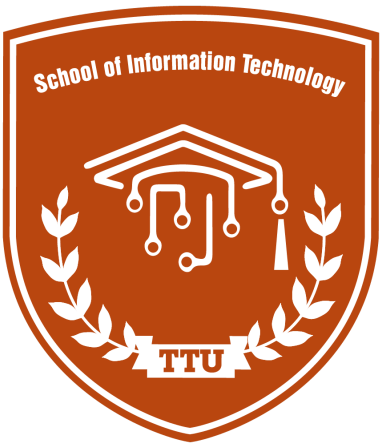


Socket Programming Project

“A Multi-user Chat Application”

Nguyen Cao Dien - 2102048

Ho Phuoc Lanh - 2102114

Vu Phuong Anh - 2102176

Lien Hai Nam - 2102115

School of Information Technology, Tan Tao University

CS440: Computer Networks

Dr. Truong Huu Tram

Dr. Le Quoc Huy

June 16, 2024

INTRODUCTION

In the era of digital communication, real-time interaction has become a crucial aspect of both personal and professional communication. The Socket Programming Project, titled "A Multi-user Chat Application," aims to create a robust platform that facilitates seamless and instantaneous communication among multiple users. This project leverages the power of WebSockets to establish a persistent connection between clients and the server, enabling real-time data transmission. By incorporating essential features such as user authentication, private messaging, and group chat capabilities, this application provides a comprehensive solution for modern communication needs.

PROJECT OVERVIEW

A socket is one endpoint of a two-way communication link between two programs running on the network. The socket mechanism provides a means of inter-process communication (IPC) by establishing named contact points between which the communication takes place.
This project uses sockets to create a Multi-User Chat Application that allows users to create and join group chats. It features user authentication, private messaging, group chat rooms, and a persistent display of users on the dashboard.

Implementation Details

1. **Server:**
    a. **Introduction:**
        - The server-side implementation of a Websocket-based chat application designed for real-time communication. The server handles user authentication, message routing, and room management, ensuring seamless interaction between clients. The server is built using Python, leveraging the Websocket library to maintain persistent connections.

    b. **Server Architecture:**

    The server architecture is centered around three main components:
        - **WebSocket Server**: facilitates connection handling, message routing, and client management.
        - **MongoDB Database**: stores user credentials and chat history.
        - **Chat Manager**: manage chat rooms, broadcasts messages, and handles user memberships.

c. **Core Functionalities:**

- **WebSocket Server Initialization**: the server is initialized to listen for incoming WebSocket connections on port '6789'. This is achieved using Python's 'websocket.serve' method, which sets up the server to run indefinitely, awaiting client connections.
- **Client Connection Handling**: when a new client connects, the server adds the Websocket connection to the set of connected clients. It also initiates a loop to asynchronously listen for incoming messages from the client. Each message is processed based on its type.
- **User Authentication**: authentication is a crucial aspect of the server, ensuring that only verified users can interact with the chat system. When an authentication message is received, the server validates the provided token against the MongoDB Database. Upon successful validation, the server sends an acknowledgement back to the client and updates the client's connection status.
- **Private Messaging**: the server supports private messaging between users. It uses a unique room name, generated from the sorted phone numbers of the sender and receiver information from the database and ensures that messages are delivered to the correct user.
- **Room Management**: Users can join existing chat rooms or create new ones. When a join room request is received, the server adds the user to the specified room, allowing them to participate in the conversation. Similarly, the server handles requests for creating new rooms, ensuring that each room has a unique name.
- **Group Chat** : the server also facilitates group chats, allowing users to create chat rooms involving multiple participants. When a create group chat request is received, the server creates a new room and invites the specified users, ensuring they can all join the conversation.

2. **User Authentication**

a. **Definition:**

1. Authentication is the process of verifying that a fact or document is genuine. In computer science, authentication typically involves verifying a user's identity. This is

usually done by the user providing credentials, which are agreed-upon pieces of information shared between the user and the system.

2. The user authentication process in this API includes checking credentials, generating and saving a security token, sending the token to the WebSocket server for authentication, and using the token to authenticate the user to other endpoints. This ensures that only authorized users can access and use the API's resources and services.

b. **Process:**

- Receive user sign in information including phone and password through request body. After, search for the user in the database based on the phone number and check if the password from the request matches the password in the database.

1. Sign In (/signin endpoint):

1.1. **Functionality**: Authenticates a user and issues a token.

1.2. **Request**: Expects a JSON object with phone and password.

1.3. **Process**:

1.3.1. Retrieves the user by phone number from the database.

1.3.2. Checks if the provided password matches the stored password.

1.3.3. If the credentials are valid, generate a token and save it in the database.

1.3.4. Sends the token to a WebSocket server for additional authentication.

1.3.5. If WebSocket authentication is successful, it returns the token and user details.

1.3.6. If authentication fails, it returns an error.

2. Token Validation:

2.1 WebSocket Integration: The /signin endpoint uses the WebSocket server to validate the authentication token. This is done by sending the token to the WebSocket server and awaiting its response.

3. Access Control:

3.1 Endpoints with Token Validation: Some endpoints require token validation to ensure that the user making the request is authenticated. For example, the /search_rooms endpoint checks the token before retrieving the rooms.

3. Room Management

Chat management is a critical component in modern communication systems, especially in multi-user chat applications. It encompasses the mechanisms and processes that facilitate the creation, maintenance, and operation of chat rooms where users can interact in real-time. Effective chat management ensures seamless communication, user engagement, and the overall reliability of the chat system.

This part delves into the intricacies of chat management within our multi-user chat application. We will explore its functions, including creating, joining, leaving, and removing chat rooms, the interaction between users and APIs, and the relationship between chat management and data storage.

What is Chat Management?

Chat management refers to the administrative and operational activities required to maintain a chat system. It involves managing chat rooms, user sessions, message broadcasting, and ensuring proper synchronization among participants. In a multi-user chat application, chat management is pivotal for handling concurrent connections and facilitating smooth communication between users.

Functions of Chat Management

Chat management in our application is implemented through a series of functions that interact with users and the system's backend. Here, we provide a detailed explanation of these functions:

The primary functions of chat management in our application include:

***Creating Chat Rooms*:**

Establishing new chat rooms where users can join and interact. The function generates a unique name for the new chat room. This is critical to avoid conflicts with existing rooms and to ensure that each room can be uniquely referenced.

The create_room function initializes a new chat room in the system. This involves assigning a unique name to the room and storing its details in the database. When a room is created, it is added to the list of available rooms, making it accessible for users to join.

Database Storage: The details of the new room, such as its name, creation timestamp, and participants, are stored in the MongoDB database. This ensures persistence, meaning the room will be available even if the server restarts. Data Stored in the Database:

- Room Name: A unique identifier for the room, allowing users to join by name.

- Creation Timestamp: The date and time when the room was created, providing context for the room's activity.
- Participants: An initial list of users who are part of the room at the time of creation. This can be empty initially if no users are specified during creation.
- Room Metadata: Any additional information relevant to the room, such as the room's purpose, description, or settings.
- Database Interaction: The function uses an instance of the MongoDB class to interact with the database. The create_room method of this class adds a new document to the rooms collection, containing the room's name and metadata. The MongoDB document structure might look like this:

```
{
  "name": "unique_room_name",
  "created_at": "2024-06-15T12:34:56Z",
  "users": [],
  "metadata": {
    "description": "This is a sample chat room",
    "settings": {
      "is_private": false,
      "max_participants": 50
    }
```

Room List Update: The newly created room is added to an in-memory list of available rooms. This list is kept up-to-date to facilitate quick access and efficient room management. The in-memory list allows the server to quickly reference active rooms without querying the database each time, enhancing performance. The room's details are stored in a dictionary or similar data structure within the ChatManager class, enabling fast lookups and modifications. For example:

```
{
  "unique_room_name": {
    "created_at": "2024-06-15T12:34:56Z",
    "users": []
  }
```

***Joining Chat Rooms*:**

Allowing users to enter existing chat rooms and participate in discussions. The first step in the join function is to verify the existence of the specified chat room. It prevents errors that could arise from attempting to join a non-existent room.This method queries the rooms collection in MongoDB to find a document that matches the specified room name. If the room is found, the method returns the room's details. If not, it returns None, indicating that the room does not exist. Once the room's existence is confirmed, the user's WebSocket connection and username are added to the room's participant list, which is stored in-memory.

Moreover, The room's participant list is a data structure (e.g., a dictionary or list) maintained in the ChatManager class. The participant list is updated to include the new user's WebSocket connection and username. This ensures that the user is registered in the room and can participate in real-time communication. After adding the user to the room, a system message is broadcasted to all current participants of the room, informing them of the new member. A JSON message is created to notify participants of the new user's entry.

The message includes the type of message (system), the event (join), and the content of the message (e.g., the username and room name).

```
{
   "type": "system",
   "event": "join",
   "message": "username has joined the room room_name."
}
```

The function uses the broadcast_message function to send the notification. The broadcast_message function constructs the JSON message and sends it to all WebSocket connections in the room except the new user's. This ensures that the new user's entry is communicated to all other participants in the room.The broadcast_message function uses asyncio to handle asynchronous message sending, ensuring non-blocking communication.The function employs asyncio to send messages concurrently without blocking the main execution thread. This is achieved using asyncio.gather, which aggregates multiple asynchronous tasks and runs them concurrently. Each task involves calling the *safe_send* method to send a message to a specific WebSocket connection. This method

method handles potential connection errors gracefully, ensuring that the application remains stable even if some connections are closed.

***Leaving Chat Rooms*:**

Enabling users to exit chat rooms, with proper notifications sent to other participants. The leave function handles user exits from chat rooms, ensuring proper removal from the participant list and notifying remaining users.It identifies the user's session that needs to leave the room by matching the WebSocket connection with the room's participant list. Let see the detailed process below:

User Identification: The first step in the leave function is to identify the user's session that needs to leave the room. This is done by matching the WebSocket connection with the room's participant list.

Participant List Search: The function iterates through the participant list of each room to find the user's WebSocket connection. This ensures that the user is correctly identified, even if they are part of multiple rooms.

Participant List Update: Once the user is identified, they are removed from the room's participant list. This update is crucial for maintaining the current state of the room's active participants.

In-Memory Data Structure: The participant list, stored in-memory, is updated to remove the user's WebSocket connection. This ensures that subsequent operations on the room do not involve the user who has left.

Database Update: If the system uses persistent session storage, the participant's removal is also updated in the database. This ensures that the change is reflected across system restarts and maintains data consistency.

Database Interaction: The function would call a method to update the participant list in the MongoDB database. This typically involves removing the user's identifier from the room's document in the rooms collection.

Notification to Participants: After updating the participant list, a system message is broadcasted to the remaining participants, informing them of the user's departure. This is important for keeping all participants aware of the current room status.

Message Construction: A JSON message is created to notify participants of the user's departure. The message includes the type (system), the event (leave), and the content of the message (e.g., the username and room name).

```
{
   "type": "system",
   "event": "leave",
   "message": "username has left the room room_name."
}
```

Message Broadcasting: Similar to the join process, the departure message is sent using the broadcast_message function. This ensures that all users in the room are informed of the change. The broadcast_message function constructs the JSON message and sends it to all WebSocket connections in the room. Asyncio is used to handle the message broadcasting asynchronously, ensuring non-blocking communication.If the user has no other active sessions in the application, their WebSocket connection is closed gracefully. This step ensures that server resources are freed and that the user is properly logged out of the system. The WebSocket connection is closed using the appropriate method, ensuring that any remaining resources are released.

***Removing Chat Rooms*:**

Deleting chat rooms when they are no longer needed, ensuring that resources are freed up.The first step in the remove function is to identify the room that needs to be deleted. This is crucial to ensure that only the specified room is removed from the system. The function checks the in-memory list of active rooms to identify the room to be deleted. This list is maintained in the ChatManager class for quick access and management.To confirm the room's existence in the database and retrieve its details, the function calls the get_room_by_name method of the MongoDB class. This method queries the rooms collection in MongoDB to find a document that matches the specified room name.Once the room is identified, all data associated with it, including participant lists and message history, is cleared from both the in-memory structures and the database. The participant list and any other room-specific data stored in memory are removed.The delete_room method of the MongoDB class is called to remove the room document from the rooms collection in the database, which ensures that the room data is permanently deleted and cannot be accessed in the future.

All current participants are notified of the room's deletion. This step ensures that users are aware that the room is no longer available for interaction. A JSON message is created to inform participants about the room's deletion. The message includes the type (system), the

event (delete), and the content of the message (e.g., the room name). A final message is sent to all participants in the room, informing them that the room is being deleted. The function uses the broadcast_message function to send the deletion notification to all WebSocket connections associated with the room. Asyncio is used to handle the message broadcasting asynchronously, ensuring non-blocking communication.

```
{
   "type": "system",
   "event": "delete",
   "message": "The room room_name has been deleted."
}
```

Any resources allocated to the room, such as memory and database records, are released and cleaned up. This step ensures that the system remains efficient and free of unnecessary data.The in-memory data structures holding room-related information are cleared to free up memory.The room document is deleted from the database using the delete_room method, ensuring that no residual data remains.

Chat management is a fundamental component of our multi-user chat application, enabling users to create, join, leave, and remove chat rooms efficiently. By integrating with the API and the MongoDB database, these functions ensure seamless communication and data management, contributing to a robust and scalable chat system. Through careful design and implementation, our chat management system supports real-time interaction and enhances the user experience in a dynamic and responsive manner.

Private Chat:

The create_private_room function is responsible for creating private chat rooms within the chat application, which ensures that only two specified users, identified by their phone numbers, can join and communicate with each other. For example, private chat's format must be :"private_room_1234567890_0987654321". The system checks the phone numbers of both users to ensure they exist in the database. If either phone number is invalid or does not exist, the room creation process is aborted.

4. Front-end

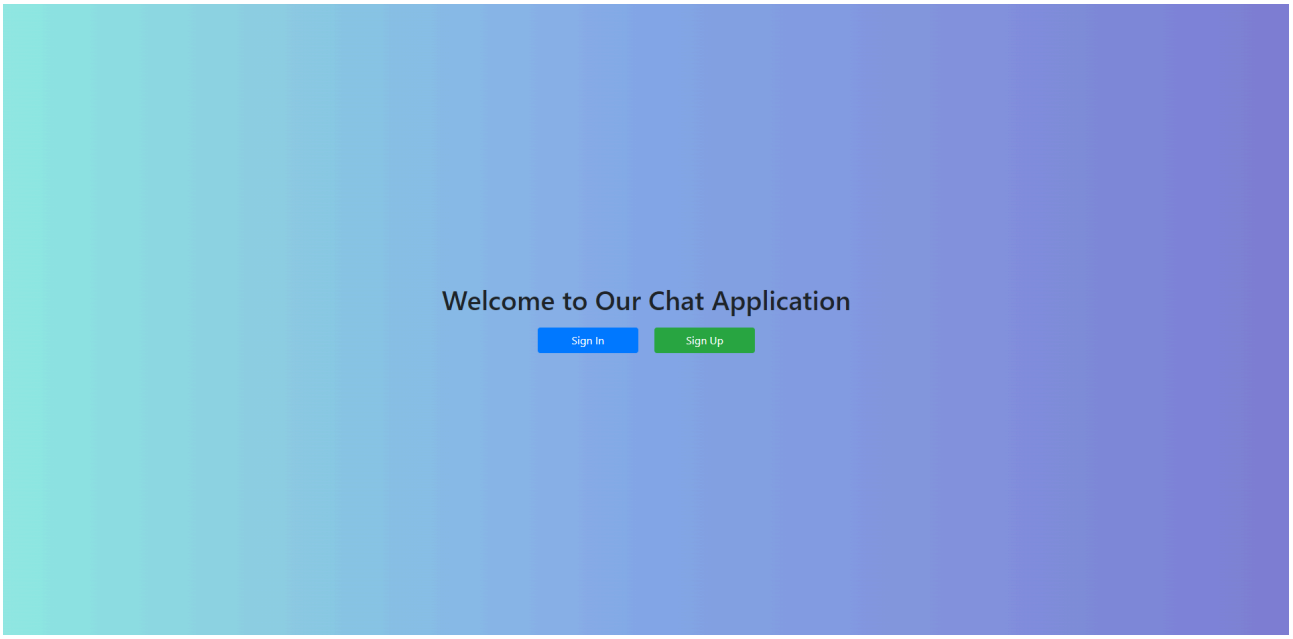
a. Design Choices

The front-end of the multi-user chat application is designed to provide a user-friendly and interactive experience. The application is built using HTML, CSS, and JavaScript,

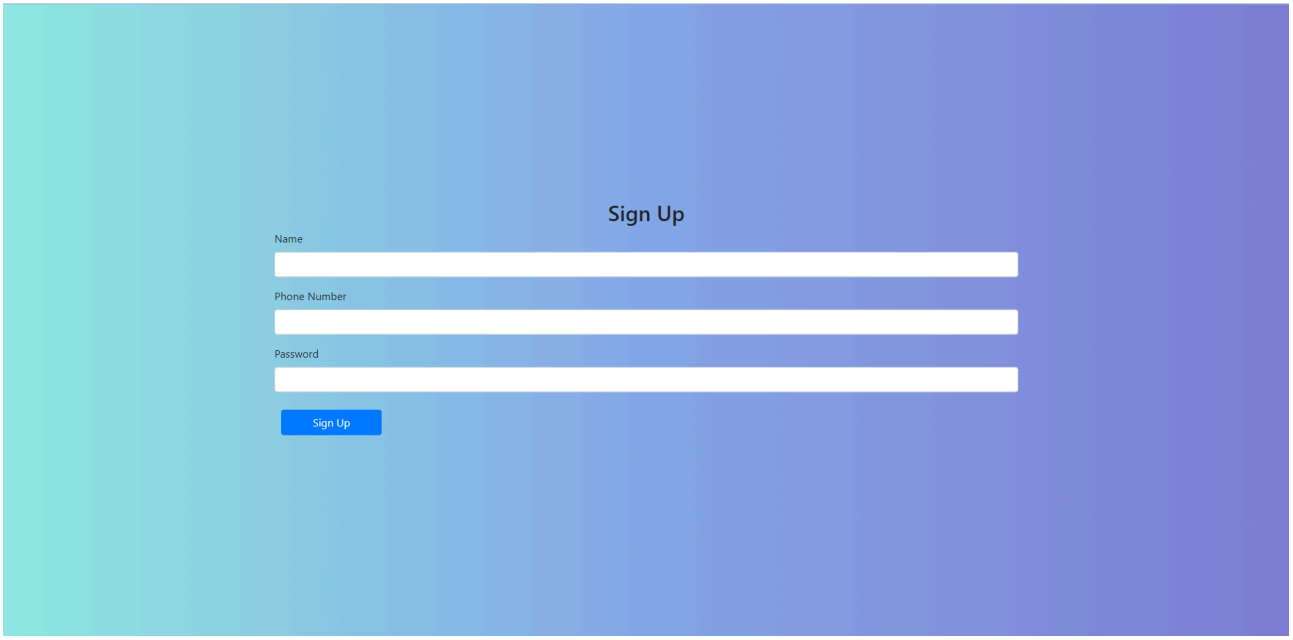
leveraging jQuery and Bootstrap for a responsive and attractive UI. The design choices focus on simplicity, ease of navigation, and real-time interaction to ensure a seamless chat experience.

b. User Interface Components

- Welcome Page: The welcome page offers two options: "Sign In" and "Sign Up." It uses Bootstrap for a clean layout and includes buttons that redirect users to the respective sign-in and sign-up pages.

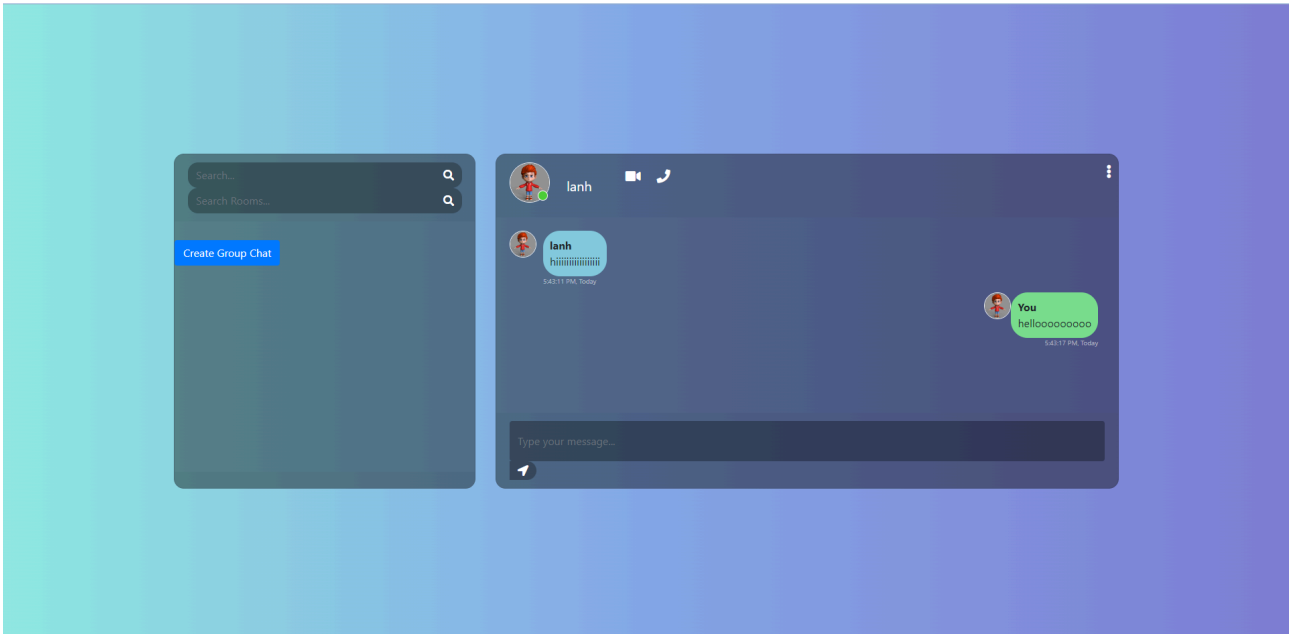


- Sign-Up Page: This page includes a form for new users to register by providing their name, phone number, and password. jQuery is used to handle form submission, sending the data to the backend via an AJAX POST request.



- Sign-In Page: The sign-in page includes a form for existing users to log in. Similar to the sign-up page, it uses jQuery for form submission and communicates with the backend to authenticate users.

- Chat Page: The main chat interface is divided into two sections:
  - Contacts Section: This section includes a search bar for users and rooms, lists of search results, and a button to create group chats.
  - Chat Section: This section displays the current chat room name, a message display area, and an input field for sending messages.



c. Algorithms and Data Structures

- WebSocket Connection: The chat application establishes a WebSocket connection to the server to handle real-time communication. When a user signs in, a WebSocket connection is initiated, and the user is authenticated via a token.
- Search and Display:
    - User and Room Search: As users type in the search input fields, AJAX GET requests are sent to the backend to search for users and rooms. The results are dynamically displayed in the respective lists.
    - Search Results Display: Search results are displayed as list items. Each result item includes the user's name and phone number or the room name, making it easy for users to identify and select.
- Message Handling:
    - Sending Messages: When a user sends a message, it is first checked for content and then sent to the server via the WebSocket connection. The message includes the sender's name, the room name, and the message content.
    - Receiving Messages: Incoming messages from the WebSocket server are displayed in the chat area. Messages are distinguished as either incoming or outgoing and are formatted accordingly for better readability.
- Group Chat Creation:
    - Modal for Group Chat: A Bootstrap modal is used to gather information for creating a group chat, including the group name and user phone numbers.
    - User Search for Group Chat: Users can search for other users to add to the group chat. Selected users are added to a list displayed in the modal.
- Data Storage and Retrieval:
    - Local Storage: The application uses local storage to store the authentication token and the user's phone number. This data is used to maintain the user's session and simplify subsequent requests.
    - Dynamic Content Loading: The application dynamically loads content such as search results and messages, ensuring the UI remains responsive and up-to-date with the latest data.

5. Challenges

1. **Real-Time Communication Management:**

- Challenge: Ensuring real-time communication between multiple users while maintaining data consistency and handling network latency was complex.
- Solution: Implementing WebSockets allowed for efficient real-time communication. Challenges with network latency were mitigated by using asynchronous programming with asyncio, ensuring the server could handle multiple connections simultaneously without blocking.

2. **Authentication and Security:**

- Challenge: Securing user authentication and ensuring only authenticated users could access the chat features.
- Solution: Tokens were used for user authentication. Upon sign-in, a token is generated and stored locally. This token is then used to authenticate WebSocket connections and API requests. Implementing CORS policies and ensuring secure data transmission over WebSockets also contributed to enhancing security.

3. **Dynamic UI Updates:**

- Challenge: Keeping the user interface updated with real-time data (e.g., incoming messages, user status updates) without causing performance issues.
- Solution: Efficient use of JavaScript and jQuery for DOM manipulation ensured that updates were smooth. Implementing a proper structure for message rendering and user list updates helped maintain performance and responsiveness.

4. **User and Room Search:**

- Challenge: Implementing a responsive and accurate search feature that could handle large datasets of users and rooms.
- Solution: AJAX calls were used for searching users and rooms dynamically. The backend was optimized to handle search queries efficiently, and results were rendered in real-time on the front end, providing a seamless search experience.

5. **Handling Multiple Chat Rooms and Private Chats:**

- Challenge: Managing multiple chat rooms and private chats simultaneously without causing confusion or data overlap.
- Solution: Each chat room and private chat was assigned a unique identifier. WebSocket messages were tagged with these identifiers to ensure they were routed to the correct rooms. The frontend UI was designed to clearly display which chat room or private chat was active.

6. **Cross-Browser Compatibility:**

- Challenge: Ensuring the application works consistently across different web browsers.
- Solution: Extensive testing was conducted on multiple browsers to identify and fix compatibility issues. Leveraging modern web standards and polyfills where necessary helped in maintaining cross-browser compatibility.

6. Future Improvements

1. **End-to-End Encryption**

Implementing end-to-end encryption for messages to ensure that only the intended recipients can read the messages, thereby enhancing security and privacy.

2. **Media Sharing**

Adding the capability to share media files (images, videos, documents) within chat rooms and private chats to make the application more versatile.

3. **Advanced Search and Filters**

Improving the search functionality with advanced filters and search options, allowing users to search for messages within chats, search by date, or filter users based on various criteria.

4. **User Presence and Status Indicators**

Adding features to display user presence (online/offline) and status (e.g., away, busy) to provide better context to the chat interactions.

5. **Message Reactions and Emojis**

Implementing message reactions and a rich set of emojis to enhance user interaction and engagement within chats.

6. **Administrative Tools**

Providing administrative tools for room moderators to manage users, moderate content, and maintain the overall health of chat rooms.

CONCLUSION

The development of the Multi-user Chat Application has provided significant insights into the complexities and potential of real-time communication systems. The project demonstrates the practical application of socket programming in creating interactive and responsive communication platforms through the implementation of WebSocket-based

architecture, user authentication, and dynamic chat room management. Despite encountering challenges such as dynamic UI updates and ensuring seamless user experiences, the project achieved its goals by employing efficient data handling and real-time synchronization techniques. Looking forward, the application has ample room for enhancements, including end-to-end encryption, media sharing, and advanced search functionalities, to further elevate user engagement and security.