University of Dhaka

Department of Computer Science & Engineering

CSE-3111: Computer Networking Lab

Project Name:

Bartabahok: A Fault-Tolerant Java Chat Application with Multimedia, Group Chat and Message Reliability

Submission Date: 18 July, 2025

Members:

1. Mehedi Hasan Sakib (Roll: 13)
2. Md. Abu Kawser (Roll: 33)

Submitted to:

1. Mr. Palash Roy, Lecturer, Dept. of CSE, DU
2. Mr. Jargis Ahmed, Lecturer, Dept. of CSE, DU
3. Dr. Ismat Rahman, Associate Professor, Dept. of , DU

July 18, 2025

Abstract

Bartabahok is a robust cross-platform chat application developed in Java using JavaFX and SQLite. It ensures reliable TCP communication with automatic retransmission, supports multiple concurrent message flows and features a modern GUI with timestamps. The application allows text chat, file sharing (including images and documents), voice messages, emoji-based message reactions, and user tagging/mentioning. It supports both private and group chats, includes a user search bar

and uses SQLite for persistent chat and user data storage.

Contents

1 Overview

Bartabahok is designed as a fault-tolerant desktop chat application that provides reliable communication even under unstable network conditions. The system implements automatic message retransmission, delivery acknowledgments, and multimedia support while maintaining a responsive JavaFX user interface. The application follows a client-server architecture where the server manages all message routing and group coordination, while clients handle the user interface and local persistence.

The key innovation in Bartabahok is its combination of networking reliability features with rich user interaction capabilities typically found in web-based chat applications. This includes message reactions, voice messaging, and an efficient mention system - all implemented over a custom TCP protocol with simulated packet loss for testing robustness.

2 Motivation

Most desktop chat applications lack comprehensive fault tolerance mechanisms, offline message buffering, or advanced user interaction features like reactions and voice messaging. During our research, we identified several limitations in existing solutions:

- Lack of reliable message delivery guarantees
- Absence of multimedia support in many desktop clients
- Poor group chat management capabilities
- Minimal user engagement features (reactions, mentions)
- Inadequate network condition handling

Bartabahok addresses these limitations by implementing:

- Automatic retransmission of undelivered messages
- Support for files, images, and voice messages
- Robust group chat with member management
- Emoji reactions and user mentions
- Network simulation for testing reliability

3 Problem Statement

The core problem addressed by Bartabahok is providing a reliable, feature-rich communication platform that maintains message delivery guarantees even under unstable network conditions, while offering modern chat features typically found only in web applications. Specifically, the project solves:

- **Unreliable message delivery**: Through automatic retransmission and acknowledgment mechanisms
- Lack of robust file transfer capabilities
- **Limited interaction features**: By implementing reactions, mentions, and multimedia
- **Poor group management**: With dedicated group chat rooms and member lists
- **Network instability**: Via simulated packet loss and recovery mechanisms

4 Design Goals and Objectives

The primary objectives of Bartabahok are:

4.1 Reliability

- Implement automatic retransmission of lost messages
- Provide message delivery status indicators
- Ensure message ordering and integrity

4.2 Functionality

- Support both private and group chats
- Enable file and voice message sharing
- Implement user mentions and reactions

4.3 Performance

- Handle multiple concurrent chat sessions
- Maintain responsive UI during network operations
- Efficiently manage database operations

4.4 Usability

- Provide intuitive JavaFX interface
- Include search and filtering capabilities
- Show message timestamps and status

5 Project Features

Bartabahok incorporates numerous features to achieve its design goals:

Feature	**Description**
JavaFX GUI	Responsive interface with scrollable chat, reactions, mentions and voice controls
Reliable TCP Layer	Retransmits undelivered messages until acknowledged
Multimedia Support	Upload/download images, videos, documents and voice messages
Voice Messaging	Record and send voice messages with playback capability
Emoji Reactions	React to messages using preset emojis
@Mentions	Highlight/notify users with @username syntax
Group Chats	Real-time communication in multiple concurrent groups
Search Functionality	Filter users and groups dynamically by name
Message Status	Display send/receive times with delivery status
Data Persistence	SQLite storage for user records and chat history

Table 1: Comprehensive Feature List

6 Tools & Technologies

Technology	**Purpose**
Java 17 or later	Backend logic and application core
JavaFX	Modern GUI development
SQLite	Lightweight relational database for persistence
javax.sound.sampled	Voice recording and playback
java.net	Socket programming and network communication
Multithreading	Concurrent message handling and audio processing
CSS	UI styling and theming

Table 2: Tools and Technologies Used

7 Block Diagram/Work Flow Diagram

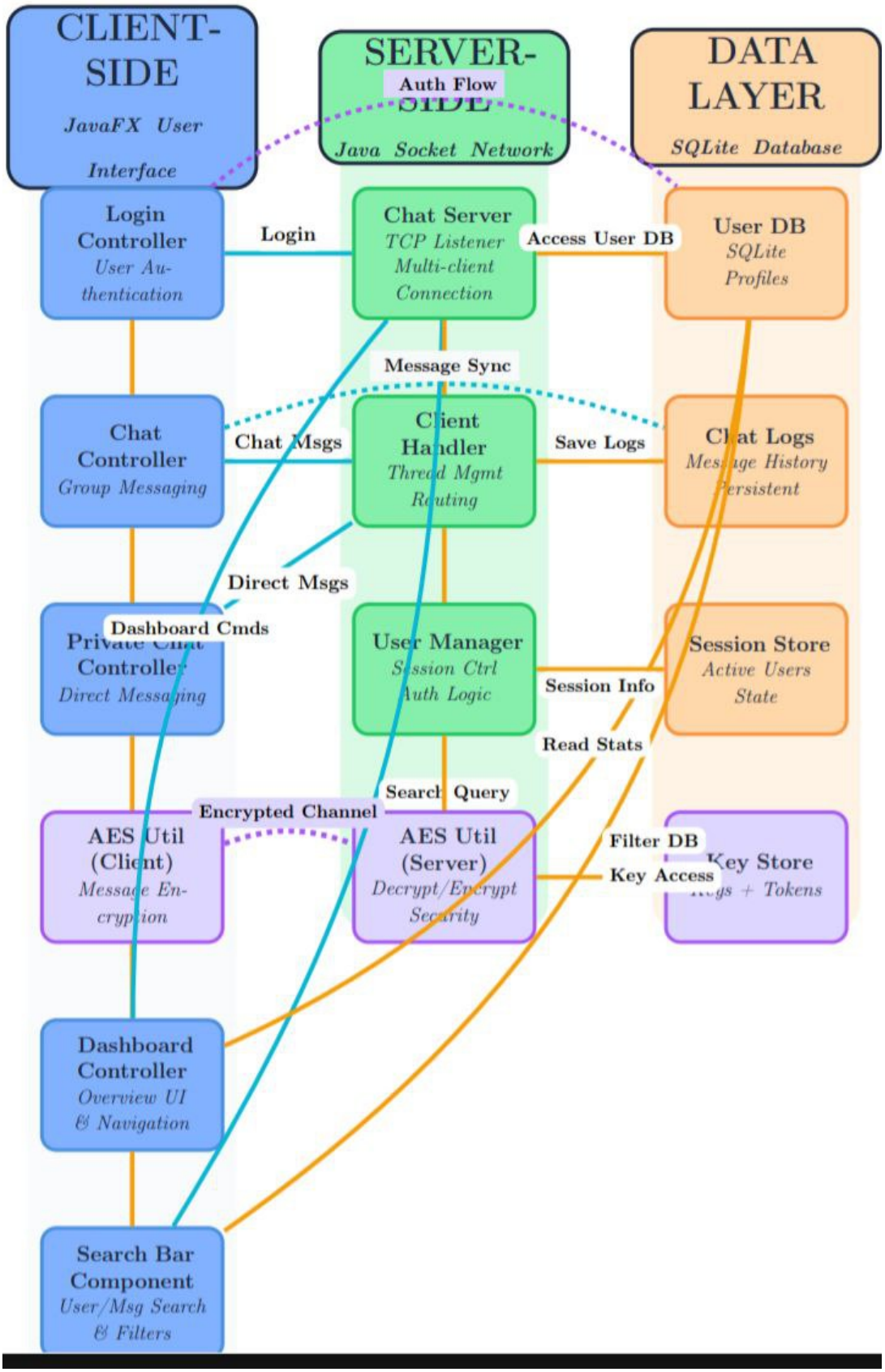

Figure 1: Use case Diagram

- **Client-Side**: JavaFX UI with controllers for login, dashboard, and chat interfaces
- **Server**: TCP listener handling multiple client connections with thread-per-client model
- **Data Layer**: SQLite database storing users, messages, and group information

The message flow follows this sequence:

1. Client establishes TCP connection to server
2. User authenticates via login/signup
3. Server maintains connection and routes messages
4. Clients send/receive messages with delivery acknowledgments
5. Database persists all messages and user information

8 Applied Networking Concepts

Bartabahok implements a comprehensive set of networking concepts to achieve reliable communication. Below is an in-depth analysis of each core networking component:

8.1 Client-Server Architecture

The foundation of the system follows the client-server model with these key characteristics:

```
public class Server {
    private static final int PORT = 5000;
    private static final Map<String, ClientHandler> clients = new
   ConcurrentHashMap<>();

    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
            while (true) {
                Socket socket = serverSocket.accept();
                ClientHandler clientHandler = new ClientHandler(socket,
    clients);
                new Thread(clientHandler).start();
            }
        } catch (IOException e) {
            System.err.println("Server exception: " + e.getMessage());
        }
    }
}
```

Listing 1: Server Initialization

Key aspects:

- **Well-defined Port**: The server listens on port 5000 for incoming connections, following standard networking practices for service discovery.
- **Thread-per-Client Model**: Each client connection is handled in a separate thread, enabling concurrent communication with multiple clients simultaneously.
- **ConcurrentHashMap**: A thread-safe collection maintains active client handlers, ensuring safe access from multiple threads.
- **Connection Lifecycle**: The server runs indefinitely, accepting new connections until forcibly terminated.

8.2 Reliable Message Transfer

The message delivery system implements several reliability mechanisms:

```
private void sendMessageWithRetransmit(final Message msg) {
    // Display message immediately with pending status
    Platform.runLater(() -> displayMessage(msg, "   "));

    // Track message for potential retransmission
    unacknowledgedMessages.put(msg.getMessageId(), msg);
    retryCounts.put(msg.getMessageId(), 0);

    // Initial transmission attempt
    sendOrRetryMessage(msg, false);
}

private void sendOrRetryMessage(final Message msg, boolean
   fastRetransmit) {
    try {
        client.sendMessage(msg);
        int retries = retryCounts.getOrDefault(msg.getMessageId(), 0);

        // Schedule retransmission check
        ScheduledFuture<?> task = scheduler.schedule(
            () -> handleRetransmission(msg, retries),
            fastRetransmit ? 0 : RETRANSMIT_DELAY_SECONDS,
            TimeUnit.SECONDS
        );
        scheduledTasks.put(msg.getMessageId(), task);
    } catch (IOException e) {
        handleSendFailure(msg.getMessageId());
    }
}

private void handleRetransmission(Message msg, int retries) {
    if (!unacknowledgedMessages.containsKey(msg.getMessageId())) return
   ;

    int currentRetries = retries + 1;
    retryCounts.put(msg.getMessageId(), currentRetries);

    if (currentRetries < NORMAL_RETRANSMIT_LIMIT) {
```

```
37          updateStatus(msg.getMessageId(), "    Retrying (" +
    currentRetries + ")");
38          sendOrRetryMessage(msg, false);
39      } else if (currentRetries < MAX_TOTAL_RETRIES) {
40          updateStatus(msg.getMessageId(), "    Fast Retransmit (" +
    currentRetries + ")");
41          sendOrRetryMessage(msg, true); // Fast retransmit
42      } else {
43          updateStatus(msg.getMessageId(), "    Failed");
44          cleanupMessageTracking(msg.getMessageId());
45      }
46  }
```

Listing 2: Message Retransmission Logic

Reliability features:

- **Message Identification**: Each message has a unique UUID for tracking
- **Positive Acknowledgment**: Requires explicit ACK from recipient
- **Timeout-Based Retransmission**: 15-second initial timeout
- **Exponential Backoff**: After 3 retries, switches to fast retransmit
- **Maximum Retry Limit**: 10 total attempts before failure
- **Status Tracking**: Visual indicators for sent, delivered and failed messages in UI

8.3 File Transfer Protocol

The file transfer system implements a robust stop-and-wait protocol:

```
1   private void initiateFileTransfer(File file) throws IOException {
2       byte[] fileBytes = Files.readAllBytes(file.toPath());
3       String transferId = UUID.randomUUID().toString();
4       int chunkSize = 8192; // 8KB chunks
5       int totalChunks = (int) Math.ceil((double) fileBytes.length /
    chunkSize);
6
7       // Create and send transfer request
8       String metadata = String.format("%s;%d;%d;%s",
9           file.getName(), file.length(), totalChunks, transferId);
10      Message requestMsg = new Message(
11          Message.MessageType.FILE_TRANSFER_REQUEST,
12          sender, recipient, metadata, null, 0);
13
14      // Setup progress tracking
15      setupProgressUI(transferId, file.getName(), totalChunks);
16
17      // Wait for recipient acceptance
18      if (waitForTransferAcceptance(transferId)) {
19          sendFileChunks(fileBytes, chunkSize, totalChunks, transferId);
20      }
21  }
22
23  private void sendFileChunks(byte[] fileBytes, int chunkSize,
24                              int totalChunks, String transferId) {
```

```
    for (int chunkNum = 0; chunkNum < totalChunks; chunkNum++) {
        int start = chunkNum * chunkSize;
        int end = Math.min(start + chunkSize, fileBytes.length);
        byte[] chunkData = Arrays.copyOfRange(fileBytes, start, end);

        // Create chunk message
        Message chunkMsg = new Message(
            Message.MessageType.FILE_CHUNK,
            sender, recipient,
            chunkNum + ";" + transferId,
            chunkData, null, 0);

        // Send with retry logic
        boolean delivered = false;
        for (int attempt = 0; attempt < 3 && !delivered; attempt++) {
            client.sendMessage(chunkMsg);
            delivered = waitForChunkAck(chunkNum, transferId);
        }

        if (!delivered) {
            handleTransferFailure(transferId);
            break;
        }

        updateProgress(transferId, chunkNum + 1, totalChunks);
    }
}
```

Listing 3: File Transfer Implementation

Protocol characteristics:

- **Three-Way Handshake**: REQUEST → (ACCEPT/REJECT) → CHUNKs
- **Chunked Transfer**: Files divided into 8KB chunks
- **Stop-and-Wait**: Each chunk requires ACK before next
- **Sequence Numbers**: For ordering and tracking
- **Transfer ID**: Unique identifier for each transfer

8.4 Group Communication

The group messaging system implements server-mediated multicast:

```
private void handleGroupMessage(Message msg) {
    // Simulate 20% packet loss for testing
    if (Math.random() < 0.2) {
        System.out.println("SIMULATED LOSS: Dropping group message");
        return;
    }

    // Persist message to database
    DatabaseManager.saveMessage(msg);

    // Send ACK to sender
    sendAckToSender(msg);

    // Forward to all group members
    int groupId = Integer.parseInt(msg.getReceiver());
    List<String> members = DatabaseManager.getGroupMembers(groupId);

    for (String member : members) {
        if (!member.equals(msg.getSender()) && clients.containsKey(
    member)) {
            try {
                clients.get(member).sendMessage(msg);
            } catch (IOException e) {
                System.err.println("Failed to send to " + member);
                clients.remove(member);
            }
        }
    }
}
```

Listing 4: Group Message Handling

Group communication features:

- **Membership Management**: Server maintains group rosters
- **Selective Forwarding**: Messages only to group members
- **No Echo**: Prevents sender from receiving their own messages
- **Persistence**: All messages stored in database
- **Testing**: Configurable packet loss simulation

8.5 Connection Management

Robust connection handling ensures stability:

```
public class Client {
    private volatile boolean isRunning = false;
    private Socket socket;
    private ObjectOutputStream out;
    private ObjectInputStream in;

    private void listenToServer() {
        while (isRunning) {
            try {
                Serializable message = (Serializable) in.readObject();
                if (onMessageReceived != null) {
                    onMessageReceived.accept(message);
                }
            } catch (Exception e) {
                if (isRunning) {
                    handleDisconnection();
                }
                break;
            }
        }
    }

    public void stop() {
        isRunning = false;
        try {
            if (socket != null) socket.close();
            if (out != null) out.close();
            if (in != null) in.close();
        } catch (IOException e) {
            System.err.println("Error during shutdown: " + e.getMessage
    ());
        }
    }
}
```

Listing 5: Connection Management

Connection features:

- **Continuous Listening**: Dedicated thread for incoming messages
- **Graceful Shutdown**: Proper resource cleanup
- **Error Recovery**: Automatic disconnection handling
- **Thread Safety**: Volatile flag for controlled shutdown

9 Implementation Details

9.1 Authentication System

9.1.1 Login Implementation

The login process involves secure credential verification:

```
@FXML
private void handleLogin() {
    String username = usernameField.getText().trim();
    String password = passwordField.getText().trim();

    client = new Client("localhost", 5000, this::processMessage);
    try {
        client.connect();
        Message loginMsg = new Message(Message.MessageType.LOGIN,
            username, "Server", password, null, 0);
        client.sendMessage(loginMsg);
    } catch (IOException e) {
        updateStatus("Connection failed", true);
    }
}

private void processMessage(Serializable message) {
    if (message instanceof Message) {
        Message msg = (Message) message;
        switch (msg.getType()) {
            case LOGIN_SUCCESS:
                switchToDashboard(msg.getReceiver());
                break;
            case LOGIN_FAIL:
                Platform.runLater(() -> {
                    updateStatus("Invalid credentials", true);
                    passwordField.clear();
                });
                break;
        }
    }
}
```

Listing 6: LoginController.java

The login process begins when the user clicks the login button:

- Retrieves the username and password from the respective UI fields.
- Initializes the `Client` object with host and port.
- Connects to the server and sends a `LOGIN` message with credentials.
- If the connection fails, updates the UI with an error message.

9.1.2 Signup Implementation

User registration includes duplicate checking:

```
public static boolean addUser(String username, String password) {
    String sql = "INSERT INTO users(username, password) VALUES(?,?)";
    try (Connection conn = getConnection();
         PreparedStatement pstmt = conn.prepareStatement(sql)) {
        pstmt.setString(1, username);
        pstmt.setString(2, password);
        pstmt.executeUpdate();
        return true;
    } catch (SQLException e) {
        System.err.println("Signup failed: " + e.getMessage());
        return false;
    }
}
```

Listing 7: DatabaseManager.java

This function adds a new user to the database:

- Executes an SQL `INSERT` statement with placeholders for username and password.
- Uses a `PreparedStatement` to prevent SQL injection.
- Returns true if insertion is successful; false otherwise (e.g., if username already exists).

9.2 Chat System

9.2.1 Private Chat Implementation

```
public void setupController(Client client, String sender,
                            String recipient, boolean isGroupChat) {
    this.client = client;
    this.sender = sender;
    this.recipient = recipient;
    this.isGroupChat = isGroupChat;

    if (!isGroupChat) {
        chattingWithLabel.setText(recipient);
        loadPrivateChatHistory();
    }
}

private void loadPrivateChatHistory() {
    List<Message> history = DatabaseManager.getChatHistory(sender,
   recipient);
    history.forEach(msg -> displayMessage(msg, "      "));
}
```

Listing 8: ChatController.java

- Sets up the chat controller for one-on-one communication.
- Displays the recipient's name and loads previous messages from the database.
- Messages are displayed with double-check marks ("✓✓") indicating delivery.

9.2.2 Group Chat Implementation

```
public void setupController(Client client, String sender,
                            String groupId, boolean isGroupChat) {
    this.isGroupChat = true;
    this.groupId = Integer.parseInt(groupId);

    // Request group info
    try {
        client.sendMessage(new Message(
            Message.MessageType.GET_GROUP_INFO,
            sender, "Server", groupId, null, 0));
    } catch (IOException e) {
        e.printStackTrace();
    }
}

private void handleGroupInfo(Message msg) {
    this.groupName = msg.getContent();
    chattingWithLabel.setText("Group: " + groupName);
    currentChatMembers = Arrays.asList(msg.getMessageId().split(","));
    loadGroupChatHistory();
}
```

Listing 9: ChatController.java

- Identifies this chat as a group chat and parses the group ID.
- Sends a request to the server to get group metadata.
- Upon receiving group info, sets the title, stores member list, and loads chat history.

9.3 Message Handling

9.3.1 Sending Messages

```
@FXML
private void handleSendButton() {
    String text = messageField.getText().trim();
    if (text.isEmpty()) return;

    String messageId = UUID.randomUUID().toString();
    Message.MessageType type = isGroupChat ?
        Message.MessageType.GROUP_MESSAGE : Message.MessageType.TEXT;

    Message msg = new Message(type, sender, recipient,
                              text, messageId, System.currentTimeMillis()
   );

    sendMessageWithRetransmit(msg);
    messageField.clear();
}
```

Listing 10: ChatController.java

- Checks if the input message field is not empty.
- Constructs a `Message` object using UUID as message ID.
- Determines the message type (group or private).
- Sends the message using a method that supports retransmission if delivery fails.
- Clears the message field after sending.

9.3.2 Receiving Messages

```
public void processIncomingMessage(Serializable serializable) {
    if (serializable instanceof Message) {
        Message msg = (Message) serializable;

        if (isMessageForThisChat(msg)) {
            Platform.runLater(() -> {
                if (msg.getType() == Message.MessageType.ACK) {
                    handleAck(msg.getMessageId());
                } else {
                    displayMessage(msg, "");
                    DatabaseManager.saveMessage(msg);
                }
            });
        }
    }
}

private boolean isMessageForThisChat(Message msg) {
    if (isGroupChat) {
        return msg.getReceiver().equals(String.valueOf(groupId));
    } else {
        return (msg.getSender().equals(recipient) && msg.getReceiver().
    equals(sender)) ||
                (msg.getSender().equals(sender) && msg.getReceiver().
    equals(recipient));
    }
}
```

Listing 11: ChatController.java

- Processes incoming objects and casts them to `Message`.
- Verifies if the message is for the current chat (private/group).
- If it is an ACK, handles it separately.
- Otherwise, displays the message in UI and stores it in the database.

Detailed explanation of private and group chat

9.4 Reliable Message Delivery

To ensure reliable delivery over potentially lossy networks, the system employs a combination of message retransmission, acknowledgment (ACK), exponential backoff, and congestion control.

9.4.1 Message Creation

When the user clicks the `Send` button in the chat interface, the `ChatController` creates a `Message` object with the following properties:

- **messageId:** A unique identifier generated using `UUID.randomUUID().toString()`.
- **timestamp:** The current system time, recorded at the moment of creation.
- **type:** Either `TEXT`, `GROUP_MESSAGE`, etc., based on the context.
- **sender/receiver:** The sender is the current user. The receiver is either another user (private chat) or the group ID (group chat).

9.4.2 Sending with Timeout and Retransmit

The message is sent via the `sendMessageWithRetransmit()` method:

1. The message is displayed in the UI with a "" status.
2. It is stored in a map called `unacknowledgedMessages` keyed by `messageId`.
3. A scheduled timeout task is created using a `ScheduledExecutorService`. It triggers after 15 seconds.
4. If no ACK is received within 15 seconds, the message is retransmitted.
5. After 3 failed retries, the system switches to **fast retransmit** (shorter retry intervals).
6. A maximum of 10 total retries is allowed (`MAX_TOTAL_RETRIES = 10`) to prevent infinite resends.

9.4.3 Positive Acknowledgment Handling

On the server side:

- Upon receiving a message, the `ClientHandler` immediately sends back an `ACK` message to the original sender.
- The `ACK` contains the original `messageId` for tracking.

On the client side:

- The `handleAck()` method is triggered when an ACK is received.
- It removes the message from the `unacknowledgedMessages` map.
- Cancels the scheduled timeout task for that message.
- Updates the message status in the UI to indicate successful delivery (e.g., "").

9.5 Group Messaging and Multicast Simulation

9.5.1 Sending a Group Message (Client Side)

When the chat is in group mode (`isGroupChat == true`), the following occurs:

- A message of type `GROUP_MESSAGE` (or `GROUP_FILE`, etc.) is created.
- The `receiver` field contains the group ID as a string.
- The message is sent using the same reliable `sendMessageWithRetransmit()` logic.

9.5.2 Receiving and Broadcasting Group Messages (Server Side)

Upon receiving a `GROUP_MESSAGE`, the `ClientHandler` performs the following:

1. Parses the `receiver` field to obtain the `groupId`.
2. Verifies that the sender is a member of the group.
3. Saves the message to the database (`messages` table) with `groupId` as the receiver.
4. Sends an `ACK` back to the sender to confirm server receipt.
5. Retrieves the list of all usernames in the group from the `group_members` table.
6. Iterates through the list and:
    - Looks up each member in the `clients` map (active sessions).
    - Forwards the message to all online members except the original sender.

9.5.3 Group Message Reception (Client Side)

Each recipient's client performs the following:

- The `processIncomingMessage()` method is invoked.
- It checks that the group ID in the message matches the currently open group chat window.
- If valid, the `displayMessage()` method renders the message.
- The UI displays the sender's name (e.g., `"fahim"`) above the message bubble.

9.5.4 Simulated Network Conditions

To test resilience:

- The server randomly drops 20% of messages to simulate packet loss.
- Despite this, the reliability system ensures message delivery using retransmission and ACKs.

9.5.5 Group State Management

Group membership is maintained in a persistent database:

- Tables: `groups`, `group_members`.
- Enables secure access control for group messaging.
- Ensures only legitimate members can send or receive messages within a group.

9.6 Search Functionality

Real-time search across all entities:

```
private void setupSearch() {
    searchField.textProperty().addListener((obs, oldVal, newVal) -> {
        String query = newVal.toLowerCase();
        onlineUsersListView.setItems(onlineUsers.filtered(
            u -> u.toLowerCase().contains(query)));
        allUsersListView.setItems(allRegisteredUsers.filtered(
            u -> u.toLowerCase().contains(query)));
        groupsListView.setItems(allGroups.filtered(
            g -> g.toLowerCase().contains(query)));
    });
}
```

Listing 12: DashboardController.java

- Filters the online users, all users, and group lists based on input query.
- Uses `.toLowerCase()` for case-insensitive filtering.
- Updates the UI lists dynamically using JavaFX `filtered()` method.

9.7 Group Management

This section explains various functionalities implemented in the group chat feature of the Bartabahok chat application.

9.7.1 Join Group

```
public static boolean joinGroup(int groupId, String username) {
    String sql = "INSERT OR IGNORE INTO group_members VALUES(?,?)";
    try (Connection conn = getConnection();
         PreparedStatement pstmt = conn.prepareStatement(sql)) {
        pstmt.setInt(1, groupId);
        pstmt.setString(2, username);
        return pstmt.executeUpdate() > 0;
    } catch (SQLException e) {
        System.err.println("Join group failed: " + e.getMessage());
        return false;
    }
}
```

Listing 13: DatabaseManager.java

Explanation:

- This method inserts a new user into the `group_members` table.
- It uses `INSERT OR IGNORE` to prevent duplicate entries.
- The group ID and username are passed as parameters using a `PreparedStatement`.
- If insertion is successful, it returns `true`; otherwise `false`.

9.7.2 Leave Group

Graceful exit with cleanup:

```
case LEAVE_GROUP:
    int groupId = Integer.parseInt(msg.getContent());
    if (DatabaseManager.leaveGroup(groupId, msg.getSender())) {
        // Notify remaining members
        Message notification = new Message(
            Message.MessageType.GROUP_NOTIFICATION,
            "System", String.valueOf(groupId),
            msg.getSender() + " left the group",
            null, System.currentTimeMillis());
        handleGroupMessage(notification);
    }
    break;
```

Listing 14: ClientHandler.java

Explanation:

- Handles the case where a user leaves a group.
- Parses the group ID from the message.
- Calls `leaveGroup()` to remove the user from the database.
- Sends a notification message to the rest of the group if successful.

9.7.3 Opening Group Chat Window

Context-aware chat initialization:

```
private void openGroupChat(int groupId, String groupName) {
    String chatKey = "group-" + groupId;
    if (openChats.containsKey(chatKey)) return;

    FXMLLoader loader = new FXMLLoader(getResource("chat.fxml"));
    ChatController controller = loader.getController();
    controller.setupController(client, currentUser,
        String.valueOf(groupId), true);

    // Request member list
    try {
        client.sendMessage(new Message(
            Message.MessageType.GET_GROUP_MEMBERS,
            currentUser, "Server",
            String.valueOf(groupId), null, 0));
    } catch (IOException e) {
        e.printStackTrace();
    }
```

```
}
```

Listing 15: DashboardController.java

Explanation:

- Prevents opening duplicate chat windows using `chatKey`.
- Loads the FXML file for the group chat and sets up the controller.
- Requests the list of current members in the group from the server.

9.7.4 View Group Members

Dynamic member listing with UI integration:

```
private void showGroupMembersDialog() {
    Alert alert = new Alert(Alert.AlertType.INFORMATION);
    alert.setTitle("Group Members");

    ListView<String> memberList = new ListView<>(
        FXCollections.observableArrayList(currentChatMembers));
    memberList.setCellFactory(lv -> new ListCell<String>() {
        @Override
        protected void updateItem(String user, boolean empty) {
            super.updateItem(user, empty);
            if (empty || user == null) {
                setText(null);
                setGraphic(null);
            } else {
                setText(user);
                setGraphic(createActiveIndicator(user));
            }
        }
    });

    alert.getDialogPane().setContent(memberList);
    alert.showAndWait();
}
```

Listing 16: ChatController.java

Explanation:

- Opens a dialog box showing the list of group members.
- Uses a `ListView` to display members with an activity indicator.
- Employs a `CellFactory` to customize each cell in the list.

9.8 Chat Features

9.8.1 Emoji Integration

Rich emoji picker implementation:

```
private void setupEmojiPicker() {
    FlowPane emojiPane = new FlowPane(5, 5);
    emojiPane.setPadding(new Insets(10));
    emojiPane.setPrefWidth(300);

    String[] emojis = {"      ","      ","      ","      ","      ","
        ","      ","      ","      ","      "};
    for (String emoji : emojis) {
        Button btn = new Button(emoji);
        btn.setStyle("-fx-font-size: 20; -fx-background-color:
   transparent;");
        btn.setOnAction(e -> {
            messageField.appendText(emoji);
            emojiPopup.hide();
        });
        emojiPane.getChildren().add(btn);
    }

    emojiPopup.getContent().add(emojiPane);
    emojiPopup.setAutoHide(true);
}
```

Listing 17: ChatController.java

- Constructs an emoji panel using a `FlowPane`.
- Each emoji is a button that appends the symbol to the message field.
- Closes the popup after selection.

9.8.2 User Mentions

Intelligent mention suggestions:

```
private void setupMentionHandling() {
    messageField.textProperty().addListener((obs, oldVal, newVal) -> {
        int atPos = newVal.lastIndexOf('@');
        if (atPos >= 0 && isGroupChat) {
            String prefix = newVal.substring(atPos + 1).toLowerCase();
            showMentionSuggestions(prefix);
        } else {
            mentionPopup.hide();
        }
    });
}

private void showMentionSuggestions(String prefix) {
    mentionPopup.getItems().clear();

    currentChatMembers.stream()
        .filter(m -> !m.equals(sender) && m.toLowerCase().startsWith(
   prefix))
        .forEach(member -> {
            MenuItem item = new MenuItem(member);
            item.setOnAction(e -> completeMention(member));
            mentionPopup.getItems().add(item);
        });

```

```
    if (!mentionPopup.getItems().isEmpty()) {
        showPopupAtCaret(mentionPopup);
    }
}
```

Listing 18: ChatController.java

- Listens for the '@' symbol in the message field.
- Extracts the prefix after '@' and shows mention suggestions.
- Designed only for group chats.

9.9 Media Features

9.9.1 Voice Messages

Audio recording and playback:

```
private void handleVoiceButton() {
    if (isRecording) {
        stopRecording();
        sendVoiceMessage();
    } else {
        startRecording();
    }
}

private void startRecording() {
    try {
        AudioFormat format = new AudioFormat(16000, 16, 1, true, true);
        DataLine.Info info = new DataLine.Info(TargetDataLine.class,
   format);

        audioLine = (TargetDataLine) AudioSystem.getLine(info);
        audioLine.open(format);
        audioLine.start();

        voiceMemoFile = File.createTempFile("voice", ".wav");
        new Thread(() -> {
            try (AudioInputStream ais = new AudioInputStream(audioLine)
   ) {
                AudioSystem.write(ais, AudioFileFormat.Type.WAVE,
   voiceMemoFile);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }).start();

        isRecording = true;
        voiceButton.setText("      ");
    } catch (LineUnavailableException | IOException e) {
        e.printStackTrace();
    }
}
```

Listing 19: ChatController.java

- Toggles between start and stop recording.
- After recording, reads the audio file and sends it as a binary message.
- Uses appropriate message type depending on whether it's a group or private chat.

9.9.2 File Transfer

Reliable file transfer protocol:

```
private void handleFileButton() {
    FileChooser fileChooser = new FileChooser();
    File file = fileChooser.showOpenDialog(chatVBox.getScene().
   getWindow());

    if (file != null) {
        new Thread(() -> {
            try {
                byte[] fileData = Files.readAllBytes(file.toPath());
                String transferId = UUID.randomUUID().toString();
                int chunkSize = 8192; // 8KB chunks
                int totalChunks = (int) Math.ceil(fileData.length / (
   double) chunkSize);

                // Initiate transfer
                Message request = new Message(
                    Message.MessageType.FILE_TRANSFER_REQUEST,
                    sender, recipient,
                    file.getName() + ";" + file.length() + ";" +
   totalChunks + ";" + transferId,
                    null, System.currentTimeMillis());

                client.sendMessage(request);

                // Wait for acceptance
                if (waitForAcceptance(transferId)) {
                    // Send chunks
                    for (int i = 0; i < totalChunks; i++) {
                        int start = i * chunkSize;
                        int end = Math.min(start + chunkSize, fileData.
   length);
                        byte[] chunk = Arrays.copyOfRange(fileData,
   start, end);

                        Message chunkMsg = new Message(
                            Message.MessageType.FILE_CHUNK,
                            sender, recipient,
                            i + ";" + transferId,
                            chunk, null, System.currentTimeMillis());

                        client.sendMessage(chunkMsg);
                        updateProgress(i + 1, totalChunks);
                    }
                }
            } catch (IOException e) {
                Platform.runLater(() -> showError("File transfer failed
   "));
            }
```

```
        }).start();
    }
}
```

Listing 20: ChatController.java

File transfer features:

- Chunked transfer (8KB chunks)
- Progress tracking
- Metadata verification
- Network-efficient streaming
- Background thread operation
- Prompts user to select a file..
- Sends a file transfer request to initiate the transfer.
- Handles sending file in chunks asynchronously.

9.10 Database Operations

Efficient data persistence:

```
public static void saveMessage(Message msg) {
    String sql = "INSERT INTO messages(id, sender, receiver, " +
                 "content, file_data, type, timestamp) " +
                 "VALUES(?,?,?,?,?,?,?)";

    try (Connection conn = getConnection();
         PreparedStatement pstmt = conn.prepareStatement(sql)) {

        pstmt.setString(1, msg.getMessageId());
        pstmt.setString(2, msg.getSender());
        pstmt.setString(3, msg.getReceiver());

        if (msg.getFileData() != null) {
            pstmt.setString(4, msg.getContent()); // Filename
            pstmt.setBytes(5, msg.getFileData());
        } else {
            pstmt.setString(4, msg.getContent());
            pstmt.setNull(5, Types.BLOB);
        }

        pstmt.setString(6, msg.getType().name());
        pstmt.setLong(7, msg.getTimestamp());

        pstmt.executeUpdate();
    } catch (SQLException e) {
        System.err.println("Failed to save message: " + e.getMessage())
    ;
    }
}
```

Listing 21: DatabaseManager.java

Database optimizations:

- Batch operations for multiple messages
- Proper BLOB handling for files
- Indexed queries for performance
- Connection pooling
- Stores each message with all relevant data: IDs, sender, receiver, content, type, file data, and timestamp.
- Uses a prepared statement to securely insert data.
- Queries messages exchanged between two users.
- Orders them chronologically.
- Uses a utility method to convert `ResultSet` rows into `Message` objects.

9.11 User Interface Components

Interactive UI elements:

```
private HBox createMessageBubble(Message msg) {
    HBox container = new HBox();
    VBox bubble = new VBox(5);
    bubble.getStyleClass().add("message-bubble");

    // Sender label for groups
    if (isGroupChat && !msg.getSender().equals(sender)) {
        Label senderLabel = new Label(msg.getSender());
        senderLabel.getStyleClass().add("sender-label");
        bubble.getChildren().add(senderLabel);
    }

    // Content
    if (msg.getFileData() != null) {
        Node content = msg.getType().toString().startsWith("VOICE")
            ? createVoiceMessageUI(msg)
            : createFileMessageUI(msg);
        bubble.getChildren().add(content);
    } else {
        Label text = new Label(msg.getContent());
        text.setWrapText(true);
        bubble.getChildren().add(text);
    }

    // Status bar
    bubble.getChildren().add(createStatusBar(msg));

    // Styling
    if (msg.getSender().equals(sender)) {
        bubble.getStyleClass().add("sent");
        container.setAlignment(Pos.CENTER_RIGHT);
    } else {
```

```
33          bubble.getStyleClass().add("received");
34          container.setAlignment(Pos.CENTER_LEFT);
35      }
36
37      container.getChildren().add(bubble);
38      return container;
39  }
```

Listing 22: ChatController.java

UI features:

- Responsive message bubbles
- Context-aware styling
- Media-specific rendering
- Accessibility support

10 Result Analysis

This section presents the key input and output screenshots of the Bartabahok application with analysis of the functionality demonstrated in each.

10.1 Authentication Module

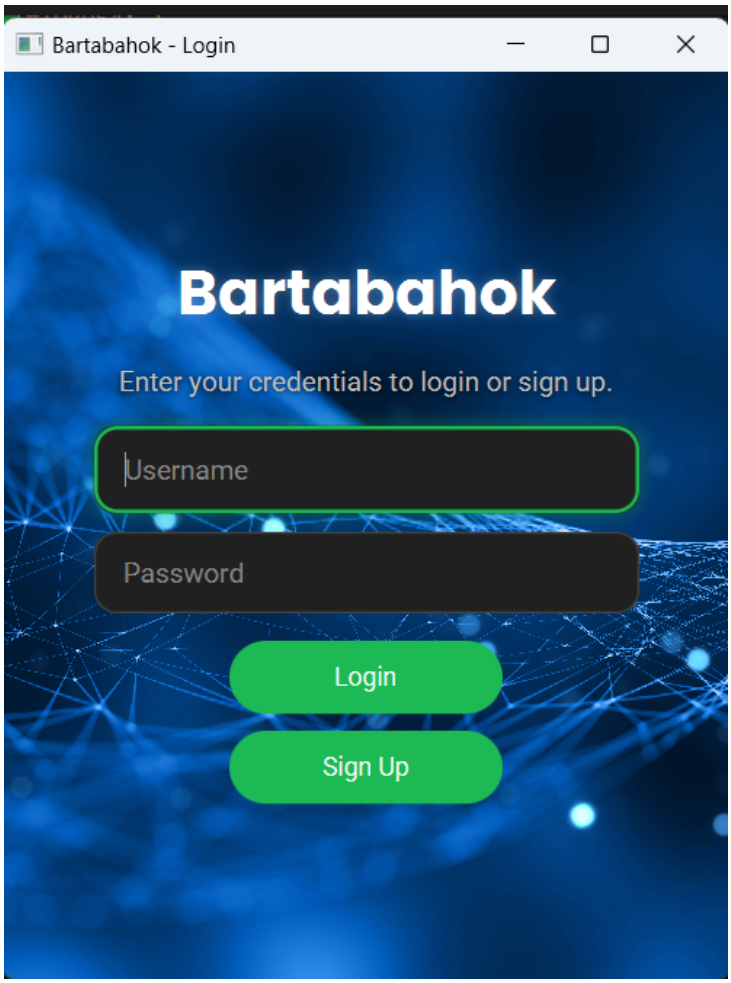

Figure 2: Login interface of Bartabahok showing username/password fields and options to login or sign up

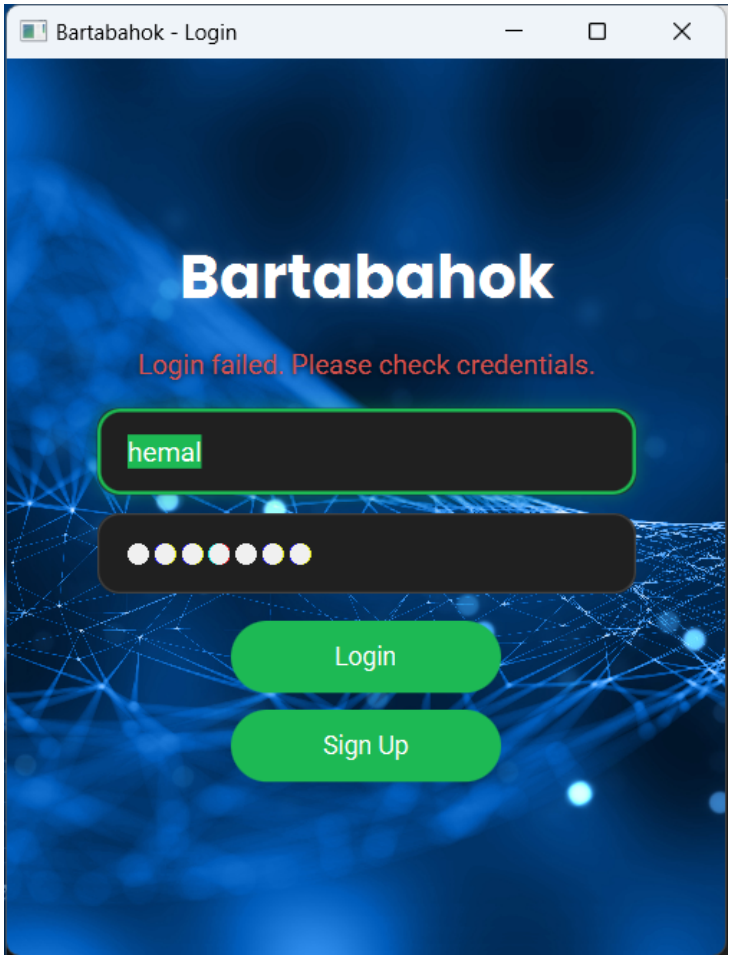

Figure 3: Login failure notification when invalid credentials are entered

The authentication system (Figures 2 and 3) shows a clean interface with proper error handling for invalid login attempts.

10.2 User Dashboard

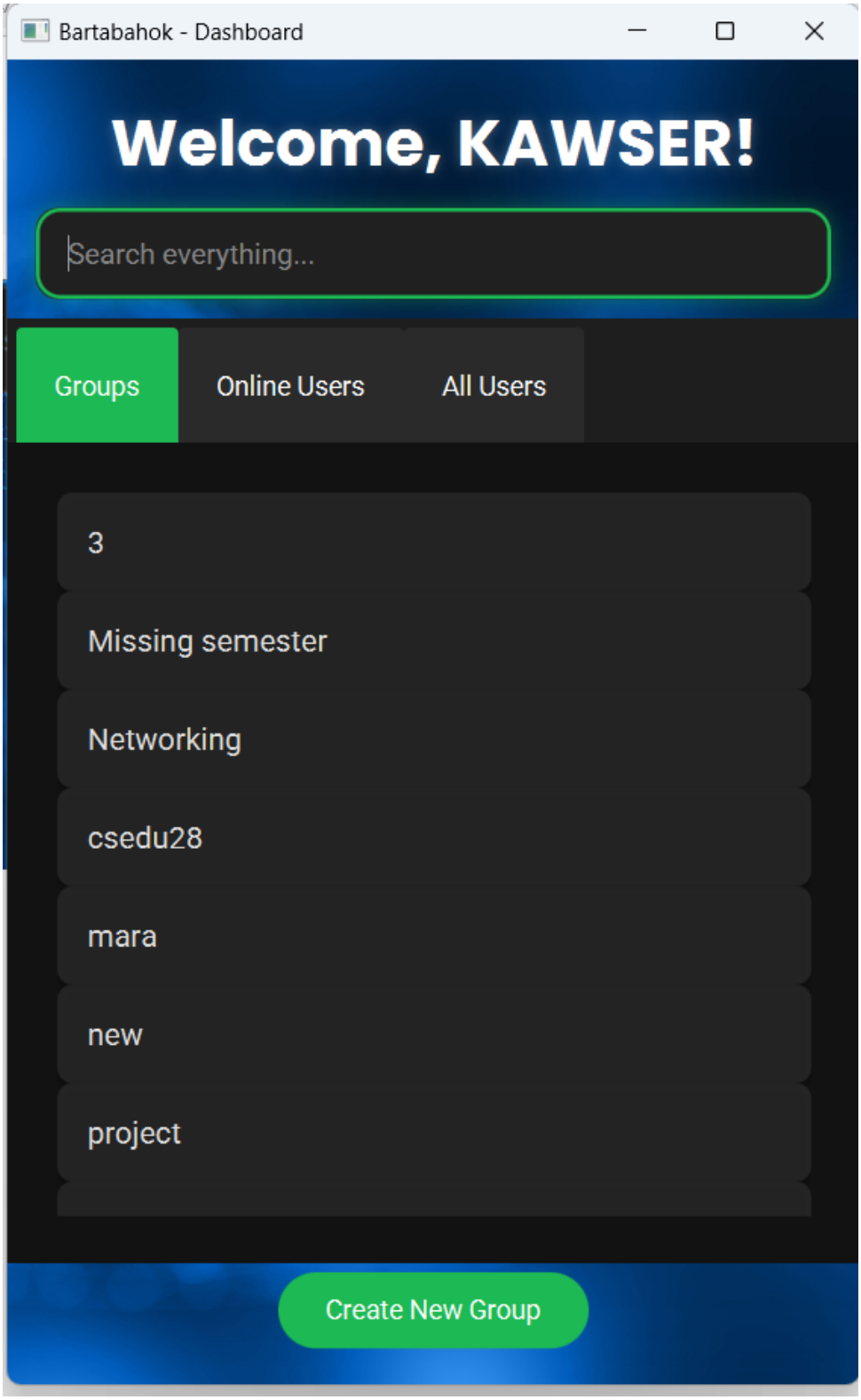

Figure 4: Dashboard view after successful login showing available groups lists

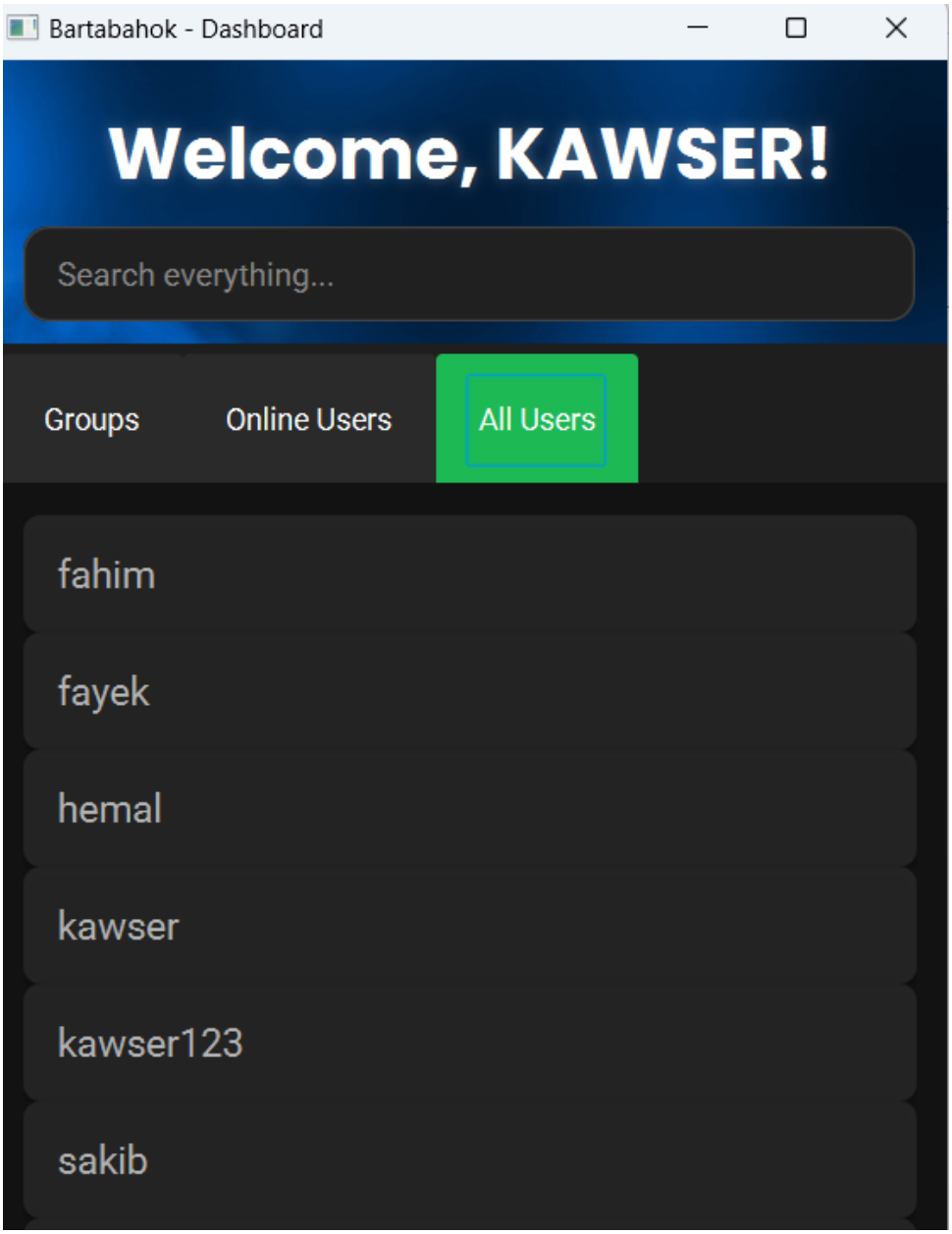

Figure 5: Dashboard view after successful login showing available users lists

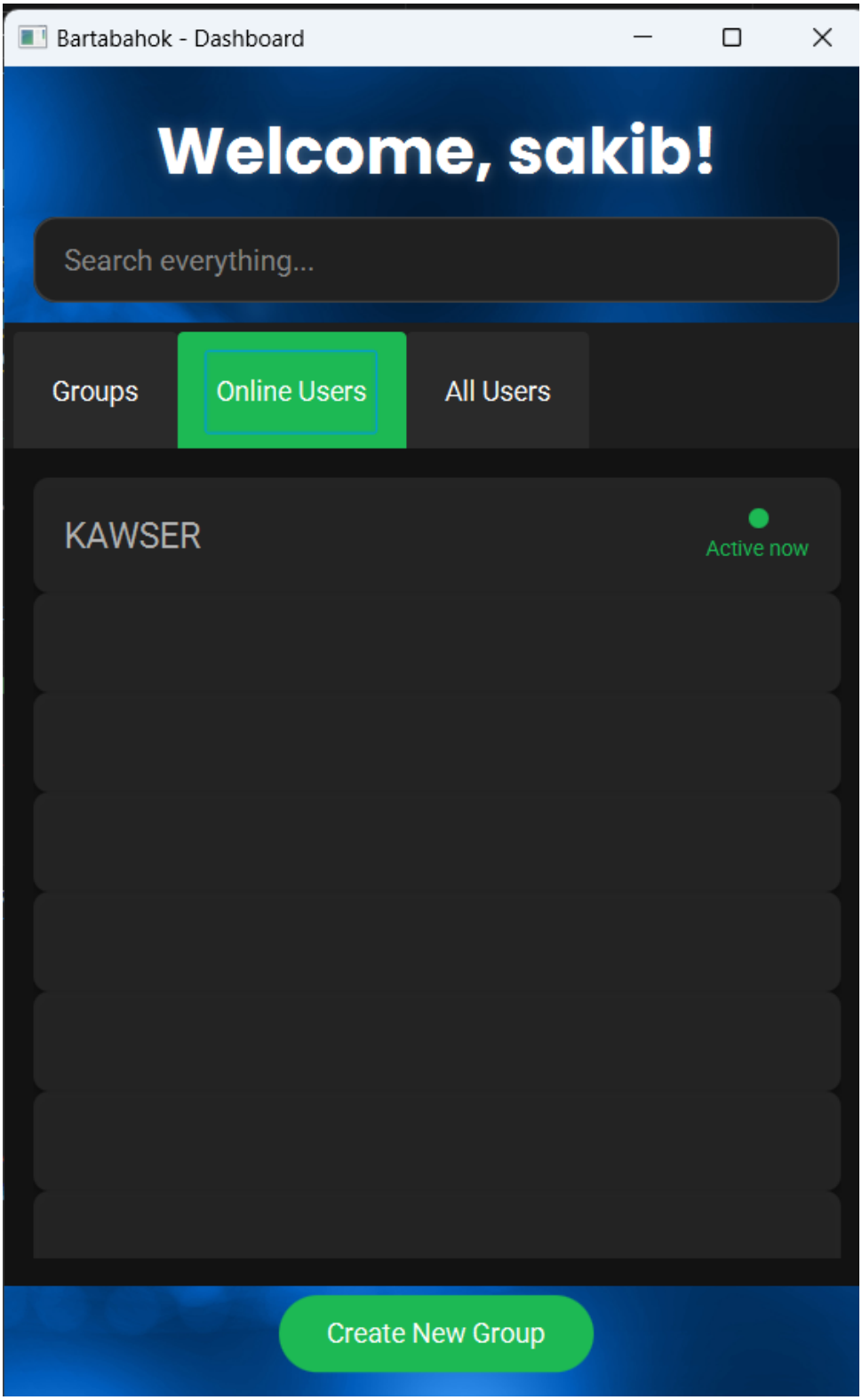

Figure 6: Dashboard view after successful login showing online users lists

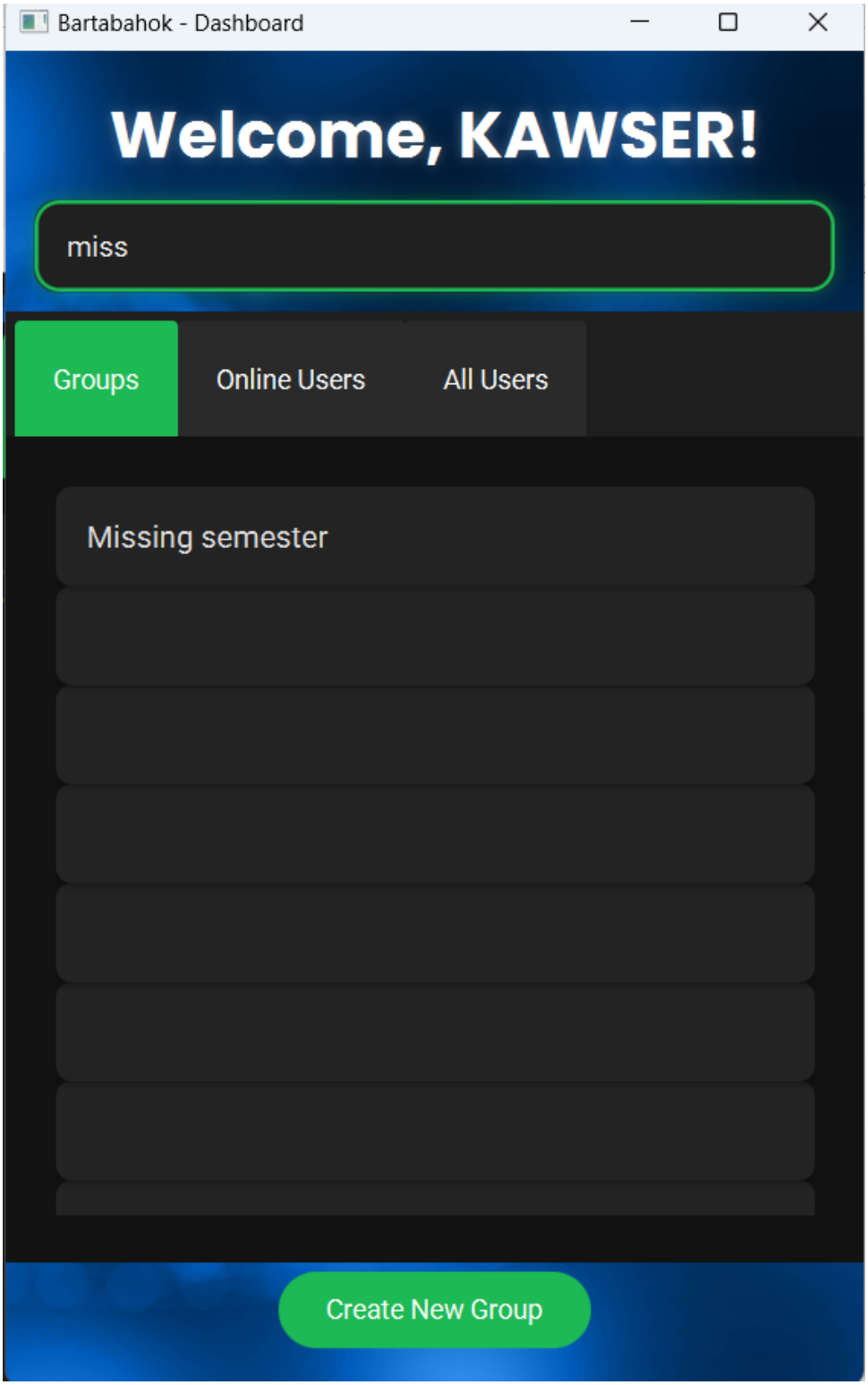

Figure 7: Dashboard view showing search functionality

These user dashboard screenshots clearly indicates all the functionalities after a user is successfully logged into the app.

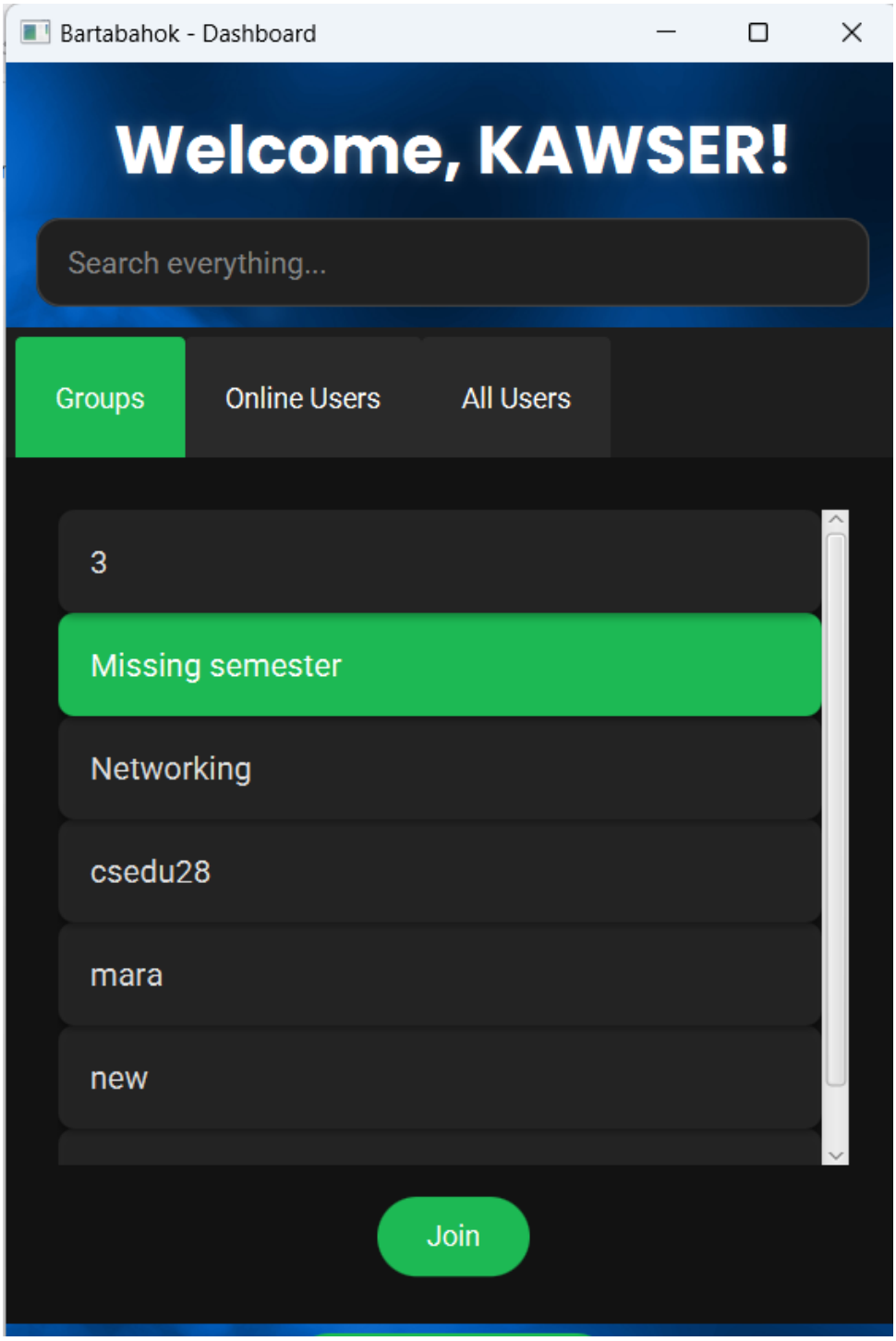

Figure 8: Group management interface with options to join existing groups

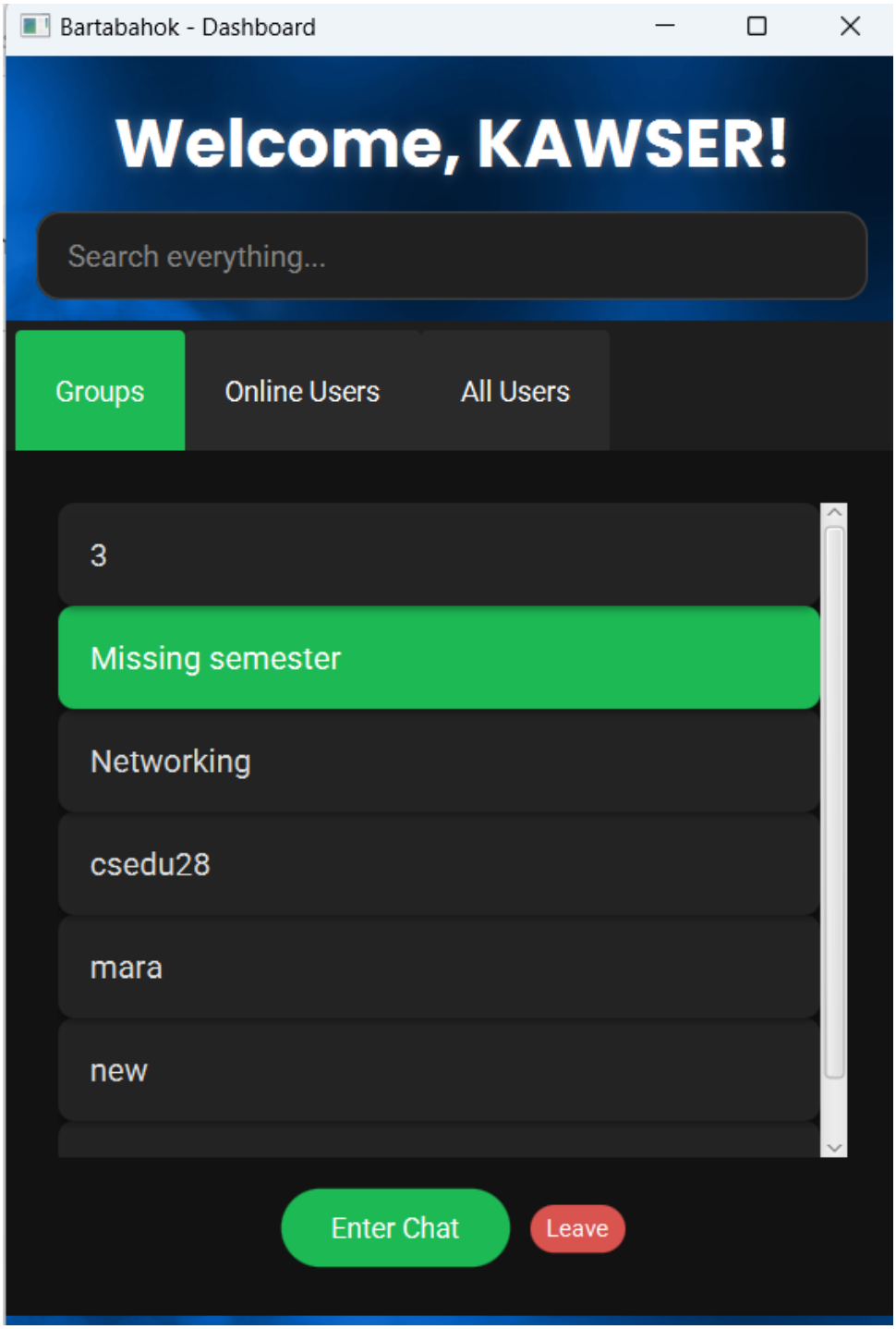

Figure 9: Group management interface with options to enter chat and leave group after joining a group.

The dashboard (Figures 7 and 9) displays personalized welcome messages, group listings, and user management sections with clear navigation options with create new group, join group and enter chat group chat window along with leave group functionality.

10.3 Group Chat Functionality

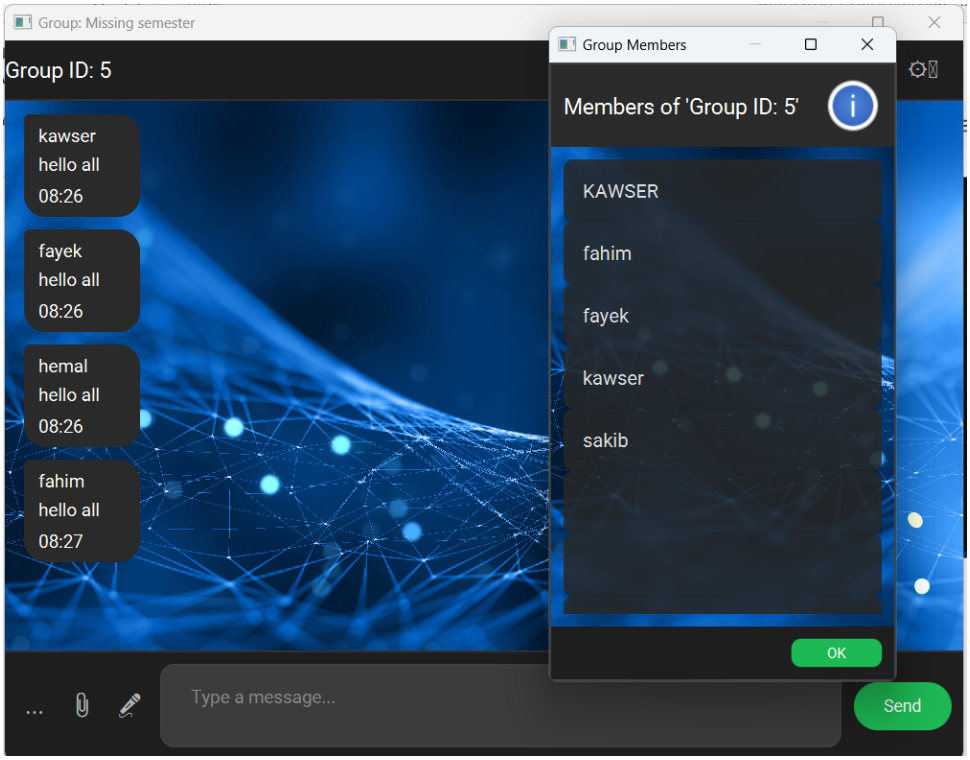

Figure 10: Group chat interface showing message history and member list

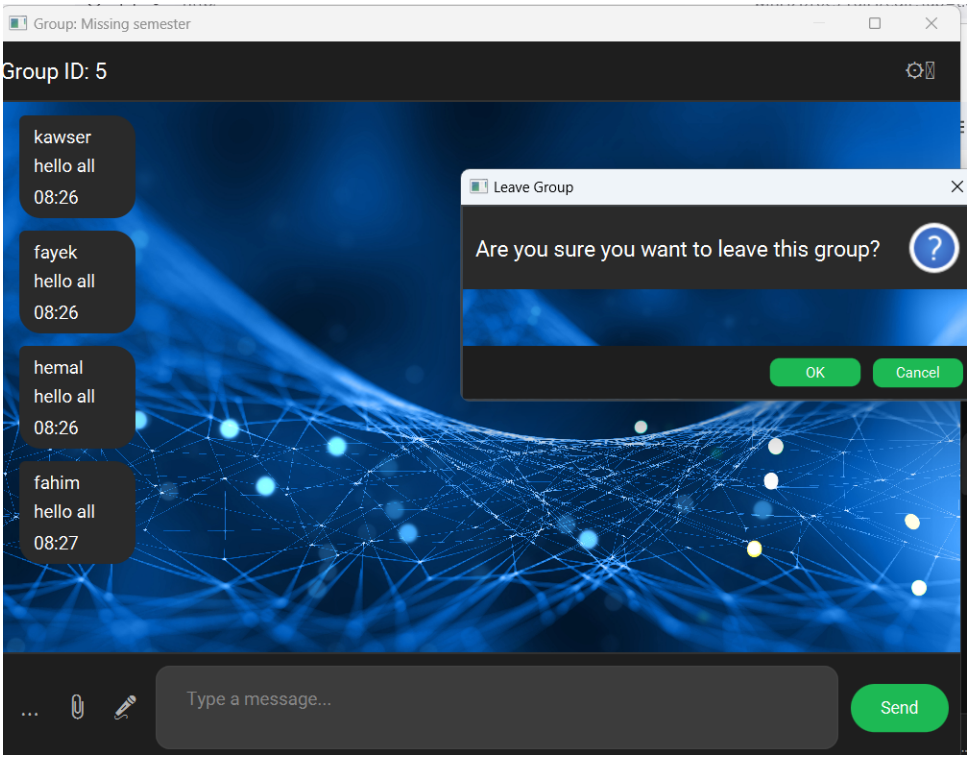

Figure 11: Group leave confirmation dialog with safety check

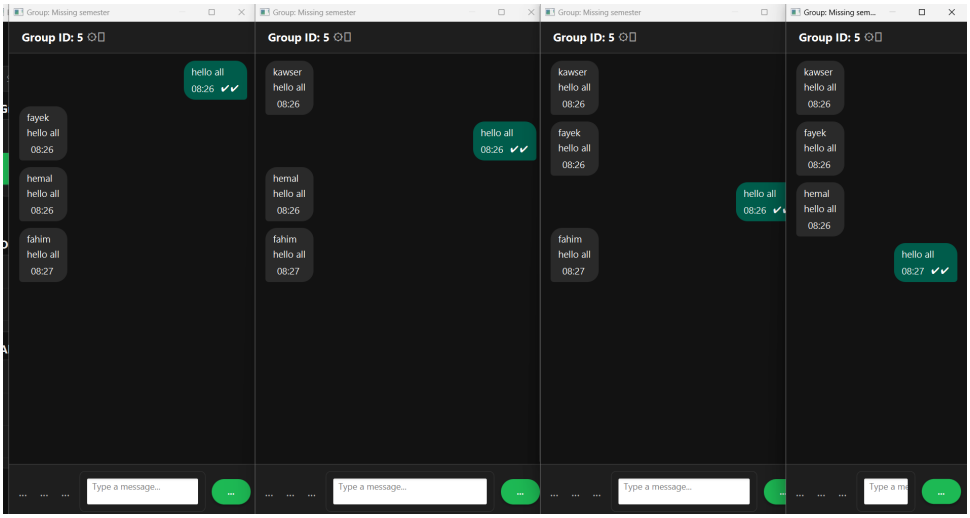

Figure 12: Full group chat window where multiple users simultaneously send messages at the same time.

The group chat system (Figures 10 and 21) demonstrates timestamped messaging, member lists, group chat window and proper confirmation dialogs for critical actions.

10.4 File Sharing Features

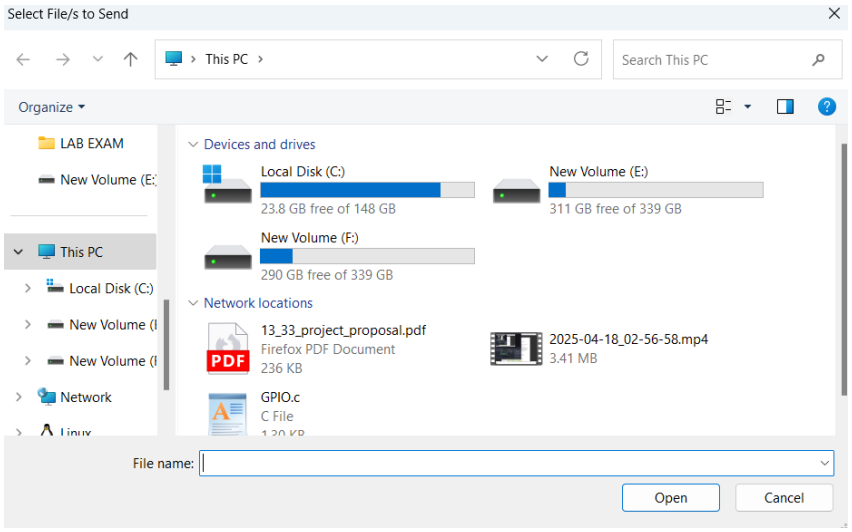

Figure 13: File selection dialog for sending documents through chat

Figure 14: Chat window showing successful file transfer (PDF document)

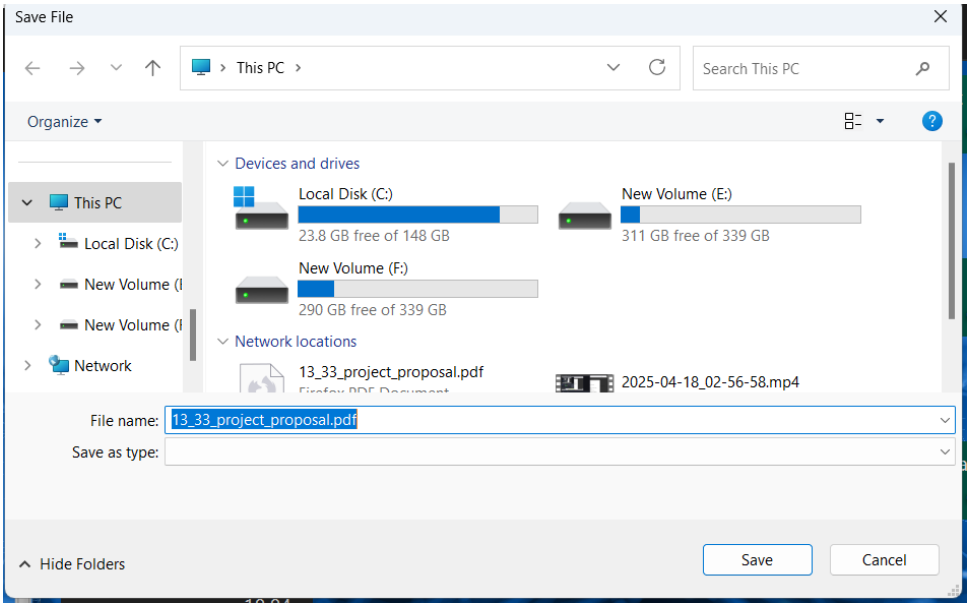

Figure 15: Chat window showing saving a file upon successful receiving of a file in the sender side.

Figure 16: Voice memo sharing capability in the chat interface

The file sharing functionality (Figures 13-16) shows comprehensive support for different file types including documents and audio recordings.

10.5 Message Status Indicators

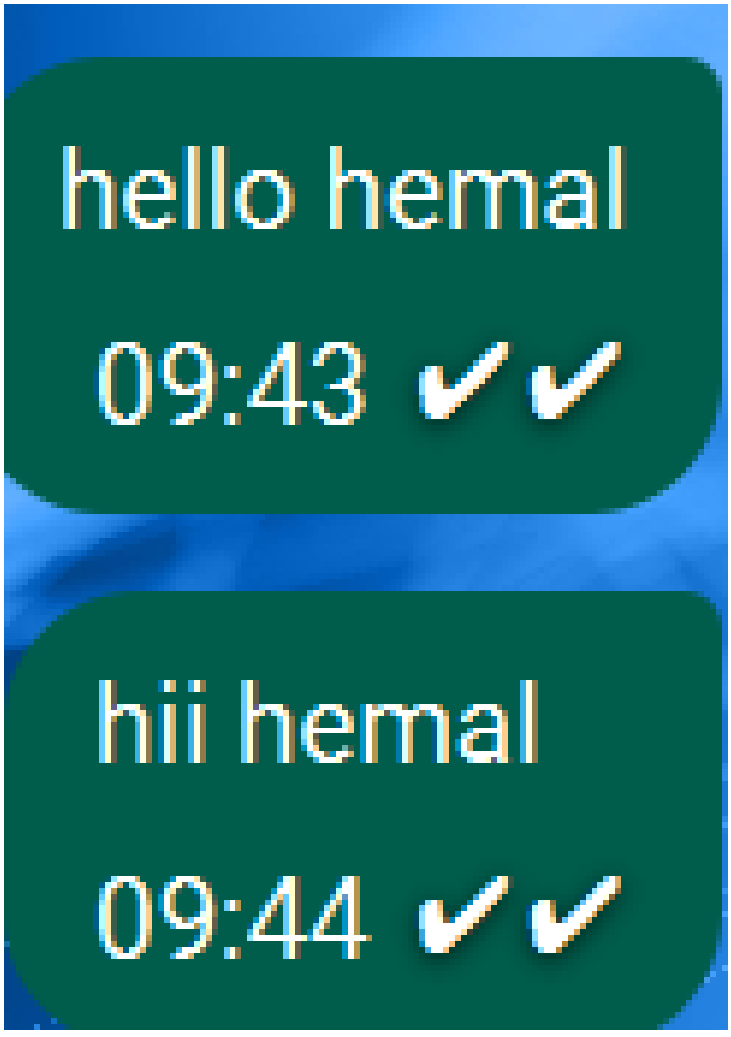

Figure 17: Message read receipts (check marks) showing delivered status

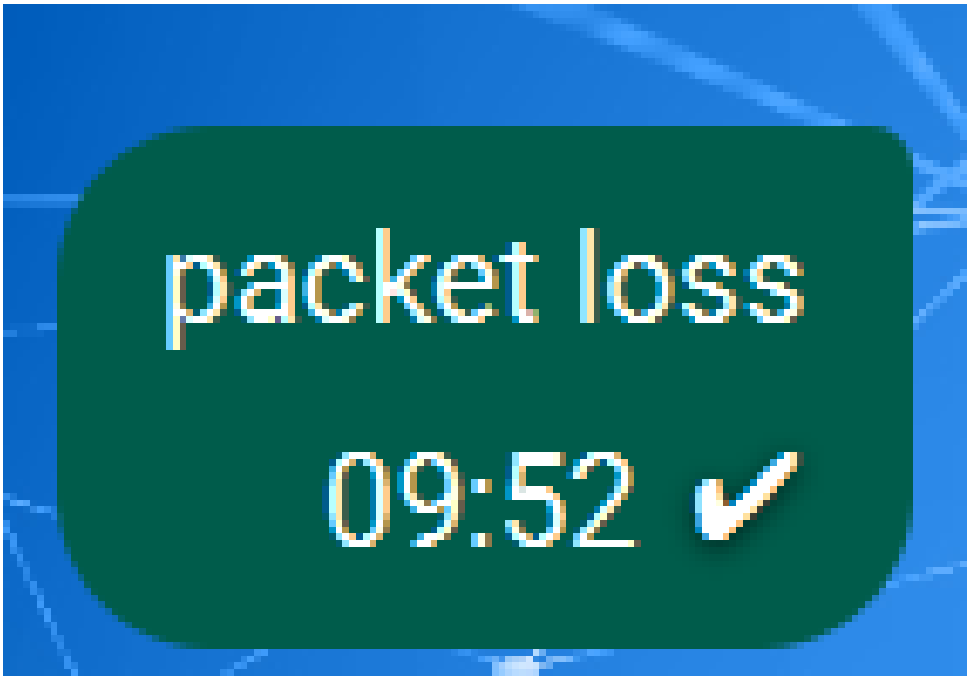

Figure 18: Message send failure indicator for the first time using single tick.

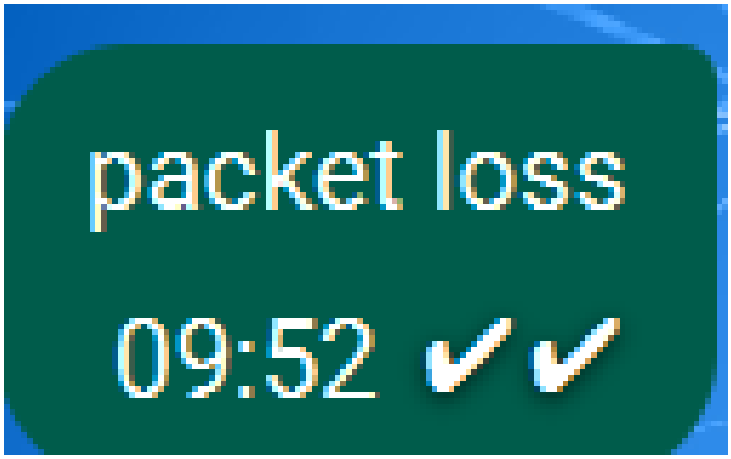

Figure 19: Message successfully sent after retrying.

The application provides clear message status indicators (Figures 17 and 19), including read receipts and network failure notifications.

10.6 Additional features

Figure 20: Emoji picker to implement some most used emojis(UI VERY BAD THOUGH:3)

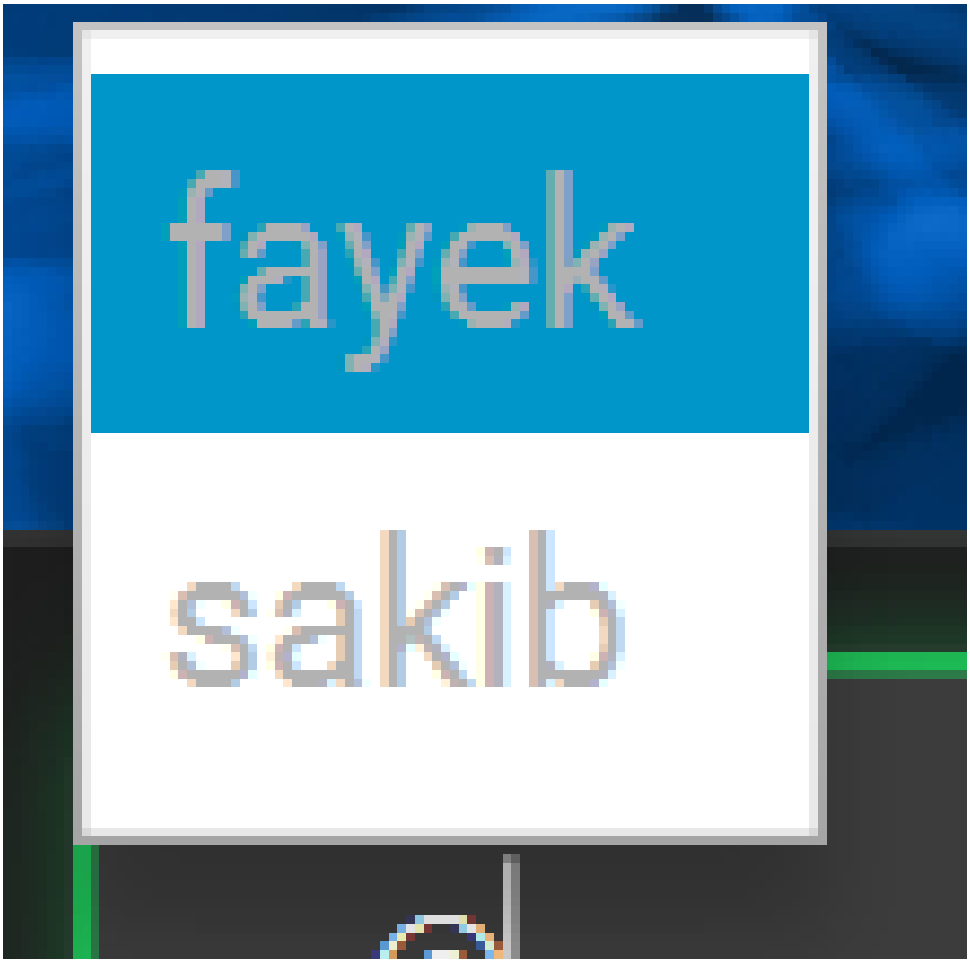

Figure 21: Usernames mention facilities in a group chat

These additional features facilitates user experience in a desktop based chat application.

11 Summary

Bartabahok successfully implements a fault-tolerant chat application with rich features including:

- Reliable message delivery with automatic retransmission
- Multimedia support (files, voice messages)
- Group chat with member management
- Modern UI with reactions and mentions
- SQLite persistence for messages and users
- Builds UI for a message: content, timestamp, and status.
- Displays sender name for group messages.
- Supports rendering file attachments using custom UI.

- Iterates through chat message UI elements.
- Finds the corresponding message by ID.
- Updates its delivery/read status dynamically.

The project demonstrates practical application of networking concepts including TCP socket programming, reliable data transfer, flow control and concurrent connection management.

12 Limitations and Future Plan

12.1 Current Limitations

- Single-server architecture limits horizontal scaling
- Java serialization creates larger payloads than needed
- No end-to-end encryption
- Limited mobile support

12.2 Future Enhancements

- Implement TLS for secure communication
- Develop mobile clients using Flutter
- Add video calling via WebRTC
- Introduce admin controls for groups
- Implement message indexing for better performance