.         logging.info("Send to " + self.registryName + ":" + str(self.registryUDPPort) + " -> "
+ message)
396.         self.udpClientSocket.sendto(message.encode(), (self.registryName,
self.registryUDPPort))
397.
398.         # Assuming you expect a response from the registry
399.
400.         # Schedule the next hello message
401.         self.timer = threading.Timer(1, self.sendKeepAliveMessage, args=[username])
402.         self.timer.start()
403.
404.     def connectServer(self):
405.         starting_message = "HELLO_P2P"
406.         logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " +
starting_message)
407.         self.tcpClientSocket.send(starting_message.encode())
408.         response = self.tcpClientSocket.recv(1024).decode().split()
409.         logging.info("Received from " + self.registryName + " -> " + " ".join(response))
410.         status_code = int(response[2])
411.         if status_code == "<200>":
412.             print(Fore.GREEN + "Connected to the registry...")
413.
414.     def createChatroom(self, name):
415.         message = "CREATE-CHAT-ROOM " + name + " " + self.loginCredentials[0]
416.         logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " +
message)
417.         self.tcpClientSocket.send(message.encode())
418.         response = self.tcpClientSocket.recv(1024).decode().split()
419.         logging.info("Received from " + self.registryName + " -> " + " ".join(response))
420.         status_code = response[2]
421.         if status_code == "<200>":
422.             self.chatroom = name
423.             print(Fore.GREEN + "A chatroom with name " + name + " has been created...\n")
424.             time.sleep(1)
425.             self.connect_to_chatroom(self.loginCredentials[0])
426.             return True
427.         else:
428.             return False
429.
430.     def joinChatroom(self, name):
431.         message = "JOIN-CHAT-ROOM " + name + " " + self.loginCredentials[0]
432.         logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " +
message)
433.         self.tcpClientSocket.send(message.encode())
434.         response = self.tcpClientSocket.recv(1024).decode().split()
435.         logging.info("Received from " + self.registryName + " -> " + " ".join(response))
436.         status_code = response[2]
437.         if status_code == "<200>":
438.             print(Fore.GREEN + "You have joined the room " + name + " successfully...\n")
439.             time.sleep(0.5)
440.             self.chatroom = name
441.             self.connect_to_chatroom(response[3])
442.             return True
443.         return False
444.
445.     def findChatRooms(self):
446.         chatrooms_list = []
447.         message = "SHOW-ROOM-LIST"
```

## P2P chatting Application

```
448.         logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " +
message)
449.         self.tcpClientSocket.send(message.encode())
450.         response = self.tcpClientSocket.recv(1024).decode()
451.         logging.info("Received from " + self.registryName + " -> " + " ".join(response))
452.         status_code = response.split()[2]
453.         if status_code == "<200>":
454.             # Extract the list part from the received message
455.             list_start_index = response.find("<200>") + len("<200>")
456.             chatrooms_list_str = response[list_start_index:].strip()
457.
458.             # Split the string into a list
459.             chatrooms_list = list(chatrooms_list_str.split(' '))[:-1]
460.             return chatrooms_list
461.         return chatrooms_list
462.
463.     def exitChatroom(self, username):
464.         message = "ROOM-EXIT " + username + " " + self.chatroom
465.         logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " +
message)
466.         self.tcpClientSocket.send(message.encode())
467.         response = self.tcpClientSocket.recv(1024).decode().split()
468.         logging.info("Received from " + self.registryName + " -> " + " ".join(response))
469.         status_code = response[2]
470.         if status_code == "<200>":
471.             return True
472.         return False
473.
474.     def connect_to_chatroom(self, host):
475.         search_status = self.search_user(host, False)
476.         # if searched user is found, then its port number is retrieved and a client thread is
created
477.         if search_status and search_status != 0:
478.             search_status = search_status.split(":")
479.             # creates the server thread for this peer, and runs it
480.             self.peerServer = PeerServer(self.loginCredentials[0], int(search_status[1]))
481.             self.peerServer.start()
482.             self.peerClient = PeerClient(int(search_status[1]), self.loginCredentials[0],
self.peerServer,
483.                                         self.chatroom)
484.             self.peerClient.start()
485.             self.peerClient.join()
486.             # Loop in the chatting until user exits
487.             self.peerClient.group_chat()
488.             # Exit from chatroom
489.             self.exitChatroom(self.loginCredentials[0])
490.
491.     def getRoomPeers(self):
492.         room_peers = []
493.         message = "DISCOVER-ROOM-PEERS " + self.chatroom
494.         logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " +
message)
495.         self.tcpClientSocket.send(message.encode())
496.         response = self.tcpClientSocket.recv(1024).decode()
497.         logging.info("Received from " + self.registryName + " -> " + " ".join(response))
498.         status_code = response.split()[2]
499.         if status_code == "<200>":
500.             # Assuming peers are present in the response starting from index 3
501.             list_start_index = response.find("<200>") + len("<200>")
502.             peerlist_list_str = response[list_start_index:].strip()
503.
504.             # Split the string into a list
505.             room_peers = peerlist_list_str.split()
506.
```

## P2P chatting Application

```
507.         # Print the chatrooms list
508.
509.         print(Fore.CYAN, str(room_peers))
510.         return list(room_peers)
511.     return room_peers
512.
513.
514. # log file initialization
515. logging.basicConfig(filename="logs/peer.log", level=logging.INFO)
516. # peer is started
517. main = peerMain()
518.
```

### 3.3 Registry Class:

```
1. from socket import *
2. import threading
3. import select
4. import logging
5.
6. from colorama import Fore
7.
8. import db
9. import ssl
10.
11.
12. # This class is used to process the peer messages sent to registry
13. # for each peer connected to registry, a new client thread is created
14. class ClientThread(threading.Thread):
15.     # initializations for client thread
16.     def __init__(self, ip, port, tcpClientSocket):
17.         threading.Thread.__init__(self)
18.         # ip of the connected peer
19.
20.         self.ip = ip
21.         # port number of the connected peer
22.         self.port = port
23.         # socket of the peer
24.         self.tcpClientSocket = tcpClientSocket
25.         # Create SSL context
26.         self.context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
27.         self.context.load_cert_chain(certfile="security/server.crt",
keyfile="security/server.key")
28.         # username, online status and udp server initializations
29.         self.username = None
30.         self.isOnline = True
31.         self.udpServer = None
32.         print("New thread started for " + ip + ":" + str(port))
33.
34.     # main of the thread
35.     def run(self):
36.         # locks for thread which will be used for thread synchronization
37.         self.lock = threading.Lock()
38.         print(Fore.BLUE + "Connection from: " + self.ip + ":" + str(port))
39.         print(Fore.BLUE + "IP Connected: " + self.ip)
40.
41.         while True:
42.             try:
43.                 # waits for incoming messages from peers
44.                 message = self.tcpClientSocket.recv(1024).decode().split()
```

## P2P chatting Application

```
45.         logging.info("Received from " + self.ip + ":" + str(self.port) + " -> " + "
".join(message))
46.         # JOIN #
47.         if message[0] == "REGISTER":
48.             # join-exist is sent to peer,
49.             # if an account with this username already exists
50.             if db.is_account_exist(message[1]):
51.                 response = "REGISTER <EXIST> <300>"
52.                 print("From-> " + self.ip + ":" + str(self.port) + " " + response)
53.                 logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
54.                 self.tcpClientSocket.send(response.encode())
55.             # join-success is sent to peer,
56.             # if an account with this username is not exist, and the account is created
57.             else:
58.                 db.register(message[1], message[2])
59.                 response = "REGISTER <SUCCESS> <200>"
60.                 logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
61.                 self.tcpClientSocket.send(response.encode())
62.         # LOGIN #
63.         elif message[0] == "LOGIN":
64.             # login-account-not-exist is sent to peer,
65.             # if an account with the username does not exist
66.             if not db.is_account_exist(message[1]):
67.                 response = "AUTH <FAILURE> <404>"
68.                 logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
69.                 self.tcpClientSocket.send(response.encode())
70.             # login-online is sent to peer,
71.             # if an account with the username already online
72.             elif db.is_account_online(message[1]):
73.                 response = "AUTH <ONLINE> <300>"
74.                 logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
75.                 self.tcpClientSocket.send(response.encode())
76.             # login-success is sent to peer,
77.             # if an account with the username exists and not online
78.             else:
79.                 # retrieves the account's password, and checks if the one entered by
the user is correct
80.                 retrieved_pass = db.get_password(message[1])
81.                 # if password is correct, then peer's thread is added to threads list
82.                 # peer is added to db with its username, port number, and ip address
83.                 if retrieved_pass == message[2]:
84.                     self.username = message[1]
85.                     self.lock.acquire()
86.                     try:
87.                         tcpThreads[self.username] = self
88.                     finally:
89.                         self.lock.release()
90.
91.                 db.user_login(message[1], self.ip, self.port)
92.                 # login-success is sent to peer,
93.                 # and a UDP server thread is created for this peer, and thread is
started
94.                 # timer thread of the udp server is started
95.                 response = "AUTH <SUCCESS> <200>"
96.                 logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
97.                 self.tcpClientSocket.send(response.encode())
98.                 self.udpServer = UDPServer(self.username, self.tcpClientSocket)
99.                 self.udpServer.start()
100.                self.udpServer.timer.start()
```

## P2P chatting Application

```
101.             # if password not matches and then login-wrong-password response is
sent
102.             else:
103.                 response = "AUTH <FAILURE> <404>"
104.                 logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
105.                 self.tcpClientSocket.send(response.encode())
106.             # LOGOUT #
107.             elif message[0] == "LOGOUT":
108.                 # if user is online, removes the user from onlinePeers list and removes the
thread for this user
109.                 # from tcpThreads socket is closed and timer thread of the udp for this
user is cancelled
110.                 if db.is_account_online(self.username):
111.                     db.user_logout(message[1])
112.                     self.lock.acquire()
113.                     try:
114.                         if self.username in tcpThreads:
115.                             del tcpThreads[self.username]
116.                     finally:
117.                         self.lock.release()
118.                     print(Fore.BLUE + self.ip + ":" + str(self.port) + " is logged out")
119.                     self.tcpClientSocket.close()
120.                     self.udpServer.timer.cancel()
121.                     break
122.                 else:
123.                     self.tcpClientSocket.close()
124.                     break
125.
126.             # SEARCH #
127.             elif message[0] == "SEARCH_USER":
128.                 # checks if an account with the username exists
129.                 if db.is_account_exist(message[1]):
130.                     # checks if the account is online
131.                     # and sends the related response to peer
132.                     if db.is_account_online(message[1]):
133.                         peer_info = db.get_peer_ip_port(message[1])
134.                         response = "SEARCH_USER_RESPONSE <SUCCESS> <200> " +
str(peer_info[0]) + ":" + str(
135.                             peer_info[1])
136.                         logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
137.                         self.tcpClientSocket.send(response.encode())
138.                     else:
139.                         response = "SEARCH_USER_RESPONSE <NOT_ONLINE> <300>"
140.                         logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
141.                         self.tcpClientSocket.send(response.encode())
142.                 # enters if username does not exist
143.                 else:
144.                     response = "SEARCH_USER_RESPONSE <NOT_FOUND> <404>"
145.                     logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
146.                     self.tcpClientSocket.send(response.encode())
147.             # online peers discovery
148.             elif message[0] == "DISCOVER_PEERS":
149.                 peer_list = db.get_online_peer_list()
150.                 # remove the requesting user from the list
151.                 if peer_list:
152.                     for peer in peer_list:
153.                         if peer['username'] == message[2]:
154.                             peer_list.remove(peer)
155.                 if peer_list and len(peer_list) > 0:
156.                     # detailed list
```



## P2P chatting Application

```
157.         if message[1] == "DETAILED":
158.             response = "PEER_LIST <SUCCESS> <200> " + ' '.join(
159.                 f"{peer['username']} ({peer['ip']}:{peer['port']})" for peer in
peer_list
160.             )
161.
162.             logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
163.             self.tcpClientSocket.send(response.encode())
164.         # partial list
165.     else:
166.         usernames = [peer['username'] for peer in peer_list]
167.         response = "PEER_LIST <SUCCESS> <200> " + ' '.join(usernames)
168.         logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
169.         self.tcpClientSocket.send(response.encode())
170.     # failure empty list
171.     else:
172.         response = "PEER_LIST <FAILURE> <404>"
173.         logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
174.         self.tcpClientSocket.send(response.encode())
175. elif message[0] == "CREATE-CHAT-ROOM":
176.     # CREATE-exist is sent to peer,
177.     # if a room with this username already exists
178.     if db.is_room_exist(message[1:-1]):
179.         response = "CREATION <FAILURE> <404>"
180.         print("From-> " + self.ip + ":" + str(self.port) + " " + response)
181.         logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
182.         self.tcpClientSocket.send(response.encode())
183.     else:
184.         db.add_chat_room(message[1], message[2])
185.         db.add_online_peer_chatroom(message[2], message[1])
186.         response = "CREATION <SUCCESS> <200>"
187.         logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
188.         self.tcpClientSocket.send(response.encode())
189.
190. elif message[0] == "ROOM-EXIT":
191.     if db.is_room_exist(message[2]):
192.         db.remove_peer_from_chatroom(message[1], message[2])
193.         response = "ROOM-EXIT-RESPONSE <SUCCESS> <200>"
194.         logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
195.         self.tcpClientSocket.send(response.encode())
196.     else:
197.         response = "ROOM-EXIT-RESPONSE <FAILURE> <404>"
198.         logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
199.         self.tcpClientSocket.send(response.encode())
200.
201. elif message[0] == "JOIN-CHAT-ROOM":
202.     # checks if an account with the username exists
203.     if db.is_room_exist(message[1]):
204.         # checks if the room exists
205.         # and sends the related response to peer
206.         peers = db.get_chatroom_peers(message[1])
207.         peers.append(message[2])
208.         peers = list(set(peers))
209.         room_host = db.get_chatroom_host(message[1])
210.         db.update_chatroom(message[1], peers, room_host)
```

## P2P chatting Application

```
213.         db.add_online_peer_chatroom(message[2], message[1])
214.         response = "JOIN <SUCCESS> <200> " + room_host
215.
216.         logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
217.         self.tcpClientSocket.send(response.encode())
218.     else:
219.         response = "JOIN <FAILURE> <404>"
220.         logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
221.         self.tcpClientSocket.send(response.encode())
222.
223.         # enters if username does not exist
224.
225.
226.     elif message[0] == "SHOW-ROOM-LIST":
227.         chat_rooms_list = db.get_chat_rooms_list()
228.         if chat_rooms_list is not None:
229.             response = "ROOMS-LIST <SUCCESS> <200> " + ' '.join(
230.                 f"{chatroom['name']} {chatroom['peers']} {chatroom['host']}."
231.                 for chatroom in chat_rooms_list
232.             )
233.
234.             logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
235.             self.tcpClientSocket.send(response.encode())
236.         else:
237.             response = "ROOM-LIST <FAILURE> <404>"
238.             logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
239.             self.tcpClientSocket.send(response.encode())
240.
241.     elif message[0] == "DISCOVER-ROOM-PEERS":
242.         peer_room_list = db.get_chatroom_peers(message[1])
243.         if peer_room_list is not None:
244.             response = "PEER-LIST <SUCCESS> <200> " + ' '.join(
245.                 f"{peer}" for peer in peer_room_list
246.             )
247.             logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
248.             self.tcpClientSocket.send(response.encode())
249.         else:
250.             response = "PEER-LIST <FAILURE> <404>"
251.             logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
252.             self.tcpClientSocket.send(response.encode())
253.
254.     except OSError as oErr:
255.         logging.error("OSError: {0}".format(oErr))
256.         response = "HELLO_BACK " + "FAILURE " + "404"
257.         logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
258.         db.user_logout(self.username)
259.     except ConnectionResetError as cErr:
260.         logging.error("ConnectionResetError: {0}".format(cErr))
261.         db.user_logout(self.username)
262.     except IndexError as iErr:
263.         logging.error("IndexError: {0}".format(iErr))
264.
265.         # function for resetting the timeout for the udp timer thread
266.         def resetTimeout(self):
267.             self.udpServer.resetTimer()
268.
269.
270. # implementation of the udp server thread for clients
```

## P2P chatting Application

```
271. class UDPServer(threading.Thread):
272.
273.     # udp server thread initializations
274.     def __init__(self, username, clientSocket):
275.         threading.Thread.__init__(self)
276.         self.username = username
277.         self.default_timeout = 3
278.         # timer thread for the udp server is initialized
279.         self.timer = threading.Timer(self.default_timeout, self.waitKeepAliveMessage)
280.         self.tcpClientSocket = clientSocket
281.
282.         # if hello message is not received before timeout
283.         # then peer is disconnected
284.         def waitKeepAliveMessage(self):
285.
286.             if self.username is not None:
287.                 db.user_logout(self.username)
288.                 if self.username in tcpThreads:
289.                     del tcpThreads[self.username]
290.                 self.tcpClientSocket.close()
291.                 print(Fore.BLUE + "Removed " + self.username + " from online peers")
292.
293.         # resets the timer for udp server
294.         def resetTimer(self):
295.             self.timer.cancel()
296.             self.timer = threading.Timer(self.default_timeout, self.waitKeepAliveMessage)
297.             self.timer.start()
298.
299.
300. # tcp and udp server port initializations
301. print("Registry started...")
302. port = 15600
303. portUDP = 15500
304.
305. # db initialization
306. db = db.DB()
307.
308. # gets the ip address of this peer
309. # first checks to get it for Windows devices
310. # if the device that runs this application is not windows
311. # it checks to get it for macOS devices
312. hostname = gethostname()
313. try:
314.     host = gethostbyname(hostname)
315. except gaierror:
316.     import netifaces as ni
317.
318.     host = ni.ifaddresses('en0')[ni.AF_INET][0]['addr']
319.
320. print("Registry IP address: " + host)
321. print("Registry port number: " + str(port))
322.
323. # onlinePeers list for online account
324. onlinePeers = {}
325. # accounts list for accounts
326. accounts = {}
327. # tcpThreads list for online client's thread
328. tcpThreads = {}
329.
330. # tcp and udp socket initializations
331. tcpSocket = socket(AF_INET, SOCK_STREAM)
332. udpSocket = socket(AF_INET, SOCK_DGRAM)
333. tcpSocket.bind((host, port))
334. udpSocket.bind((host, portUDP))
```

## P2P chatting Application

```
335. tcpSocket.listen(1000)
336. print("Listening for incoming connections...")
337.
338. # input sockets that are listened
339. inputs = [tcpSocket, udpSocket]
340.
341. # log file initialization
342. logging.basicConfig(filename="logs/registry.log", level=logging.INFO)
343.
344. # as long as at least a socket exists to listen registry runs
345. while inputs:
346.     # monitors for the incoming connections
347.     readable, writable, exceptional = select.select(inputs, [], [])
348.     for s in readable:
349.         # if the message received comes to the tcp socket
350.         # the connection is accepted and a thread is created for it, and that thread is started
351.         if s is tcpSocket:
352.             tcpClientSocket, addr = tcpSocket.accept()
353.             newThread = ClientThread(addr[0], addr[1], tcpClientSocket)
354.             newThread.tcpClientSocket =
newThread.context.wrap_socket(newThread.tcpClientSocket, server_side=True)
355.             newThread.start()
356.             response = "HELLO_BACK " + "SUCCESS " + "200 "
357.             logging.info("Send to " + addr[0] + ":" + str(addr[1]) + " -> " + response)
358.             newThread.tcpClientSocket.send(response.encode())
359.         # if the message received comes to the udp socket
360.         elif s is udpSocket:
361.             # received the incoming udp message and parses it
362.             message, clientAddress = s.recvfrom(1024)
363.             message = message.decode().split()
364.             # checks if it is a hello message
365.             if message[0] == "KEEP_ALIVE":
366.                 # checks if the account that this hello message
367.                 # is sent from is online
368.                 if message[1] in tcpThreads:
369.                     # resets the timeout for that peer since the hello message is received
370.                     tcpThreads[message[1]].resetTimeout()
371.                     logging_message = "KEEP_ALIVE <SUCCESS> <200>"
372.                     # Send the response back to the UDP client
373.
374. # registry tcp socket is closed
375. tcpSocket.close()
376.
```