**A blue and white logo

Description automatically generated**

Ain Shams University Faculty of Engineering

CSE351: Computer Networks

Under the surveillance of Professor Ayman Bahaa.

**Peer-to-Peer Multi-User Chatting Application**

**Group 7**

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

Phase 2

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

Table of Contents

[Outputs of the code 5](#_Toc154092689)

[CODE: 15](#_Toc154092690)

[Code of database: 15](#_Toc154092691)

[Description of Database Code: 16](#_Toc154092692)

[Code of Peer: 17](#_Toc154092693)

[Description of Peer code: 22](#_Toc154092694)

[Code of Registry: 24](#_Toc154092695)

[Description of Registry code: 29](#_Toc154092696)

[Code of Utility : 31](#_Toc154092697)

[Description of Utility code: 31](#_Toc154092698)

Table of Figures

[Figure 1: Sign up page 6](#_Toc154092699)

[Figure 2: Account record in the DB (password is hashed) 7](#_Toc154092700)

[Figure 3: Login bad scenario 7](#_Toc154092701)

[Figure 4: Login Scenario 8](#_Toc154092702)

[Figure 5: Detailed Find online users 9](#_Toc154092703)

[Figure 6: Simple Find online user 9](#_Toc154092704)

[Figure 7: Search User (FOUND) 10](#_Toc154092705)

[Figure 8: Output of registry 11](#_Toc154092706)

[Figure 9: Search user (NOT EXIST) 12](#_Toc154092707)

[Figure 10: Search User (OFFLINE) 13](#_Toc154092708)

[Figure 11: Find User (no one online +invalid option) 13](#_Toc154092709)

[Figure 12: Logout 14](#_Toc154092710)

[Figure 13: exit 15](#_Toc154092711)

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

A screenshot of a computer

Description automatically generated

Figure 1: Sign up page

A close up of numbers

Description automatically generated

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

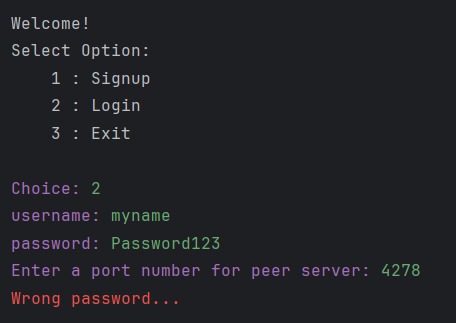


Figure 3: Login bad scenario

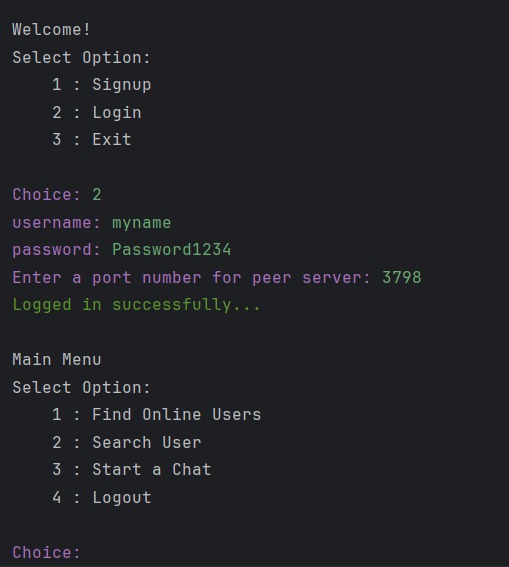


Figure 4: Login Scenario

A screenshot of a computer

Description automatically generated

Figure 5: Detailed Find online users

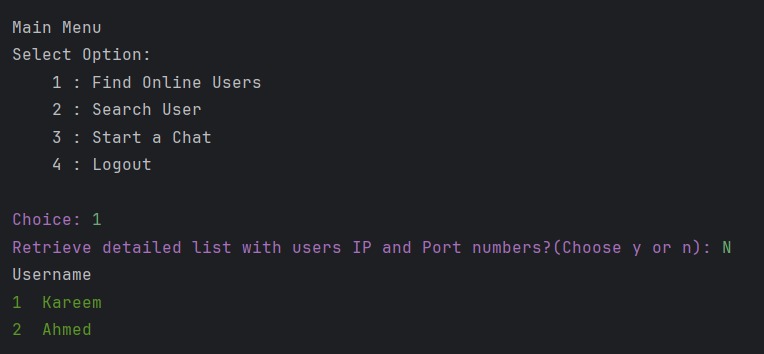


Figure 6: Simple Find online user

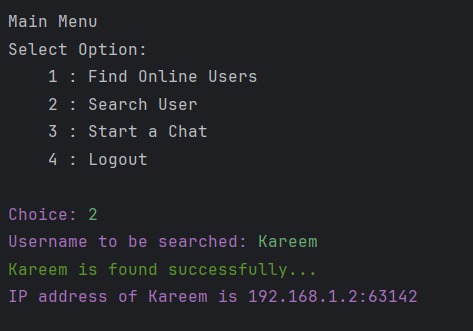


Figure 7: Search User (FOUND)

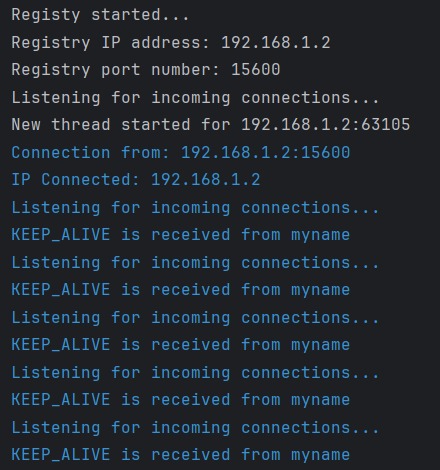


Figure 8: Output of registry

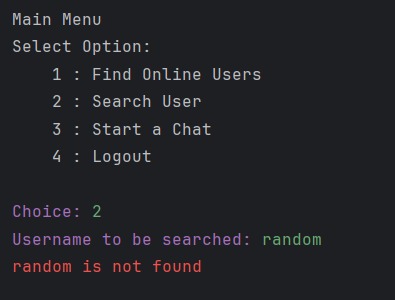


Figure 9: Search user (NOT EXIST)

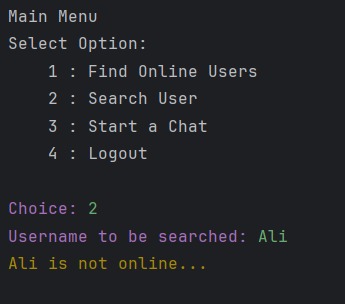


Figure 10: Search User (OFFLINE)

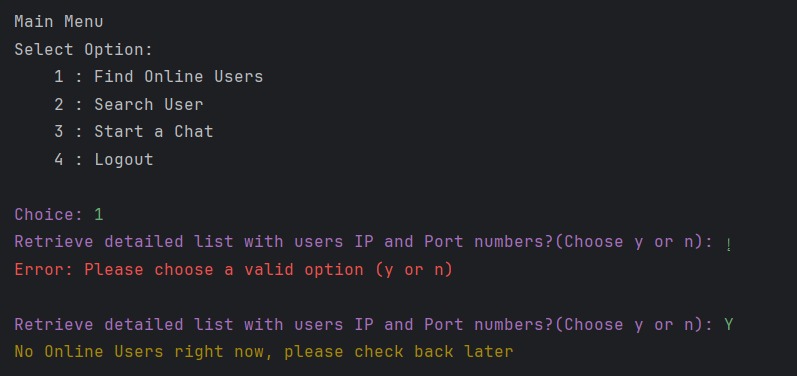


Figure 11: Find User (no one online +invalid option)

A screenshot of a computer program

Description automatically generated

Figure 12: Logout

A screen shot of a computer

Description automatically generated

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.

60. def get\_online\_peer\_list(self):

61. online\_peers\_cursor = self.db.online\_peers.find()

62. online\_peers\_list = list(online\_peers\_cursor)

63. return online\_peers\_list

64.

## Description of Database Code:

Database Operations with MongoDB

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

Class: DB

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

Methods:

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

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

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

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

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

user\_login(username, ip, port): Logs in a user by inserting their details (username, ip, and port) into the 'online\_peers' collection.

user\_logout(username): Logs out a user by removing their details from the 'online\_peers' collection based on the username.

get\_peer\_ip\_port(username): Retrieves the IP address and port number associated with a specific username from the 'online\_peers' collection.

get\_online\_peer\_list(): Fetches a list of online peers by querying the 'online\_peers' collection and returning a list of documents.

Usage:

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

## Code of Peer:

1. import logging

2. import threading

3. import time

4. from socket import \*

5. import ssl

6. from colorama import Fore

7. import utility

8.

9.

10. # main process of the peer

11. class peerClient:

12.

13. # peer initializations

14. def \_\_init\_\_(self):

15. # ip address of the registry

16. self.registryName = input("Enter IP address of registry: ")

17. # self.registryName = 'localhost'

18. # port number of the registry

19. self.registryPort = 15600

20. # tcp socket connection to registry

21. self.tcpClientSocket = socket(AF\_INET, SOCK\_STREAM)

22. # Create an SSL context

23. context = ssl.create\_default\_context()

24. context.check\_hostname = False

25. context.verify\_mode = ssl.CERT\_NONE

26. # Wrap the socket with SSL

27. self.tcpClientSocket = context.wrap\_socket(self.tcpClientSocket, server\_hostname=self.registryName)

28. # Connect to the server

29. self.tcpClientSocket.connect((self.registryName, self.registryPort))

30. self.connectServer()

31. # initializes udp socket which is used to send hello messages

32. self.udpClientSocket = socket(AF\_INET, SOCK\_DGRAM)

33. # udp port of the registry

34. self.registryUDPPort = 15500

35. # login info of the peer

36. self.loginCredentials = (None, None)

37. # online status of the peer

38. self.isOnline = False

39. self.timer = None

40. # User Interface

41. self.state = 0

42. self.states = {1: "Welcome!", 2: "Main Menu"}

43. self.options = {1: {1: "Signup", 2: "Login", 3: "Exit"},

44. 2: {1: "Find Online Users", 2: "Search User", 3: "Start a Chat", 4: "Logout"}}

45.

46. # log file initialization

47. logging.basicConfig(filename="logs/peer.log", level=logging.INFO)

48. # as long as the user is not logged out, asks to select an option in the menu

49. while True:

50. # menu selection prompt

51. if self.state == 0:

52. print(Fore.MAGENTA + "P2P Chat Started")

53. self.state = 1

54.

55. print(Fore.RESET + '\n' + self.states[self.state] + '\nSelect Option:')

56. for option\_number, option\_name in self.options[self.state].items():

57. print("\t" + str(option\_number) + " : " + option\_name)

58. choice = input(Fore.MAGENTA + "\nChoice: ")

59. self.handle\_user\_request(choice)

60.

61. def handle\_user\_request(self, choice):

62. selection = self.options[self.state][int(choice)]

63.

64. if selection == "Signup":

65. # Creates an account with the username and password entered by the user

66. username = input("username: ")

67. password = input("password: ")

68. self.createAccount(username, password)

69.

70. elif selection == "Login" and not self.isOnline:

71. # Asks for the username and the password to login

72. username = input("username: ")

73. password = input("password: ")

74. # asks for the port number for server's tcp socket, will be needed in late phases

75. peer\_server\_port = int(input("Enter a port number for peer server: "))

76.

77. status = self.login(username, password, peer\_server\_port)

78. # is user logs in successfully, peer variables are set

79. if status == 1:

80. self.isOnline = True

81. self.loginCredentials = (username, password)

82. # hello message is sent to registry

83. self.sendKeepAliveMessage(self.loginCredentials[0])

84. self.state = 2

85.

86. elif selection == "Logout":

87. # User is logged out and peer variables are set, and server and client sockets are closed

88. if self.isOnline:

89. self.logout(1)

90. self.isOnline = False

91. self.loginCredentials = (None, None)

92. print(Fore.GREEN + "Logged out successfully")

93. self.tcpClientSocket.close()

94. exit(0)

95.

96. elif selection == "Exit":

97. # Exits the program:

98. self.logout(2)

99. self.tcpClientSocket.close()

100. exit(0)

101.

102. elif selection == "Find Online Users":

103. # Prompt user for the users list mode and return it

104. while True:

105. option = input(Fore.MAGENTA + "Retrieve detailed list with users IP and Port numbers?(Choose y or n): ")

106. if option == 'Y' or option == 'y':

107. self.find\_online\_user("DETAILED")

108. return

109. elif option == 'N' or option == 'n':

110. self.find\_online\_user("SIMPLE")

111. return

112. else:

113. print(Fore.RED + "Error: Please choose a valid option (y or n)\n")

114.

115. elif selection == "Search User":

116. # If user is online, then user is asked for a username that is wanted to be searched

117. if self.isOnline:

118. username = input("Username to be searched: ")

119. search\_status = self.search\_user(username)

120. # if user is found its ip address is shown to user

121. if search\_status is not None and search\_status != 0:

122. print(Fore.MAGENTA + "IP address of " + username + " is " + search\_status)

123. time.sleep(1)

124.

125. elif selection == "Start a Chat":

126. print(Fore.RED + "Not available in this phase")

127.

128. # if choice is cancel timer for hello message is cancelled

129. elif choice == "CANCEL":

130. self.timer.cancel()

131. else:

132. print(Fore.RED + "Invalid Option Selected, please try again.\n")

133.

134. # account creation function

135. def createAccount(self, username, password):

136. # join message to create an account is composed and sent to registry

137. # if response is "success" then informs the user for account creation

138. # if response is "exist" then informs the user for account existence

139. message = "REGISTER " + username + " " + utility.hash\_password(password)

140. response = self.send\_credentials(message)

141. # Process the response from the registry

142.

143. if response[2] == "<200>":

144. print(Fore.GREEN + "Account created successfully.")

145. time.sleep(1)

146. elif response[2] == "<300>":

147. print(Fore.YELLOW + "Username already exists. Choose another username or login.")

148. time.sleep(1)

149. elif response[2] == "<404>":

150. print(Fore.RED + "Failed to create an account. Please try again.")

151. time.sleep(1)

152.

153. def send\_credentials(self, message):

154. logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)

155. self.tcpClientSocket.send(message.encode())

156. response = self.tcpClientSocket.recv(1024).decode()

157. logging.info("Received from " + self.registryName + " -> " + response)

158. return response.split()

159.

160. # login function

161. def login(self, username, password, peerServerPort):

162. # a login message is composed and sent to registry

163. # an integer is returned according to each response

164. message = "LOGIN " + username + " " + utility.hash\_password(password) + " " + str(peerServerPort)

165. response = self.send\_credentials(message)

166. if response[2] == "<200>":

167. print(Fore.GREEN + "Logged in successfully...")

168. time.sleep(1)

169. return 1

170. elif response[2] == "<300>":

171. print(Fore.YELLOW + "Account is already online...")

172. time.sleep(1)

173. return 2

174. elif response[2] == "<404>":

175. print(Fore.RED + "Wrong password...")

176. time.sleep(1)

177. return 3

178.

179. # logout function

180. def logout(self, option):

181. # a logout message is composed and sent to registry

182. # timer is stopped

183. if option == 1:

184. message = "LOGOUT " + self.loginCredentials[0]

185. self.timer.cancel()

186. else:

187. message = "LOGOUT"

188. logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)

189. self.tcpClientSocket.send(message.encode())

190.

191. # function for searching an online user

192. def search\_user(self, username):

193. # a search message is composed and sent to registry

194. # custom value is returned according to each response

195. # to this search message

196. message = "SEARCH\_USER " + username

197. logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)

198. self.tcpClientSocket.send(message.encode())

199. response = self.tcpClientSocket.recv(1024).decode().split()

200. logging.info("Received from " + self.registryName + " -> " + " ".join(response))

201. if response[2] == "<200>":

202. print(Fore.GREEN + username + " is found successfully...")

203. time.sleep(1)

204. return response[3]

205. elif response[2] == "<300>":

206. print(Fore.YELLOW + username + " is not online...")

207. time.sleep(1)

208. return 0

209. elif response[2] == "<404>":

210. print(Fore.RED + username + " is not found")

211. time.sleep(1)

212. return None

213.

214. def find\_online\_user(self, option):

215. message = "DISCOVER\_PEERS " + option + " " + self.loginCredentials[0]

216. logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)

217. self.tcpClientSocket.send(message.encode())

218. response = self.tcpClientSocket.recv(1024).decode().split()

219. logging.info("Received from " + self.registryName + " -> " + " ".join(response))

220. if response[2] == "<200>":

221. response = response[3:]

222. if option == "DETAILED":

223. print(Fore.RESET + "# Username".ljust(18) + "(IP:Port)")

224. for i in range(0, len(response), 2):

225. print(Fore.GREEN + f"{i+1} {response[i]:15}{response[i+1]}")

226. else:

227. print(Fore.RESET + "Username")

228. for username in response:

229. print(Fore.GREEN + username)

230. time.sleep(1)

231. elif response[2] == "<404>":

232. print(Fore.YELLOW + "No Online Users right now, please check back later")

233. time.sleep(1)

234.

235. # function for sending hello message

236. # a timer thread is used to send hello messages to udp socket of registry

237. def sendKeepAliveMessage(self, username):

238. message = "KEEP\_ALIVE " + username

239. logging.info("Send to " + self.registryName + ":" + str(self.registryUDPPort) + " -> " + message)

240. self.udpClientSocket.sendto(message.encode(), (self.registryName, self.registryUDPPort))

241.

242. # Assuming you expect a response from the registry

243.

244. # Schedule the next hello message

245. self.timer = threading.Timer(1, self.sendKeepAliveMessage, args=[username])

246. self.timer.start()

247.

248. def connectServer(self):

249. starting\_message = "HELLO\_P2P"

250. logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + starting\_message)

251. self.tcpClientSocket.send(starting\_message.encode())

252. response = self.tcpClientSocket.recv(1024).decode().split()

253. logging.info("Received from " + self.registryName + " -> " + " ".join(response))

254. status\_code = int(response[2])

255. if status\_code == 200:

256. print(Fore.GREEN + "Connected to the registry...")

257.

258.

259. # peer is started

260. main = peerClient()

261.

## Description of Peer code:

The peer basic Client have the follwing functionalities:

User Registration/Login/Logout:

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

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

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

User Interaction:

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

Different options are handled within the handle\_user\_request method based on user input.

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

Input Handling:

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

Menu Navigation:

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

Action Execution:

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

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

Signup/Login/Logout/Exit:

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

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

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

Exiting the program involves logging out and closing connections.

Find Online Users/Search User/Start a Chat:

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

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

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

Cancel/Invalid Choice:

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

Registry Communication:

Communication with the registry server happens via TCP sockets.

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

Keep-Alive Mechanism:

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

Logging:

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

User Interface:

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

**The color coding is mainly as follows:**

Green : Successful operation

Red : Error

Yellow : Abnormal Scenario but not error

Magenta : Prompt user for input

Blue : Logs at the server

More details will be added after implementing the chatting functionality

## Code of Registry:

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)

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

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.

167. logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)

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

169. # partial list

170. else:

171. usernames = [peer['username'] for peer in peer\_list]

172. response = "PEER\_LIST <SUCCESS> <200> " + ' '.join(usernames)

173. logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)

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

175. # failure empty list

176. else:

177. response = "PEER\_LIST <FAILURE> <404>"

178. logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)

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

180.

181.

182. except OSError as oErr:

183. logging.error("OSError: {0}".format(oErr))

184. response = "HELLO\_BACK " + "FAILURE " + "404"

185. logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)

186.

187. db.user\_logout(self.username)

188.

189. # function for resettin the timeout for the udp timer thread

190.

191. def resetTimeout(self):

192. self.udpServer.resetTimer()

193.

194.

195. # implementation of the udp server thread for clients

196. class UDPServer(threading.Thread):

197.

198. # udp server thread initializations

199. def \_\_init\_\_(self, username, clientSocket):

200. threading.Thread.\_\_init\_\_(self)

201. self.username = username

202. self.default\_timeout = 3

203. # timer thread for the udp server is initialized

204. self.timer = threading.Timer(self.default\_timeout, self.waitKeepAliveMessage)

205. self.tcpClientSocket = clientSocket

206.

207. # if hello message is not received before timeout

208. # then peer is disconnected

209. def waitKeepAliveMessage(self):

210.

211. if self.username is not None:

212. notification = "TIMEOUT " + self.username

213. self.tcpClientSocket.send(notification.encode())

214. db.user\_logout(self.username)

215. if self.username in tcpThreads:

216. del tcpThreads[self.username]

217. self.tcpClientSocket.close()

218. print(Fore.BLUE +"Removed " + self.username + " from online peers")

219.

220. # resets the timer for udp server

221. def resetTimer(self):

222. self.timer.cancel()

223. self.timer = threading.Timer(self.default\_timeout, self.waitKeepAliveMessage)

224. self.timer.start()

225.

226.

227. # tcp and udp server port initializations

228. print("Registy started...")

229. port = 15600

230. portUDP = 15500

231.

232. # db initialization

233. db = db.DB()

234.

235. # gets the ip address of this peer

236. # first checks to get it for windows devices

237. # if the device that runs this application is not windows

238. # it checks to get it for macos devices

239. hostname = gethostname()

240. try:

241. host = gethostbyname(hostname)

242. except gaierror:

243. import netifaces as ni

244.

245. host = ni.ifaddresses('en0')[ni.AF\_INET][0]['addr']

246.

247. print("Registry IP address: " + host)

248. print("Registry port number: " + str(port))

249.

250. # onlinePeers list for online account

251. onlinePeers = {}

252. # accounts list for accounts

253. accounts = {}

254. # tcpThreads list for online client's thread

255. tcpThreads = {}

256.

257. # tcp and udp socket initializations

258. tcpSocket = socket(AF\_INET, SOCK\_STREAM)

259. udpSocket = socket(AF\_INET, SOCK\_DGRAM)

260. tcpSocket.bind((host, port))

261. udpSocket.bind((host, portUDP))

262. tcpSocket.listen(1000)

263.

264. # input sockets that are listened

265. inputs = [tcpSocket, udpSocket]

266.

267. # log file initialization

268. logging.basicConfig(filename="registry.log", level=logging.INFO)

269.

270. # as long as at least a socket exists to listen registry runs

271. while inputs:

272.

273. print("Listening for incoming connections...")

274. # monitors for the incoming connections

275. readable, writable, exceptional = select.select(inputs, [], [])

276. for s in readable:

277. # if the message received comes to the tcp socket

278. # the connection is accepted and a thread is created for it, and that thread is started

279. if s is tcpSocket:

280. tcpClientSocket, addr = tcpSocket.accept()

281. newThread = ClientThread(addr[0], addr[1], tcpClientSocket)

282. newThread.tcpClientSocket = newThread.context.wrap\_socket(newThread.tcpClientSocket, server\_side=True)

283. newThread.start()

284. response = "HELLO\_BACK " + "SUCCESS " + "200 "

285. logging.info("Send to " + addr[0] + ":" + str(addr[1]) + " -> " + response)

286. newThread.tcpClientSocket.send(response.encode())

287. # if the message received comes to the udp socket

288. elif s is udpSocket:

289. # received the incoming udp message and parses it

290. message, clientAddress = s.recvfrom(1024)

291. message = message.decode().split()

292. # checks if it is a hello message

293. if message[0] == "KEEP\_ALIVE":

294. # checks if the account that this hello message

295. # is sent from is online

296. if message[1] in tcpThreads:

297. # resets the timeout for that peer since the hello message is received

298. tcpThreads[message[1]].resetTimeout()

299. print("KEEP\_ALIVE is received from " + message[1])

300. loggingmessage = "KEEP\_ALIVE <SUCCESS> <200>"

301.

302. logging.info(

303. "Received from " + clientAddress[0] + ":" + str(clientAddress[1]) + " -> " + " ".join(message))

304. # Send the response back to the UDP client

305.

306. # registry tcp socket is closed

307. tcpSocket.close()

308.

## Description of Registry code:

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

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

Initialization:

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

Timeout Handling:

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

Timer Reset:

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

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

Main Section:

Initializes TCP and UDP sockets for communication.

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

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.