**Design Document**

Computer Networks – CSE351

**Submitted to**

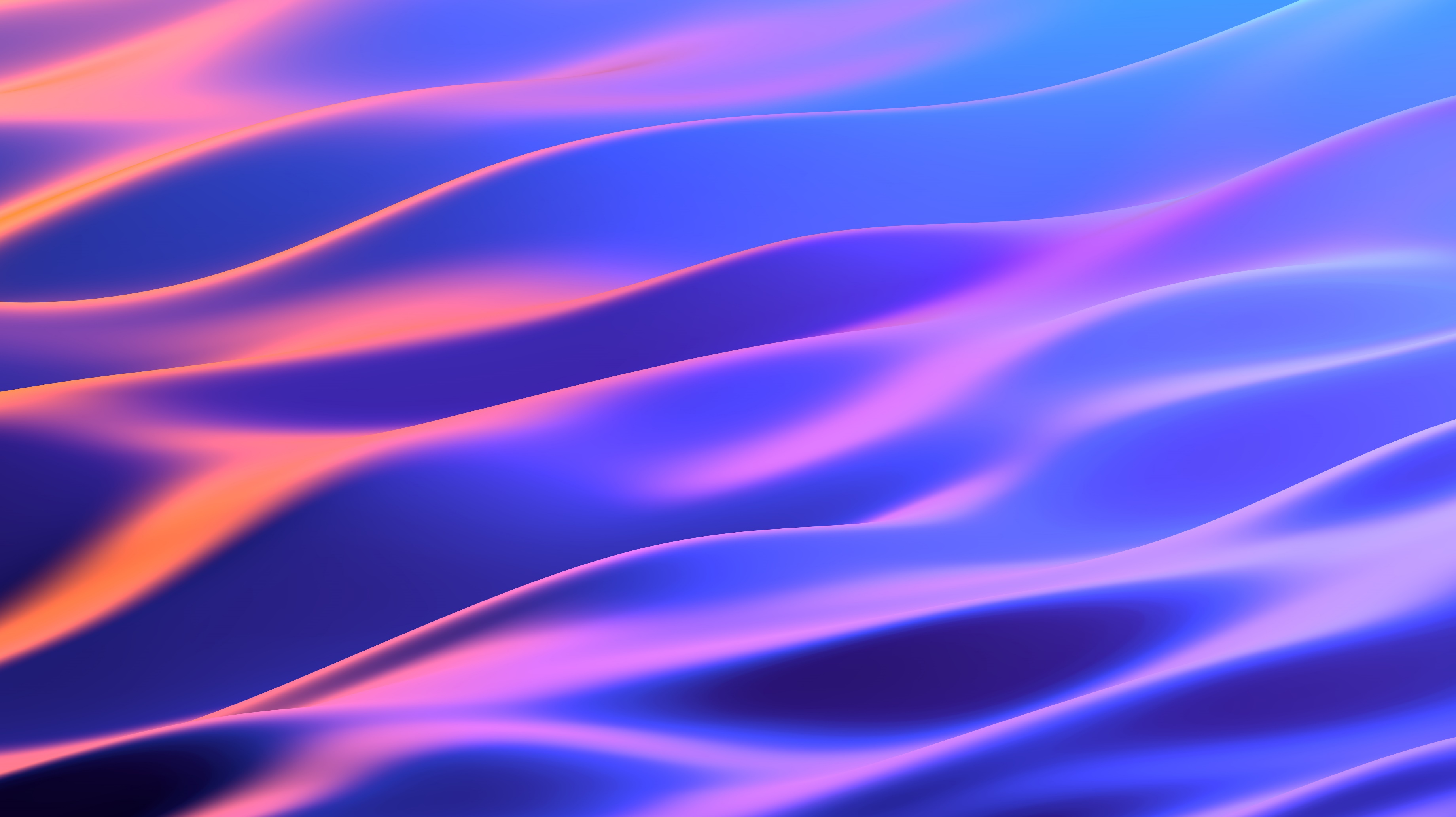
Prof. Ayman Bahaa El-dine

Eng. Noha Wahdan

**Group: 20**

|  |  |
| --- | --- |
| Ahmed Wael Ibrahim Mohamed | 20P3343 |
| Donia Sameh Farouk Mohamed | 20P3424 |
| Joliana Emad Kamal Naguib | 20P3292 |
| Mohamed Amr El-Mallah | 20P3485 |





Contents

[Overview 6](#_Toc154081686)

[Project Duration 6](#_Toc154081687)

[Purpose 6](#_Toc154081688)

[Methodology 6](#_Toc154081689)

[User Stories 7](#_Toc154081690)

[Cost Analysis 9](#_Toc154081691)

[User Authentication 12](#_Toc154081692)

[Authentication Mechanisms for Client-Server application 12](#_Toc154081693)

[Key Considerations for Selection 13](#_Toc154081694)

[System Architecture 14](#_Toc154081695)

[Client-Server Model 14](#_Toc154081696)

[Definition and characteristics 14](#_Toc154081697)

[Components of Client-Server Architecture 14](#_Toc154081698)

[Peer-to-Peer Model 15](#_Toc154081699)

[Definition and Characteristics 15](#_Toc154081700)

[Components of P2P Architecture 16](#_Toc154081701)

[Software Components 18](#_Toc154081702)

[Component Diagram 18](#_Toc154081703)

[Client-Peer Component 18](#_Toc154081704)

[Chat Room Component 19](#_Toc154081705)

[Database Component 19](#_Toc154081706)

[Server-Peer Component 20](#_Toc154081707)

[Sequence Diagrams 21](#_Toc154081708)

[Login 21](#_Toc154081709)

[Logout 22](#_Toc154081710)

[Signup 23](#_Toc154081711)

[One-to-One Chatting 24](#_Toc154081712)

[Group Chatting 25](#_Toc154081713)

[Data Flow Diagram 26](#_Toc154081714)

[Communication Protocols 27](#_Toc154081715)

[Client-Server Protocol 27](#_Toc154081716)

[Peer-to-Peer Protocol 28](#_Toc154081717)

[Message Formats 33](#_Toc154081718)

[Implementation 37](#_Toc154081719)

[Progress Comments 37](#_Toc154081720)

[Database 38](#_Toc154081721)

[Registry (Server) 40](#_Toc154081722)

[Peer (Client) 47](#_Toc154081723)

[Application Preview 61](#_Toc154081724)

[Starting Applications 61](#_Toc154081725)

[Create Account 62](#_Toc154081726)

[Login 64](#_Toc154081727)

[Get online peers 66](#_Toc154081728)

Design Document

# Overview

Project Duration

4 weeks until January 4th, 2024.

Development Team

* Ahmed Wael Ibrahim
* Mohamed Amr
* Donia Sameh
* Joliana Emad

Purpose

The development of a Peer-to-Peer Messaging application serves as a major task submission for the Computer Networks course instructed by Prof. Ayman Bahaa El-Dine and Eng. Noha Wahdan.

Dependencies

Development will depend on the code base provided by prof. Ayman on LMS.

Methodology

The project will adhere to the **Agile Software Development methodology**. Agile is an iterative and collaborative approach to software development, emphasizing adaptability and continuous improvement. It involves breaking the project into manageable iterations or "sprints" to facilitate ongoing development, feedback incorporation, and adjustments. This methodology promotes close teamwork, frequent reassessment of priorities, and adaptation to changes. Its core focus lies in delivering functional software that aligns with evolving requirements while ensuring customer satisfaction through effective communication and teamwork.

# User Stories

1. User Authentication:

* **Requirement:** Users must authenticate with a unique username and password.
* **User Story:** As a user, I want to create an account with a unique username and password.
* **User Story:** As a user, I want to log in using my registered username and password.

2. Basic Client-Server Setup:

* **Requirement:** Implement a basic server application to handle multiple client connections.
* **User Story:** As a user, I want to connect to the server using a client application.
* **User Story:** As a user, I want to see a list of online users.

3. Chat Room Functionality:

* **Requirement:** Users can create and join chat rooms.
* **User Story:** As a user, I want to create a new chat room.
* **User Story:** As a user, I want to join an existing chat room.
* **User Story:** As a user, I want to see a list of available chat rooms.

4. Group Messaging in Chat Rooms:

* **Requirement:** Users can send and receive messages within a chat room.
* **User Story:** As a user, I want to send a message to everyone in the chat room.
* **User Story:** As a user, I want to see messages from other users in the chat room.
* **User Story:** As a user, I want to receive notifications for new messages.

5. One-to-One Chat Functionality:

* **Requirement:** Users can initiate one-to-one chat sessions.
* **User Story:** As a user, I want to send a private message to another user.
* **User Story:** As a user, I want to receive private messages from others.
* **User Story:** As a user, I want to be notified of new private messages.

6. Message Formatting and Features:

* **Requirement:** Support basic text formatting (e.g., bold, italics) in messages.
* **User Story:** As a user, I want to format my messages to emphasize certain words.
* **Requirement:** Users can share hyperlinks in messages.
* **User Story:** As a user, I want to click on a hyperlink shared in a message to open a browser.

7. Error Handling and Resilience:

* **Requirement:** Implement robust error handling for unexpected scenarios.
* **User Story:** As a user, I want to receive meaningful error messages for troubleshooting.
* **Requirement:** Automatically reconnect users in case of a network interruption.

8. User Interface (UI) Enhancements:

* **Requirement:** Develop a command-line interface for simplicity.
* **User Story:** As a user, I want a clean and intuitive command-line interface.
* **Requirement:** Add color-coded messages for better visual distinction.
* **User Story:** As a user, I want to easily identify different types of messages.

9. Documentation:

* **Requirement:** Create user documentation covering installation, configuration, and usage.
* **User Story:** As a user, I want a comprehensive guide to set up and use the application.
* **Requirement:** Technical documentation detailing system architecture, protocols, and codebase structure.

10. Testing:

* **Requirement:** Conduct unit testing for each implemented feature.
* **User Story:** As a developer, I want to ensure each component functions correctly in isolation.
* **Requirement:** Perform integration testing to ensure seamless interactions between different components.
* **Requirement:** Conduct stress testing to evaluate system performance under high loads.

11. Scalability:

* **Requirement:** Design the system to handle an increasing number of users and chat rooms efficiently.
* **User Story:** As a developer, I want to optimize data structures and algorithms for scalability.

# Cost Analysis

Developing a robust Peer-to-Peer Multi-User Chatting Application involves various costs, including:

**Development Costs:**

* **Software Development:**
  + **Resource Time:**
    - Programming (Python) - Analyzing, designing, coding, and testing the application.
    - Network Engineering - Designing and implementing the network architecture.
    - Security Engineering - Implementing secure authentication and communication protocols.
  + **Tools and Frameworks:**
    - Python development environment (IDE)
    - Specific libraries for networking, cryptography, and UI/UX
    - Cloud-based development platform (optional)
  + **Testing and Quality Assurance:**
    - Unit testing, integration testing, and user acceptance testing
    - Automated testing tools
* **Documentation:**
  + Creating user manuals, API documentation, and internal technical documentation

**Infrastructure Costs:**

* **Deployment:**
  + Cloud-based server (optional)
  + Domain name and SSL certificate
  + Load balancer (optional)
* **Hosting:**
  + Monthly or annual fees for cloud server or other hosting services
  + Bandwidth costs depending on user activity

**Maintenance Costs:**

* Bug Fixes: Addressing issues reported by users
* Feature Enhancements: Implementing new features and functionality
* Security Updates: Maintaining security patches and updates for libraries and frameworks
* Version Control: Managing code changes and releases

**Additional Costs:**

* Project Management: Planning, scheduling, and coordinating development activities
* Legal and Regulatory Compliance: Ensuring compliance with data privacy regulations
* Third-Party Services: APIs, libraries, or other paid services
* Marketing and Promotion: Advertising and promoting the application to attract users

**Cost Estimation:**

Due to the project's scope and varying factors, providing a definitive cost estimate is difficult. However, here's a rough breakdown:

* Development: $5,000 - $20,000+
* Infrastructure: $500 - $2,000+ per month
* Maintenance: $1,000 - $5,000+ per month

**Cost Optimization Strategies:**

* **Open-source libraries and frameworks:**
  + Utilize freely available libraries and frameworks for various functionalities, reducing licensing costs.
* **Cloud-based development and hosting:**
  + Leverage cloud platforms for development and deployment to reduce infrastructure costs and maintenance overhead.
* **Agile development methodology:**
  + Focus on rapid prototyping and iterative development to ensure resource efficiency and early feedback.
* **Community-driven development:**
  + Encourage contributions from open-source communities to leverage shared resources and expertise.

Overall, the cost of developing and maintaining the Peer-to-Peer Multi-User Chatting Application will depend on various factors like project complexity, team size, and chosen technologies. Implementing cost-optimization strategies can significantly reduce expenses and ensure project viability.

# User Authentication

## Authentication Mechanisms for Client-Server application

1. **Username and Password:**
   * **Description:** This is the fundamental authentication method where users provide a unique username and password during both registration and login. The application stores and verifies these credentials to ensure secure access.
   * **Implementation:** The passwords and usernames are securely hashed before storage, using strong cryptographic algorithms. This prevents storing plaintext and enhances overall security.
2. **Account Registration:**
   * **Description:** Users can create an account by selecting a unique username and establishing a password. This step is essential for setting up user identities within the application.
   * **Implementation:** During registration, ensure that the chosen username is unique and not already in use. Store user information securely, possibly in a database.
3. **Hashing:**
   * **Description:** Usernames, Passwords are hashed to protect them from unauthorized access, ensuring they are not stored in a readable format. The project will be built over SHA256 hashing library. The hashed credentials are stored in the database and will be compared with the hashed user inputs. The hashing mechanism protects users from getting exposed if the system is hacked.
   * **Implementation:** using hashlib library to encrypt username and passwords before sending the to the database.
4. **Login:**
   * **Description:** Users can login to use the application features.
   * **Implementation:** when the user enters username and password it is hashed then compared with the hashed version in the database to ensure that the login information is correct then the application menu appears to use its features.

## Key Considerations for Selection

* Security: Protect user credentials and prevent unauthorized access.
* Scalability: Handle increasing numbers of users effectively.
* User Experience: Ensure a smooth and seamless authentication process for users.

# System Architecture

In this section, the different architectures that might be used in the design of the system will be discussed and compared.

## Client-Server Model

A diagram of a cloud computing

Description automatically generated

### Definition and characteristics

* A client-server model is a networking computing system design that illustrates a relationship between two or more clients, where the client computers (Hosts) request and receive services or resources from a powerful centralized server computer.
* It enables several clients to access files from a single server, preserving consistency across all devices where Data is stored at server side only.
* This model is widely used in various computing applications, ranging from simple file sharing to complex web-based applications.

### Components of Client-Server Architecture

1. **Client**
   1. Client is end-user device or application that initiates requests for services or resources from a powerful centralized server.
   2. Clients can be computers, smartphones, tablets, or any other device capable of making requests.
2. **Server**
   1. The server is a powerful computer or software application that provides services or resources in response to client requests.
   2. Servers are designed to handle multiple requests simultaneously and can be specialized for specific tasks such as web hosting, database management, or file storage.
3. **Network**
   1. Is the communication infrastructure that allows clients and servers with each other.
   2. This can be a local area network (LAN), a wide area network (WAN), or the internet.
   3. The network facilitates the transmission of data between clients and servers.

## Peer-to-Peer Model

A diagram of a computer network

Description automatically generated

### Definition and Characteristics

* A decentralized network architecture where each participant in the network, referred to as a peer.
* There is no dedicated server and clients. Each node(peer) acts as a server and as well as client which means peers communicate directly with each other to share resources.

So, each node in the network can request a service and can provide a service & store its own data.

A diagram of a computer connection

Description automatically generated

* One-to one Chatting where Peer1 can directly communicate with Peer2 through P2P Connection after Authentication.

A diagram of a network of computers

Description automatically generated

* Chat Rooms where All Peers can communicate directly with each other through P2P Connection by sending Broad messages.

### Components of P2P Architecture

1. **Client**
   1. Each participant in the P2P network is referred to as a “peer”, Each peer has the capability to request resources or services from other peers as well as provide resources or services to others. Peers are equal in terms of functionality, and there is no centralized control.
   2. Peers communicate with each other using P2P communication protocols.
   3. Peers interact with server only for authentication purposes, when a peer joins the network, it might need to authenticate itself before participating fully & for chat room management.

**Examples of Request sent:**

* Chat requests in order to initiate peer-to-peer chat
* Create/Join chat room requests in order to create/join specific chat room
* Sending/Receiving messages in chat room requests

**Examples of Requests sent (Authentication):**

* User login request
* User logout request
* User signup request 🡪 Register if no account exists

1. **Server**
   1. In P2P Network, Server is not a Centralized Entity - Instead, It typically serves specific roles Such as :
      1. Authentication**:** The server plays a role in authenticating peers. When a peer wants to join the network or access certain services, it may send authentication requests to the server to verify its identity. The server can verify the identity of the peer based on credentials, ensuring secure access to the P2P network.
      2. Chat Room Management**:**  The server manages the process of joining/leaving rooms. Peers may request to join/leave specific rooms, and the server coordinates these requests, ensuring that the processes is controlled and organized. And provides a list of Available Chat rooms.
   2. Server facilitates the establishment of P2P connections.
2. **Database**
   1. The database component is responsible for managing and storing data shared among peers. Unlike traditional client-server architectures where a centralized server manages the database, P2P databases are often distributed across multiple peers in a decentralized manner.
   2. Server interacts with database to verify login data of client and store account if user is not registered.
   3. Database provides a list of online peers.

# Software Components

## Component Diagram

A diagram of a computer

Description automatically generated

Assumptions

* Server is intended only for Authentication and Chat Room Management
* Client 1 (acts as client) provides login data to Client peer for sending messages to server peer either one-to one or group chatting ,Client 2(acts as a server) on the other hand provides login data to Server peer component in order to exchange messages through p2p connection.

In the upcoming section each component will be described separately.

## Client-Peer Component

Description

Represents Server- side application responsible for user authentication, Managing p2p connections and Handling Chat Rooms Requests by client peer.

Roles

1. **Authentication Request Management**

Request Authentication of client to database for validation of stored accounts to ensure secure and valid access to the P2P network.

1. **Chat Room Joining Management**

When user requests joining chat room, server will provide a list of available chat rooms to client peer to be able to join target chat room.

1. **P2P Communication Management**

Request user status to database whether server peer is online/offline so that client peer can initiate P2P connection if server peer is online.

## Chat Room Component

Description

Manages direct communication between peers in p2p connection by exchanging messages across it.

Roles

1. **Handling Messages of Client Peer**

Manages processing, and presentation of messages sent by the Client-Peer Component to Ensure effective handling of messages, including displaying them to the user, managing message history, and providing a seamless messaging experience within the P2P network.

1. **Facilitates Messages Transmission**

Enables the client to communicate messages to the Server peer without relying on a centralized messaging channel. This feature supports efficient and direct data transfer between the client and the server peer through p2p connection.

## Database Component

Description

System responsible for managing ,organizing and storing data within a peer-to-peer (P2P) architecture.

Roles

1. **User Registration:**

When a user joins the P2P network or requests to access specific resources, they need to register with the system.

Registration involves providing necessary information (username, password, etc.), and this information is securely stored in the distributed database.

1. **Authentication Requests:**

When a user attempts to log in, the peer initiates an authentication request. This request was sent to database for verification.

1. **Provides List of Online Peers:**

When a client peer wants to send a message to server peer, it must be online So, request is sent by server to database to check status and then database respond by sending list of online peers.

## Server-Peer Component

Description

emphasize the server's role in a peer-to-peer (P2P) architecture**.**

Roles

1. **P2P Communication**

Server peer interact with client-peer component when client-peer sends a message through P2P connection.

1. **Chat Room Interactions**

Server peer also can join chat room by providing username of client.

## Sequence Diagrams

### Login

A screenshot of a computer screen

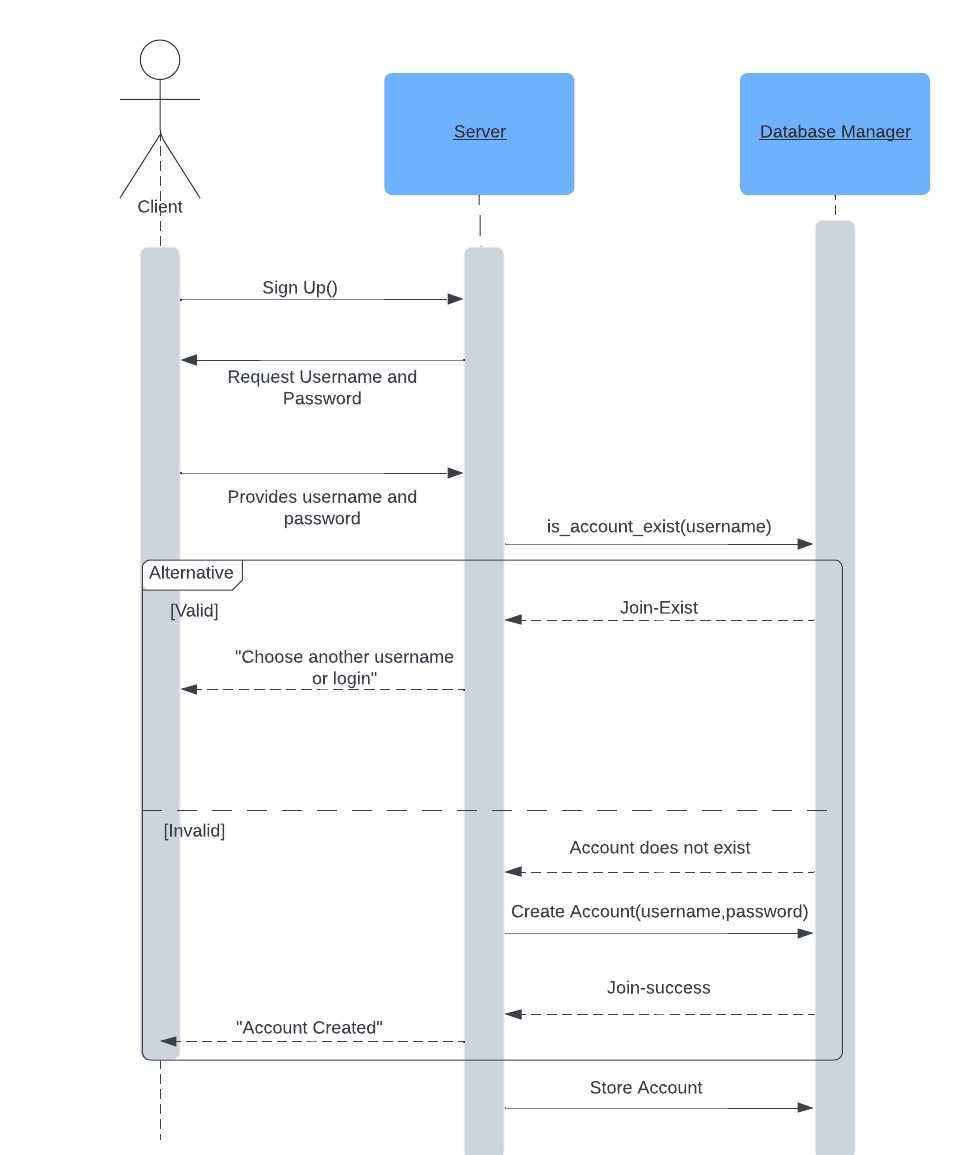
Description automatically generated

### Logout

A diagram of a login

Description automatically generated

### Signup



### One-to-One Chatting

A diagram of a project

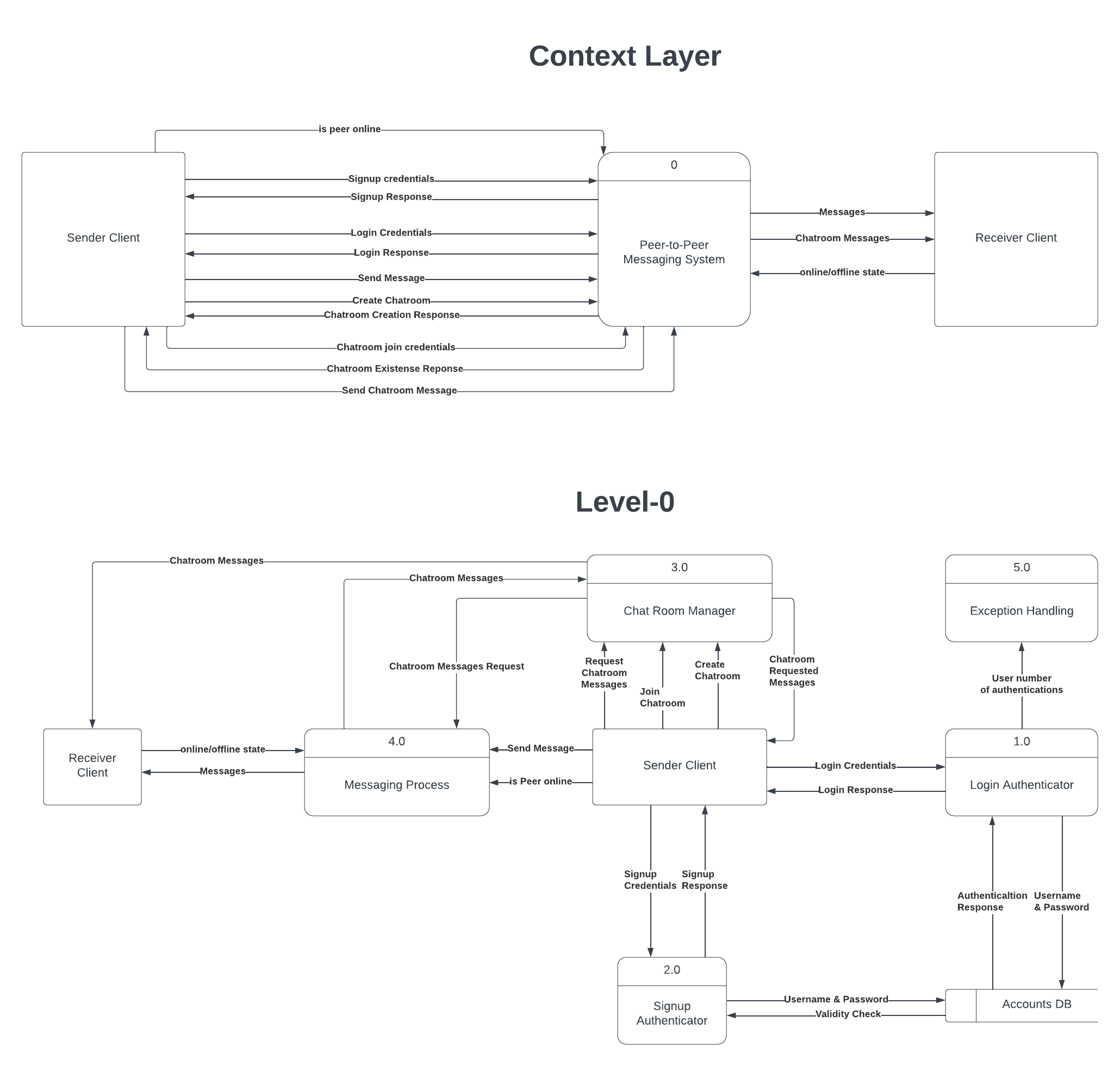
Description automatically generated

### Group Chatting

A blueprint of a building

Description automatically generated

## Data Flow Diagram



# Communication Protocols

## Client-Server Protocol

**Establishing Client Server Connections**

Establish a TCP connection first, and it should be established successfully after that.

Client requests to create ana account to the server then server sends that account is created.

To create authentication, clients first send requests to log in. If the username and password are incorrect, the server rejects the request (404 not found). If the login details are correct, the server responds with an acceptance (200 ok).

Server maintain a list of available online peers.

Handshaking occurs, the client requests to send a message to server, and server checks if the other client is online. If the other client is online the server which delivers message. If the other client is offline, then the request is rejected.

Client sends requests to create or join chat rooms then server manage the creation and joining of chat rooms.

**A diagram of a network

Description automatically generated**

## Peer-to-Peer Protocol

Establishing Peer-to-Peer Connections

Each user in the chat room will act as a peer and have their own instance of the database, representing their local database connection. Peers need a way to discover and connect with each other. This can be achieved by implementing a peer discovery mechanism, where peers exchange their IP addresses and port numbers.

You can modify the database to discover and connect with other peers in the chat room to retrieve a list of online peers from the database.

Joining the Chat Room

When a peer wants to join the chat room, they can use the peer discovery mechanism to find other peers. Once the peer obtains the IP addresses and port numbers of other peers, they can establish direct connections with them.

The peer can use the database to mark themselves as online and store their IP address and port number in the online peer’s collection.

Peer-to-Peer Message Exchange

With peer connections established, peers can send and receive messages directly to/from each other. You can create a messaging system where peers exchange messages using their established connections. Peers can use the database to retrieve the IP address and port number of a specific peer before sending a message to them.

Consider implementing a message format to determine the structure of the messages exchanged between peers. This format should include information such as the type of message, sender, receiver, and the actual content of the message. You can define a message format using a specific syntax, such as JSON or plain text.

Broadcasting Messages to the Group

To implement group chat functionality, each peer needs to broadcast messages to all other peers in the chat room. You can modify the code to include a method that sends a message to all connected peers. This method can iterate through the list of connected peers obtained from the online peer’s collection and use their IP addresses and port numbers to send the message.

Leaving the Chat Room

When a peer wants to leave the chat room, they can use the logout method of the database to mark themselves as offline and remove their entry from the online peers collection. Additionally, the peer should close their UDP socket and terminate any established connections with other peers.

Create an account

Client is requesting from the server to create an account then the server stores the username and password in the database.

A screen shot of a computer screen

Description automatically generated

Login to an account

The client requests from the server to login then server checks in database the username, password, IP address and port number.

If login details are accepted, then the database returns the list of online peers and server sends to the client that it is accepted (200 ok).

A screen shot of a computer

Description automatically generated

If login details are rejected, then the database returns invalid details and server sends to the client that it is rejected (404 not found).

A screen shot of a computer

Description automatically generated

One to one chat

To Establish peer connections, each peer should have a mechanism to establish connections with other peers. This can be done using IP addresses and port numbers. Peers can connect to each other using sockets, such as TCP or UDP sockets. The server socket is used to accept incoming connections from other peers. Using TCP connection for one-to-one chat.

Client 2 must be online (logged in) to be able to receive the message so the server checks in database if client 2 is online and server sends to client 1 that request accepted (200 ok) and the connection will be initiated.

If client 1 requests to send chat to client 2 and client 2 is offline the server will reject the request and server will send to the client a rejection response (404 not found).

If client 1 wants to start a chat with client 2, it sends to server a request to chat with client 2 then the server sends IP address and port number of client 2 to client 1 and sends IP address and port number of client 1 to client 2 to make a peer connection.

When two peers establish a connection, they need to perform a handshake to negotiate the communication parameters. This handshake can include exchanging information about supported protocols, capabilities, and establishing a shared encryption key.

Client 1’s message will be encrypted and sent to client 2 and will be decrypted to be read.

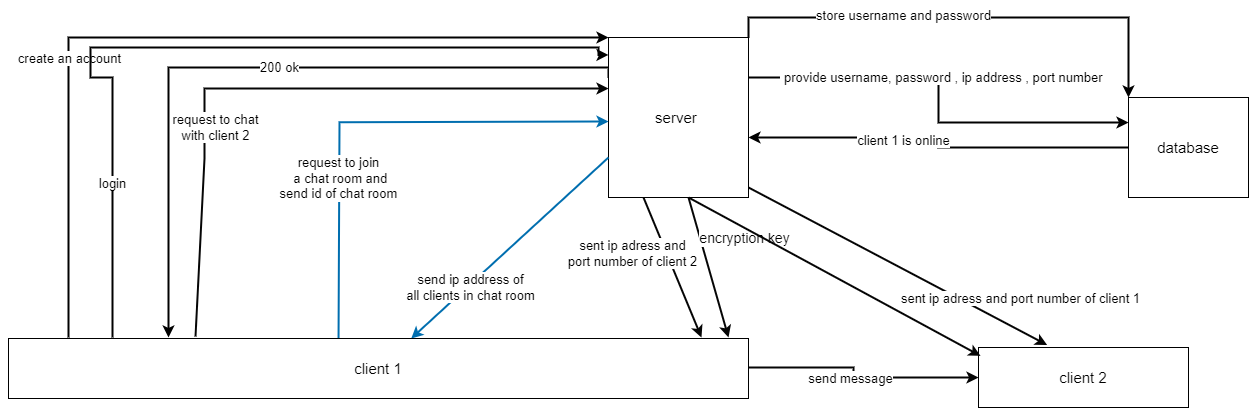
If client 1 requests to start a chat with client 2, server checks the list of online peers in database, if client 2 is busy with other chat the server sends to client 1 that the request is rejected, and connection will not be initiated.

**A diagram of a diagram

Description automatically generated**

Chat room

Client 1 requests to join a chat room and id of group chat that wants to join from a server then server sends IP address of all clients in chat room then it starts the broadcasting to send the messages to all the clients in the group and starts a UDP connection.



Client 1 requests to create a new chat room then the server adds the new chat room to the database.

Client 1 requests to see a list of available chat rooms then the server requests the list from the database then the database sends the list to the server which is resent to Client 1.

A diagram of a building

Description automatically generated

Logout

Client 1 requests to logout from server then server sends the username to the database then database send that the user is offline the server sends to the client 200 ok (logged out successfully)

A diagram of a diagram

Description automatically generated

## Message Formats

|  |  |
| --- | --- |
| Chat request message Format | Chat Request:  {  “Action”: "CHAT-REQUEST",  “Username\_sender”: “Medhat2990”,  “Username\_receiver”: “Fawzy9000”  }  Acceptance Response:  {  “Status”: “OK”,  “Message”: "Client with OK message is created... and sending messages"  }  Chat Response:  {  “Status”: “REJECT”,  “Message”: "client of requester is closing..."  }  Chat Response:  {  “Status”: “BUSY”,  “Message”: "Receiver peer is busy"  } |
| Quit Message Format | Quit request:  “Action”: ":q",  “Username\_sender”: “Medhat2990”,  “Username\_receiver”: “Fawzy9000”  “Message”: “Hello, how are you?”  }  Received message response is empty  {  “Status”: “empty-message-not-chatting”,  “Message”: “chat suddenly ended”  }  Message response is not an empty  {  “Status”: “:q-is-sent”,  “Message”: “User you're chatting with ended the chat”  } |
| Create Account Message Format | Create account request:  {  “Action”: "JOIN",  “username”: “medhat2990”,  “password”: “pass2990”  }  Create Account Response:  {  “Status”: "join-success",  “Message”: "Account created..."  }  Create Account Response:  {  “Status”: "join-exist",  “Message”: "choose another username or login..."  } |
| Login Message Format | Login request:  {  “Action”: " LOGIN",  “username”: “medhat2990”,  “password”: “pass2990”  }  Login response:  {  “Status”: "login-success",  “Message”: "Logged in successfully..."  }  Login response:  {  “Status”: "login-account-not-exist",  “Message”: "Account does not exist..."  }  Login response:  {  “Status”: "login-online",  “Message”: " Account is already online..."  }  Login response:  {  “Status”: "login-wrong-password",  “Message”: "Wrong password..."  } |
| Logout Message Format | Login request:  {  “Action”: " LOGOUT",  “username”: “medhat2990”,  }  Logout response:  {  “Status”: “user is offline”,  “Message”: “Logout successfully”  } |
| Search User Message Format | Search user request:  {  “Action”: " SEARCH-username"  }  Search user response:  {  “Status”: "search-success",  “Message”: " is found successfully..."  }  Search user response:  {  “Status”: "search-user-not-online",  “Message”: " is not online..."  }  Search user response:  {  “Status”: "search-user-not-found",  “Message”: " is not found"  } |
| Hello Message Format | Hello message request:  {  “Action”: " HELLO username",  } |
| User authentication | user request:  {  “Action”: "AUTHENTICATE",  “username”: “medhat2990”,  “password”: “pass2990”  }  user response:  {  “Status”: " authentication-check",  “Message”: " authentication done successfully "  } |
| Joining a chat room message format | Join request:  {  “Action”: "JOIN-CHAT-ROOM",  “chat room id”: “general ”  }  Join Response:  {  “Status”: "join-success",  “Message”: "joined chat room successfully", |
| Send a message to chat room | Send request:  {  “Action”: " SEND-TO-CHAT-ROOM",  “chat\_room\_id”: “general ”,  “Message”: “Hi, everyone”  }  Send response:  {  “Status”: "send-success",  “Message”: "sent successfully"  } |
| Send a message to one-to-one chat | Send request:  {  “Action”: " SEND-TO-ONE-TO-ONE-CHAT",  “Username\_sender”: “Medhat2990”,  “Username\_receiver”: “Fawzy9000”  “Message”: “Hello, how are you?”  }  Send response:  {  “Status”: "send-success",  “Message”: "sent successfully"  } |

# Implementation

This section showcases the current progress.

The project repository link: <https://github.com/Julianaa04/P2PChat>

## Progress Comments

Added Features

* Ability to view currently online users (online Peers)
* Improved Database functionality by refactoring DB code to the latest version of MongoDB package
* Improved User Interface and added colors
* Improved Performance through extensive exception handling and edge cases testing
* Improved system security by hashing registered username and password using SHA256 hashing function

Added Packages

These packages can be added to system using pip install command in CLI

* **Colorama:** Used for improving user interface.
* **Hashlib:** Used for hashing username and password
* **MongoDB:** Used for creating and maintaining database

## Database

|  |
| --- |
| from pymongo import MongoClient  class DB:      # Constructor method      def \_\_init\_\_(self):          self.client = MongoClient('mongodb://localhost:27017/')          self.db = self.client['p2p-chat']      # checks if an account with the username exists      def is\_account\_exist(self, username):          user\_exists = self.db.accounts.find\_one({'username': username})          if user\_exists is not None:              return True          else:              return False      # registers a user      def register(self, username, password):          account = {              "username": username,              "password": password          }          self.db.accounts.insert\_one(account)      # retrieves the password for a given username      def get\_password(self, username):          return self.db.accounts.find\_one({"username": username})["password"]      # checks if an account with the username online      def is\_account\_online(self, username):          if self.db.online\_peers.count\_documents({"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"] |

## Registry (Server)

|  |
| --- |
| '''      ##  Implementation of registry      ##  150114822 - Eren Ulaş  '''  from socket import \*  import threading  import select  import logging  import db  import hashlib  # 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("New thread started for " + ip + ":" + str(port))      # main of the thread      def run(self):          # locks for thread which will be used for thread synchronization          self.lock = threading.Lock()          print("Connection from: " + self.ip + ":" + str(port))          print("IP Connected: " + self.ip)          while True:              try:                  # waits for incoming messages from peers                  message = self.tcpClientSocket.recv(1024).decode().split()                  logging.info("Received from " + self.ip + ":" + str(self.port) + " -> " + " ".join(message))                  # message[1]=username, message[2]=password                  # Hashing of username and password using SHA256 Hashing Method                  username = hashlib.sha256(message[1].encode()).hexdigest()                    #   JOIN    #                  if message[0] == "JOIN":                      # join-exist is sent to peer,                      # if an account with this username already exists                      password = hashlib.sha256(message[2].encode()).hexdigest()                      if db.is\_account\_exist(username):                          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 != exist, and the account is created                      else:                          db.register(username, password)                          response = "join-success"                          logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)                          self.tcpClientSocket.send(response.encode())                    #   LOGIN    #                  elif message[0] == "LOGIN":                      password = hashlib.sha256(message[2].encode()).hexdigest()                      # login-account-not-exist is sent to peer,                      # if an account with the username does not exist                      if not db.is\_account\_exist(username):                          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(username):                          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                          retrievedPass = db.get\_password(username)                          # 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                            ################# WAEL EDITED HERE #################                          if retrievedPass == password:                              self.username = message[1]                              self.lock.acquire()                              try:                                  tcpThreads[self.username] = self                              finally:                                  self.lock.release()                              db.user\_login(username, self.ip, message[3])                              # 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 username != None and db.is\_account\_online(username):                          db.user\_logout(username)                          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(username):                          # checks if the account is online                          # and sends the related response to peer                          if db.is\_account\_online(username):                              peer\_info = db.get\_peer\_ip\_port(username)                              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] == "ONLINE":                      response = list(tcpThreads.keys())                      self.tcpClientSocket.send(str(response).encode())              except OSError as oErr:                  logging.error("OSError: {0}".format(oErr))              except threading.ThreadError as te:                  logging.error(f"Thread error: {te}")              except IndexError as ie:                  logging.error(f"Index error: {ie}")              except Exception as e:                  logging.error(f"An unexpected error occurred: {e}")      # 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 != received before timeout      # then peer is disconnected      def waitHelloMessage(self):          if self.username != 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()  '''''''''''''''''''''''''''''''''''''''''''''''''''''''''''                      REGISTRY MAIN CODE  '''''''''''''''''''''''''''''''''''''''''''''''''''''''''''  # tcp and udp server port initializations  print("Registy started...")  port = 15699  portUDP = 15599  # 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 != windows  # it checks to get it for macos devices  hostname=gethostname()  try:      host=gethostbyname(hostname)  except gaierror:      import netifaces as ni      host = ni.ifaddresses('en0')[ni.AF\_INET][0]['addr']  print("Registry IP address: " + host)  print("Registry port number: " + str(port))  # onlinePeers list for online account  onlinePeers = {}  # accounts list for accounts  accounts = {}  # tcpThreads list for online client's thread  tcpThreads = {}  #tcp and udp socket initializations  tcpSocket = socket(AF\_INET, SOCK\_STREAM)  udpSocket = socket(AF\_INET, SOCK\_DGRAM)  try:      tcpSocket.bind((host,port))  except OSError as oErr:      logging.error(f"Error binding TCP socket: {oErr}")  try:      udpSocket.bind((host,portUDP))  except OSError as oErr:      logging.error(f"Error binding UDP socket: {oErr}")  tcpSocket.listen(5)  # input sockets that are listened  inputs = [tcpSocket, udpSocket]  # log file initialization  logging.basicConfig(filename="registry.log", level=logging.INFO)  # as long as at least a socket exists to listen registry runs  while inputs:      print("Listening for incoming connections...")      # monitors for the incoming connections      readable, writable, exceptional = select.select(inputs, [], [])      for s in readable:          try:              # 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 == 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 == 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("Hello is received from " + message[1])                          logging.info("Received from " + clientAddress[0] + ":" + str(clientAddress[1]) + " -> " + " ".join(message))          except OSError as se:              logging.error(f"Socket error: {se}")          except Exception as e:              logging.error(f"An unexpected error occurred: {e}")          except threading.ThreadError as te:              logging.error(f"Thread error: {te}")          except IndexError as ie:              logging.error(f"Index error: {ie}")  # registry tcp socket is closed  tcpSocket.close() |

## Peer (Client)

|  |
| --- |
| '''      ##  Implementation of peer      ##  Each peer has a client and a server side that runs on different threads      ##  150114822 - Eren Ulaş  '''  from socket import \*  import threading  import time  import select  import logging  import colorama  from colorama import Back, Fore, Style  # Server side of peer  class PeerServer(threading.Thread):      # Peer server initialization      def \_\_init\_\_(self, username, peerServerPort):          threading.Thread.\_\_init\_\_(self)          # keeps the username of the peer          self.username = username          # tcp socket for peer server          self.tcpServerSocket = socket(AF\_INET, SOCK\_STREAM)          # port number of the peer server          self.peerServerPort = peerServerPort          # 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        # main method of the peer server thread      def run(self):          print("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'          # socket initializations for the server of the peer          self.tcpServerSocket.bind((self.peerServerHostname, self.peerServerPort))          self.tcpServerSocket.listen(4)          # inputs sockets that should be listened          inputs = [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==self.tcpServerSocket:                          # 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(self.username + " is connected from " + str(addr))                              self.connectedPeerSocket = connected                              self.connectedPeerIP = addr[0]                      # if the socket that receives the data is the one that                      # is used to communicate with a connected peer, then enters here                      else:                          # 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":                              # 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==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 != 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":                              self.isChatRequested = 0                              inputs.clear()                              inputs.append(self.tcpServerSocket)                              # connected peer ended the chat                              if len(messageReceived) == 2:                                  print("User you're chatting with ended the chat")                                  print("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("User you're chatting with suddenly ended the chat")                              print("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))              except Exception as e:                  logging.error(f"An unexpected error occurred: {e}")              except threading.ThreadError as te:                  logging.error(f"Thread error: {te}")              except IndexError as ie:                  logging.error(f"Index error: {ie}")  # 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):          threading.Thread.\_\_init\_\_(self)          # keeps the ip address of the peer that this will connect          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)          # 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      # main method of the peer client thread      def run(self):          print("Peer client started...")          # connects to the server of other peer          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==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()    # main process of the peer  class peerMain:      # peer initializations      def \_\_init\_\_(self):          # ip address of the registry          self.registryName = input(Fore.LIGHTWHITE\_EX+"Enter IP address of registry: ")          #self.registryName = 'localhost'          # port number of the registry          self.registryPort = 15699          # 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 = 15599          # login info of the peer          self.loginCredentials = (None, None)          # online status of the peer          self.isOnline = False          # server port number of this peer          self.peerServerPort = None          # server of this peer          self.peerServer = None          # client of this peer          self.peerClient = None          # timer initialization          self.timer = None            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              def get\_colored\_input(choices):                  for key, value in choices.items():                      print(                          f"{value['number\_color']}{key}.{Style.RESET\_ALL} {value['text\_color']}{value['text']}{Style.RESET\_ALL}")                  user\_input = input(Fore.LIGHTWHITE\_EX + "Choose: ")                  return user\_input              if not self.isOnline:                  choices = {                      '1': {'text': 'Create account', 'number\_color': Fore.LIGHTCYAN\_EX, 'text\_color': Fore.LIGHTWHITE\_EX},                      '2': {'text': 'Login', 'number\_color': Fore.LIGHTCYAN\_EX, 'text\_color': Fore.LIGHTWHITE\_EX},                      '3': {'text': 'Exit', 'number\_color': Fore.LIGHTCYAN\_EX, 'text\_color': Fore.LIGHTWHITE\_EX},                  }              else:                  choices = {                      '1': {'text': 'Search', 'number\_color': Fore.LIGHTCYAN\_EX, 'text\_color': Fore.LIGHTWHITE\_EX},                      '2': {'text': 'Start a chat', 'number\_color': Fore.LIGHTCYAN\_EX, 'text\_color': Fore.LIGHTWHITE\_EX},                      '3': {'text': 'Exit', 'number\_color': Fore.LIGHTCYAN\_EX, 'text\_color': Fore.LIGHTWHITE\_EX},                      '4': {'text': 'Logout', 'number\_color': Fore.LIGHTCYAN\_EX, 'text\_color': Fore.LIGHTWHITE\_EX},                      '5': {'text': 'Get Online Peers', 'number\_color': Fore.LIGHTCYAN\_EX, 'text\_color': Fore.LIGHTWHITE\_EX},                  }              choice = get\_colored\_input(choices)              # if choice is 1, creates an account with the username              # and password entered by the user              if choice=="1" and not self.isOnline:                  username = input(Fore.LIGHTMAGENTA\_EX +"username: ")                  password = input(Fore.LIGHTMAGENTA\_EX +"password: ")                    self.createAccount(username, 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.LIGHTMAGENTA\_EX +"username: ")                  password = input(Fore.LIGHTMAGENTA\_EX +"password: ")                  # asks for the port number for server's tcp socket                  sock = socket()                  sock.bind(('', 0))                  peerServerPort = sock.getsockname()[1]                  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                      # creates the server thread for this peer, and runs it                      self.peerServer = PeerServer(self.loginCredentials[0], self.peerServerPort)                      self.peerServer.start()                      # hello message is sent to registry                      self.sendHelloMessage()                  # if choice is 1 and user is online, then user is asked                  # for a username that is wanted to be searched              elif choice == "1" and self.isOnline:                  username = input(Fore.LIGHTMAGENTA\_EX + "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 2 and user is online, then user is asked                  # to enter the username of the user that is wanted to be chatted              elif choice == "2" and self.isOnline:                  username = input(Fore.LIGHTMAGENTA\_EX + "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.peerClient = PeerClient(searchStatus[0], int(searchStatus[1]), self.loginCredentials[0],                                                       self.peerServer, None)                      self.peerClient.start()                      self.peerClient.join()                  #if choice is 6 and user is online, the user requests the list of online peers right now                  # The user sends the request to the server and the server shall respond with a message containing the users              elif choice == "5" and self.isOnline:                  msg = self.getOnlinePeers()                  print(msg + '\n\n')                      # if choice is 4 and user is logged in, then user is logged out                      # # and peer variables are set, and server and client sockets are closed              elif choice == "4" and self.isOnline:                  self.logout(1)                  self.isOnline = False                  self.loginCredentials = (None, None)                  self.peerServer.isOnline = False                  self.peerServer.tcpServerSocket.close()                  if self.peerClient != None:                      self.peerClient.tcpClientSocket.close()                  print(Fore.LIGHTGREEN\_EX + "Logged out successfully")                  main = peerMain()                      #if choice is 5 ,"Exit" process is done, peer will not be logged in & program is terminated              # is peer is not logged in and exits the program              elif choice=="3":                  self.logout(2)              # 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 == "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")                  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()      # 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.LIGHTGREEN\_EX+"Account created...")          elif response == "join-exist":              print(Fore.LIGHTYELLOW\_EX+"choose another username or login...")      # 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.LIGHTGREEN\_EX+"Logged in successfully...")              return 1          elif response == "login-account-not-exist":              print(Fore.LIGHTRED\_EX+"Account does not exist...")              return 0          elif response == "login-online":              print(Fore.LIGHTYELLOW\_EX+"Account is already online...")              return 2          elif response == "login-wrong-password":              print(Fore.LIGHTRED\_EX+"Wrong username or password...")              return 3        # 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" + ' 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(username + Fore.LIGHTGREEN\_EX+" is found successfully...")              return response[1]          elif response[0] == "search-user-not-online":              print(username + Fore.LIGHTYELLOW\_EX+" is not online...")              return 0          elif response[0] == "search-user-not-found":              print(username +Fore.LIGHTRED\_EX+ " is not found")              return None        def getOnlinePeers(self):          message = "ONLINE" + ' online'          self.tcpClientSocket.send(message.encode())          response = self.tcpClientSocket.recv(1024).decode()          return ("Online Peers: " + response)      # 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()    # peer is started  main = peerMain() |

# Application Preview

## Starting Applications

1. (Starting Server) Run registry.py on server device
2. Provide the user with IP address of server device
3. (Starting user-side application) Run peer.py
   1. The user is prompted to enter the provided IP address of the server device
   2. The user is given a main menu of all features

A screenshot of a computer program

Description automatically generated

Figure a: (Starting Server) Registry.py

A screenshot of a computer program

Description automatically generated

Figure 1b: (Starting user app) peer.py

## Create Account

1. User is prompted to enter username and password to create the account
2. Account is validated on the server side on the database
3. Similar account is found
   1. The user is prompted to enter the provided IP address of the server device, the user is re-prompted to enter different credentials and returns to menu
4. Account is successfully created
   1. The user is notified and The account is successfully created and the hashed credentials are stored in the database

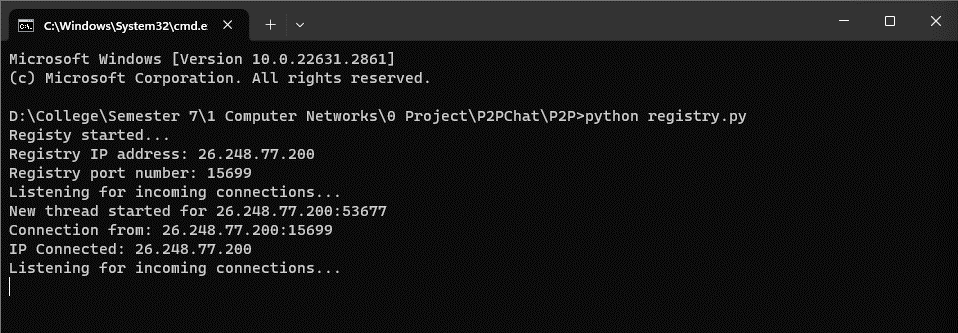


Figure 2a: Server-side of account creation

A screenshot of a computer

Description automatically generated

Figure 2b: Client-side account creation

A screenshot of a computer

Description automatically generated

Figure 2c: Database View

## Login

1. User is prompted to enter username and password to login
2. Unsuccessful login
   1. User is notified and is redirected to menu
3. Successful login
   1. User is notified
   2. The peer application keeps sending hello message to server
   3. The server keeps a timer to check for hello messages if they are not sent and the timer timeouts the client is removed from online users.

A screenshot of a computer screen

Description automatically generated

Figure 3a: Server-side login

A screenshot of a computer

Description automatically generated

Figure 3b: User-side login

## Get online peers

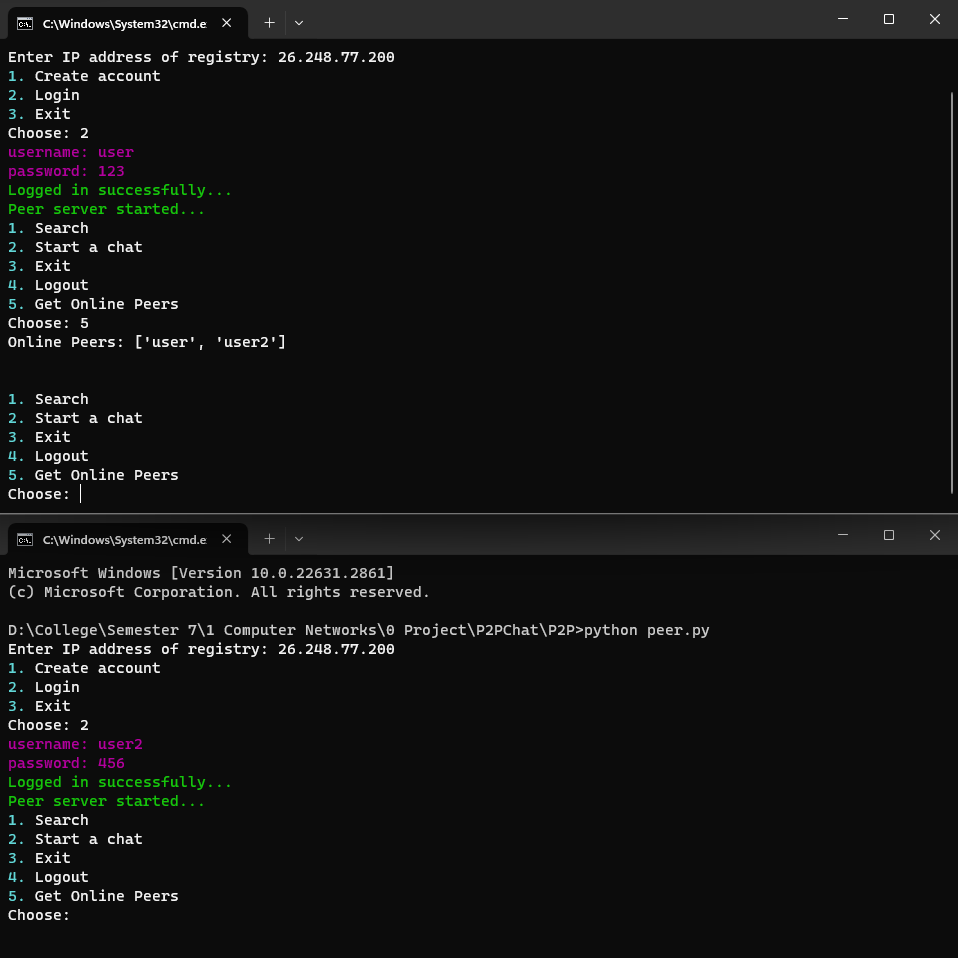


Figure 4: Getting online peers when 2 users are online

To avoid program crashes, the user is supposed to logout before exiting the program.

To logout enter 4 🡪 the program is reset and the user is required to re-enter the server ip.

To fully exit the program enter 3

Note: In some cases the user is required to exit the program twice to ensure the account is logged out and removed from the list of the online peers on the server.