



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

Peer-to-Peer Multi-User Chatting Application

Group 7

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

Phase 2

Table of Contents

<i>Outputs of the code.....</i>	<i>6</i>
<i>CODE:.....</i>	<i>16</i>
Code of database:	16
Description of Database Code:	17
Code of Peer:	18
Description of Peer code:	23
Code of Registry:.....	25
Description of Registry code:	30
Code of Utility :.....	32
Description of Utility code:	32

Table of Figures

Figure 1: Sign up page.....	6
Figure 2: Account record in the DB (password is hashed)	7
Figure 3: Login bad scenario	7
Figure 4: Login Scenario	8
Figure 5: Detailed Find online users.....	9
Figure 6: Simple Find online user	9
Figure 7: Search User (FOUND).....	10
Figure 8: Output of registry	11
Figure 9: Search user (NOT EXIST)	12
Figure 10: Search User (OFFLINE)	13
Figure 11: Find User (no one online +invalid option)	13
Figure 12: Logout	14
Figure 13: exit	15

Abstract

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

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

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

Outputs of the code

```
Enter IP address of registry: 192.168.1.2
Connected to the registry...
P2P Chat Started

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

Choice: 1
username: myname
password: Password1234
Account created successfully.
```

Figure 1: Sign up page

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

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

```
Welcome!  
Select Option:  
    1 : Signup  
    2 : Login  
    3 : Exit  
  
Choice: 2  
username: myname  
password: Password123  
Enter a port number for peer server: 4278  
Wrong password...
```

Figure 3: Login bad scenario

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

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

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

Choice:
```

Figure 4: Login Scenario

P2P chatting Application

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

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

Figure 5: Detailed Find online users

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

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

Figure 6: Simple Find online user

Main Menu

Select Option:

- 1 : Find Online Users
- 2 : Search User
- 3 : Start a Chat
- 4 : Logout

Choice: 2

Username to be searched: Kareem

Kareem is found successfully...

IP address of Kareem is 192.168.1.2:63142

Figure 7: Search User (FOUND)

```
Registry started...
Registry IP address: 192.168.1.2
Registry port number: 15600
Listening for incoming connections...
New thread started for 192.168.1.2:63105
Connection from: 192.168.1.2:15600
IP Connected: 192.168.1.2
Listening for incoming connections...
KEEP_ALIVE is received from myname
Listening for incoming connections...
KEEP_ALIVE is received from myname
Listening for incoming connections...
KEEP_ALIVE is received from myname
Listening for incoming connections...
KEEP_ALIVE is received from myname
Listening for incoming connections...
KEEP_ALIVE is received from myname
```

Figure 8: Output of registry

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

Choice: 2
Username to be searched: random
random is not found
```

Figure 9: Search user (NOT EXIST)

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

Choice: 2
Username to be searched: Ali
Ali is not online...
```

Figure 10: Search User (OFFLINE)

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

Choice: 1
Retrieve detailed list with users IP and Port numbers?(Choose y or n): !
Error: Please choose a valid option (y or n)

Retrieve detailed list with users IP and Port numbers?(Choose y or n): Y
No Online Users right now, please check back later
```

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

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

Choice: 4
Logged out successfully

Process finished with exit code 0
```

Figure 12: Logout

```
Connected to the registry...
P2P Chat Started

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

Choice: 3

Process finished with exit code 0
```

Figure 13: exit

CODE:

Code of database:

```
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.    # registers a user
25.    def register(self, username, password):
26.        account = {
27.            "username": username,
28.            "password": password
29.        }
30.        self.db.accounts.insert_one(account)
31.
32.    # retrieves the password for a given username
33.    def get_password(self, username):
34.        return self.db.accounts.find_one({"username": username})["password"]
35.
36.    # checks if an account with the username online
37.    def is_account_online(self, username):
38.        count = self.db.online_peers.count_documents({'username': username})
39.        return count > 0
40.
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.        }
49.        self.db.online_peers.insert_one(online_peer)
50.
51.    # logs out the user
52.    def user_logout(self, username):
53.        self.db.online_peers.delete_one({"username": username})
54.
55.    # retrieves the ip address and the port number of the username
56.    def get_peer_ip_port(self, username):
57.        res = self.db.online_peers.find_one({"username": username})
58.        return (res["ip"], res["port"])
59.
```


P2P chatting Application

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

Description of Database Code:

Database Operations with MongoDB

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

Class: DB

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

Methods:

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

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

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

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

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

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

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

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

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

Usage:

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

P2P chatting Application

Code of Peer:

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

P2P chatting Application

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

P2P chatting Application

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

P2P chatting Application

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

P2P chatting Application

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

P2P chatting Application

Description of Peer code:

The peer basic Client have the follwing functionalities:

User Registration/Login/Logout:

Users can register by creating an account with a username and password.

Registered users can log in using their credentials and a specified port number for the peer server.

Users can log out, which terminates the connection to the registry and closes sockets.

User Interaction:

The program prompts the user with various options like creating an account, logging in, searching for users, finding online users, etc.

Different options are handled within the `handle_user_request` method based on user input.

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

Input Handling:

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

Menu Navigation:

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

Action Execution:

Based on the user's input, it triggers specific actions like creating an account, logging in, logging out, searching for users, finding online users, etc.

Here's a breakdown of how it handles various user choices:

Signup/Login/Logout/Exit:

If the user chooses to sign up, it initiates the account creation process.

For login, it prompts the user for credentials and attempts to log in.

If the user chooses to log out, it initiates the logout process.

Exiting the program involves logging out and closing connections.

Find Online Users/Search User/Start a Chat:

Searching for online users can be done in detailed or simple mode, providing information about users currently online.

Searching for a specific user prompts the user to input the username and attempts to find the user's IP address if they are online.

Starting a chat is not available in this phase, as indicated in the code.

Cancel/Invalid Choice:

P2P chatting Application

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

Registry Communication:

Communication with the registry server happens via TCP sockets.

Different messages (e.g., registration, login, logout, search) are sent to the registry, and responses are received accordingly.

Keep-Alive Mechanism:

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

Logging:

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

User Interface:

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

The color coding is mainly as follows:

Green : Successful operation

Red : Error

Yellow : Abnormal Scenario but not error

Magenta : Prompt user for input

Blue : Logs at the server

More details will be added after implementing the chatting functionality

Code of Registry:

```

1. '''
2.     ## Implementation of registry
3.     ## 150114822 - Eren Ulaş
4. '''
5.
6. from socket import *
7. import threading
8. import select
9. import logging
10.
11. from colorama import Fore
12.
13. import db
14. import ssl
15.
16.
17. # This class is used to process the peer messages sent to registry
18. # for each peer connected to registry, a new client thread is created
19. class ClientThread(threading.Thread):
20.     # initializations for client thread
21.     def __init__(self, ip, port, tcpClientSocket):
22.         threading.Thread.__init__(self)
23.         # ip of the connected peer
24.
25.         self.ip = ip
26.         # port number of the connected peer
27.         self.port = port
28.         # socket of the peer
29.         self.tcpClientSocket = tcpClientSocket
30.         # Create SSL context
31.         self.context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
32.         self.context.load_cert_chain(certfile="security/server.crt",
keyfile="security/server.key")
33.         # username, online status and udp server initializations
34.         self.username = None
35.         self.isOnline = True
36.         self.udpServer = None
37.         print("New thread started for " + ip + ":" + str(port))
38.
39.     # main of the thread
40.     def run(self):
41.         # locks for thread which will be used for thread synchronization
42.         self.lock = threading.Lock()
43.         print(Fore.BLUE+"Connection from: " + self.ip + ":" + str(port))
44.         print(Fore.BLUE+"IP Connected: " + self.ip)
45.
46.         while True:
47.             try:
48.                 # waits for incoming messages from peers
49.                 message = self.tcpClientSocket.recv(1024).decode().split()
50.                 logging.info("Received from " + self.ip + ":" + str(self.port) + " -> " + "
".join(message))
51.                 # JOIN #
52.                 if message[0] == "REGISTER":
53.                     # join-exist is sent to peer,
54.                     # if an account with this username already exists
55.                     if db.is_account_exist(message[1]):
56.                         response = "REGISTER <EXIST> <300>"
57.                         print("From-> " + self.ip + ":" + str(self.port) + " " + response)
58.                         logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)

```

P2P chatting Application

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

P2P chatting Application

```
114.         # from tcpThreads socket is closed and timer thread of the udp for this
user is cancelled
115.         if db.is_account_online(self.username):
116.             db.user_logout(message[1])
117.             self.lock.acquire()
118.             try:
119.                 if self.username in tcpThreads:
120.                     del tcpThreads[self.username]
121.             finally:
122.                 self.lock.release()
123.             print(Fore.BLUE+self.ip + ":" + str(self.port) + " is logged out")
124.             self.tcpClientSocket.close()
125.             self.udpServer.timer.cancel()
126.             break
127.         else:
128.             self.tcpClientSocket.close()
129.             break
130.
131.         # SEARCH #
132.         elif message[0] == "SEARCH_USER":
133.             # checks if an account with the username exists
134.             if db.is_account_exist(message[1]):
135.                 # checks if the account is online
136.                 # and sends the related response to peer
137.                 if db.is_account_online(message[1]):
138.                     peer_info = db.get_peer_ip_port(message[1])
139.                     response = "SEARCH_USER_RESPONSE <SUCCESS> <200> " +
str(peer_info[0]) + ":" + str(
140.                         peer_info[1])
141.                     logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
142.                     self.tcpClientSocket.send(response.encode())
143.                 else:
144.                     response = "SEARCH_USER_RESPONSE <NOT_ONLINE> <300>"
145.                     logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
146.                     self.tcpClientSocket.send(response.encode())
147.             # enters if username does not exist
148.             else:
149.                 response = "SEARCH_USER_RESPONSE <NOT_FOUND> <404>"
150.                 logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
151.                 self.tcpClientSocket.send(response.encode())
152.         # online peers discovery
153.         elif message[0] == "DISCOVER_PEERS":
154.             peer_list = db.get_online_peer_list()
155.             # remove the requesting user from the list
156.             if peer_list:
157.                 for peer in peer_list:
158.                     if peer['username'] == message[2]:
159.                         peer_list.remove(peer)
160.             if peer_list and len(peer_list) > 0:
161.                 # detailed list
162.                 if message[1] == "DETAILED":
163.                     response = "PEER_LIST <SUCCESS> <200> " + ' '.join(
164.                         f"{peer['username']} ({peer['ip']}:{peer['port']})" for peer in
peer_list
165.                     )
166.                     logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " +
response)
167.                     self.tcpClientSocket.send(response.encode())
168.                 # partial list
169.             else:
```

P2P chatting Application

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

P2P chatting Application

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

P2P chatting Application

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

Description of Registry code:

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

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

Initialization:

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

Timeout Handling:

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

Timer Reset:

`resetTimer`: Resets the timer for the UDP server when necessary, canceling the existing timer and starting a new one.

This class is responsible for managing the keep-alive mechanism for the UDP connections to ensure that users stay connected within a specific time frame.

Main Section:

Initializes TCP and UDP sockets for communication.

Monitors incoming connections and creates a new ClientThread for TCP connections.

P2P chatting Application

Manages incoming UDP messages to keep clients online by sending keep-alive messages.

The code communicates with a MongoDB database (db module) for account management (registration, login/logout) and maintains lists of online peers and threads for each user.

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

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

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

REGISTER: Checks if an account exists with the username. If it does, sends a response indicating existence; otherwise, registers a new account.

LOGIN: Handles login operations, checking account existence, online status, and password verification. If successful, adds the user to the list of online peers.

LOGOUT: Logs the user out by removing them from online status and closing connections.

SEARCH_USER: Searches for a specific user and responds with their online status or presence details.

DISCOVER_PEERS: Retrieves a list of online peers, either detailed or partial, based on the request.

Exception Handling: Catches OSError and handles it by logging the error and sending a failure response.

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

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

Code of Utility :

```
1. import hashlib
2.
3.
4. def hash_password(password):
5.     # Encode the password string to bytes before hashing
6.     password_bytes = password.encode('utf-8')
7.     # Create an SHA-256 hash object
8.     sha256 = hashlib.sha256()
9.     # Update the hash object with the password bytes
10.    sha256.update(password_bytes)
11.    # Get the hexadecimal representation of the hashed password
12.    hashed_password = sha256.hexdigest()
13.    return hashed_password
14.
```

Description of Utility code:

The Hashing technique for the application is described as follows:

Input:

The function takes a password string as input.

Encoding to Bytes:

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

Hashing:

It creates an SHA-256 hash object sha256.

The update() method updates the hash object with the encoded password bytes.

This effectively hashes the password using the SHA-256 algorithm.

Hexadecimal Representation:

The hexdigest() method generates the hexadecimal representation of the hashed password.

Hexadecimal representation is commonly used to present hash values in a human-readable format.

Return:

The function returns the hashed password as a hexadecimal string.

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