Northeastern University

CS6650 Scalable Distributed Systems Spring 2023

Project Report- GPT Chatroom

Team: Kartikey Dwivedi, Sonal Varshney, Zheng Yin

Goal

The goal of this project is to create a multi-agent chat application where multiple clients can connect to a room of their choice and communicate with each other while also integrating OpenAI's GPT API as a chatbot that can answer clients' questions and engage in conversations. The chat rooms are designed to enable users to share text-based messages. The clients can specify when they want to chat with GPT or other connected clients. The application has a user friendly interface which allows users to easily join a chat room and send messages to other active clients. This system is designed as a multithreaded, fault tolerant distributed system that can handle a large number of concurrent users while also being able to recover from failures without affecting the overall functionality of the chat room.

The application provides a platform for users to chat with each other and ask ChatGPT questions, making it a useful tool for communication and learning.

High-level Design

The application uses a server-client architecture, where RMI (Remote Method Invocation) is used for communication between the server and clients. RMI allows remote objects to invoke methods on other remote objects, making it a suitable choice for distributed applications like this chat application.

The server is responsible for handling user registration, room creation, message storage, and integration with the ChatGPT API. Users will need to register with the server to access the chat rooms, and the server will maintain a database of all registered users. Room creation is also managed by the server, with users being able to join pre-existing rooms or create new ones. Messages sent by users will be stored on the server and retrieved when needed. The server also integrates with the ChatGPT API to allow users to ask questions and receive responses from the language model.

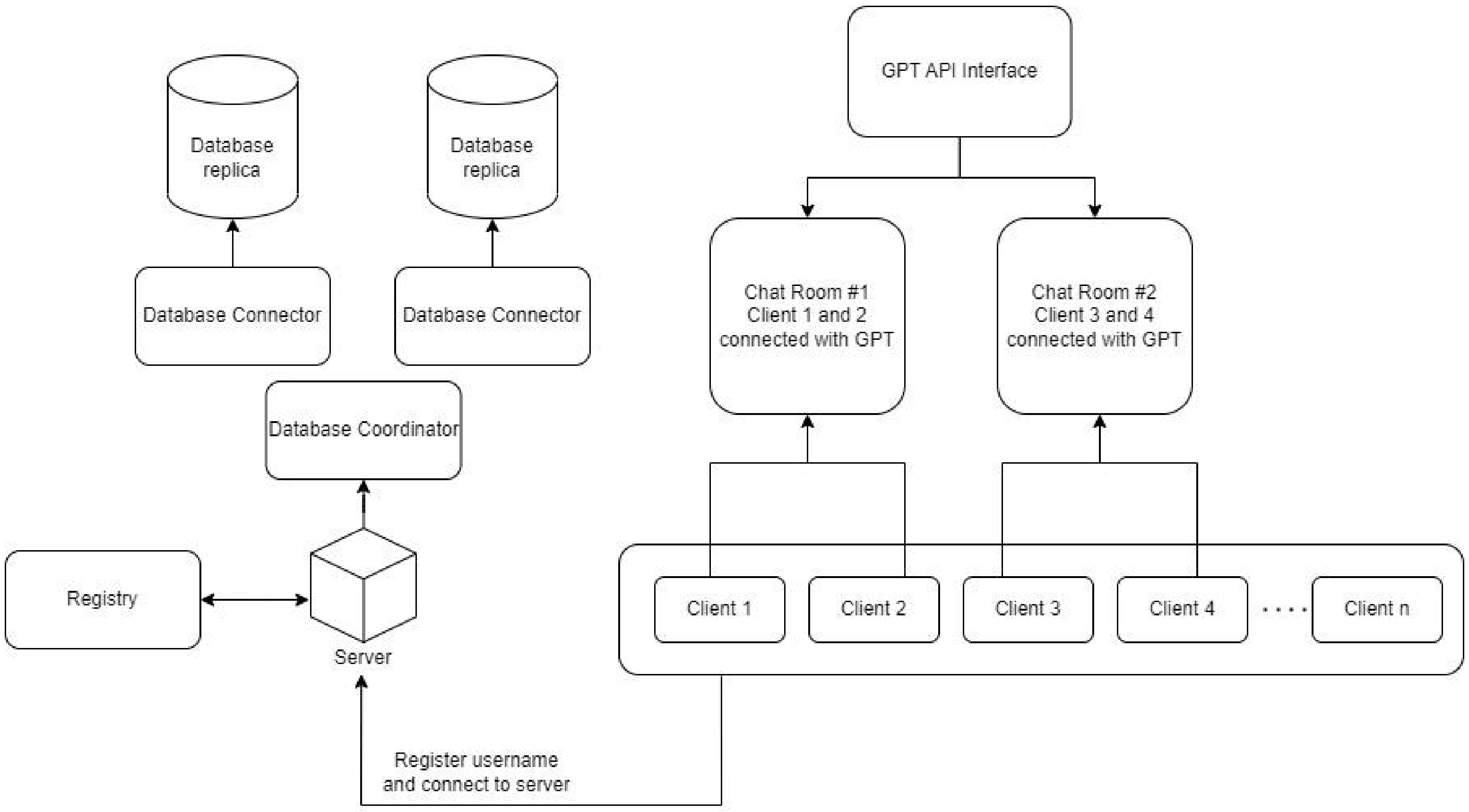
Clients can join chat rooms by providing a username and selecting a room. Once in a room, clients can send text messages that will be broadcast to all active users in the room. Clients will also be able to view the messages sent by other users in the room in their chat window.

The chat application uses Java Swing to create a login window and chat console for the logged-in user. The login window allows users to enter their username to access the chat room in the application, while the chat console displays messages sent by other users in the room. The chat console will also provide users with options to send messages, view active users, and view previous messages.

The application uses a MySQL database to store and retrieve messages sent by users in the chat rooms. The database allows for efficient querying and retrieval of messages based on user and room, and ensures data consistency across the application. The database is integrated with the server to ensure that messages are saved and retrieved correctly.

Database replicas are used for high availability. This ensures that the application remains available even if one of the replicas goes down. If one replica goes down, the application will still be able to access the database through the remaining replicas.

Architecture Overview



The architecture of the application is shown in the diagram above. There can be n clients which can connect to any chat room. If the chat room already exists, the client joins other active users on it, else a new room is created. In the diagram, it is shown that Client 1 and 2 have joined Chat Room #1 and Client 3 and 4 have joined Chat Room #2. This means that 1 and 2 chat with each other and can only see each other's messages.

There is a GPT API interface which provides real time response to any prompt entered by the user in any chat room of the application. Whenever the user wants GPT to reply, they can specify their message with a "@BOT" at the beginning of their message.

The client and server architecture is shown where multiple clients can connect to a server using RMI communication.

The server is responsible for registering the clients, creating rooms or adding clients to the rooms of their choice and broadcasting messages throughout active users in each room. The server stores and retrieves all the user information and messages sent in a MySQL database. The database uses distributed transactions to ensure data is inserted correctly to the tables. As seen in the diagram, there are two replicas of the database. This is to ensure high availability. In case one of the databases is down, the other database can retrieve data efficiently for the application. There is a database coordinator that uses the two phase commit protocol to make sure that the data being entered in both of the replicas is accurate and matching.

Tools and Libraries used

- The application is developed in Java for the backend
- Java Swing is used to develop the user interface for the chat room application
- Remote Method Invocation for client and server-side communications through simple method calls on the server object
- MySQL database to store the chat history and clients connected to the system
- Executor Service for handling thread pools
- ChatGPT API for getting real time responses

Key Algorithms

This application implements 4 main algorithms used in distributed systems to ensure scalability, availability, fault tolerance and robustness.

1. **Replicated Data Management**

In the multi-client chat application, replicated data management refers to the process of maintaining consistent and synchronized copies of data across multiple replicas of a MySQL database.

By using replicated databases, the chat application can continue to function even if one replica goes down. If one replica fails, the chat application can still read and write to the other replicas, ensuring that clients can continue to communicate with each other. Also, the chat application can scale horizontally by adding more replicas to handle increasing traffic. This means that as the number of clients increases, the chat application can handle the increased load by adding more database replicas.

Replicating data ensures that all clients see the same messages and updates, regardless of which replica they are connected to. This is important for a chat application, as clients need to see the same conversation in real-time to be able to communicate effectively.

In this project, there are two databases being used. Each database has a "clients" table and a "messages" table. Both the insertion of a client and message into the databases are done using a two phase commit protocol which is explained below.

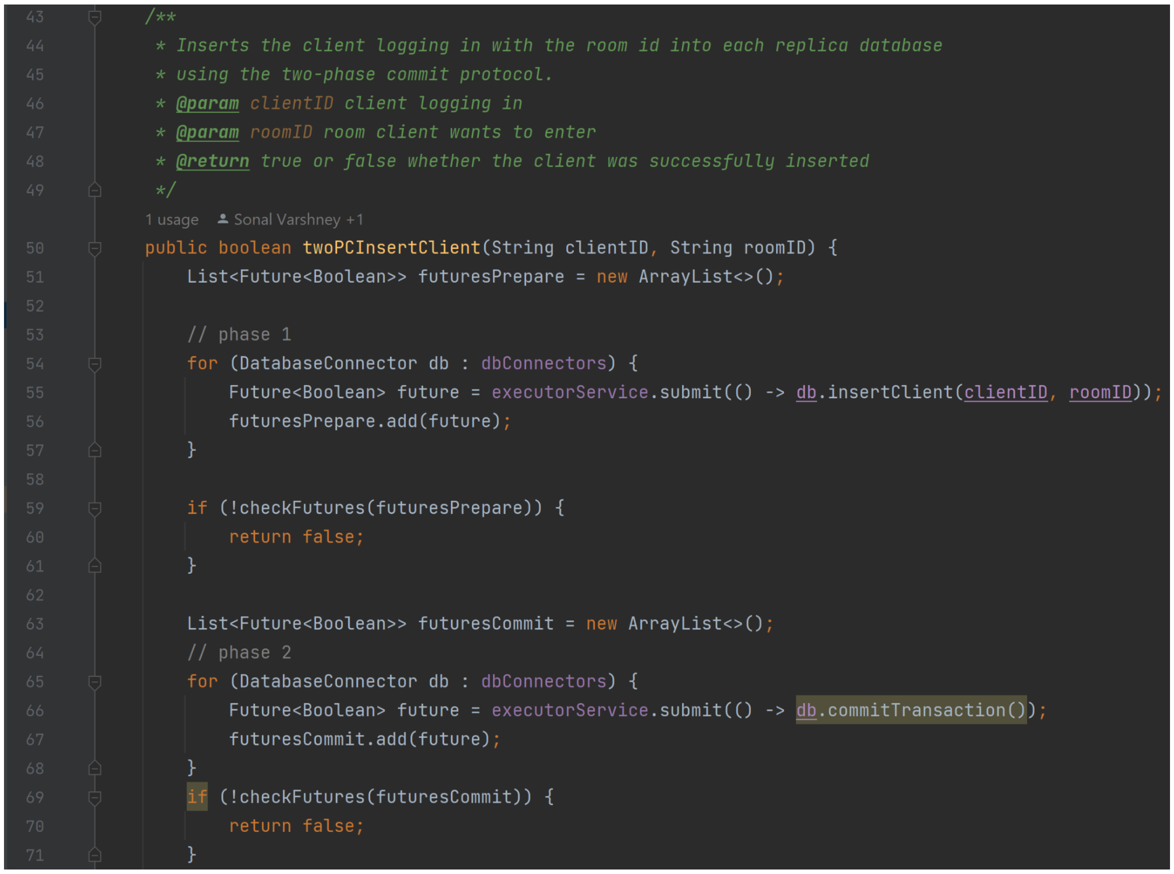
2. Two phase commit protocol

To make sure that the replicas are consistent, a two-phase commit protocol is used, which is a distributed algorithm that ensures all replicas of the database agree to commit or rollback a transaction. The two-phase commit protocol has two phases: the prepare phase and the commit phase.

In the prepare phase, the coordinator node sends a "prepare" message to all replicas, asking them to vote on whether or not to commit the transaction. Each replica responds with a vote to the coordinator node. If all replicas vote to commit the transaction, the coordinator sends a commit message to all replicas, instructing them to commit the transaction. If any replica votes to abort the transaction, the coordinator sends an abort message to all replicas, instructing them to rollback the transaction.

By using this protocol, the two replicas of the MySQL database can ensure that they always contain consistent data. For example, when a client sends a message to the chat application, the message is stored in both replicas of the database using the two-phase commit protocol. This ensures that all clients see the same messages, regardless of which replica they are connected to.

This protocol is seen as an example in the below code snippet from DatabaseCoordaintor.java. The protocol is being used to insert a client into the database replicas. Line 54-57 goes through each database connection and submits the method to "insertClient" to the executor service. The methods then execute parallely using Futures. Line 59 checks if any of the replicas is not ready to commit. In this case the method returns false and the insertion fails. Similar process is done for the commit phase from line 64.



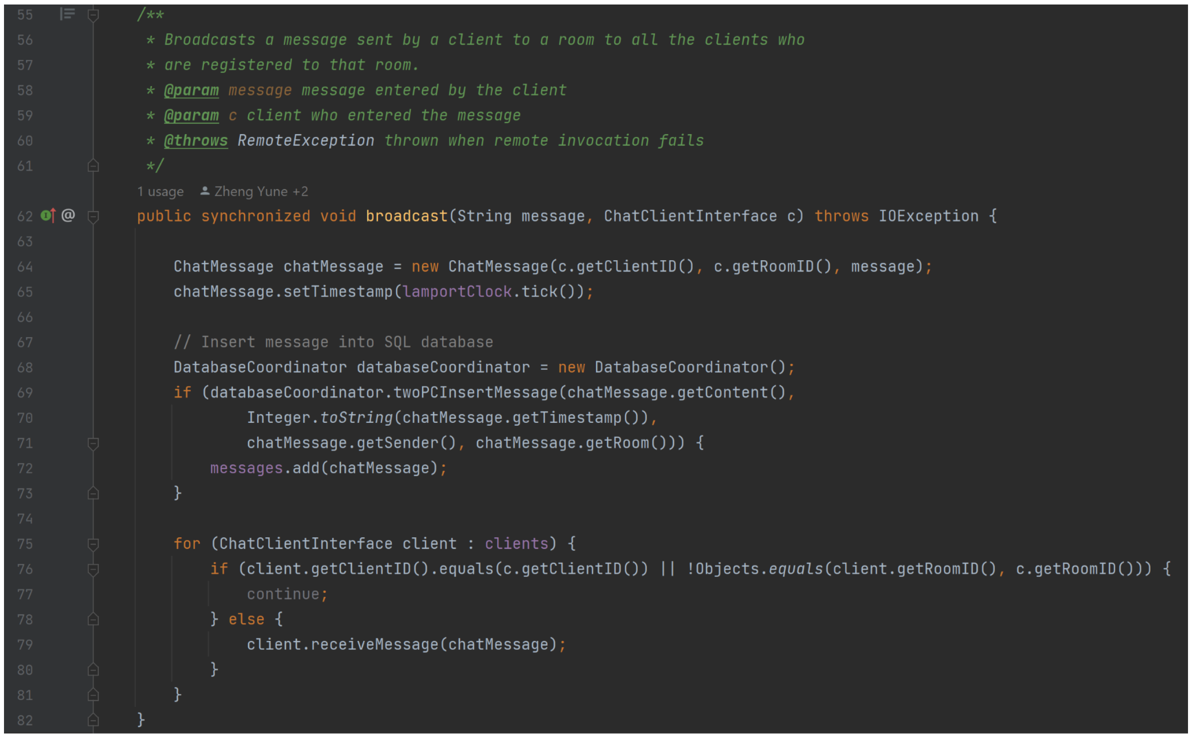
```
43  /**
44   * Inserts the client logging in with the room id into each replica database
45   * using the two-phase commit protocol.
46   * @param clientID client logging in
47   * @param roomID room client wants to enter
48   * @return true or false whether the client was successfully inserted
49   */
    1 usage   Sonal Varshney +1
50  public boolean twoPCInsertClient(String clientID, String roomID) {
51      List<Future<Boolean>> futuresPrepare = new ArrayList<>();
52
53      // phase 1
54      for (DatabaseConnector db : dbConnectors) {
55          Future<Boolean> future = executorService.submit(() -> db.insertClient(clientID, roomID));
56          futuresPrepare.add(future);
57      }
58
59      if (!checkFutures(futuresPrepare)) {
60          return false;
61      }
62
63      List<Future<Boolean>> futuresCommit = new ArrayList<>();
64      // phase 2
65      for (DatabaseConnector db : dbConnectors) {
66          Future<Boolean> future = executorService.submit(() -> db.commitTransaction());
67          futuresCommit.add(future);
68      }
69      if (!checkFutures(futuresCommit)) {
70          return false;
71      }
```

3. Group communication

Group communication in distributed systems refers to the process of exchanging messages or data among multiple participants, known as a group or multicast group, where each member can send or receive information from others in the group. Group communication is an essential component in many distributed systems applications, such as multi-player games, collaborative editing tools, and chat applications.

Broadcasting is a technique used in multi-client chat applications to distribute messages to all clients in the chat room simultaneously. In broadcasting, a client sends a message to the server, and the server then forwards the message to all other clients in the chat room. This allows for real-time communication between multiple participants in a chat room without the need for each participant to individually send messages to all other participants.

In this application, broadcasting has been implemented such that all the clients of a specific room receive messages in real time. This is done using a publish-subscribe model where a client subscribes to a particular room and the server maintains a list of these clients. When a client sends a message to the server, the server publishes the message to all clients subscribed to the chat room. This allows each client to receive the message without the sender needing to address it to each individual recipient. In the ChatServerV.java file, the list of client objects is maintained as “clients”. Each client object consists of the client id and the room they are subscribed to. So when broadcasting a new message in the server, line 76 shows the filter to broadcast only to the room the message was sent to.



```
55  /**
56   * Broadcasts a message sent by a client to a room to all the clients who
57   * are registered to that room.
58   * @param message message entered by the client
59   * @param c client who entered the message
60   * @throws RemoteException thrown when remote invocation fails
61   */
    1 usage  Zheng Yune +2
62  public synchronized void broadcast(String message, ChatClientInterface c) throws IOException {
63
64      ChatMessage chatMessage = new ChatMessage(c.getClientID(), c.getRoomID(), message);
65      chatMessage.setTimestamp(lamportClock.tick());
66
67      // Insert message into SQL database
68      DatabaseCoordinator databaseCoordinator = new DatabaseCoordinator();
69      if (databaseCoordinator.twoPCInsertMessage(chatMessage.getContent(),
70              Integer.toString(chatMessage.getTimestamp()),
71              chatMessage.getSender(), chatMessage.getRoom())) {
72          messages.add(chatMessage);
73      }
74
75      for (ChatClientInterface client : clients) {
76          if (client.getClientID().equals(c.getClientID()) || !Objects.equals(client.getRoomID(), c.getRoomID())) {
77              continue;
78          } else {
79              client.receiveMessage(chatMessage);
80          }
81      }
82  }
```

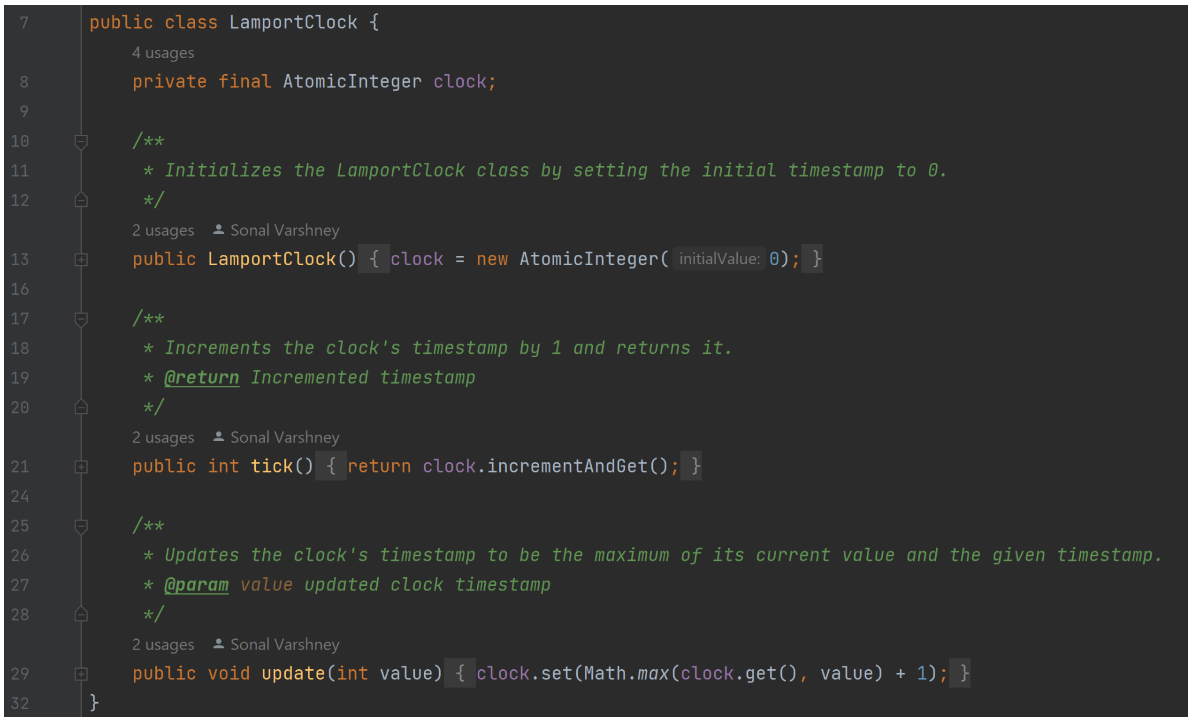
4. Lamport clocks

In a chat application, multiple clients may be sending messages to each other at the same time, and it's important to ensure that the messages are delivered in the order they were sent.

Lamport clocks work by assigning a unique timestamp to each event in the system, including message sends and receives. These timestamps are not based on a global clock, but instead are derived from the local clocks of the individual clients. Each client increments its own timestamp for each event it generates and includes this timestamp with the message. When a message is received, the receiving client compares its own timestamp to the timestamp in the message to determine the order of the events.

Using Lamport clocks in a chat application ensures that messages are delivered in the order they were sent, even if the clients' local clocks are not synchronized with each other. This is important because without a way to order events, messages could be delivered out of order, leading to confusion and miscommunication.

The below class shows the methods used to use the lamport clocks. The tick and update methods are used to increment timestamps and compare to determine the ordering.



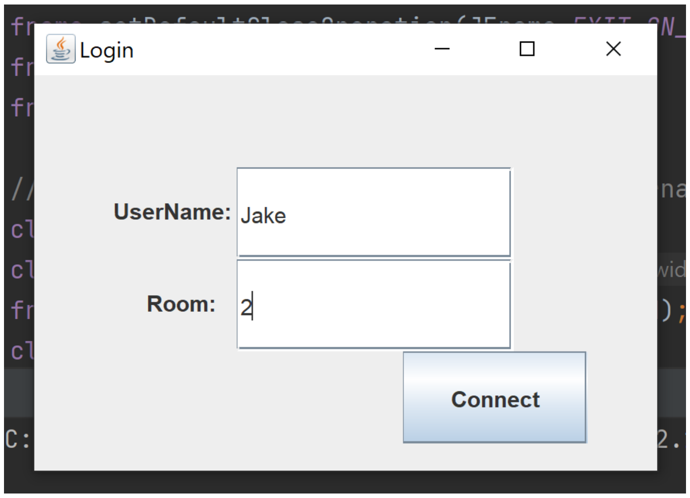
```
public class LamportClock {
    4 usages
    private final AtomicInteger clock;

    /**
     * Initializes the LamportClock class by setting the initial timestamp to 0.
     */
    2 usages  Sonal Varshney
    public LamportClock() { clock = new AtomicInteger( initialValue: 0); }

    /**
     * Increments the clock's timestamp by 1 and returns it.
     * @return Incremented timestamp
     */
    2 usages  Sonal Varshney
    public int tick() { return clock.incrementAndGet(); }

    /**
     * Updates the clock's timestamp to be the maximum of its current value and the given timestamp.
     * @param value updated clock timestamp
     */
    2 usages  Sonal Varshney
    public void update(int value) { clock.set(Math.max(clock.get(), value) + 1); }
}
```

In the broadcast function of the Server class, the tick function is used to define the timestamp of a message.

```
ChatMessage chatMessage = new ChatMessage(c.getClientID(), c.getRoomID(), message);
chatMessage.setTimestamp(lamportClock.tick());
```

Example runs:

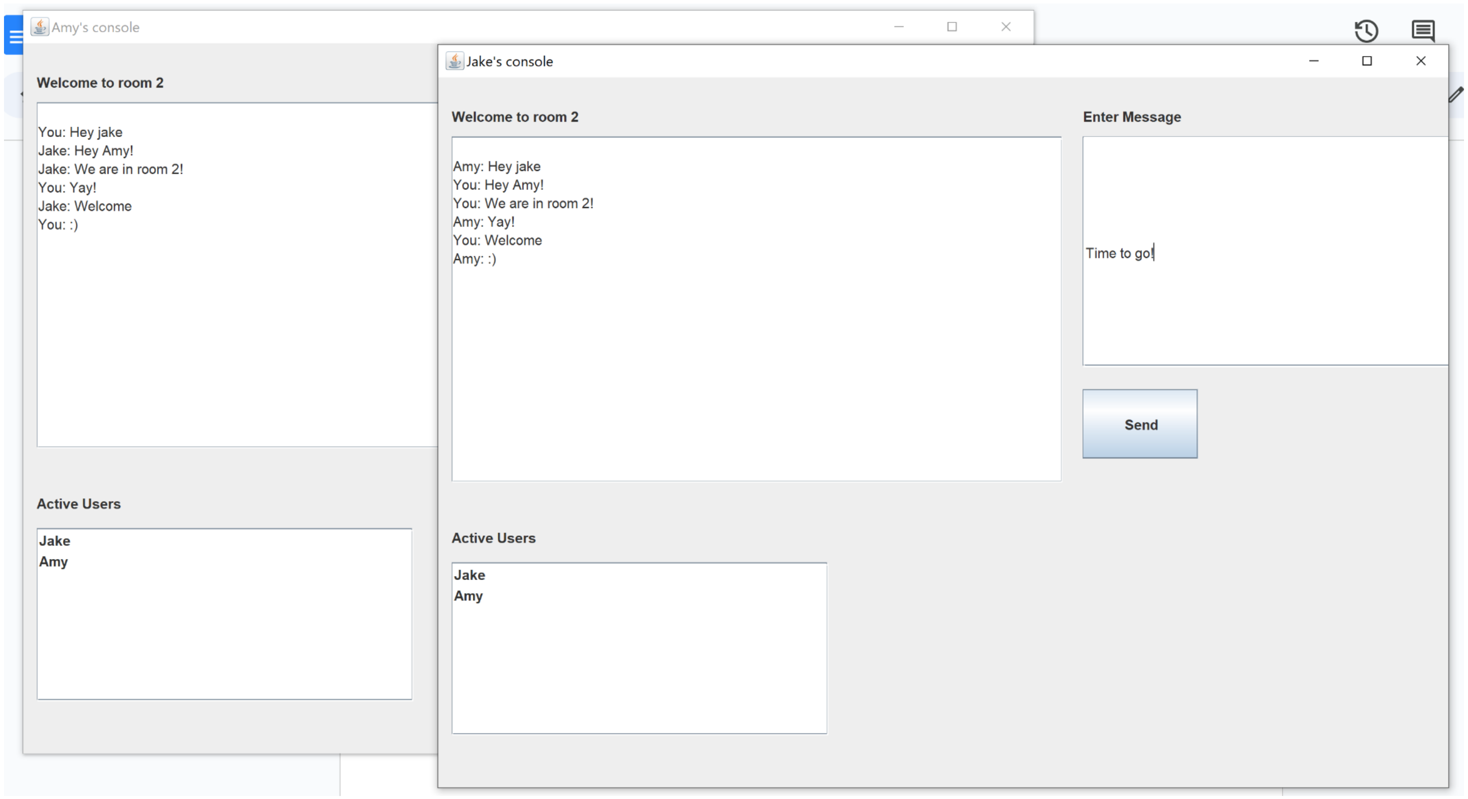
1. Connect to the server
2. Starting the client launches the login window. Enter username and room id:



UserName: Jake

Room: 2

3. After clicking on Connect, the client's chat console opens. Here there is a chat message board which displays the chat history of the room, a chat text area where the client can enter a message to send to the room, and a list of active users from that room.
4. Another client can login to the same room and the broadcasting is demonstrated below. We can see two client consoles where each client is connected to room 2. Each message sent by them has been displayed in the chat message board. The active users list shows



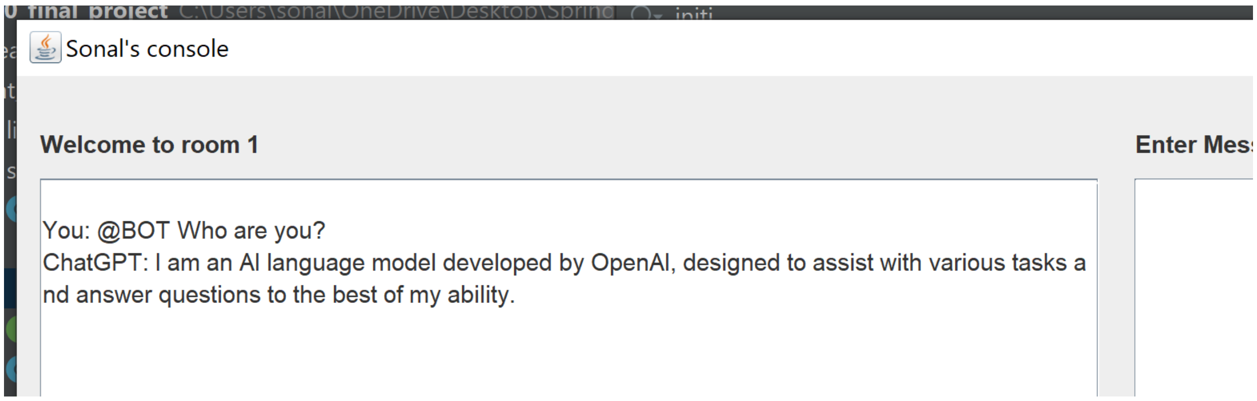
```
You: Hey jake
Jake: Hey Amy!
Jake: We are in room 2!
You: Yay!
Jake: Welcome
You: :)
```

```
Amy: Hey jake
You: Hey Amy!
You: We are in room 2!
Amy: Yay!
You: Welcome
Amy: :)
```

```
Time to go!
```

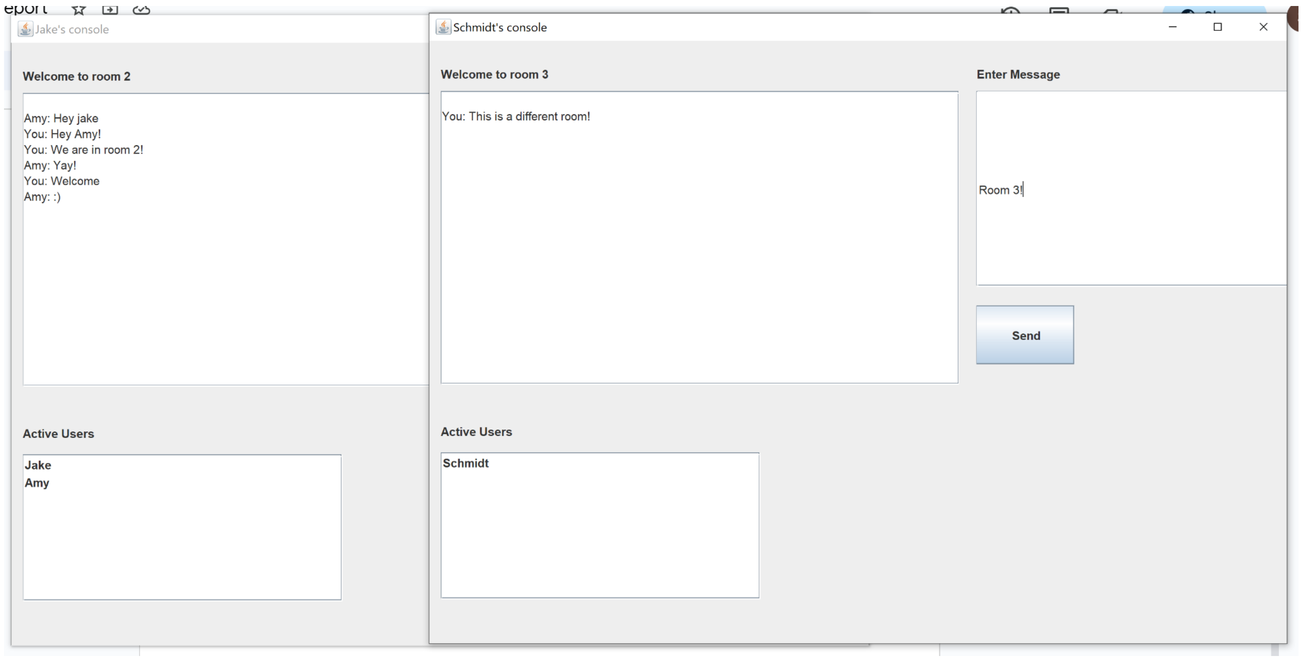
two of the clients connected.

5. An example of GPT:



```
You: @BOT Who are you?
ChatGPT: I am an AI language model developed by OpenAI, designed to assist with various tasks a
nd answer questions to the best of my ability.
```

6. A user can disconnect from the application by pressing the X mark on the window. This will shut down the client, but their chat messages sent will be stored. So when a new client enters room 2, they will be able to view the exited clients messages.


Jake's console

Welcome to room 2

Amy: Hey jake
You: Hey Amy!
You: We are in room 2!
Amy: Yay!
You: Welcome
Amy: :)

Active Users

Jake
Amy

Schmidt's console

Welcome to room 3

You: This is a different room!

Enter Message

Room 3!

Active Users

Schmidt

7. If another room is created, broadcasting makes sure that the messages are received only in the rooms they were sent to as shown above. A new client joined room 3 and does not have the messages used in room 2.

Future Scope

- Add authentication when the user wants to log in to the application.
- Extend text based messages to use multimedia.
- Improve GUI to make it appealing and add features for user convenience.
- Remove central coordinator dependency.