WHISPER DESIGN DOCUMENT

Prepared by:

Moamen Ahmed Ali Amin 20p9447

Donia Ahmed Nasr 20p9451

Mohamed Khaled Elazhary 20p9416

Mina Shawket Shaker 20p9476

Contents

Introduction

Project Scope

The Whisper project focuses on developing a Peer-to-Peer Multi-User Chatting Application that emphasizes text-based communication, drawing inspiration from popular platforms like Clubhouse. The application will use both client-server and peer-to-peer architectures to facilitate group and one-to-one conversations. A custom protocol is made to match the application's functionalities to ensure efficient communication.

The main implementation of the project will include:

1- Client-Server Architecture:
   - Implementation of a basic server application capable of handling multiple client connections.
   - TCP connection for secure user authentication.
2- Peer-to-Peer Architecture:
   - Integration of peer-to-peer connections to enable efficient communication between users.
   - Specialized protocol to support peer-to-peer interactions.
3- Group and One-to-One Conversations:
   - Implementation of features allowing users to engage in chat rooms.
   - Support for one-to-one conversations using the peer-to-peer architecture.
4- Custom Protocol:
   - Development and implementation of a custom protocol tailored to match Whisper's functionalities.
   - Utilization of the protocol for reliable and efficient communication.

Goal and Objectives

Our goal is to create a simple and intuitive chatting application that uses peer to peer communication for sending chat messages.

Using peer to peer for chatting would significantly reduce the load on the server and boost the scalability of the of the chat room.

In this first phase we use TCP and simple broadcasting methods for group chatting later we intend to optimize the chatting by instead of using broadcasting we can use advanced techniques like the ones used in BitTorrent for example.

Functionalities

Functional requirements

1) User Authentication
   - Users can create an account with a unique username and password.
   - Users must securely log in using their registered credentials.
2) Chat Room Functionality
   - Users can create new chat rooms.
   - Users can join existing chat rooms and see a list of available ones.
3) Group Messaging in Chat Rooms
   - Users can send messages to everyone in the chat room.
   - Users receive real-time messages in the chat room.
4) One-to-One Chat Functionality
   - Users can send messages to everyone in the chat room.
   - Users receive real-time notifications for new messages in the private chat.
5) User Status
   - Implement user status indicators (online, offline) visible to others in the chat.

Non-functional requirements

1) Usability
   - Clear instructions for account creation, chat room interactions, and private messaging.
   - Understandable error messages to the user to easily use the app.
2) Security
   - User credentials are securely stored using encryption and hashing.
   - Implement measures for preventing unauthorized access.
3) Scalability
   - The system should be designed to handle a growing number of users and chat rooms efficiently without loss in performance.
4) Portability
   - The application should be designed to run on multiple platforms, ensuring compatibility with various operating systems.
5) Performance
   - Communication protocols (TCP and UDP) are optimized for efficient message delivery.
6) UI command line
   - CLI supports straightforward and intuitive commands.
   - Implementation of color-coded messages for visual distinction.
7) Support for hyperlinks:
   - Hyperlinks are highlighted and the user can open hyperlinks.

Authentication Mechanism

User Registration

- Users register with the system by providing a unique username and a secure password.
- Passwords are hashed using Argon2 hashing algorithm and saved in the database.

User Login

- When a user attempts to log in, the server receives a login request containing the provided username and password.
- The server retrieves the hashed password associated with the given username from the database.

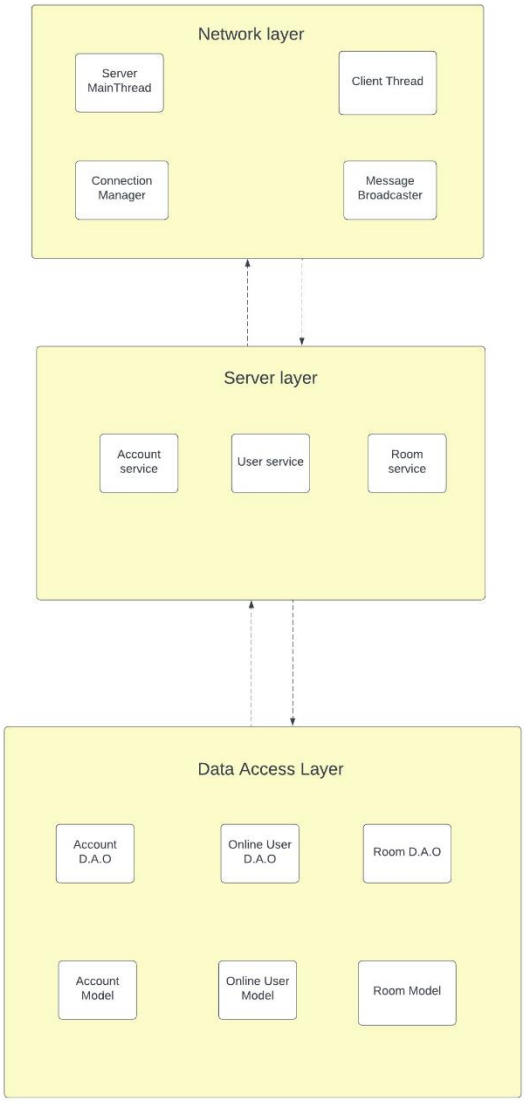
Password Hash Verification

- The server hashes the provided password using the same hashing algorithm used during registration.
- It compares the calculated hash with the stored hash in the database.
- If the hashes match, the password is correct, and authentication is successful.

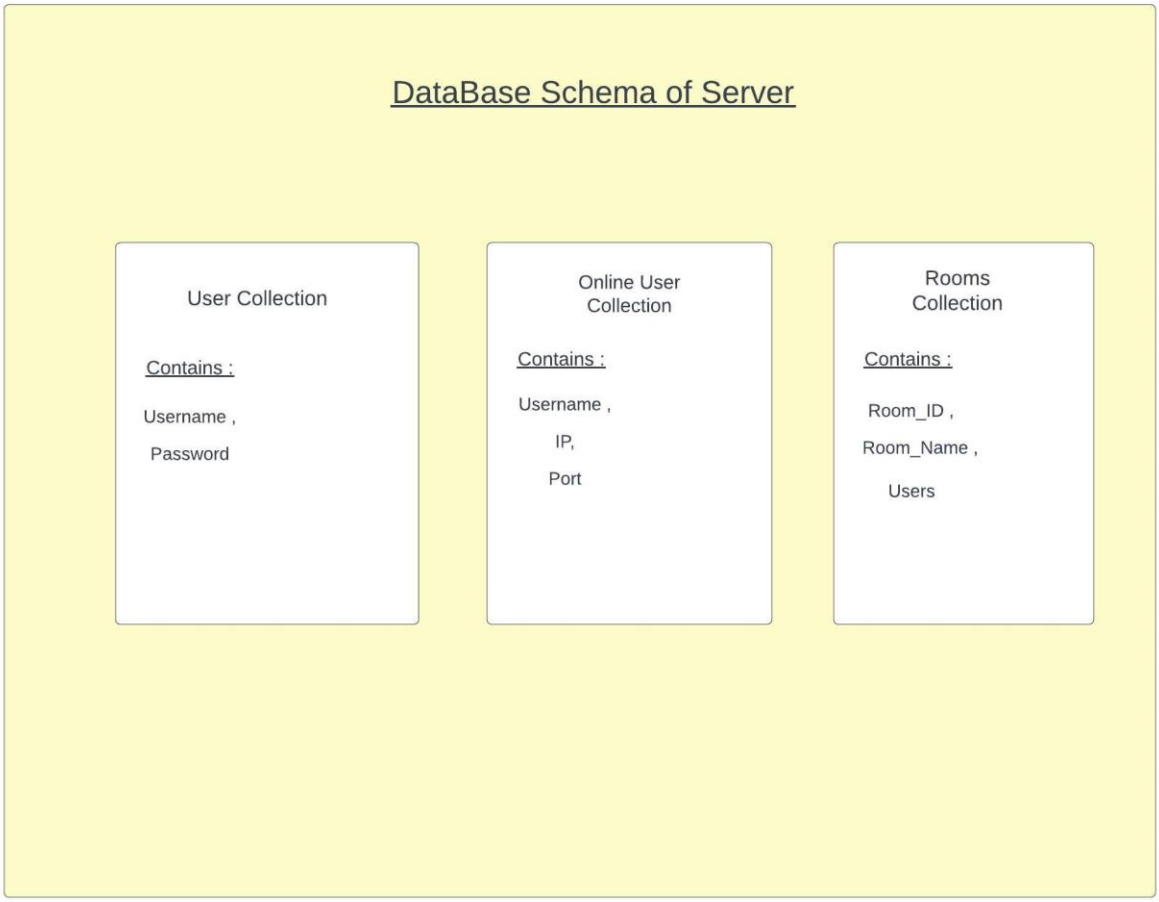
System Architecture

The system Architecture is divided into Server architecture which handles the server side jobs and Peer Architecture which contains the applications that the peers run.

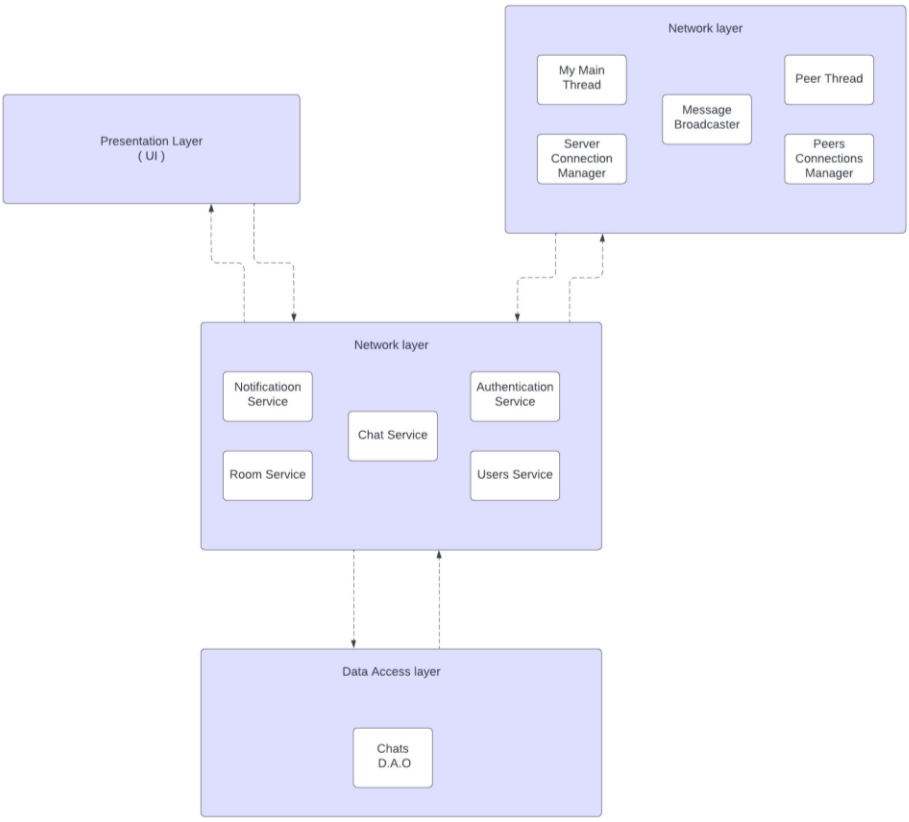
Server Architecture:



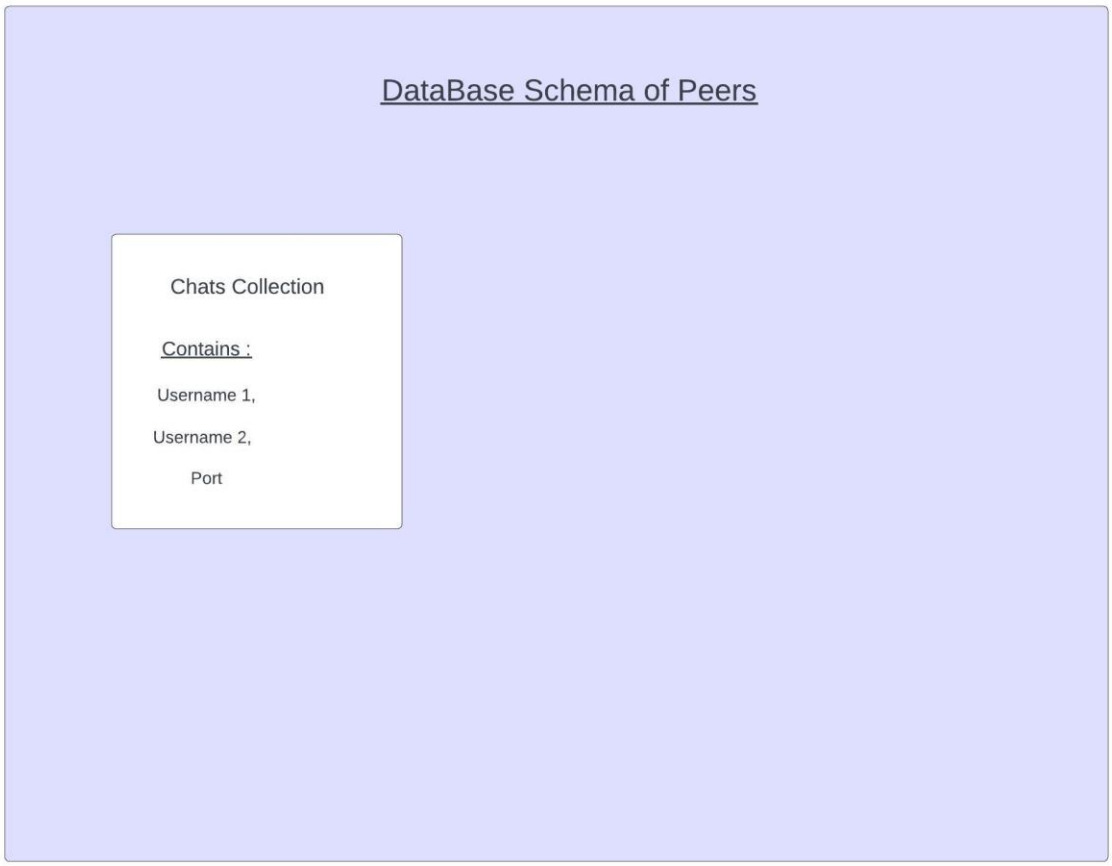
Server Database



Peer Architecture



Peer Database



Application Protocol

Client-Server (TCP)

User Request: Create Account (`CREATE_ACC`)

The `CREATE_ACC` command is used by clients to request the creation of a new user account. This command includes the desired username and password for the new account.

- `<username>`: The desired username for the new account.
- `<password>`: The password associated with the new account.

Server Response:

Upon receiving the `CREATE_ACC`, the server responds with a status code indicating the outcome of the account creation request.

If the requested username is available and the account is successfully created, the server responds with:
STATUS_CODE: 1 (ACC_CREATED)

If the requested username is already taken, and the account creation fails, the server responds with:
STATUS_CODE: 0 (USERNAME_TAKEN)

User Request: Login (`LOGIN`)

The `LOGIN` command is used by clients to authenticate and log into an existing account. This command includes the username and password associated with the user account.

- `<username>`: The username of the existing account.
- `<password>`: The password associated with the account.

Server Response:

Upon receiving the `LOGIN` command, the server responds with a status code indicating the outcome of the login attempt.

If the provided username and password match a valid user account, the server responds with:
STATUS_CODE: 1 (LOGIN_SUCC)

If the authentication fails (e.g., incorrect username or password), the server responds with:
STATUS_CODE: 0 (AUTH_FAIL)

User Request: Logout (`LOGOUT`)

The `LOGOUT` command is used by clients to request logging out. This command includes the username of the user initiating the logout.

- `<username>`: The username of the user requesting the logout.

Sequence Diagram for Sign-up/Login and Logout

User Request: Create Room (`CREATE_ROOM`)

The `CREATE_ROOM` command is used by clients to request the creation of a new chat room. This command includes the username of the user initiating the request.

- '<username>': The username of the user creating the chat room.

Server Response:

Upon receiving the `CREATE_ROOM` command, the server responds with a `1 ROOM_CREATED` message indicating the successful creation of the chat room. Additionally, the server generates a unique room ID and includes it in the response.

- <room_id>: ID of the newly created room.

User Request: List Rooms (`LIST_ROOMS`)

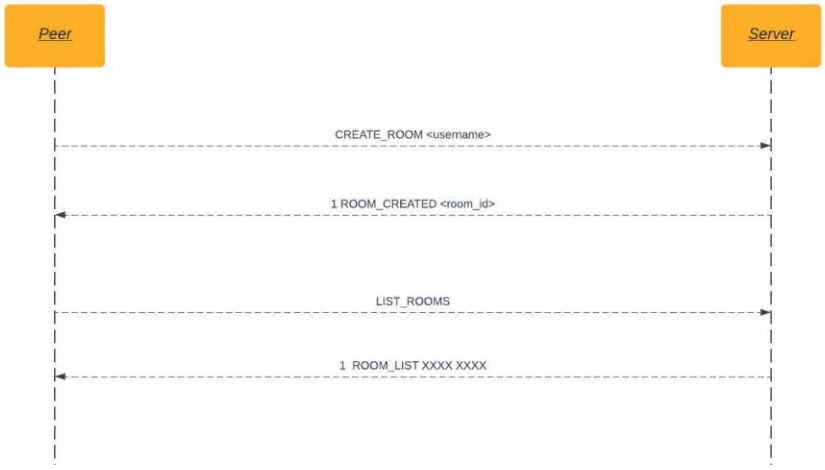
The 'LIST_ROOMS' command is used by clients to request a list of all available chat rooms. This command does not require additional parameters.

Server Response:

Upon receiving the `LIST_ROOMS` command, the server responds with a status code indicating a successful request for the list of rooms. Additionally, the server includes information about all available chat rooms.

If the list of chat rooms is successfully retrieved, the server responds with: STATUS_CODE: `1 ROOM_LIST (Rooms)`

Sequence Diagram for Creating and Listing Rooms



User Request: List Users (`LIST_USERS`)

The `LIST_USERS` command is used by clients to request a list of all online users.

Server Response:

Upon receiving the `LIST_USERS` command, the server responds with a status code indicating a successful request for the list of Online. Additionally, the server includes information about all users.

If the list of users is successfully retrieved, the server responds with: STATUS_CODE: `1 USERS_LIST (users)`

Sequence Diagram for Listing Users

Client-Server (UDP)

User Update: Online Status (`KEEP`)

The `KEEP` message is sent by clients to inform the server about their online status. This UDP message is used to update the server on whether the user is currently online.

- <username>: The username of the user sending the KEEP message.

Sequence diagram for keeping connection

Peer-To-Peer (TCP)

Peer Action: Sending Chat Message (`CHAT`)

When peer1 wishes to communicate with peer2 in a private chat, peer1 sends a `CHAT` message to peer2 and vice versa, passing along the username.

- `<username>`: The username of peer1 sending the message.

Peer Action: Close Chat (`BYE`)

The `BYE` message is sent by peers to inform the other participant that they want to close the private chat.

- `<username>`: The username of the peer initiating the chat closure.

Sequence Diagram for private Chatting and Closing private Chat

Peer-Server-Peer (TCP)

User Request: Join Room (`JOIN_ROOM`)

The `JOIN_ROOM` command is used by clients to request joining a specific chat room. This command includes the room ID of the desired chat room.

- `<room_id>`: The unique identifier of the chat room the user wants to join.

Server Response:

Upon receiving the `JOIN_ROOM` command, the server responds with a status code indicating the outcome of the request.

- If the requested room is unavailable (doesn't exist), the server responds with: STATUS_CODE: 0 (ROOM_UNAVAILABLE)
- If the user is already a member of the requested chat room, the server responds with: STATUS_CODE: 0 (ALREADY_JOINED)

Else the server sends a `REQUEST_INFO` message to all the peers in the room. This message contains the user information (username, IP address, and port number) and is asking the peers in the room to send their information to the new peer so that he can start sending messages to them.

- `<username>`: The username of the user that wants to join the room.
- `<ip>`: The IP address of the user that wants to join the room.
- `<port>`: The port number of the user that wants to join the room.
- `<room_id>`: The ID of the room of interest.

Peers Response:

Upon receiving the `REQUEST_INFO` message, the peers in the room send to the address specified in the message their information in a `PEER_INFO` message.

- `<username>`: The username of the peer in the chat room.
- `<room_id>`: The ID of the room of interest.

Sequence Diagram for joining room

User Request: Leave Room (`LEAVE_ROOM`)

When a peer decides to leave a chat room, they initiate the `LEAVE_ROOM` command, notifying the server of their departure.

Peer Request: LEAVE_ROOM <room_id>

- `<room_id>`: The unique identifier of the chat room the peer wants to leave.

Server Response:

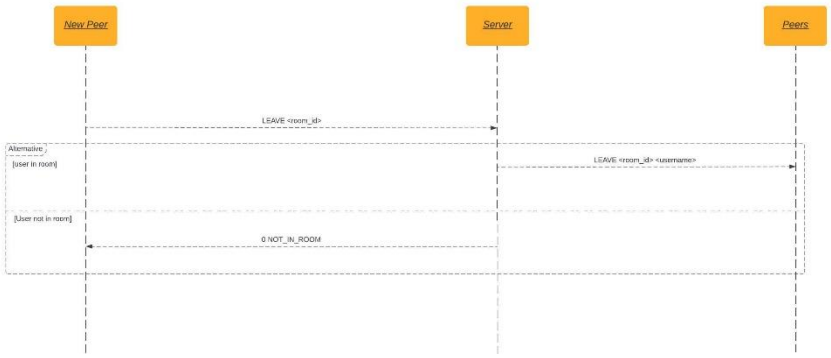
Upon receiving the `LEAVE_ROOM` message, the server check if the user was actually in the room, if so, the server sends `LEAVE_ROOM` message to other peers in the room:

- `<room_id>`: ID of the room the peer left.
- `<username>`: username of the peer that left the room.

If the user is not in the room, the server replies with a `0 NOT_IN_ROOM` message.

Note: The `LEAVE_ROOM` message is overloaded, if the server receives `LEAVE_ROOM`, it notifies the peers in the room to remove the user from the room. If a peer receives a `LEAVE_ROOM`, this means that the server wants it to remove another peer from the room.

Sequence Diagram for Leaving Room



User Request: Chat Request (`CHAT_REQUEST`)

The `CHAT_REQUEST` command is used by clients to send a chat request to another user.

`<username>`: The username of the user to whom the chat request is being sent.

Server Response:

Upon receiving the `CHAT_REQUEST` command, the server responds with a status code indicating the outcome of the request.

If the user is not online the server responds with `0 USER_NOT_ONLINE`

If the user is online the server sends a `REQUEST_INFO` message to the other peer:

- `<username>`: The username of the user that wants to chat.
- `<ip>`: The IP address of the user that wants to chat.
- `<port>`: The port number of the user that wants to chat.

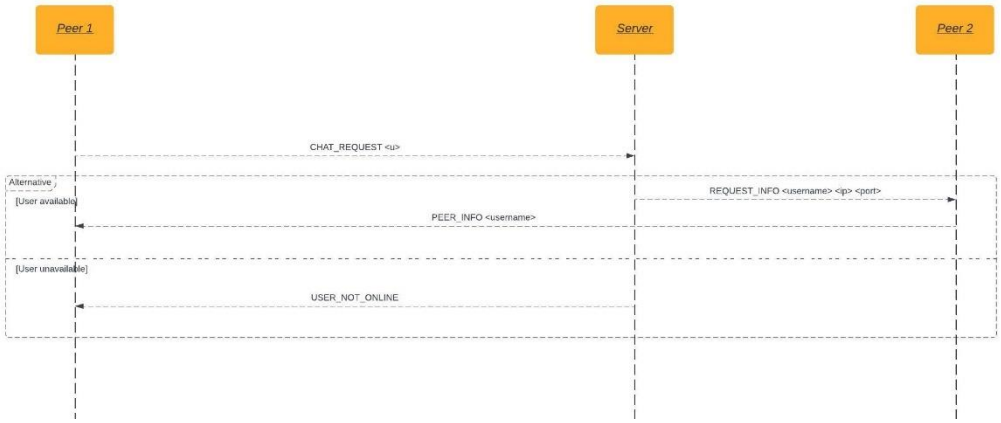
Peer Response:

Upon receiving the `REQUEST_INFO` message, the other peer sends a `PEER_INFO` message to the requesting peer to start the connection and the chat.

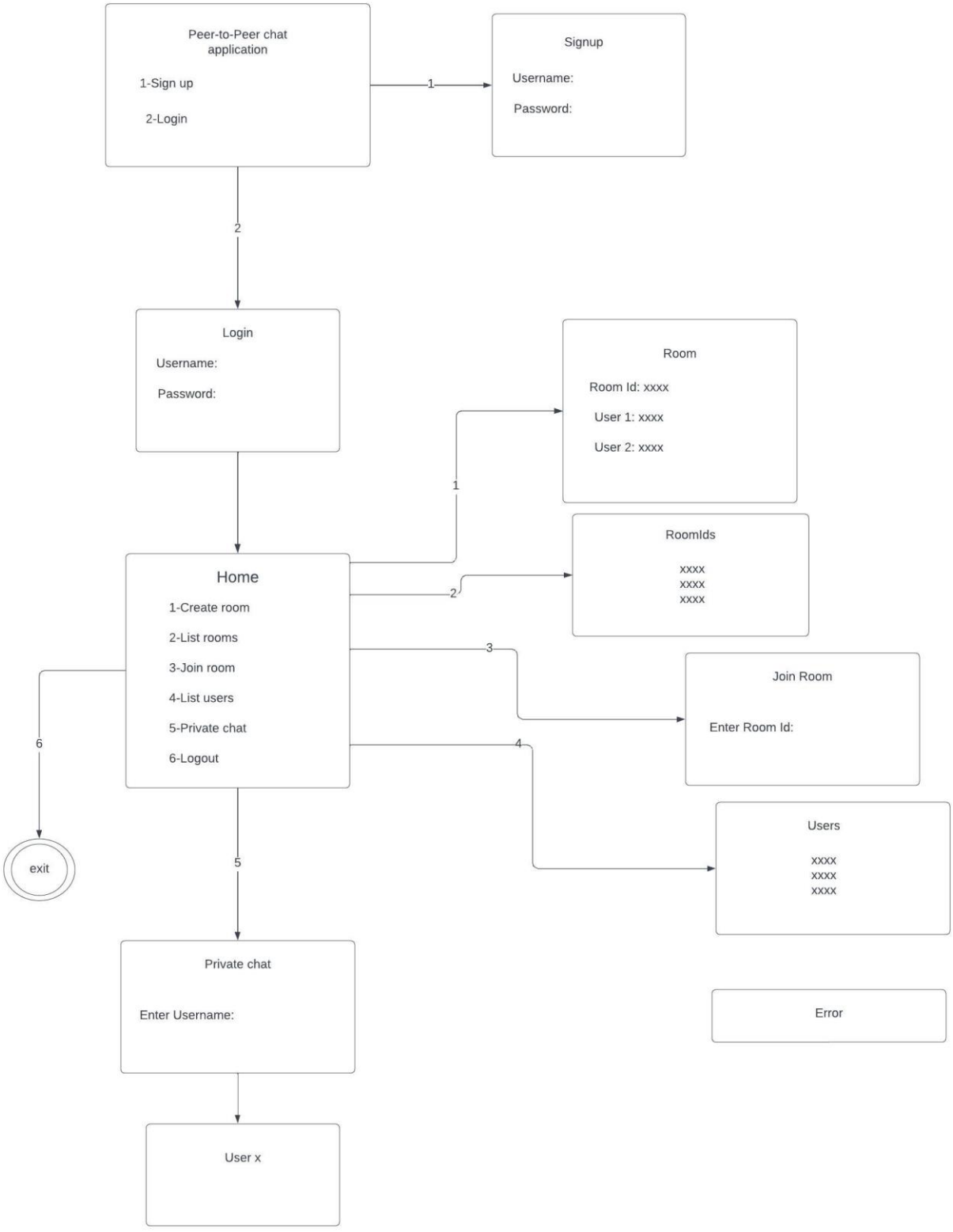
- `<username>`: The username of the user that accepted the chat.

Note: `PEER_INFO` message is overloaded, if it contained a room ID then the receiver is getting information of a peer in a room which he wants to join. If the message did not contain a room ID then this is the information of another peer which he requested to chat with.

Sequence Diagram for requesting private chat



UI State Diagram



Further optimizations

The current communication mechanism of group chatting is that when a peer wants to send a message, he simply sends the message to all the TCP sockets of the peers he is currently connected to. Alternatively, the peers could be connected by a network where each peer is connected only to a small number of peers, and when a message is sent the message can be relayed through the graph in a behavior similar to breadth first search where each peer relays the message only to the peers he is connected to. This would of course introduce message latency since the message would need to hop between multiple users, for this the graph data structure can be optimized to resemble a balanced tree, this way the number of hops for the message is minimized. This discussed method would need further research and could be implemented in later phases.

Another optimization is to transfer the database carrying the information about the current users in a room from the server to the peers. In this implementation, each peer would carry a copy of the other users in the chat room. This would reduce the load on the server to manage an increasing number of chatting rooms and for a peer to join a room he would only get the information about the owner of the room from the server.

In conclusion, the current system is designed for simplicity and the strategy is to build a simple system and then optimize it. Currently peer to peer communication is only used for delivering chat messages, which significantly reduces the load in the server because chat messages are assumed to be the bulk for the communication. In later phases we intend to research to offload more tasks from the server and to assign them to the peers to boost scalability.