**Presented To :**

**Dr. Ayman Baha T.A Noha Wahdan**

CSE351-Computer networks

**ABDULRAHMAN MAGED ABDULRAHMAN HELMY 20P7467 Hazem Zaki Aboukhalil Zaki Ahmed 20p7516 Mohamed Ali Ahmed Mohamed Eid Zaitun 20p7519 Mazen Tayseer Ahmed Abdelaal 20p7460**

Peer to Peer-Chatting Application Team 42

Table of Contents

[1. Summary 1](#_Toc154786670)

[2. Objective 2](#_Toc154786671)

[3. Scope of the Application 2](#_Toc154786672)

[3.1. User Registration and Authentication 2](#_Toc154786673)

[3.2. Private Messaging: 2](#_Toc154786674)

[3.3. Friend List: 3](#_Toc154786680)

[3.4. Chatting Rooms: 3](#_Toc154786681)

[3.5. Nearby Global Online Peers: 3](#_Toc154786682)

[3.6. Messages Notification Tray: 3](#_Toc154786691)

[3.7. Most Populated Global Chat Rooms: 3](#_Toc154786692)

[3.8. Real-time Updates: 3](#_Toc154786693)

[4. Timeline 4](#_Toc154786694)

[4.1. Phase 1 Project Planning and Design 4](#_Toc154786695)

[4.2. Phase 2 Basic Client-Server Setup 4](#_Toc154786696)

[4.3. Phase 3 Peer-to-Peer Architecture and Chat Rooms : 4](#_Toc154786697)

[4.4. Phase 4 One-to-One Chatting and Protocol Optimization : 4](#_Toc154786698)

[5. Cost Estimation 5](#_Toc154786699)

[5.1. Cost Analysis 5](#_Toc154786700)

[5.2. Development Cost: 5](#_Toc154786701)

[5.3. Maintenance Costs: 5](#_Toc154786702)

[5.4. Hosting Costs: 5](#_Toc154786703)

[5.5. Marketing Costs: 5](#_Toc154786704)

[6. SYSTEM ARCHITECTURE (MODULES, COMPONENTS, AND INTERACTIONS) 6](#_Toc154786705)

[6.1. User Authentication: 6](#_Toc154786706)

[6.1.1. Components: 6](#_Toc154786707)

[6.1.2. Interactions: 6](#_Toc154786708)

[6.2. Server: 6](#_Toc154786709)

[6.2.1. Components 6](#_Toc154786710)

[6.2.2. Interactions: 6](#_Toc154786711)

[6.3. Client: 6](#_Toc154786712)

[6.3.1. Components: 6](#_Toc154786713)

[6.3.2. Interactions: 6](#_Toc154786714)

[6.4. System Architecture Diagram 7](#_Toc154786715)

[7. Communication Protocol 8](#_Toc154786716)

[7.1. Peer 2 Peer 8](#_Toc154786717)

[7.1.1. HMMZP Protocol 8](#_Toc154786718)

[7.2. Client Server 9](#_Toc154786719)

[7.2.1. HMMZC Protocol 9](#_Toc154786720)

[8. Functionalities 10](#_Toc154786721)

[8.1. Friend List : 10](#_Toc154786722)

[8.1.1. User-Friendly Interface 10](#_Toc154786723)

[8.1.2. Status Indicators 10](#_Toc154786724)

[8.1.3. Dynamic Population 10](#_Toc154786725)

[8.2. Nearby Online Peers : 11](#_Toc154786726)

[8.2.1. Geolocation Integration: 11](#_Toc154786727)

[8.2.2. Real-time Updates: 11](#_Toc154786728)

[8.2.3. User Privacy Controls: 11](#_Toc154786729)

[8.3. Messages Notifications : 12](#_Toc154786730)

[8.3.1. Real-time Notifications: 12](#_Toc154786731)

[8.3.2. Convenient Access: 12](#_Toc154786732)

[8.3.3. Message Preview: 12](#_Toc154786733)

[8.4. Global Chat Rooms List 13](#_Toc154786734)

[8.4.1. Dynamic Ranking: 13](#_Toc154786735)

[8.4.2. Real-time Participant Count: 13](#_Toc154786736)

[9. User Authentication 14](#_Toc154786737)

[9.1. Sign up: 14](#_Toc154786738)

[9.2. Log in: 14](#_Toc154786739)

[9.3. Sequence Diagram 15](file:///C:\programs\Mozakra\Fall%2023\Networks\Project\Project%20Doc\Networks%20whole%20doc.docx#_Toc154786751)

[10. GitHub Repository 16](#_Toc154786752)

[11. Phase 2 Summary 16](#_Toc154786753)

[12. Files 16](#_Toc154786754)

[12.1. Requirements.txt 16](#_Toc154786755)

[12.2. Peer.py 17](#_Toc154786756)

[12.3. Db.py 24](#_Toc154786757)

[12.4. Registry.py 26](#_Toc154786758)

[12. Screenshots of working application 32](#_Toc154786770)

[12.1. The start of the application 32](#_Toc154786771)

[12.2. Creation and log in of account 33](#_Toc154786772)

[12.3. Log out 33](#_Toc154786773)

[12.4. Log in directly and testing password 34](#_Toc154786774)

[12.5. Testing Search and Chat with offline user 34](#_Toc154786775)

[12.6. Testing password length check and confirmation 35](#_Toc154786776)

[12.7. Searching for online clients 35](#_Toc154786777)

[13. Phase 3 Summary 35](#_Toc154786778)

[13.1. New added file 36](#_Toc154786779)

[13.1.1. message\_formatter.py 36](#_Toc154786780)

[13.2. Modified Files 37](#_Toc154786781)

[13.2.1. registry.py 37](#_Toc154786782)

[13.2.2. peer.py 45](#_Toc154786783)

[13.2.3. db.py 63](#_Toc154786784)

[14. Screenshot of working code 67](#_Toc154786785)

# Summary

In an era dominated by digital connectivity, our proposed Peer-to-Peer Chatting Application revolutionizes the way individuals interact and communicate. This innovative platform seamlessly integrates chatting rooms, private messaging, and a dynamic friend list, offering users a comprehensive and personalized communication experience.

We have always encountered the Traditional messaging applications that often lack the versatility to cater and diverse communication needs , Users find themselves navigating between multiple platforms for group discussions, private conversations, and maintaining connections.

Our Peer-to-Peer Chatting Application addresses this gap by amalgamating chatting rooms, private messaging, and a friend list into a unified, user-friendly interface Command Line . This multifaceted solution provides a one-stop-shop for all communication requirements, streamlining the user experience.

Our users should expect from the application to easily communicate and send private messages to fellow friends , join a chatting room by typing its unique name (ID) , create a Chatting room that others can join , receive private messages from friends and messages inside the chatting room.

Our Peer-to-Peer Chatting Application represents a leap forward in the realm of digital communication. By seamlessly blending chatting rooms, private messaging, and friend list functionalities, we offer users a cohesive and enjoyable communication experience. Embrace the future of peer-to-peer interaction with our innovative and user-centric solution.

# Objective

Peer to Peer application Aimed to be user-friendly for sending and receiving Private messages and Creating and joining Chat Rooms .

Users can anticipate seamless communication and private messaging within the application, allowing them to easily interact with friends. They can join chatting rooms by entering the unique name (ID), create their own chatting rooms for others to join, and receive private messages from friends, as well as messages within the chatting rooms.

Also they can see who is online in a friendly looking Friend list and Global online Peers List.

1. Scope of the Application:

The application aims to provide a comprehensive and user-friendly platform for communication and social interaction. The primary scope encompasses the following features and functionalities:

* 1. User Registration and Authentication:

Users can create accounts, providing essential information for authentication.

Robust authentication mechanisms ensure secure access to the application.

## Private Messaging:

Users can send and receive private messages to and from their friends.

End-to-end encryption ensures the confidentiality and security of private conversations.



## Friend List:

Users can maintain a dynamic friend list, displaying the online or offline status of their friends.

The friend list facilitates quick access to private messaging and enhances user connections.

## Chatting Rooms:

Users can join existing chatting rooms by entering unique names (IDs) or create their own rooms.

Chatting rooms support real-time conversations, promoting group interactions and discussions.

## ***Nearby Global Online Peers***:

The application provides a list of online peers in close proximity, fostering spontaneous connections among users sharing the same location.



## Messages Notification Tray:

A notification tray informs users of incoming private messages, ensuring timely awareness and efficient message management.

## Most Populated Global Chat Rooms:

Users can explore and join global chat rooms based on popularity and diverse topics, fostering a sense of community on a global scale.

## Real-time Updates:

The application offers real-time updates for various features, including online status changes, message notifications, and room participation.

# Timeline

The expected Project timeline should go as follow , we will Deliver the project in 4 Phases,

* 1. Phase 1 Project Planning and Design :

Delivery date : 7/12/2023

Deliverables : Project Proposal , system architecture, components, and communication protocols

* 1. Phase 2 Basic Client-Server Setup :

Delivery date : 21/12/2023

Deliverables : Basic Client-Server application and TCP connection for user authentication

## Phase 3 Peer-to-Peer Architecture and Chat Rooms :

Delivery date : 28/12/2023

Deliverables : Modified server to support peer-to-peer connections. Implement of chat room creation and joining functionalities , basic text-based communication within chat rooms.

## Phase 4 One-to-One Chatting and Protocol Optimization :

Delivery date : 4/01/2023

Deliverables : Implement one-to-one chat functionality , Optimize TCP and UDP usage based on different scenarios, Conduct performance testing and document results , we will Finalize the user interface and overall user experience

# Cost Estimation

## Cost Analysis

This will cover all the expected costs of developing, maintaining, Hosting, and marketing the application.

## Development Cost:

The application will be finished in one month by 4 developers, each get paid $20/hour and work 160 hours/month:

* Development cost per developer: $20/hour \* 160 hours = $3,200
* Total cost for four developers: $3,200 \* 4 = $12,800

## Maintenance Costs:

The application will need 2 developers to work 20 additional hours per month for updates and bug fixes.

* Cost per Developer = $20/hour \* 20 hours = $400
* Total cost for 2 developers = 2 Developers \* $400/Developer = $800

## Hosting Costs:

The application will be hosted on a cloud with a database like AWS, Azure, etc... to store our peers on, costs will increase will how many users are registered to the application.

* Costs for 1000 users or less: $20/month.
* Costs for 10000 users or less: $80/month.
* Costs for more than 10000 users: $150/month.

## Marketing Costs:

The application will need ads on social media platforms so that people can get to know the application more and start to use it.

* Total Marketing Costs: $500/month.

# SYSTEM ARCHITECTURE (MODULES, COMPONENTS, AND INTERACTIONS)

## User Authentication:

Description: this module manages the authentication of users that are trying to login into the application.

### Components:

* Authentication Controller

### Interactions:

Clients interact with the user authentication module for registration and login. The user credentials are stored and verified in the database.

## Server:

Description: this module is responsible for handling client connections, creating chat rooms either private chat rooms or group chats.

* + 1. Components:
* Chat Room Controller
* Private Chat Room Controller
* User Controller

### Interactions:

* clients connect to the server.
* Server manages connecting users and creating chat rooms.
* Server manages messages coming into and out of chat rooms
* Server interacts with user controller to create or delete user accounts through the authentication controller

## Client:

Description: this modules contains the user interface and the peer application.

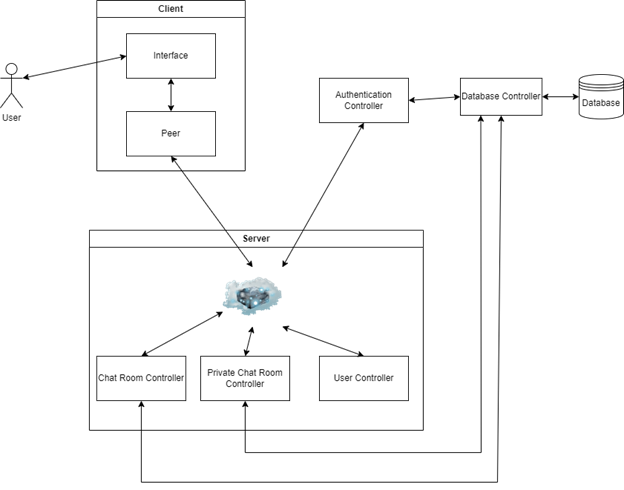
### Components:

* Peer
* Interface

### Interactions:

* The user interacts with the interface
* The interface interacts with the peer application.
* The user interacts with the peer application through the interface.

## System Architecture Diagram



# Communication Protocol

## Peer 2 Peer

### HMMZP Protocol

* **Structure of Sent message**:

Operation number: (number from 1 to x to indicate which operation to make)

Date: (date when message was sent)

Time: (time when message was sent)

Source Peer: (peer number message coming from)

Destination Peer: (peer number message going to)

Source Socket: (socket number of peer message coming from)

Destination Socket: (socket number of peer message going to)

Username: (username)

* **Structure of Response message:**
  + Operation Success: (1 for success or 0 for failure)
  + Operation Success Details: (details about operation success or failure)
  + Date: (date when message was sent)
  + Time: (time when message was sent)
  + Source Peer: (peer number message coming from)
  + Destination Peer: (peer number message going to)
  + Source Socket: (socket number of peer message coming from)
  + Destination Socket: (socket number of peer message going to)
  + Data Format: (format of the incoming data)
  + Body of Response: (incoming data)

## Client Server

### HMMZC Protocol

* **Structure of Sent message:**

Operation number: (number from 1 to x to indicate which operation to make)

Date: (date when message was sent)

Time: (time when message was sent)

Source Host: (Source Ip address )

Destination Host : (Destination Ip address )

Source Socket: (socket number of Host message coming from)

Destination Socket: (socket number of Host message going to)

Username: (username)

* **Structure of Response message:**

Operation Success : (1 for success or 0 for failure)

Server Response Code : (200 ok , 404 Not Found,….etc)

Operation Success Details: (details about operation success or failure)

Date: (date when message was sent)

Time: (time when message was sent)

Source Host : (Source Ip address)

Destination Host: (Destination Ip address)

Source Socket: (socket number of Host message coming from)

Destination Socket: (socket number of Host message going to)

Data Format: (format of the incoming data)

Body of Response: (incoming data)

# Functionalities

## Friend List :

In our communication application, we introduce a robust and user-centric "Friend List" functionality designed to enhance user experience and foster real-time connectivity. This feature elegantly displays the online or offline status of each friend, providing users with a quick and intuitive way to stay informed about their network's availability.

* + 1. User-Friendly Interface**:**

The Friend List is seamlessly integrated into the application's interface, ensuring a straightforward and intuitive user experience. Each friend is represented in a structured and easily navigable format.

* + 1. Status Indicators**:**

Users can effortlessly identify the online or offline status of each friend through clear and visually distinct indicators , Status indicators are designed to be easily recognizable, with green indicating online status and red indicating offline status.

* + 1. Dynamic Population**:**

The Friend List dynamically populates based on the user's network, ensuring real-time updates on the status of each friend , The list accommodates scalability, allowing users to expand their network without sacrificing performance.

## Nearby Online Peers :

In our ongoing commitment to providing a dynamic and inclusive communication platform, we are thrilled to introduce the "Nearby Online Peers List" functionality. This feature enriches user interaction by presenting a curated list of online peers in close proximity, fostering real-time connections and community engagement.

### Geolocation Integration:

The Nearby Online Peers List leverages geolocation data to identify and display online peers in the user's vicinity.

This integration ensures relevance and immediacy in connecting with nearby users.

### Real-time Updates:

The list dynamically updates to reflect the current online status of peers in proximity, enabling users to discover and engage with nearby individuals instantaneously.

Users can expect a continuously refreshed list, maintaining accuracy in the representation of nearby online peers.

### User Privacy Controls:

Our implementation prioritizes user privacy by incorporating robust controls.

Users have the ability to customize their visibility within the nearby peers list, ensuring a secure and comfortable experience.

## Messages Notifications :

Continuing our dedication to enhancing the user experience, we are pleased to introduce the "Private Message Notification Tray." This pivotal functionality ensures that users are promptly informed and can seamlessly manage incoming private messages, fostering efficient and uninterrupted communication within our platform.

### Real-time Notifications:

The Notification Tray provides real-time alerts for every private message directed towards the user.

Users are instantly notified, eliminating delays and ensuring timely responses to incoming messages.

### Convenient Access:

The tray is strategically positioned for easy accessibility, allowing users to quickly view and respond to private messages without interrupting their current activities.

Notifications are designed to be unobtrusive yet attention-grabbing for an optimal user experience.

### Message Preview:

Users receive a concise preview of incoming private messages within the notification, enabling them to gauge the content and urgency without navigating away from their current screen.

This feature facilitates efficient message prioritization.

## Global Chat Rooms List

In our continuous pursuit of creating a vibrant and engaging communication environment, we are delighted to introduce the "Most Populated Global Chat Rooms List." This feature provides users with an avenue to connect with a diverse and expansive community, facilitating dynamic conversations on a global scale

### Dynamic Ranking:

The list dynamically ranks and displays the most populated global chat rooms, offering users insights into the trending and active conversations based on Room Population.

Rooms are categorized based on popularity, ensuring users can easily identify and join discussions aligned with their interests.

### Real-time Participant Count:

Users gain real-time visibility into the number of participants within each global chat room.

This feature assists users in making informed decisions about which rooms to join based on their preferred level of engagement.

# User Authentication

User Authentication can be shown below with a sequence diagram.

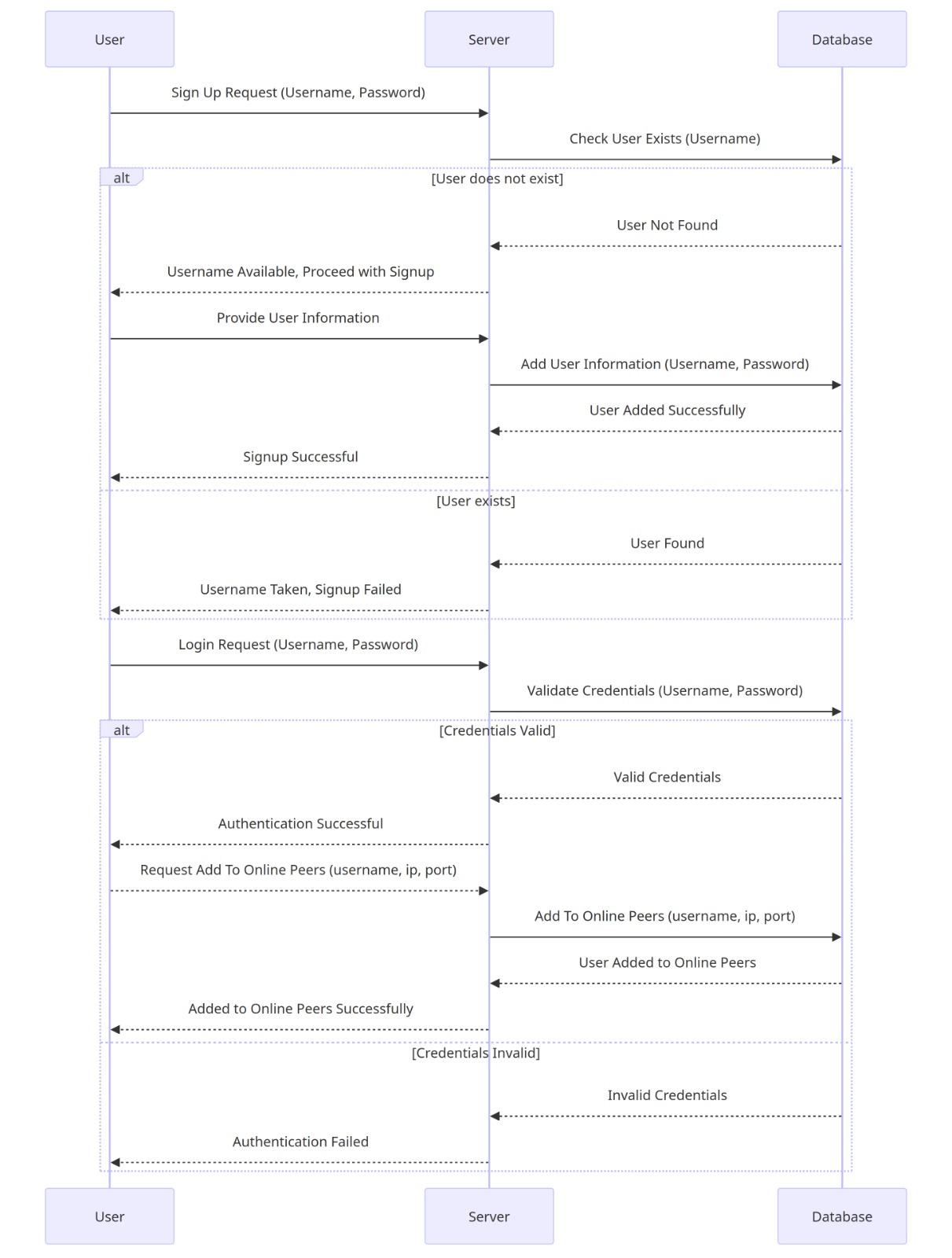
## Sign up:

* User will send a sign-up request with his username and password.
* Server will check if username exists.
* If it does not exist: Sign up will be successful.
* If exists: Sign up will not be successful.

## Log in:

* User will send a log in request with his username and password.
* Server will validate the given data to see if the username belongs to the password.
* If validated: Log in will be successful, then the username, IP address and port number to online peers table in database.

If not validated: Log in will not be successful





## Sequence Diagram

# GitHub Repository

<https://github.com/MazenTayseer/Chatting-P2P-Application>

# Phase 2 Summary

We continue the development procedure of Peer 2 Peer chatting application by making a simple client – server communication over TCP , Login / Sign up , begin chat , search for another client to chat with , and log out if logged in , this was achieved by Making Peer.py , registry.py , db.py , and requirements.txt that contains all the PIP install packages in order for the application to run

Beginning by the Peer.py file , this file contains the client side of the App , the user can communicate through the command line who is user friendly and colorful for easily recognizing error messages or successful procedures.

Registry.py is responsible for the server side of the application that receives the requests from the client and responds to it moreover, it has access to the db.py which stores and retrieves the data from the database , so that it can access encrypted passwords and check if user is already exist.

Lastly the db.py file contain the necessary functions to connect to the database and respond to the Registry.py file that have requests from the client side.

# Files

## Requirements.txt

A screenshot of a computer

Description automatically generated

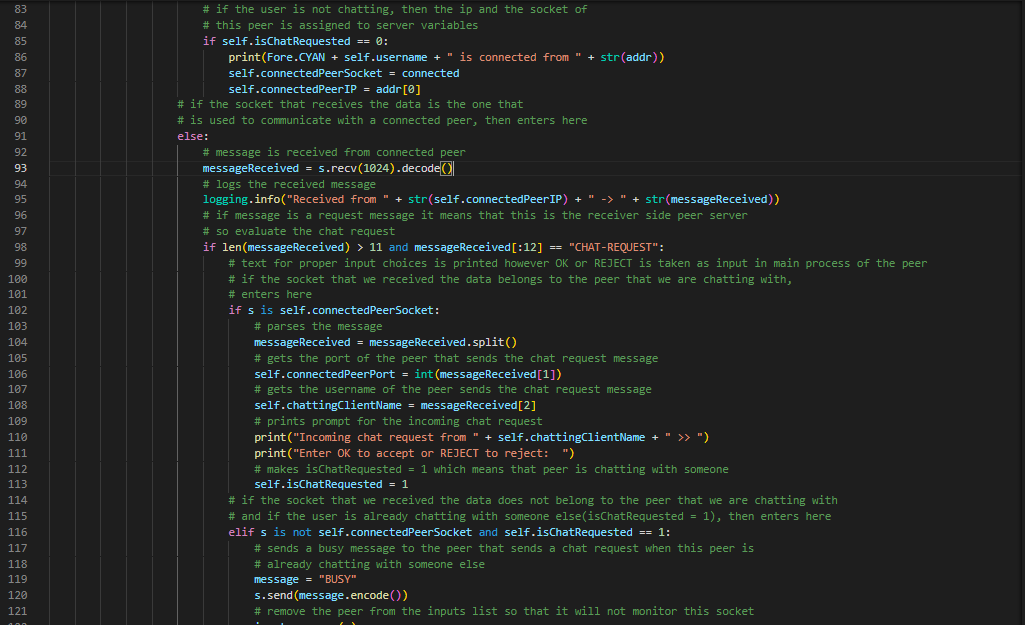
## Peer.py

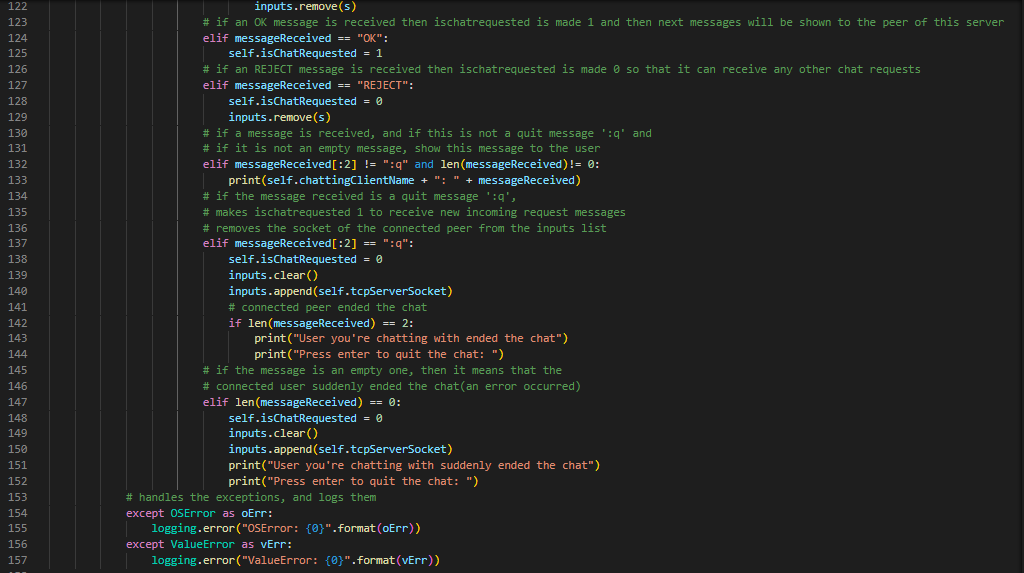
A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated





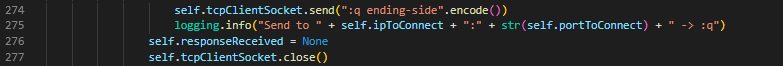
A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

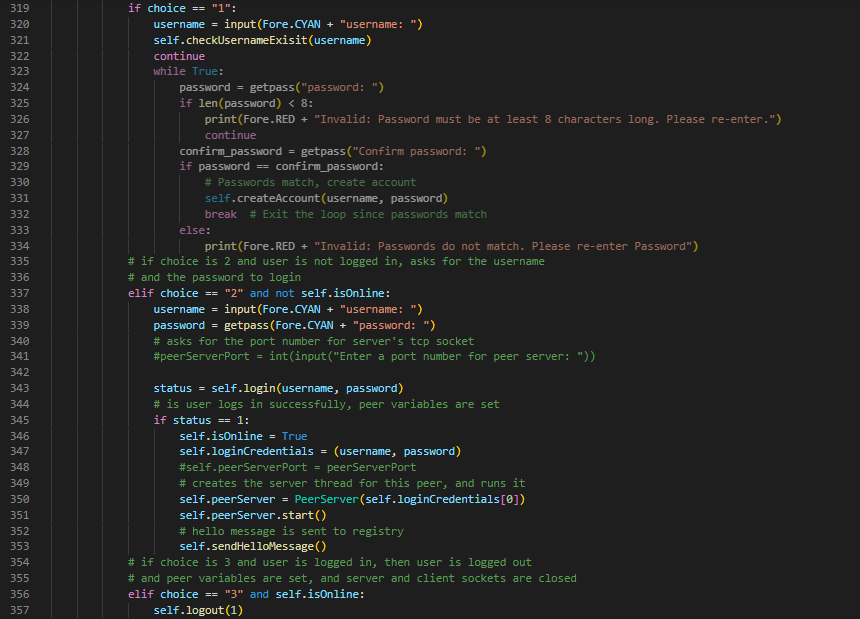
Description automatically generated

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated



A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

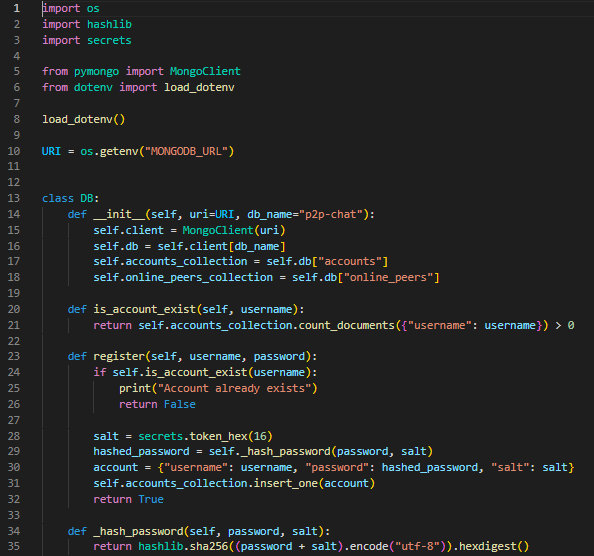
A screen shot of a computer program

Description automatically generated

A screen shot of a computer screen

Description automatically generated

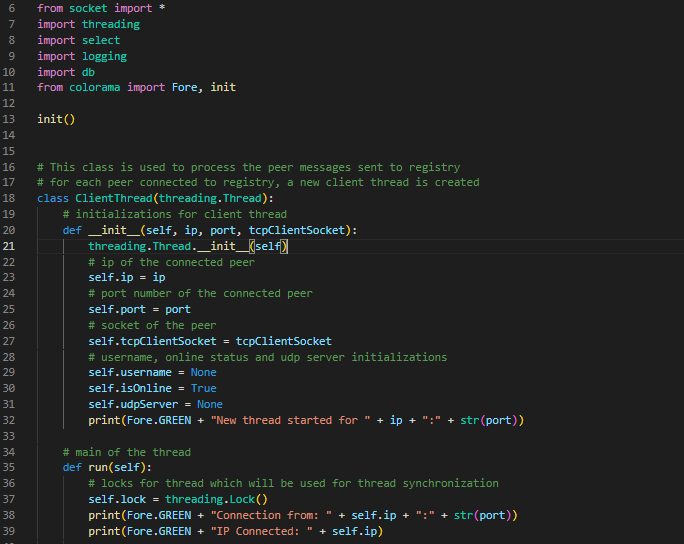
## Db.py



A screen shot of a computer program

Description automatically generated

## Registry.py



A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

A computer screen shot of a program

Description automatically generated

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated



# Screenshots of working application

## The start of the application

A screenshot of a computer program

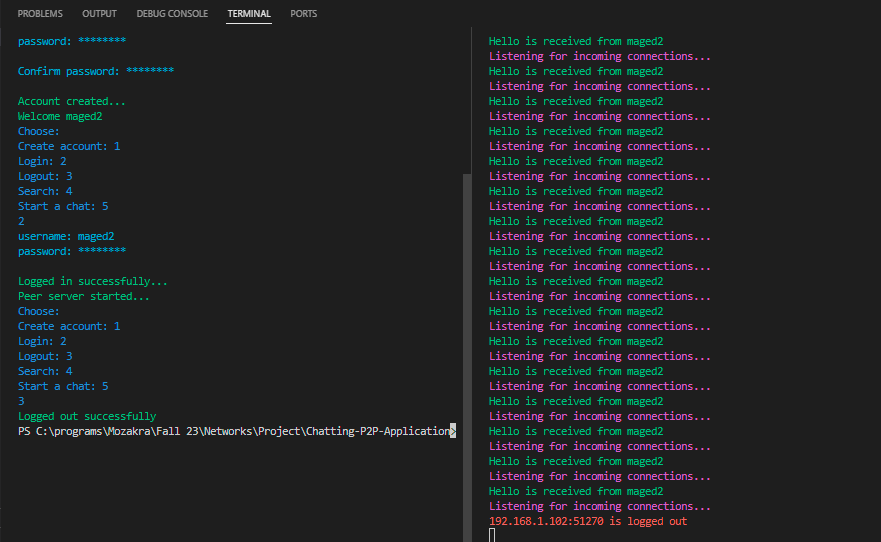
Description automatically generated

## Creation and log in of account

A screenshot of a computer

Description automatically generated

## Log out

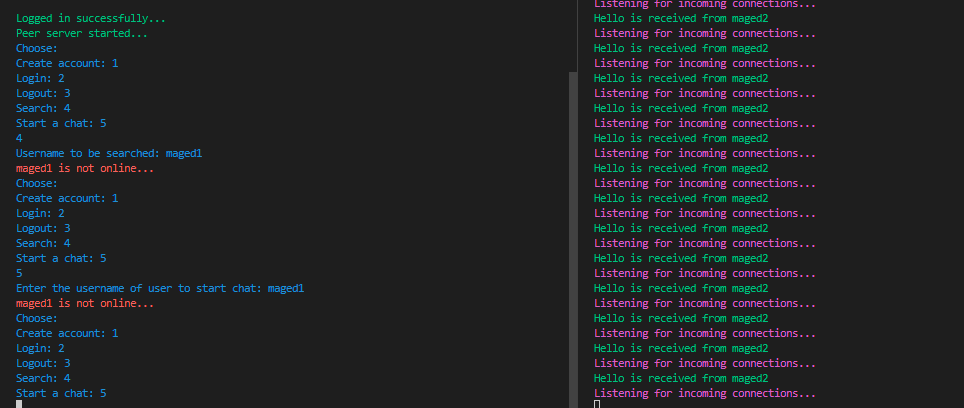


## Log in directly and testing password

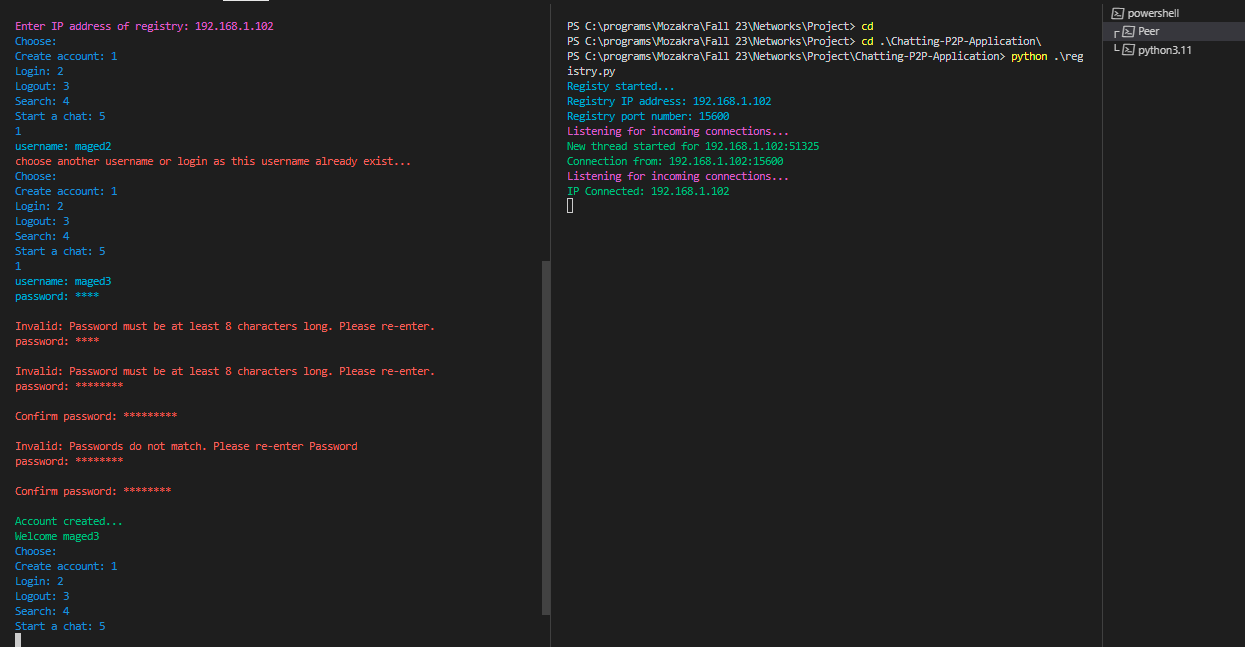
A screenshot of a computer screen

Description automatically generated

## Testing Search and Chat with offline user



## Testing password length check and confirmation



## Searching for online clients

A screen shot of a computer screen

Description automatically generated

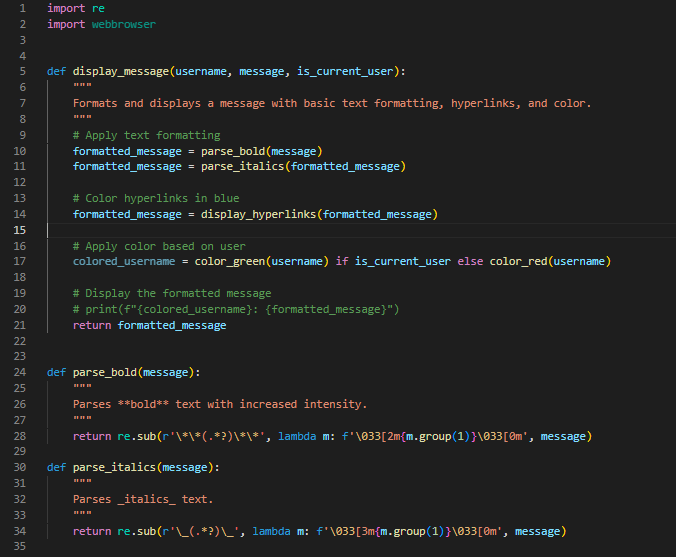
# Phase 3 Summary

We implemented and integrated the peer to peer architecture with the capabilities for the user to create a chatting room with a unique ID , join a chatting room using the room ID , handled the changes that could happen when a user is in a chatting room like leaving suddenly or disconnecting or when a user joins the room , by adding message\_formatter.py which handles the message format and styles in chat rooms.

We also modified the db , registry and peer.py files in order to achieve phase 3 objective

## New added file

### message\_formatter.py



A screen shot of a computer program

Description automatically generated

## Modified Files

### registry.py

from socket import \*

import threading

import select

import logging

import db

from colorama import Fore, init

# This class is used to process the peer messages sent to registry

# for each peer connected to registry, a new client thread is created

class ClientThread(threading.Thread):

    # initializations for client thread

    def \_\_init\_\_(self, ip, port, tcpClientSocket):

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

        # ip of the connected peer

        self.ip = ip

        # port number of the connected peer

        self.port = port

        # socket of the peer

        self.tcpClientSocket = tcpClientSocket

        # username, online status and udp server initializations

        self.username = None

        self.isOnline = True

        self.udpServer = None

        print(Fore.GREEN +"New thread started for " + ip + ":" + str(self.port))

    # main of the thread

    def run(self):

        # locks for thread which will be used for thread synchronization

        self.lock = threading.Lock()

        print(Fore.MAGENTA +"Connection from: " + self.ip + ":" + str(port))

        print(Fore.GREEN +"IP Connected: " + self.ip)

        while True:

            try:

                # waits for incoming messages from peers

                message = self.tcpClientSocket.recv(1024).decode().split()

                # if message is empty or something went wrong

                if not message:

                    print(Fore.LIGHTRED\_EX+"No message received, client may have disconnected.")

                    break

                logging.info("Received from " + self.ip + ":" + str(self.port) + " -> " + " ".join(message))

                if message[0] == "CHECK-USERNAME-EXISTS":

                    if db.is\_account\_exist(message[1]):

                        response = "username-exist"

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

                    else:

                        response = "username-not-found"

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

                #   JOIN    #

                if message[0] == "JOIN":

                    # join-exist is sent to peer,

                    # if an account with this username already exists

                    if db.is\_account\_exist(message[1]):

                        response = "join-exist"

                        print("From-> " + self.ip + ":" + str(self.port) + " " + response)

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

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

                    # join-success is sent to peer,

                    # if an account with this username is not exist, and the account is created

                    else:

                        db.register(message[1], message[2])

                        response = "join-success"

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

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

                #   LOGIN    #

                elif message[0] == "LOGIN":

                    # login-account-not-exist is sent to peer,

                    # if an account with the username does not exist

                    if not db.is\_account\_exist(message[1]):

                        response = "login-account-not-exist"

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

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

                    # login-online is sent to peer,

                    # if an account with the username already online

                    elif db.is\_account\_online(message[1]):

                        response = "login-online"

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

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

                    # login-success is sent to peer,

                    # if an account with the username exists and not online

                    else:

                        # retrieves the account's password, and checks if the one entered by the user is correct

                        is\_password\_correct = db.verify\_password(message[1], message[2])

                        # if password is correct, then peer's thread is added to threads list

                        # peer is added to db with its username, port number, and ip address

                        if is\_password\_correct:

                            self.username = message[1]

                            self.lock.acquire()

                            try:

                                tcpThreads[self.username] = self

                            finally:

                                self.lock.release()

                            db.user\_login(message[1], self.ip, message[2])

                            # login-success is sent to peer,

                            # and a udp server thread is created for this peer, and thread is started

                            # timer thread of the udp server is started

                            response = "login-success"

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

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

                            self.udpServer = UDPServer(self.username, self.tcpClientSocket)

                            self.udpServer.start()

                            self.udpServer.timer.start()

                        # if password not matches and then login-wrong-password response is sent

                        else:

                            response = "login-wrong-password"

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

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

                #   LOGOUT  #

                elif message[0] == "LOGOUT":

                    # if user is online,

                    # removes the user from onlinePeers list

                    # and removes the thread for this user from tcpThreads

                    # socket is closed and timer thread of the udp for this

                    # user is cancelled

                    if len(message) > 1 and message[1] != None and db.is\_account\_online(message[1]):

                        db.user\_logout(message[1])

                        self.lock.acquire()

                        try:

                            if message[1] in tcpThreads:

                                del tcpThreads[message[1]]

                        finally:

                            self.lock.release()

                        print(self.ip + ":" + str(self.port) + " is logged out")

                        self.tcpClientSocket.close()

                        self.udpServer.timer.cancel()

                        break

                    else:

                        self.tcpClientSocket.close()

                        break

                #   SEARCH  #

                elif message[0] == "SEARCH":

                    # checks if an account with the username exists

                    if db.is\_account\_exist(message[1]):

                        # checks if the account is online

                        # and sends the related response to peer

                        if db.is\_account\_online(message[1]):

                            peer\_info = db.get\_peer\_ip\_port(message[1])

                            response = "search-success " + peer\_info[0] + ":" + peer\_info[1]

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

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

                        else:

                            response = "search-user-not-online"

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

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

                    # enters if username does not exist

                    else:

                        response = "search-user-not-found"

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

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

                elif message[0] == "CREATE":

                    # CREATE-exist is sent to peer,

                    # if an room with this username already exists

                    if db.is\_room\_exist(message[1]):

                        response = "room-exist"

                        print("From-> " + self.ip + ":" + str(self.port) + " " + response)

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

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

                    else:

                        db.register\_room(message[1])

                        response = "creation-success"

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

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

                elif message[0] == "JOINROOM":

                    # checks if an account with the username exists

                    if db.is\_room\_exist(message[1]):

                        # checks if the room exists

                        # and sends the related response to peer

                        id, peers = db.get\_room\_peers(message[1])

                        peers.append(message[2])

                        peers = list(set(peers))

                        db.update\_room(id, peers)

                        response = "success " + str(peers)

                        print(response)

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

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

                    # enters if username does not exist

                    else:

                        response = "search-fail"

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

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

                elif message[0] == "UPDATE":

                        id, peers = db.get\_room\_peers(message[1])

                        response = "updated " + str(peers)

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

                elif message[0] == "EXIT":

                        db.remove\_peer(message[1], message[2])

                        response = "SUCCESS"

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

            except OSError as oErr:

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

    # function for resettin the timeout for the udp timer thread

    def resetTimeout(self):

        self.udpServer.resetTimer()

# implementation of the udp server thread for clients

class UDPServer(threading.Thread):

    # udp server thread initializations

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

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

        self.username = username

        # timer thread for the udp server is initialized

        self.timer = threading.Timer(3, self.waitHelloMessage)

        self.tcpClientSocket = clientSocket

    # if hello message is not received before timeout

    # then peer is disconnected

    def waitHelloMessage(self):

        if self.username is not None:

            db.user\_logout(self.username)

            if self.username in tcpThreads:

                del tcpThreads[self.username]

        self.tcpClientSocket.close()

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

    # resets the timer for udp server

    def resetTimer(self):

        self.timer.cancel()

        self.timer = threading.Timer(3, self.waitHelloMessage)

        self.timer.start()

# tcp and udp server port initializations

print(Fore.MAGENTA +"Registy started...")

port = 15600

portUDP = 15500

# db initialization

db = db.DB()

# gets the ip address of this peer

# first checks to get it for windows devices

# if the device that runs this application is not windows

# it checks to get it for macos devices

hostname=gethostname()

try:

    host=gethostbyname(hostname)

except gaierror:

    import netifaces as ni

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

print(Fore.MAGENTA +"Registry IP address: " + host)

print(Fore.MAGENTA +"Registry port number: " + str(port))

# onlinePeers list for online account

onlinePeers = {}

# accounts list for accounts

accounts = {}

# tcpThreads list for online client's thread

tcpThreads = {}

#tcp and udp socket initializations

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

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

tcpSocket.bind((host,port))

udpSocket.bind((host,portUDP))

tcpSocket.listen(5)

# input sockets that are listened

inputs = [tcpSocket, udpSocket]

# log file initialization

#logging.basicConfig(filename="src/registry.log", level=logging.INFO)

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

while inputs:

    print(Fore.YELLOW +"Listening for incoming connections...")

    # monitors for the incoming connections

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

    for s in readable:

        # if the message received comes to the tcp socket

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

        if s is tcpSocket:

            tcpClientSocket, addr = tcpSocket.accept()

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

            newThread.start()

        # if the message received comes to the udp socket

        elif s is udpSocket:

            # received the incoming udp message and parses it

            message, clientAddress = s.recvfrom(1024)

            message = message.decode().split()

            # checks if it is a hello message

            if message[0] == "HELLO":

                # checks if the account that this hello message

                # is sent from is online

                if message[1] in tcpThreads:

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

                    tcpThreads[message[1]].resetTimeout()

                    print(Fore.GREEN +"Hello is received from " + message[1])

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

# registry tcp socket is closed

tcpSocket.close()

### peer.py

from socket import \*

import threading

import select

import logging

from getpass4 import getpass

from colorama import init, Fore

from utils.message\_formatter import \*

import db

# Server side of peer

class PeerServer(threading.Thread):

    # Peer server initialization

    def \_\_init\_\_(self, username, peerServerPort, roomServerPort):

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

        # keeps the username of the peer

        self.username = username

        # tcp socket for peer server

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

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

        # port number of the peer server

        self.peerServerPort = peerServerPort

        self.roomServerPort = roomServerPort

        # if 1, then user is already chatting with someone

        # if 0, then user is not chatting with anyone

        self.isChatRequested = 0

        # keeps the socket for the peer that is connected to this peer

        self.connectedPeerSocket = None

        # keeps the ip of the peer that is connected to this peer's server

        self.connectedPeerIP = None

        # keeps the port number of the peer that is connected to this peer's server

        self.connectedPeerPort = None

        # online status of the peer

        self.isOnline = True

        # keeps the username of the peer that this peer is chatting with

        self.chattingClientName = None

        self.chat = 0

        self.room = 0

    # main method of the peer server thread

    def run(self):

        print(Fore.GREEN + "Peer server started...")

        # gets the ip address of this peer

        # first checks to get it for windows devices

        # if the device that runs this application is not windows

        # it checks to get it for macos devices

        hostname=gethostname()

        try:

            self.peerServerHostname=gethostbyname(hostname)

        except gaierror:

            import netifaces as ni

            self.peerServerHostname = ni.ifaddresses('en0')[ni.AF\_INET][0]['addr']

        # ip address of this peer

        self.peerServerHostname = 'localhost'

        self.udpServerSocket.bind((self.peerServerHostname, self.roomServerPort))

        #socket initialization for chatting

        self.tcpServerSocket.bind((self.peerServerHostname, self.peerServerPort))

        self.tcpServerSocket.listen(4)

        # inputs sockets that should be listened

        inputs = [self.udpServerSocket, self.tcpServerSocket]

        # server listens as long as there is a socket to listen in the inputs list and the user is online

        while inputs and self.isOnline:

            # monitors for the incoming connections

            try:

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

                # If a server waits to be connected enters here

                for s in readable:

                    # if the socket that is receiving the connection is

                    # the tcp socket of the peer's server, enters here

                    if s is self.tcpServerSocket and self.room==0:

                        # accepts the connection, and adds its connection socket to the inputs list

                        # so that we can monitor that socket as well

                        connected, addr = s.accept()

                        connected.setblocking(0)

                        inputs.append(connected)

                        # if the user is not chatting, then the ip and the socket of

                        # this peer is assigned to server variables

                        if self.isChatRequested == 0:

                            print(Fore.CYAN + self.username + " is connected from " + str(addr))

                            self.connectedPeerSocket = connected

                            self.connectedPeerIP = addr[0]

                    # Here we handle messages from rooms

                    elif s is self.udpServerSocket and self.room == 1:

                        #socket initializatoion for rooms

                        #self.udpServerSocket.bind((self.peerServerHostname, self.peerServerPort))

                        while(1):

                            data, address = self.udpServerSocket.recvfrom(1024)

                            messageReceived = data.decode()

                            print(messageReceived)

                            if(self.room == 0):

                                break

                    # if the socket that receives the data is the one that

                    # is used to communicate with a connected peer, then enters here

                    elif self.room == 0:

                        # message is received from connected peer

                        messageReceived = s.recv(1024).decode()

                        # logs the received message

                        logging.info("Received from " + str(self.connectedPeerIP) + " -> " + str(messageReceived))

                        # if message is a request message it means that this is the receiver side peer server

                        # so evaluate the chat request

                        if len(messageReceived) > 11 and messageReceived[:12] == "CHAT-REQUEST" and self.room == 0:

                            # text for proper input choices is printed however OK or REJECT is taken as input in main process of the peer

                            # if the socket that we received the data belongs to the peer that we are chatting with,

                            # enters here

                            if s is self.connectedPeerSocket:

                                # parses the message

                                messageReceived = messageReceived.split()

                                # gets the port of the peer that sends the chat request message

                                self.connectedPeerPort = int(messageReceived[1])

                                # gets the username of the peer sends the chat request message

                                self.chattingClientName = messageReceived[2]

                                # prints prompt for the incoming chat request

                                print("Incoming chat request from " + self.chattingClientName + " >> ")

                                print("Enter OK to accept or REJECT to reject:  ")

                                # makes isChatRequested = 1 which means that peer is chatting with someone

                                self.isChatRequested = 1

                            # if the socket that we received the data does not belong to the peer that we are chatting with

                            # and if the user is already chatting with someone else(isChatRequested = 1), then enters here

                            elif s is not self.connectedPeerSocket and self.isChatRequested == 1:

                                # sends a busy message to the peer that sends a chat request when this peer is

                                # already chatting with someone else

                                message = "BUSY"

                                s.send(message.encode())

                                # remove the peer from the inputs list so that it will not monitor this socket

                                inputs.remove(s)

                        # if an OK message is received then ischatrequested is made 1 and then next messages will be shown to the peer of this server

                        elif messageReceived == "OK":

                            self.isChatRequested = 1

                        # if an REJECT message is received then ischatrequested is made 0 so that it can receive any other chat requests

                        elif messageReceived == "REJECT":

                            self.isChatRequested = 0

                            inputs.remove(s)

                        # if a message is received, and if this is not a quit message ':q' and

                        # if it is not an empty message, show this message to the user

                        elif messageReceived[:2] != ":q" and len(messageReceived)!= 0:

                            print(self.chattingClientName + ": " + messageReceived)

                        # if the message received is a quit message ':q',

                        # makes ischatrequested 1 to receive new incoming request messages

                        # removes the socket of the connected peer from the inputs list

                        elif messageReceived[:2] == ":q":

                            if(self.room == 1):

                                self.room = 0

                                #leave\_room()

                            else:

                                self.isChatRequested = 0

                                inputs.clear()

                                inputs.append(self.tcpServerSocket)

                                # connected peer ended the chat

                                if len(messageReceived) == 2:

                                    print(Fore.LIGHTRED\_EX+"User you're chatting with ended the chat")

                                    print(Fore.LIGHTRED\_EX+"Press enter to quit the chat: ")

                        # if the message is an empty one, then it means that the

                        # connected user suddenly ended the chat(an error occurred)

                        elif len(messageReceived) == 0:

                            self.isChatRequested = 0

                            inputs.clear()

                            inputs.append(self.tcpServerSocket)

                            print(Fore.LIGHTRED\_EX+"User you're chatting with suddenly ended the chat")

                            print(Fore.LIGHTRED\_EX+"Press enter to quit the chat: ")

            # handles the exceptions, and logs them

            except OSError as oErr:

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

            except ValueError as vErr:

                logging.error("ValueError: {0}".format(vErr))

# Client side of peer

class PeerClient(threading.Thread):

    # variable initializations for the client side of the peer

    def \_\_init\_\_(self, ipToConnect, portToConnect, username, peerServer, responseReceived, flag, room\_id ,room\_peers : list, registry\_name = "127.0.1.1"):

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

        # keeps the ip address of the peer that this will connect

        # ip address of the registry

        self.registryName = registry\_name

        #self.registryName = 'localhost'

        # port number of the registry

        self.registryPort = 15600

        self.ipToConnect = ipToConnect

        # keeps the username of the peer

        self.username = username

        # keeps the port number that this client should connect

        self.portToConnect = portToConnect

        # client side tcp socket initialization

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

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

        # keeps the server of this client

        self.peerServer = peerServer

        # keeps the phrase that is used when creating the client

        # if the client is created with a phrase, it means this one received the request

        # this phrase should be none if this is the client of the requester peer

        self.responseReceived = responseReceived

        # keeps if this client is ending the chat or not

        self.isEndingChat = False

        #flag to indicate room or normal chat

        self.flag = flag

        #RoomID

        self.room\_id = room\_id

        #list of room\_peers

        self.room\_peers = room\_peers

        self.isRoomEmpty = False

    def update\_peers(self):

        message = "UPDATE " + str(self.room\_id)

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

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

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

        list\_start = response.index('[')

        list\_end = response.index(']') + 1

        list\_string = response[list\_start:list\_end]

        response = response.split()

        response2 = eval(list\_string)

        self.room\_peers = response2

    def exit(self):

        #Need to access peerServerObject to get roomPortNo to remove from list

        port\_to\_remove = self.peerServer.roomServerPort

        #Then go to registry and remove him

        request = "EXIT " + str(self.room\_id) + " " + str(port\_to\_remove)

        self.tcpClientSocket.send(request.encode())

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

        #Display Message ("USERNAME Disconected")

        return response

    # main method of the peer client thread

    def run(self):

        if self.flag == '5':

            print(Fore.CYAN + "Peer client started...")

            # connects to the server of other peer

            print("Connecting to " + self.ipToConnect + ":" + str(self.portToConnect) + "...")

            self.tcpClientSocket.connect((self.ipToConnect, self.portToConnect))

            # if the server of this peer is not connected by someone else and if this is the requester side peer client then enters here

            if self.peerServer.isChatRequested == 0 and self.responseReceived is None:

                # composes a request message and this is sent to server and then this waits a response message from the server this client connects

                requestMessage = "CHAT-REQUEST " + str(self.peerServer.peerServerPort)+ " " + self.username

                # logs the chat request sent to other peer

                logging.info("Send to " + self.ipToConnect + ":" + str(self.portToConnect) + " -> " + requestMessage)

                # sends the chat request

                self.tcpClientSocket.send(requestMessage.encode())

                print("Request message " + requestMessage + " is sent...")

                # received a response from the peer which the request message is sent to

                self.responseReceived = self.tcpClientSocket.recv(1024).decode()

                # logs the received message

                logging.info("Received from " + self.ipToConnect + ":" + str(self.portToConnect) + " -> " + self.responseReceived)

                print("Response is " + self.responseReceived)

                # parses the response for the chat request

                self.responseReceived = self.responseReceived.split()

                # if response is ok then incoming messages will be evaluated as client messages and will be sent to the connected server

                if self.responseReceived[0] == "OK":

                    # changes the status of this client's server to chatting

                    self.peerServer.isChatRequested = 1

                    # sets the server variable with the username of the peer that this one is chatting

                    self.peerServer.chattingClientName = self.responseReceived[1]

                    # as long as the server status is chatting, this client can send messages

                    while self.peerServer.isChatRequested == 1:

                        # message input prompt

                        messageSent = input(self.username + ": ")

                        # sends the message to the connected peer, and logs it

                        self.tcpClientSocket.send(messageSent.encode())

                        logging.info("Send to " + self.ipToConnect + ":" + str(self.portToConnect) + " -> " + messageSent)

                        # if the quit message is sent, then the server status is changed to not chatting

                        # and this is the side that is ending the chat

                        if messageSent == ":q":

                            self.peerServer.isChatRequested = 0

                            self.isEndingChat = True

                            break

                    # if peer is not chatting, checks if this is not the ending side

                    if self.peerServer.isChatRequested == 0:

                        if not self.isEndingChat:

                            # tries to send a quit message to the connected peer

                            # logs the message and handles the exception

                            try:

                                self.tcpClientSocket.send(":q ending-side".encode())

                                logging.info("Send to " + self.ipToConnect + ":" + str(self.portToConnect) + " -> :q")

                            except BrokenPipeError as bpErr:

                                logging.error("BrokenPipeError: {0}".format(bpErr))

                        # closes the socket

                        self.responseReceived = None

                        self.tcpClientSocket.close()

                # if the request is rejected, then changes the server status, sends a reject message to the connected peer's server

                # logs the message and then the socket is closed

                elif self.responseReceived[0] == "REJECT":

                    self.peerServer.isChatRequested = 0

                    print("client of requester is closing...")

                    self.tcpClientSocket.send("REJECT".encode())

                    logging.info("Send to " + self.ipToConnect + ":" + str(self.portToConnect) + " -> REJECT")

                    self.tcpClientSocket.close()

                # if a busy response is received, closes the socket

                elif self.responseReceived[0] == "BUSY":

                    print("Receiver peer is busy")

                    self.tcpClientSocket.close()

            # if the client is created with OK message it means that this is the client of receiver side peer

            # so it sends an OK message to the requesting side peer server that it connects and then waits for the user inputs.

            elif self.responseReceived == "OK":

                # server status is changed

                self.peerServer.isChatRequested = 1

                # ok response is sent to the requester side

                okMessage = "OK"

                self.tcpClientSocket.send(okMessage.encode())

                logging.info("Send to " + self.ipToConnect + ":" + str(self.portToConnect) + " -> " + okMessage)

                print("Client with OK message is created... and sending messages")

                # client can send messsages as long as the server status is chatting

                while self.peerServer.isChatRequested == 1:

                    # input prompt for user to enter message

                    messageSent = input(self.username + ": ")

                    self.tcpClientSocket.send(messageSent.encode())

                    logging.info("Send to " + self.ipToConnect + ":" + str(self.portToConnect) + " -> " + messageSent)

                    # if a quit message is sent, server status is changed

                    if messageSent == ":q":

                        self.peerServer.isChatRequested = 0

                        self.isEndingChat = True

                        break

                # if server is not chatting, and if this is not the ending side

                # sends a quitting message to the server of the other peer

                # then closes the socket

                if self.peerServer.isChatRequested == 0:

                    if not self.isEndingChat:

                        self.tcpClientSocket.send(":q ending-side".encode())

                        logging.info("Send to " + self.ipToConnect + ":" + str(self.portToConnect) + " -> :q")

                    self.responseReceived = None

                    self.tcpClientSocket.close()

        elif self.flag == '7':

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

            print(Fore.GREEN+"Joined Room Successfully ...")

            while True :

                self.update\_peers()

                if (not self.room\_peers):

                    break

                message = input(f"{color\_green(self.username)}: ")

                message = f"\n{self.username}: {message}"

                self.update\_peers()

                if (len(message) != 0 and message.split()[1] == ":q"):

                    if self.exit() == "SUCCESS":

                        message = f"{self.username} Disconnected !"

                        for peer in self.room\_peers:

                            self.udpClientSocket.sendto(message.encode(), (self.ipToConnect, int(peer)))

                        break

                else:

                    for peer in self.room\_peers:

                        if int(peer) != self.peerServer.roomServerPort:

                            formatted\_message = display\_message(self.username, message, True)

                            self.udpClientSocket.sendto(formatted\_message.encode(), (self.ipToConnect, int(peer)))

                        # self.tcpClientSocket.connect((self.ipToConnect, int(peer)))

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

                        # self.tcpClientSocket.close()

            print(Fore.LIGHTRED\_EX+"Chat Ended!")

            self.flag = None

# main process of the peer

class peerMain:

    # peer initializations

    def \_\_init\_\_(self):

        # ip address of the registry

        self.registryName = input(Fore.MAGENTA + "Enter IP address of registry: ")

        #self.registryName = '127.0.1.1'

        # port number of the registry

        self.registryPort = 15600

        # tcp socket connection to registry

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

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

        # initializes udp socket which is used to send hello messages

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

        # udp port of the registry

        self.registryUDPPort = 15500

        # login info of the peer

        self.loginCredentials = (None, None)

        # online status of the peer

        self.isOnline = False

        # server port number of this peer

        self.peerServerPort = None

        #server port for rooms

        self.roomServerPort = None

        # server of this peer

        self.peerServer = None

        # client of this peer

        self.peerClient = None

        # timer initialization

        self.timer = None

        choice = "0"

        # log file initialization

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

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

        while choice != "3":

            # menu selection prompt

            choice = input(Fore.BLUE +"Choose: \nCreate account: 1\nLogin: 2\nLogout: 3\nSearch: 4\nStart a chat: 5\nCreate Chatroom: 6\nJoin a room: 7\n")

            # if choice is 1, creates an account with the username

            # and password entered by the user

            if choice == "1":

                username = input(Fore.CYAN +"username: ")

                if self.checkAccount(username):

                    continue

                while True:

                    password = getpass(Fore.CYAN +"password: ")

                    if len(password) < 8:

                        print(Fore.RED + "Invalid: Password must be at least 8 characters long. Please re-enter.")

                        continue

                    confirm\_password = getpass(Fore.CYAN +"Confirm password: ")

                    if password == confirm\_password:

                            # Passwords match, create account

                        self.createAccount(username, password)

                        break  # Exit the loop since passwords match

                    else:

                        print(Fore.RED + "Invalid: Passwords do not match. Please re-enter Password")

                # if choice is 2 and user is not logged in, asks for the username

            # and the password to login

            elif choice == "2" and not self.isOnline:

                username = input(Fore.CYAN + "username: ")

                password = getpass(Fore.CYAN + "password: ")

                if not self.checkAccountLogin(username):

                    print(Fore.RED+"Invalid Username")

                else :

                        # asks for the port number for server's tcp socket

                    peerServerPort = int(input(Fore.CYAN +"Enter a port number for peer server: "))

                    roomServerPort = int(input(Fore.CYAN +"Enter a port number to join rooms: "))

                    status = self.login(username, password, peerServerPort)

                    # is user logs in successfully, peer variables are set

                    if status == 1:

                        self.isOnline = True

                        self.loginCredentials = (username, password)

                        self.peerServerPort = peerServerPort

                        self.roomServerPort = roomServerPort

                        # creates the server thread for this peer, and runs it

                        self.peerServer = PeerServer(self.loginCredentials[0], self.peerServerPort, self.roomServerPort)

                        self.peerServer.start()

                        # hello message is sent to registry

                        self.sendHelloMessage()

                    continue

            # if choice is 3 and user is logged in, then user is logged out

            # and peer variables are set, and server and client sockets are closed

            elif choice == "3" and self.isOnline:

                self.logout(1)

                self.isOnline = False

                self.loginCredentials = (None, None)

                self.peerServer.isOnline = False

                self.peerServer.tcpServerSocket.close()

                if self.peerClient is not None:

                    self.peerClient.tcpClientSocket.close()

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

            # is peer is not logged in and exits the program

            elif choice == "3":

                self.logout(2)

            # if choice is 4 and user is online, then user is asked

            # for a username that is wanted to be searched

            elif choice == "4" and self.isOnline:

                username = input(Fore.CYAN +"Username to be searched: ")

                searchStatus = self.searchUser(username)

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

                if searchStatus != None and searchStatus != 0:

                    print("IP address of " + username + " is " + searchStatus)

            # if choice is 5 and user is online, then user is asked

            # to enter the username of the user that is wanted to be chatted

            elif choice == "5" and self.isOnline:

                username = input(Fore.CYAN +"Enter the username of user to start chat: ")

                searchStatus = self.searchUser(username)

                # if searched user is found, then its ip address and port number is retrieved

                # and a client thread is created

                # main process waits for the client thread to finish its chat

                if searchStatus != None and searchStatus != 0:

                    searchStatus = searchStatus.split(":")

                    self.peerServer.chat = 1

                    self.peerClient = PeerClient(ipToConnect = searchStatus[0], portToConnect = int(searchStatus[1]) , username = self.loginCredentials[0], peerServer = self.peerServer, responseReceived=None ,flag = '5', room\_id = None, room\_peers = None)

                    self.peerClient.start()

                    self.peerClient.join()

            # if this is the receiver side then it will get the prompt to accept an incoming request during the main loop

            # that's why response is evaluated in main process not the server thread even though the prompt is printed by server

            # if the response is ok then a client is created for this peer with the OK message and that's why it will directly

            # sent an OK message to the requesting side peer server and waits for the user input

            # main process waits for the client thread to finish its chat

            elif choice == "5" and not self.isOnline:

                print(Fore.RED + "You can't chat, you need to log in first")

            elif choice == "6" and self.isOnline:

            #This choice creates a new chatroom and saves it in the database

                room\_id = input(Fore.CYAN +"Enter a Room ID: ")

                self.create\_room(room\_id)

                print(Fore.GREEN +"Room Created Successfully\n")

            elif choice == "7" and self.isOnline:

            #This choice joins already existing chatroom

                room\_id = input(Fore.CYAN +"Enter a Room ID: ")

                search\_status = self.search\_room(room\_id)

                if search\_status != 0 and search\_status != None:

                    #def \_\_init\_\_(self, ipToConnect, portToConnect, username, peerServer, responseReceived, flag, room\_peers : list)

                    ipToConnect = "192.168.1.5"

                    self.peerServer.room = 1

                    self.peerClient = PeerClient(ipToConnect, None, self.loginCredentials[0], self.peerServer, None, '7', room\_id ,search\_status)

                    self.peerClient.start()

                    self.peerClient.join()

            elif choice == "OK" and self.isOnline:

                okMessage = "OK " + self.loginCredentials[0]

                logging.info("Send to " + self.peerServer.connectedPeerIP + " -> " + okMessage)

                self.peerServer.connectedPeerSocket.send(okMessage.encode())

                self.peerClient = PeerClient(self.peerServer.connectedPeerIP, self.peerServer.connectedPeerPort , self.loginCredentials[0], self.peerServer, "OK", '5' , None, None)

                self.peerClient.start()

                self.peerClient.join()

            # if user rejects the chat request then reject message is sent to the requester side

            elif choice == "REJECT" and self.isOnline:

                self.peerServer.connectedPeerSocket.send("REJECT".encode())

                self.peerServer.isChatRequested = 0

                logging.info("Send to " + self.peerServer.connectedPeerIP + " -> REJECT")

            # if choice is cancel timer for hello message is cancelled

            elif choice == "CANCEL":

                self.timer.cancel()

                break

        # if main process is not ended with cancel selection

        # socket of the client is closed

        if choice != "CANCEL":

            self.tcpClientSocket.close()

    def create\_room(self, room\_id):

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

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

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

        message = "CREATE " + room\_id

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

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

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

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

        if response == "creation-success":

            print(Fore.GREEN +"Room created...")

        elif response == "room\_exist":

            print(Fore.LIGHTRED\_EX+"Room already exits")

    # function for searching an online user

     # function for searching an online user

    def search\_room(self, room\_id):

        # a search message is composed and sent to registry

        # custom value is returned according to each response

        # to this search message

        message = "JOINROOM " + room\_id + " " + str(self.roomServerPort)

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

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

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

        # for example "success ['4001', '8001', '3001', '5001']"

        # list start = 8 which is the index of [

        # list end =  40 which is the index of ]

        # list string is extracted using the start and ending list !!!

        list\_start = response.index('[')

        list\_end = response.index(']') + 1

        list\_string = response[list\_start:list\_end]

        # converts the extracted String to an actual python list

        response2 = eval(list\_string)

        response = response.split()

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

        if response[0] == "success":

            print(Fore.GREEN + room\_id + " is found successfully...")

            return response2

        elif response[0] == "search-fail":

            print(Fore.LIGHTRED\_EX+ room\_id + " is not found")

            return 0

    # def join\_room(self, search\_status, port\_no):

    #     if search\_status == 0:

    #         print("Room Not Found!")

    #     else:

    #         message = "JOINROOM " + search\_status + " " + port\_no

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

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

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

    #         if response == 'success':

    #             print("Joined Room Successfully")

    # account creation function

    def createAccount(self, username, password):

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

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

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

        message = "JOIN " + username + " " + password

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

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

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

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

        if response == "join-success":

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

            print(Fore.GREEN + "Welcome " + username)

        elif response == "join-exist":

             print(Fore.RED + "choose another username or login as this username already exist...")

    # login function

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

        # a login message is composed and sent to registry

        # an integer is returned according to each response

        message = "LOGIN " + username + " " + password + " " + str(peerServerPort)

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

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

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

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

        if response == "login-success":

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

            return 1

        elif response == "login-account-not-exist":

            print(Fore.RED + "Account does not exist...")

            return 0

        elif response == "login-online":

            print(Fore.RED + "Account is already online...")

            return 1

        elif response == "login-wrong-password":

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

            return 3

    # login function

    # logout function

    def logout(self, option):

        # a logout message is composed and sent to registry

        # timer is stopped

        if option == 1:

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

            self.timer.cancel()

        else:

            message = "LOGOUT"

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

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

    # function for searching an online user

    def searchUser(self, username):

        # a search message is composed and sent to registry

        # custom value is returned according to each response

        # to this search message

        message = "SEARCH " + username

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

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

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

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

        if response[0] == "search-success":

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

            return response[1]

        elif response[0] == "search-user-not-online":

            print(Fore.RED+ username + " is not online...")

            return 0

        elif response[0] == "search-user-not-found":

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

            return None

    # function for sending hello message

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

    def sendHelloMessage(self):

        message = "HELLO " + self.loginCredentials[0]

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

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

        self.timer = threading.Timer(1, self.sendHelloMessage)

        self.timer.start()

    def checkAccount(self, username):

        message = "CHECK-USERNAME-EXISTS " + username

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

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

        if response == "username-exist":

            print(Fore.RED + "choose another username or login as this username already exist...")

            return True

        else:

            return False

    def checkAccountLogin(self, username):

        message = "CHECK-USERNAME-EXISTS " + username

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

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

        if response == "username-exist":

            return True

        else:

            return False

# peer is started

main = peerMain()

### db.py

import os

import hashlib

import secrets

from pymongo.errors import DuplicateKeyError, ServerSelectionTimeoutError

from pymongo import MongoClient

from dotenv import load\_dotenv

load\_dotenv()

URI = os.getenv("MONGODB\_URL")

class DB:

    def \_\_init\_\_(self, uri=URI, db\_name="p2p-chat"):

        self.client = MongoClient(uri)

        self.db = self.client[db\_name]

        self.accounts\_collection = self.db["accounts"]

        self.online\_peers\_collection = self.db["online\_peers"]

        self.is\_connection\_working()

    # checks if an account with the username exists

    def is\_account\_exist(self, username):

        if len(list(self.db.accounts.find({'username': username}))) > 0:

            return True

        else:

            return False

    # registers a user

    def register(self, username, password):

        if self.is\_account\_exist(username):

            print("Account already exists")

            return False

        salt = secrets.token\_hex(16)

        hashed\_password = self.\_hash\_password(password, salt)

        account = {"username": username, "password": hashed\_password, "salt": salt}

        self.accounts\_collection.insert\_one(account)

        return True

    def \_hash\_password(self, password, salt):

        return hashlib.sha256((password + salt).encode("utf-8")).hexdigest()

    def verify\_password(self, username, password):

        user = self.accounts\_collection.find\_one({"username": username})

        if user:

            hashed\_password = self.\_hash\_password(password, user["salt"])

            return hashed\_password == user["password"]

        else:

            return False

    #def register(self, username, password):

        #account = {

        #    "username": username,

        #    "password": password

        #}

        #self.db.accounts.insert\_one(account)

    # retrieves the password for a given username

    def get\_password(self, username):

        return self.db.accounts.find\_one({"username": username})["password"]

    # checks if an account with the username online

    def is\_account\_online(self, username):

        if len(list(self.db.online\_peers.find({"username": username}))) > 0:

            return True

        else:

            return False

    # logs in the user

    def user\_login(self, username, ip, port):

        online\_peer = {

            "username": username,

            "ip": ip,

            "port": port

        }

        self.db.online\_peers.insert\_one(online\_peer)

    # logs out the user

    def user\_logout(self, username):

        self.db.online\_peers.delete\_one({"username": username})

    # retrieves the ip address and the port number of the username

    def get\_peer\_ip\_port(self, username):

        res = self.db.online\_peers.find\_one({"username": username})

        return (res["ip"], res["port"])

    # Assuming you have imported the necessary exceptions

    def register\_room(self, room\_id, peers = []):

        # Check if the room\_id already exists in the database

        if self.db.rooms.find\_one({"room\_id": room\_id}):

            raise ValueError(f"Room with id {room\_id} already exists.")

        room = {

            "room\_id": room\_id,

            "peers": peers

        }

        # Store the room information in the database

        self.db.rooms.insert\_one(room)

    # checks if an room with the id exists

    def is\_room\_exist(self, room\_id):

        if len(list(self.db.rooms.find({'room\_id': room\_id}))) > 0:

            return True

        else:

            return False

    #Needed when we flood a message

    def get\_room\_peers(self, room\_id):

        res = self.db.rooms.find\_one({"room\_id": room\_id})

        return (res["\_id"] ,res["peers"])

    def update\_room(self, id, peers):

        filter\_criteria = {"\_id": id}

        update\_data = {

            "$set": {"peers": peers}

        }

        self.db.rooms.update\_one(filter\_criteria, update\_data)

    def remove\_peer(self, id, peer):

        filter\_criteria = {"room\_id": id}

        room = self.db.rooms.find\_one(filter\_criteria)

        new\_peers = room["peers"].remove(peer)

        update\_data = {

            "$set": {"peers": new\_peers}

        }

        self.db.rooms.update\_one(filter\_criteria, update\_data)

    def is\_connection\_working(self):

        try:

            # Attempt to ping the database server

            self.client.server\_info()

            return True  # Connection is working

        except ServerSelectionTimeoutError:

            return False  # Connection failed

def drop\_all\_records(self):

        self.accounts\_collection.delete\_many({})

        self.online\_peers\_collection.delete\_many({})

# Example usage:

db = DB()

if db.is\_connection\_working():

    print("Connection to the database is working.")

else:

    print("Connection to the database is not working.")

# Screenshot of working code

A screen shot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

# Phase 4 Summary

We enhanced The application to handle one – to – one chatting , by searching for the username of the peer we want to chat with privately

# Unit Testing

## Create Account

### Failure

A screenshot of a computer

Description automatically generated

### Successful

A screenshot of a computer

Description automatically generated

### Confirmation Password Feature

A screenshot of a computer

Description automatically generated

## Login

### Failure

A screenshot of a computer

Description automatically generated

### Successful

A screenshot of a computer

Description automatically generated

## Logout

### Successful

## A screenshot of a computer program Description automatically generated

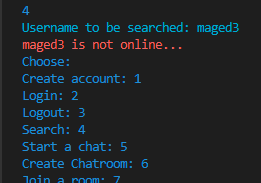
## Search

### Invalid

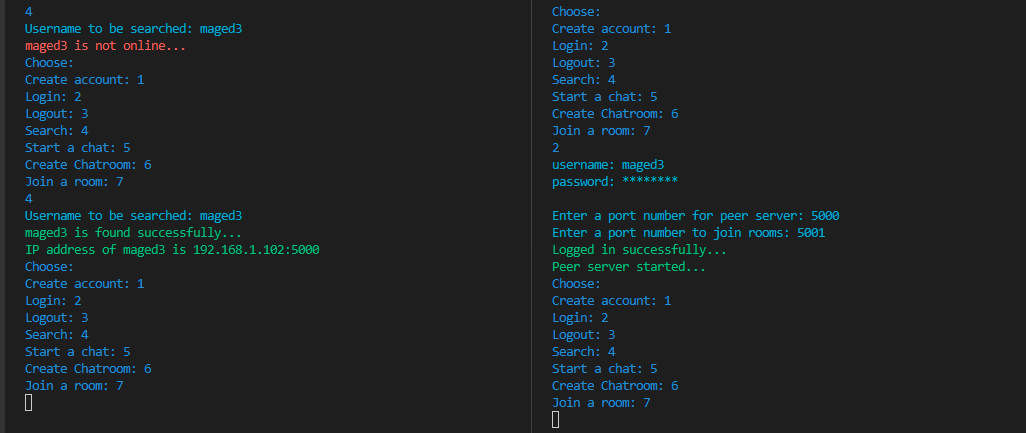
A screen shot of a computer

Description automatically generated

### Offline



### Online



## Start a chat

### Offline

A screenshot of a computer

Description automatically generated

### Online ( OK )

A screenshot of a computer

Description automatically generated

### Online (REJECT)

A screenshot of a computer

Description automatically generated

### Exit chat

A screenshot of a computer

Description automatically generated

## Create Chatroom

### Successful

A screenshot of a computer program

Description automatically generated

### Already Exist

A screenshot of a computer program

Description automatically generated

## Join a room

### Doesn’t Exist

A screen shot of a computer

Description automatically generated

### Invalid Number

### A screenshot of a computer Description automatically generated

### Successful

## Search Online Peers

### No online Peers

### Successful Search