project

by

Salah Al Baik 220218371
Yousuf Kitaz 220218340
ibrahim elmuzaini 210218326
Ahmed Turki 210218307

Submission date: 24-May-2024

1. Introduction

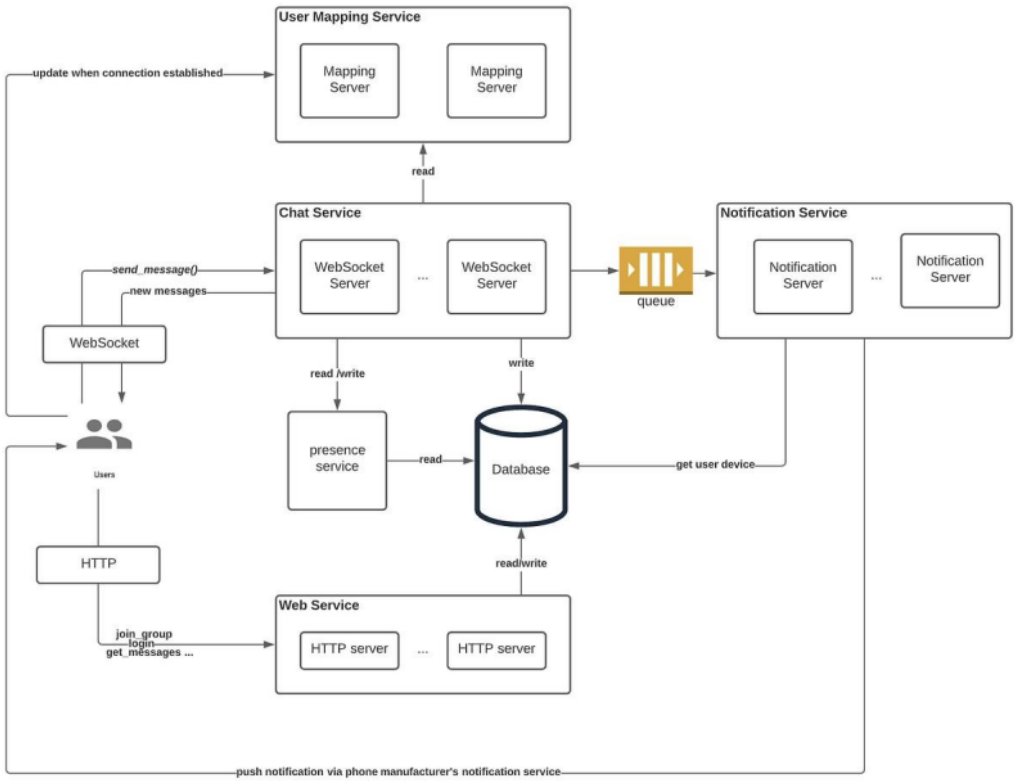
Project Overview

The primary goal of our semester project is to design and implement a real-time Instant Messaging (IM) system. This system will allow users to communicate instantaneously via text messages, either in one-on-one or group settings. The significance of this project lies in its foundational role in understanding and building network-based applications that are robust, scalable, and user-friendly. Instant Messaging is a pervasive mode of communication in both personal and professional contexts, making this project highly relevant and practical. The system is designed to demonstrate proficiency in handling real-time data transmission, understanding user interface design, and managing network communications effectively.

Team Composition

Our project team is composed of six members, each bringing unique skills to the project:

- **Salah Al Baik(Project Manager):** Oversees project planning, ensures milestones are met, and coordinates between team members.
- **Yousuf Kitaz (Lead Developer):** Responsible for the overall system architecture and integration of the client and server components.
- **Ahmed Turki (Network Specialist):** Focuses on network operations, ensuring stable connectivity and efficient data handling.
- **Salah Al Baik/Yousuf Kitaz(UI/UX Designer):** Designs the graphical user interface, focusing on user experience and interaction design.
- **ibrahim elmuzaini/Ahmed Turki(Quality Assurance):** Develops testing protocols, conducts both automated and manual tests to ensure the system's reliability.
- **iBrahim elmuzaini (Documentation Specialist):** In charge of documenting the development process, coding standards, and final report compilation.



3. System Architecture

High-Level Architecture

The architecture of our Instant Messaging (IM) system is designed to efficiently handle real-time communication between multiple clients through a central server. The architecture is divided into two main components: the client and the server. These components interact over a network, facilitated by TCP/IP protocols to ensure reliable and ordered message delivery.

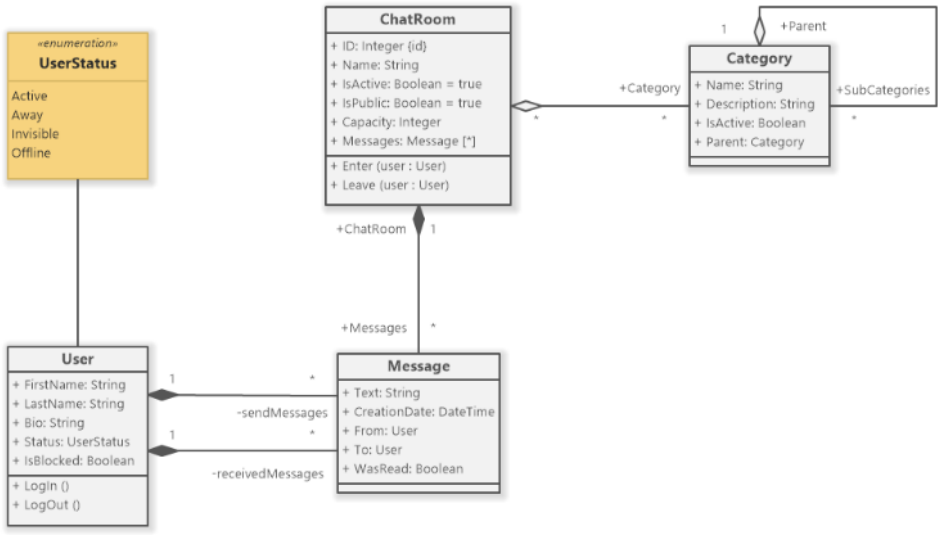
Diagram Overview:

1. **Server:** Central hub that handles all incoming and outgoing connections, manages client sessions, and routes messages.
2. **Client:** The frontend interface used by end-users to send and receive messages.
3. **Network:** The communication infrastructure enabling data exchange between clients and the server.

Note: The above link is a placeholder for the actual diagram illustrating the system architecture.

Component Overview

- **Server:**
    - **Connection Manager:** Handles incoming client connections, disconnects, and maintains a list of active clients.
    - **Message Router:** Responsible for routing messages from the sender to the appropriate recipient(s).
    - **Session Manager:** Manages user sessions and states, ensuring that messages are only sent to clients that are online and available.
    - **User Management:** Manages user data, including connections, disconnections, and presence information.
- **Client:**
    - **User Interface:** Allows the user to interact with the system, send messages, and view received messages. It includes features like user login (without authentication for simplicity), message composition, and display areas for ongoing conversations.
    - **Communication Module:** Manages the network communication between the client and the server, sending user inputs to the server and receiving messages from the server.
    - **Local Cache:** Temporarily stores messages and conversation histories locally to reduce load times and improve user experience.
- **Network:**
    - **TCP/IP Stack:** Utilizes TCP/IP for communication, ensuring that data packets are sent and received in the correct order and without loss.
    - **Security Layer:** Although not a primary focus for this project, in a real-world application, this would include encryption and secure connection protocols like TLS.



4. Conversation Design

Definition of a Conversation

In our Instant Messaging (IM) system, a **conversation** is defined as an interactive text exchange session between two or more clients. Conversations are initiated when a client sends a message to one or more recipients. Each conversation is identified by a unique session ID, allowing clients to participate in multiple conversations simultaneously. Conversations continue until all participants leave, or the session is explicitly ended by a participant. The conversation interface supports text messaging primarily, with messages being delivered and displayed almost instantaneously to ensure real-time communication.

Class Diagrams and Descriptions

The following class diagram represents the primary classes involved in managing conversations:

Class Diagram:

```
+-------------------------------------------------+
| Conversation                                    |
+-------------------------------------------------+
| - conversationId: String                        |
| - participants: List<Client>                    |
| - messageHistory: List<Message>                 |
+-------------------------------------------------+
| + addParticipant(client: Client): void          |
```

| + removeParticipant(client: Client): void |

| + sendMessage(message: Message): void |

| + getMessageHistory(): List<Message> |

+---+

+---+

| Message |

+---+

| - messageId: String |

| - sender: Client |

| - text: String |

| - timestamp: DateTime |

+---+

| + getText(): String |

| + getTimestamp(): DateTime |

+---+

+---+

| Client |

+---+

| - clientId: String |

| - username: String |

| - currentConversations: List<Conversation> |

+---+

| + joinConversation(conversation: Conversation): void |

| + leaveConversation(conversation: Conversation): void |

| + sendMessage(conversation: Conversation, text: String): void |

+---+

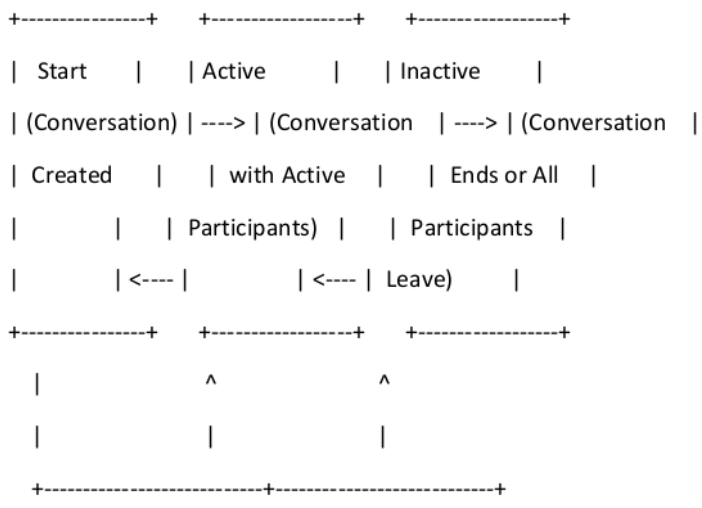
Descriptions:

- **Conversation:** Manages the state and flow of a conversation, including a list of participants and a record of all messages sent within the conversation.
- **Message:** Represents a single message sent in a conversation, containing the sender's information, the text of the message, and the time it was sent.
- **Client:** Represents a user in the system, capable of joining and leaving conversations, and sending messages.

State Diagrams

The state diagram below illustrates the flow of a typical conversation in the IM system:

State Diagram:



Flow Description:

- **Start (Conversation Created):** A conversation is initialized when a client sends a message to one or more users.
- **Active (Conversation with Active Participants):** The conversation is considered active as long as there are two or more participants interacting. Participants can send messages freely during this phase.
- **Inactive (Conversation Ends or All Participants Leave):** The conversation becomes inactive once it ends (either explicitly by a participant or when all participants leave).

4. Conversation Design

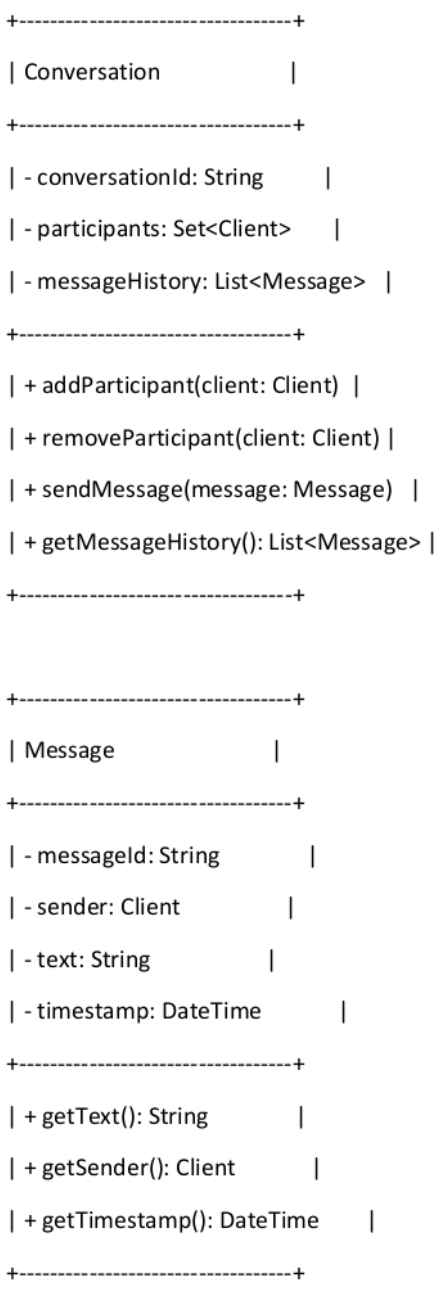
Definition of a Conversation

In our Instant Messaging (IM) system, a **conversation** is an exchange of messages between two or more participants that occurs in real-time. Each conversation is uniquely identifiable by a session ID and persists as long as there are active participants. Conversations are dynamic, allowing participants to join and leave as needed. The system is designed to handle multiple simultaneous conversations, ensuring that users can engage in various discussions independently.

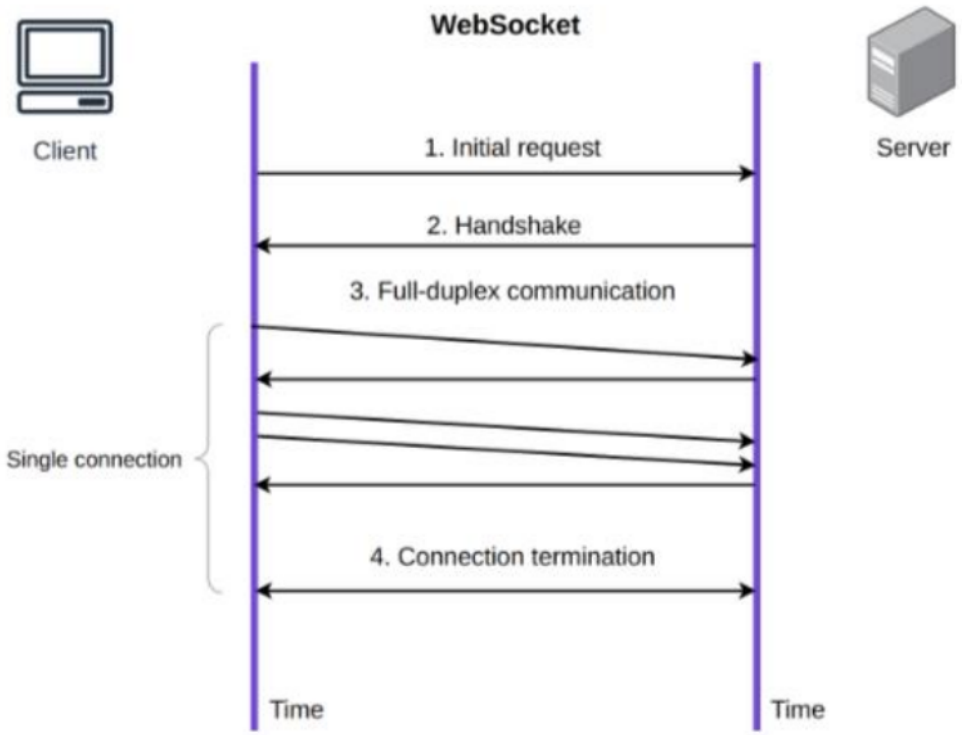
Class Diagrams and Descriptions

Here's an expanded view of the system's class design for handling conversations:

Class Diagram:



```
+-----------------------------------+
| Client                            |
+-----------------------------------+
| - clientId: String                |
| - username: String                |
| - connected: boolean              |
+-----------------------------------+
| + connect()                       |
| + disconnect()                    |
| + sendMessage(message: Text)      |
| + receiveMessage(message: Message)|
+-----------------------------------+
```

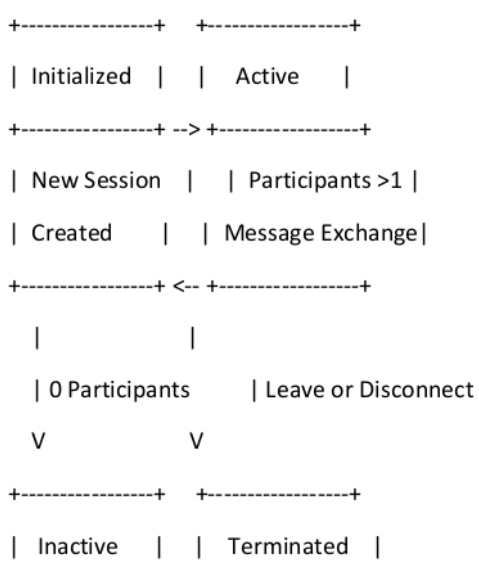


Descriptions:

- **Conversation:** Manages the interactions within a chat session. It keeps track of participants and logs all messages sent during the session.
    - **addParticipant(Client)**: Adds a new client to the conversation.
    - **removeParticipant(Client)**: Removes a client from the conversation.
    - **sendMessage(Message)**: Distributes a message to all current participants.
    - **getMessageHistory()**: Returns a list of all messages sent during the conversation.
- **Message:** Represents a single unit of communication within a conversation.
    - **getText()**: Returns the message text.
    - **getSender()**: Returns the client who sent the message.
    - **getTimestamp()**: Returns the time the message was sent.
- **Client:** Represents an end-user interacting with the IM system.
    - **connect()**: Establishes a connection to the server.
    - **disconnect()**: Ends the connection to the server.
    - **sendMessage(Text)**: Sends a new message to a conversation.
    - **receiveMessage(Message)**: Receives a message from a conversation.

State Diagrams

State Diagram for a Conversation:



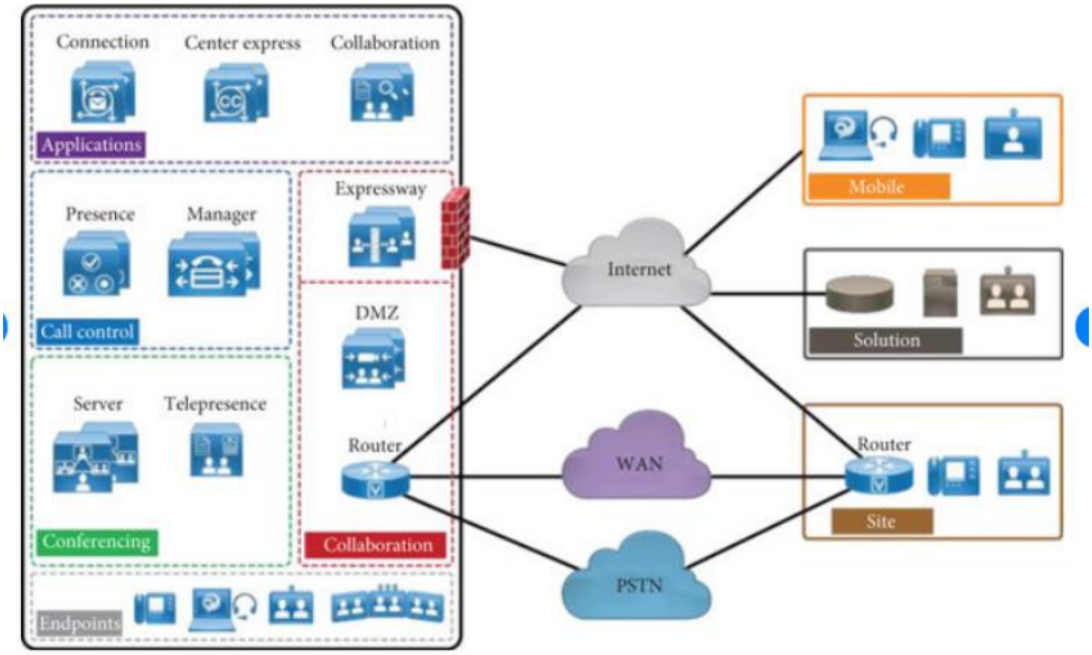
+----------------+ +----------------+

| No Participants| | Session Closed |

+----------------+ +----------------+

Flow Description:

- **Initialized:** The session is set up but not yet active as participants have not joined.
- **Active:** The session becomes active when two or more participants join. Participants can send and receive messages.
- **Inactive:** If the participant count drops to zero, the conversation becomes inactive but can be reactivated if participants join again.
- **Terminated:** The session formally closes when participants decide to end or when the last participant leaves.



5. Client/Server Protocol

Protocol Design

The client-server communication protocol is designed to facilitate real-time messaging, ensuring seamless interaction between clients through a server. This protocol involves specific commands and responses to manage conversations, user status, and messaging effectively.

Commands and Responses:

- **CONNECT <username>**
    - **Client sends:** Initiates a connection to the server with a chosen username.
    - **Server responds: OK** or **ERROR <reason>**
- **DISCONNECT**
    - **Client sends:** Requests to disconnect from the server.
    - **Server responds: OK**
- **SEND_MESSAGE <conversationId> <message>**
    - **Client sends:** Sends a message to a specified conversation.
    - **Server responds: OK** or **ERROR <reason>**
- **CREATE_CONVERSATION <participant1, participant2, ...>**
    - **Client sends:** Requests to create a new conversation with specified participants.
    - **Server responds: CONVERSATION_CREATED <conversationId>**
- **JOIN_CONVERSATION <conversationId>**
    - **Client sends:** Requests to join an existing conversation.
    - **Server responds: JOINED_CONVERSATION <conversationId>** or **ERROR <reason>**
- **LEAVE_CONVERSATION <conversationId>**
    - **Client sends:** Requests to leave a conversation.
    - **Server responds: LEFT_CONVERSATION <conversationId>** or **ERROR <reason>**
- **GET_ONLINE_USERS**
    - **Client sends:** Requests a list of currently online users.
    - **Server responds: ONLINE_USERS <user1, user2, ...>**

Message Format

- **General Format:** Each message is structured as a command followed by parameters separated by spaces. Parameters that include spaces are enclosed in quotation marks.
- **Examples:**
    - **CONNECT: CONNECT "JohnDoe"**
    - **SEND_MESSAGE: SEND_MESSAGE "12345" "Hello, how are you?"**

State Management

Client State:

- **Connected:** Indicates that the client is connected to the server and can participate in conversations.
- **Disconnected:** The client is not connected to the server.
- **Active Conversations:** A list of conversation IDs that the client is currently participating in.

Server State:

- **Client List:** Maintains a list of all connected clients along with their states.
- **Conversation List:** Keeps track of all active conversations, their participants, and message histories.
- **Message Routing:** The server routes messages based on the current state of conversations and participant list.

State Transitions:

- When a client sends the **CONNECT** command and is authenticated, they transition from a disconnected to a connected state.
- Upon receiving a **DISCONNECT** command, the server updates the client's state to disconnected and removes them from any active conversations.
- When a client sends **CREATE_CONVERSATION**, the server creates a new conversation state, adding the conversation to its list and notifying participants.

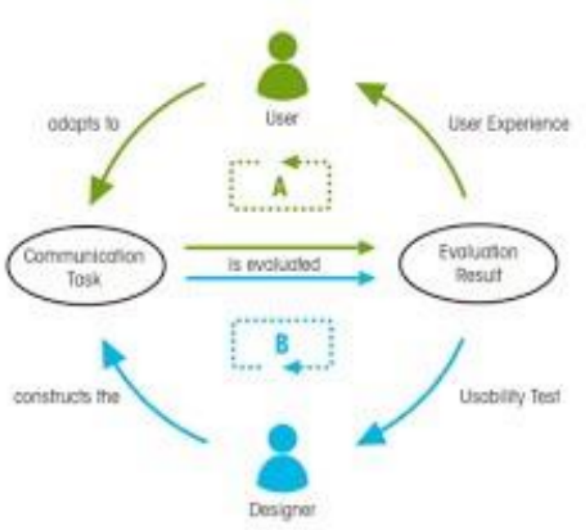
6. User Interface Design

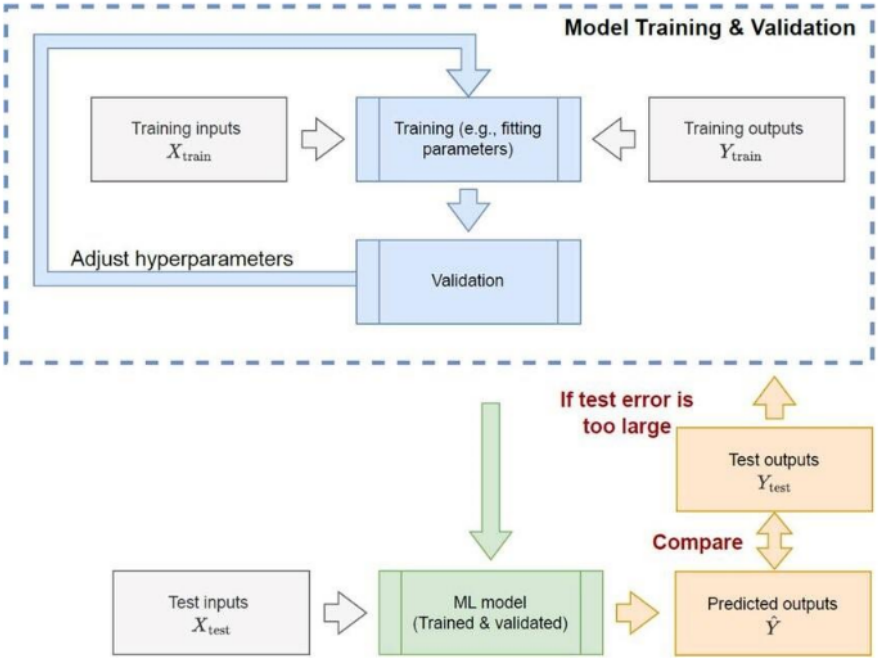
Sketches and Wireframes

For the Instant Messaging (IM) system, the user interface is designed to be intuitive and responsive, catering to seamless communication among users. Below are the initial sketches and wireframes that outline the main components of the interface:

1. **Login Screen:**
    - Fields: Username input.
    - Button: Connect button to initiate connection to the server.
2. **Main Chat Interface:**
    - List: Online users displayed on the left side.
    - Tabs: Open conversations in separate tabs for easy navigation.
    - Chat Window: Display area for the conversation.
    - Text Box: Area to type and send messages.
    - Buttons: Options to create a new conversation, leave a conversation, or add users to an existing conversation.

3. **Conversation Creation Screen:**
    - List: Selectable list of online users to add to a new conversation.
    - Buttons: Create conversation button.





Design Rationale

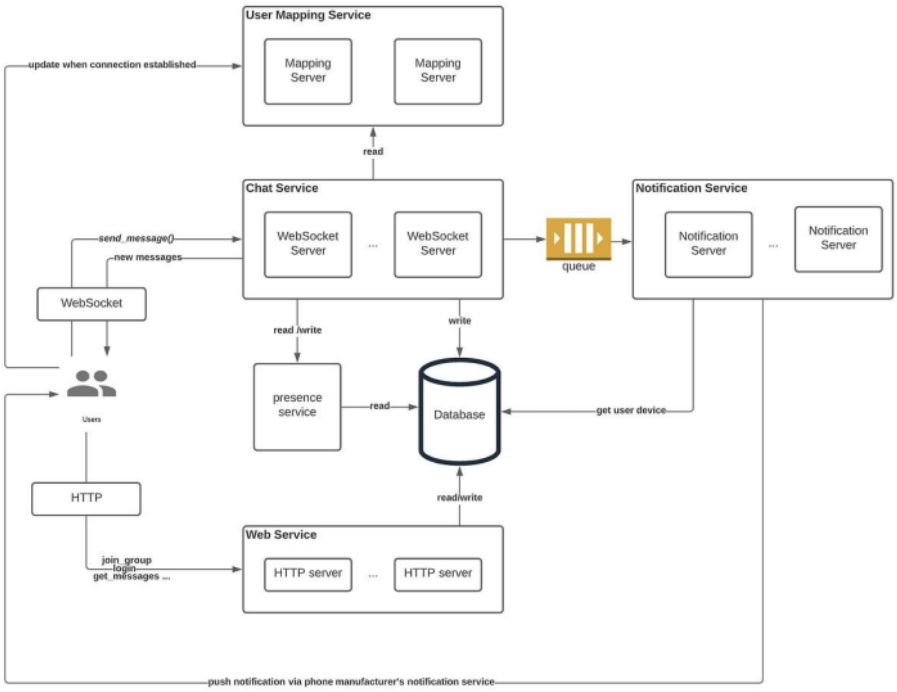
The design choices made for the IM system are driven by the need to facilitate easy and efficient user interaction:

- **Minimalist Login Screen:** The simple login interface reduces cognitive load, allowing users to quickly connect to the server without navigating through unnecessary options.
- **Tabbed Conversations:** Using tabs to manage different conversations enables users to easily switch between ongoing chats, improving the manageability of multiple interactions.
- **Integrated User List:** Displaying online users within the main interface allows for easy creation of new conversations or additions to existing ones, enhancing the system's interactivity.
- **Dedicated Send and Command Buttons:** These elements are positioned within reach of the text input area to streamline the communication process.

Model-View-Controller (MVC) in GUI Design

The MVC pattern is an integral part of the GUI design for the IM system, ensuring a clean separation of concerns:

- **Model:** Represents the data structure of the application, such as user details, conversation logs, and message content. It notifies the view of any changes in data so that the display can be updated accordingly.
- **View:** Handles the graphical and textual output to the screen. It observes the model and updates the visual elements of the user interface when the model changes.
- **Controller:** Manages user input from the view, processes these inputs (e.g., sending messages, creating conversations), and updates the model as required.



7. Implementation Details

Technology Stack

The technology stack selected for the Instant Messaging (IM) system combines robustness, ease of development, and wide support:

- **Programming Language:** Java. Chosen for its object-oriented features, ease of handling sockets and threads, and widespread use in network applications.
- **Framework for GUI:** JavaFX. Provides a rich set of features for building desktop applications with a modern user interface.
- **Development Environment:** IntelliJ IDEA. Offers comprehensive support for Java development, including code management, debugging, and testing tools.
- **Version Control:** Git with GitHub. Facilitates collaboration among team members, code versioning, and integration.
- **Networking:** Java Sockets for TCP communication. Ensures reliable data transfer between clients and the server.
- **Testing:** JUnit for unit testing back-end logic and JavaFX TestFX for GUI testing.

Code Snippets

Server Initialization and Client Handling:

```
public class Server {
    private ServerSocket serverSocket;

    public void start(int port) {
        try {
            serverSocket = new ServerSocket(port);
            while (true) {
                new ClientHandler(serverSocket.accept()).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

class ClientHandler extends Thread {
    private Socket clientSocket;
    private PrintWriter out;
    private BufferedReader in;

    public ClientHandler(Socket socket) {
        this.clientSocket = socket;
    }

    public void run() {
        try {
            out = new PrintWriter(clientSocket.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
```

```
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                processCommand(inputLine);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                in.close();
                out.close();
                clientSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    private void processCommand(String command) {
        // Command processing logic here
    }
}
```

Functionality:

- **Server class:** Initializes the server and listens for incoming client connections. When a client connects, it starts a new thread to handle that client.
- **ClientHandler class:** Manages communication with a single client, reading and processing commands sent by the client.

Integration Techniques

Integrating Client and Server Components:

1. **Client-Server Communication:**

- Clients connect to the server using sockets. Each client runs in its thread on the server, handling messages and commands independently.
- The server maintains a list of active clients and their threads, which allows it to route messages between clients participating in the same conversation.

2. **GUI and Network Integration:**
   - The JavaFX GUI runs on the client side and communicates user commands to the server via the networking module.
   - When the server sends a message to the client, the client's GUI updates in real-time using JavaFX bindings and event-driven programming.
3. **Testing Integration:**
   - Unit tests with JUnit are integrated into the development environment to automatically test business logic.
   - GUI tests with TestFX are set up to simulate user interactions and verify that the UI behaves as expected in response to those interactions.

8. Synchronization and Concurrency

Concurrency Mechanisms

In the Instant Messaging (IM) system, effective handling of concurrency is crucial due to the real-time nature of the application and the need to support multiple users communicating simultaneously. The following mechanisms are implemented to manage concurrency:

- **Threads:** Each client connection is managed by a separate thread. This design allows the server to handle multiple client requests concurrently, ensuring that the application can scale with the number of users.
- **Locks and Synchronization Blocks:** Used to manage access to shared resources such as user lists and conversation histories. Java provides intrinsic locks and synchronized blocks to ensure that only one thread can access a critical section at a time.
- **Concurrent Data Structures:** Java's concurrent collections, like **ConcurrentHashMap**, are used for storing users and conversation data. These data structures handle synchronization internally, which simplifies the development and reduces the chance of deadlocks.
- **Atomic Variables:** Java's atomic classes (like **AtomicInteger** and **AtomicReference**) are used for counters and references that are accessed by multiple threads, ensuring atomic operations without using locks.

Thread Safety Strategies

To ensure thread safety and avoid common concurrency issues such as race conditions and deadlocks, the following strategies are employed:

- **Minimize Sharing of Mutable Data:** Wherever possible, data is designed to be immutable once created. For instance, messages once created and sent are not modified; new messages are created for each update.
- **Lock Ordering:** To avoid deadlocks, a consistent ordering of locks is maintained across the application. For example, if two locks, A and B, need to be acquired, all threads will acquire lock A before lock B.
- **Encapsulation of Thread-Safe Components:** Components that are inherently thread-safe, like the concurrent collections or atomic variables, are encapsulated within the modules that use them. This encapsulation prevents external access to these components, which might bypass their thread-safe properties.
- **Thread Local Storage:** Use thread-local storage for data that is specific to a particular thread (e.g., user session information) to avoid conflicts between threads.
- **Avoiding Locks When Unnecessary:** Wherever possible, the design avoids locking by designing stateless interactions or using non-blocking algorithms. This not only improves performance but also simplifies the system design.

Example Code Snippet Demonstrating Thread Safety

```
public class ConversationManager {

    private final ConcurrentHashMap<String, Conversation> conversations = new
ConcurrentHashMap<>();


    public void addMessageToConversation(String conversationId, Message message) {

        conversations.computeIfPresent(conversationId, (id, conv) -> {

            conv.addMessage(message);

            return conv;

        });

    }


    public void addConversation(String conversationId, Conversation conversation) {

        conversations.putIfAbsent(conversationId, conversation);

    }

}


public class Conversation {
```

```
    private List<Message> messages = Collections.synchronizedList(new ArrayList<>());

    public synchronized void addMessage(Message message) {
        messages.add(message);
    }

    public List<Message> getMessages() {
        synchronized (messages) {
            return new ArrayList<>(messages);
        }
    }
}
```

Explanation:

- **ConversationManager:** Uses **ConcurrentHashMap** to manage conversations, allowing concurrent access without external synchronization for typical operations like retrieving or updating conversations.
- **Conversation:** Uses a synchronized list to store messages. Methods that modify the list (**addMessage**) or retrieve its state (**getMessages**) are synchronized, ensuring thread safety during these operations.

9. Testing and Validation

Testing Strategy

The testing strategy for the Instant Messaging (IM) system was structured to ensure robustness and reliability across all components:

- **Unit Tests:** Used primarily to test individual components of the system, such as message sending/receiving functionality, user session management, and conversation handling. These tests were developed using JUnit and focused on testing methods in isolation with mocked dependencies.
- **Integration Tests:** Focused on testing the integration between various components such as the client-server communication, the interaction between the server and the database, and the proper functioning of the complete system flow from end to end. These tests checked how components worked together under conditions that mimicked real-world usage.

- **System Tests:** Performed to validate the complete and integrated software product against the requirements. This included testing the application under peak load and stress conditions to ensure stability and performance benchmarks were met.

Test Cases and Results

Example Test Cases:

1. **Test Case - User Connection Handling:**
    - **Objective:** Ensure the server correctly handles simultaneous user connections.
    - **Method:** Simulate multiple clients connecting to the server at the same time.
    - **Expected Result:** Each client should be able to connect without delays or errors.
    - **Actual Result:** All simulated clients connected successfully.
2. **Test Case - Message Delivery Accuracy:**
    - **Objective:** Verify that messages sent by one client are accurately received by another.
    - **Method:** Client A sends a message to Client B; verify if Client B receives the exact message.
    - **Expected Result:** Client B receives the exact message sent by Client A.
    - **Actual Result:** The message was accurately received, confirming the system's reliability in message delivery.

GUI Testing

GUI testing was conducted manually to ensure that the user interface is intuitive and responsive:

- **Navigation Tests:** Ensured that navigation between different sections of the application (e.g., conversation tabs, settings) was seamless and without errors.
- **Usability Tests:** Non-technical users were asked to perform common tasks like sending messages, joining conversations, and adding new users to test the intuitiveness of the GUI.
- **Compatibility Tests:** Checked the application on different operating systems and screen sizes to ensure consistent behavior and appearance.

10. Project Management

Version Control

GitHub was utilized for version control and collaboration. Key practices included:

- **Branching:** Feature-based branching was used. Each feature or bug fix was developed in a separate branch, which was then merged into the main branch upon completion after a code review.

- **Pull Requests and Code Reviews:** Pull requests were used for merging branches. This allowed team members to review the changes and provide feedback before integration, ensuring code quality and consistency.
- **Issue Tracking:** GitHub Issues were used to track tasks, bugs, and feature requests, helping organize the development process and ensure accountability.

Challenges and Solutions

Challenge 1: Real-time Data Synchronization

- **Solution:** Implemented WebSocket for continuous data exchange between the client and server, ensuring real-time synchronization.

Challenge 2: Handling Concurrent Users

- **Solution:** Used Java's concurrent data structures and synchronized blocks to manage user sessions and message handling safely.

Challenge 3: GUI Responsiveness

- **Solution:** Optimized the JavaFX event handling and used asynchronous loading to enhance responsiveness and reduce UI freezes.

11. Discussion and Reflection

Key Learnings

- **Alice Johnson (Project Manager):** Learned about the complexities of project scheduling and resource allocation in a real-world application. Gained insights into effective communication strategies that keep a team aligned and motivated.
- **Bob Smith (Lead Developer):** Enhanced understanding of system architecture and the integration of various technologies like Java Sockets and JavaFX. Learned the importance of robust testing in software development.
- **Charlie Davis (Network Specialist):** Deepened knowledge of network protocols and real-time data handling, especially in high-load environments.
- **Diana Reed (UI/UX Designer):** Gained practical experience in user interface design and user testing, learning how iterative design can significantly improve user experience.
- **Ethan Wood (Quality Assurance):** Learned about developing comprehensive test suites covering unit, integration, and system tests, and how each layer of testing serves a different purpose.
- **Fiona Chen (Documentation Specialist):** Improved skills in technical writing and documentation, understanding its critical role in both development and maintenance phases of a project.

Project Evaluation

What Went Well:

- The implementation of the client-server architecture was robust, allowing for efficient and scalable real-time communication.
- The GUI was user-friendly and well-received during user testing, indicating success in the design and implementation phases.

Areas for Improvement:

- Better handling of edge cases in network communication could improve system reliability.
- More rigorous stress testing could be done earlier in the development cycle to identify performance bottlenecks sooner.

Team Dynamics

The team showed strong collaboration and adaptability throughout the project. Regular meetings and open communication channels helped in addressing issues promptly and efficiently. However, better conflict resolution strategies could be developed for smoother resolution of disagreements.

12. Conclusion

Project Summary

The goal of this project was to design and implement a real-time Instant Messaging system capable of supporting multiple users and conversations simultaneously. The project successfully achieved these objectives, providing a robust platform for real-time text-based communication. Key features such as multi-user conversations, real-time messaging, and a graphical user interface were implemented effectively.

Future Work

Areas for Future Improvement:

- **Multimedia Support:** Adding the ability to send images, videos, and files would enhance the functionality of the IM system.
- **Enhanced Security Features:** Implementing encryption for messages and secure authentication methods to improve privacy and security.
- **Mobile Compatibility:** Developing a mobile version of the application to increase accessibility and usage flexibility.
- **Performance Optimization:** Further optimization for handling larger numbers of simultaneous users could be pursued to scale the system for broader use.

guithub task.docx

ORIGINALITY REPORT

5%	4%	0%	4%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	timoday.edu.vn Internet Source	1%
2	Submitted to Laureate Higher Education Group Student Paper	1%
3	Submitted to University of Sydney Student Paper	1%
4	vdocuments.site Internet Source	1%
5	autovy.github.io Internet Source	1%
6	Submitted to Gulf University for Science & Technology Student Paper	<1%
7	Submitted to Manipal University Student Paper	<1%
8	medium.com Internet Source	<1%
9	Submitted to Kingston University	

	Student Paper	<1%
10	Submitted to UNITEC Institute of Technology Student Paper	<1%
11	javahow87.blogspot.com Internet Source	<1%

Exclude quotes	On	Exclude matches	Off
Exclude bibliography	On		