Java Chatting Application

A PROJECT REPORT

Submitted by

Divyansh Doshi (22BCS17108)
Manish Kumar (22BCS15641)

In partial fulfillment for the award of the degree of

BACHELORS OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

Chandigarh University
April 2025

BONAFIDE CERTIFICATE

Certified that this project report **"Java Chat Application"** is the Bonafide work of **"Manish Kumar, Divyansh Doshi"** who carried out the project work under my /our supervision.

SIGNATURE **SIGNATURE**

Head of The Department Supervisor

Computer Science Computer Science

Submitted for the project viva-voce examination held on:

INTERNAL EXAMINER **EXTERNAL EXAMINER**

TABLEOFCONTENTS

List of Figures

PROJECT REPORT: Chat Application Using Java Sockets and Swing

CHAPTER 1. INTRODUCTION

This chapter introduces the context, identifies the need for a custom chat application, outlines the specific problems addressed, details the tasks involved in the project, and provides an estimated timeline for completion.

1.1. Identification of Client/ Need/ Relevant Contemporary Issue

In today's digitally interconnected world, real-time communication is paramount. From casual social interactions to critical business collaborations, the ability to exchange information instantaneously is a fundamental expectation. While numerous sophisticated commercial chat platforms exist (like Slack, Microsoft Teams, WhatsApp, Discord), they often come with complexities, subscription costs, data privacy concerns, or features that are overkill for simpler communication needs.

There exists a persistent need, particularly within educational settings, small organizations, or hobbyist groups, for a straightforward, customizable, and understandable chat application. This need stems from several factors:

1. **Educational Purposes:** Computer science and software engineering students require practical projects to understand fundamental networking concepts (like TCP/IP, sockets, client-server architecture) and GUI development principles (event handling, component layout). Building a chat application provides a tangible and engaging way to learn these core skills. Existing commercial applications are often closed-source or too complex to serve as effective learning tools.
2. **Small Group Communication:** Small teams, clubs, or family groups might desire a simple, private chat environment without the overhead of managing accounts on large platforms or paying fees. A custom application can be deployed on a local network or a private server, offering greater control and simplicity.
3. **Prototyping and Customization:** Developers might need a basic chat framework as a starting point for more specialized communication tools. A simple Java-based solution provides a foundation that can be extended with specific features (e.g., integration with other systems, custom commands, unique interfaces).
4. **Understanding Underlying Technology:** For those interested in how real-time communication works "under the hood," building a chat application from basic principles using core libraries like Java Sockets offers invaluable insight, stripping away the layers of abstraction found in higher-level frameworks.
5. **Platform Independence:** Java's "Write Once, Run Anywhere" philosophy, coupled with Swing for the GUI, allows the application to potentially run on various operating systems (Windows, macOS, Linux) where a Java Runtime Environment (JRE) is available, addressing the need for cross-platform compatibility in diverse environments.

The relevant contemporary issue addressed is the desire for simpler, more transparent, and controllable communication tools in an era dominated by complex, often opaque, commercial platforms. This project caters to the niche requiring a fundamental, educational, and potentially customizable real-time text communication solution.

1.2. Identification of Problem

The core problem this project aims to solve is the lack of a readily available, simple, open, and educational example of a real-time, multi-user chat application built using fundamental Java technologies – specifically Java Sockets for network communication and Java Swing for the graphical user interface.

Existing solutions often fall into these categories, each presenting drawbacks for the target need:

1. **Complex Commercial Platforms:** Over-featured, potentially costly, closed-source, and not suitable for learning networking fundamentals.
2. **High-Level Frameworks/Libraries:** While simplifying development (e.g., using Netty, Java Message Service (JMS), WebSockets with web frameworks), they abstract away the core socket programming concepts, defeating the educational purpose.
3. **Console-Based Examples:** Many basic socket programming tutorials result in command-line applications, which lack the user-friendliness and visual feedback of a GUI, making them less engaging and less representative of typical user applications.
4. **Incomplete or Poorly Documented Examples:** Online tutorials or code snippets might demonstrate parts of the solution but often lack completeness, robustness, clear explanations, or a well-structured GUI implementation.

Therefore, the specific problems being addressed are:

- The difficulty for learners in finding a comprehensive example connecting low-level socket programming with a functional GUI for a practical application like chat.
- The need for a simple, self-contained client-server chat system that can be easily understood, deployed, and potentially modified for specific small-group or educational requirements.
- The absence of a demonstrative project that clearly illustrates handling multiple client connections concurrently using threads on the server-side within a basic Java context.
- The lack of a project that combines core Java networking (`java.net`) and core Java GUI (`javax.swing`) libraries effectively for this purpose, without relying on external dependencies or complex frameworks.

This project directly tackles these issues by creating a functional, documented, and relatively simple chat application using only core Java SE libraries.

1.3. Identification of Tasks

To develop the Java Socket and Swing chat application, the project is broken down into the following distinct tasks:

1. **Requirement Analysis:** Define the specific functional (e.g., sending/receiving messages, user list) and non-functional (e.g., usability, basic error handling) requirements for the chat application.
2. **System Design:**
    - Define the overall client-server architecture.
    - Design the communication protocol between the client and server (e.g., message formats, commands for connect/disconnect/message).
    - Design the server structure: main listener, client handler threads, user management.

    - Design the client structure: connection logic, GUI layout, message sending/receiving logic, GUI update mechanism.
    - Select appropriate Java Swing components for the client GUI.
3. **Server-Side Development:**
    - Implement the main server class to listen for incoming connections on a specific port (`ServerSocket`).
    - Implement the client handler mechanism, likely using threads, to manage communication with each connected client independently.
    - Implement logic within the client handler to read messages from the client (`InputStream`) and broadcast messages to other clients (`OutputStream`).
    - Implement user management (tracking connected clients, updating user lists).
    - Implement basic error handling (e.g., client disconnections).
4. **Client-Side Development:**
    - Implement the client connection logic (`Socket`) to connect to the server's IP address and port.
    - Design and implement the Swing GUI (`JFrame`, `JTextArea`, `JTextField`, `JButton`, `JList`, Layout Managers).
    - Implement event handling (`ActionListener`) for sending messages when a button is clicked or Enter is pressed in the input field.
    - Implement logic to send messages to the server (`OutputStream`).
    - Implement a mechanism (typically a separate thread) to continuously listen for incoming messages from the server (`InputStream`).
    - Implement logic to update the GUI (e.g., display received messages in the `JTextArea`, update the `JList` of users) safely from the receiving thread (using `SwingUtilities.invokeLater`).
    - Implement basic error handling (e.g., server unavailable, disconnection).
5. **Integration and Testing:**
    - Test server functionality independently (e.g., accepting connections).
    - Test client functionality independently (e.g., GUI layout, sending attempts).
    - Integrate client and server and test the end-to-end communication flow.
    - Test with multiple clients concurrently to ensure threading and message broadcasting work correctly.
    - Perform basic usability testing on the client GUI.
    - Identify and fix bugs discovered during testing.
6. **Documentation:**
    - Add comments to the code explaining key sections.
    - Prepare a final report (like this one) detailing the project's design, implementation, and findings.
7. **Packaging (Optional):** Create executable JAR files for the server and client for easier deployment.

1.4. Timeline

This is an estimated timeline for a single developer with intermediate Java experience. Durations are indicative and can vary based on complexity and experience.

- **Week 1:**
    - Requirement Analysis & Planning (1 day)
    - System Design (Architecture, Protocol, GUI Mockups) (2 days)
    - Initial Server Setup (ServerSocket, basic connection acceptance) (2 days)

- **Week 2:**
  - Server Client Handling (Threading, basic I/O) (3 days)
  - Server User Management & Broadcasting (2 days)
- **Week 3:**
  - Initial Client GUI Design & Layout (Swing components) (3 days)
  - Client Connection Logic & Basic Sending (2 days)
- **Week 4:**
  - Client Message Receiving Thread (2 days)
  - Client GUI Updates (Safe threading with `invokeLater`) (2 days)
  - Initial Integration Testing (1 day)
- **Week 5:**
  - Multi-Client Testing & Debugging (3 days)
  - Error Handling Implementation (Client & Server) (2 days)
- **Week 6:**
  - Refinement & Code Cleanup (2 days)
  - Documentation & Report Writing (3 days)
- **Week 7 (Optional):**
  - Packaging into JAR files (1 day)
  - Final Review & Buffer (4 days)

Total Estimated Time: 6-7 Weeks.

CHAPTER 2. LITERATURE REVIEW/BACKGROUND STUDY

This chapter explores the historical context of chat applications, examines existing solutions, discusses the concept of bibliometric analysis in this context, reviews the core technologies (Sockets, Swing, Threads), refines the problem definition, and sets clear goals for the project.

2.1. Timeline of the reported problem

The fundamental problem – needing effective real-time text communication – is nearly as old as networked computers themselves. The evolution relevant to this project includes:

- **Early Days (1960s-1970s):** Precursors like messaging on time-sharing systems (e.g., CTSS .MAIL). Early network protocols like ARPANET allowed basic file transfer and remote login, laying the groundwork.
- **IRC (Internet Relay Chat) (Late 1980s):** One of the earliest, widely adopted, text-based, multi-user chat protocols. It established many conventions (channels, nicknames, server-client model) still seen today. It highlighted the need for dedicated server infrastructure and client software.
- **Instant Messengers (Mid-1990s - 2000s):** Rise of ICQ, AOL Instant Messenger (AIM), MSN Messenger, Yahoo! Messenger. These introduced user-friendly GUIs, presence indicators (online/offline status), buddy lists, and one-on-one conversations, moving beyond the channel-centric IRC model. These were typically proprietary, centralized systems.
- **Web-Based Chat (Late 1990s - Present):** Chat integrated into websites (chat rooms, support widgets). Early versions used technologies like Java Applets or Flash. Later, AJAX, WebSockets, and frameworks like Node.js enabled more sophisticated browser-based real-time communication.

- **VoIP and Rich Communication (Early 2000s - Present):** Skype popularized Voice over IP combined with text chat and file transfer. Modern platforms (WhatsApp, Telegram, Signal, Discord, Slack, Teams) integrate text, voice, video, file sharing, screen sharing, bots, and extensive integrations, often using sophisticated client-server architectures and protocols (sometimes proprietary, sometimes standards-based like XMPP initially, or WebSockets).
- **Rise of Mobile (Late 2000s - Present):** Smartphones fueled the explosion of mobile-first chat apps like WhatsApp, fundamentally changing how people communicate daily. These rely heavily on efficient mobile networking and push notifications.

Throughout this evolution, the specific niche problem addressed by *this* project – the need for a *simple, understandable, Java-based socket/Swing chat application for educational or basic use* – persists because:

1. Modern platforms are vastly more complex, obscuring the fundamentals.
2. Earlier simple examples often used outdated technologies (e.g., Applets) or were console-based.
3. The focus shifted towards web and mobile, leaving fewer well-documented, modern examples of basic desktop client-server chat using core Java libraries.

2.2. Existing solutions

Numerous chat solutions exist, varying widely in complexity, features, and underlying technology.

- **Commercial/SaaS Platforms:**
  - *Examples:* Slack, Microsoft Teams, Discord, WhatsApp, Telegram, Google Chat.
  - *Pros:* Feature-rich (persistent history, search, integrations, voice/video, mobile apps), scalable, reliable (usually), managed infrastructure.
  - *Cons:* Often costly (for business features), proprietary, potential data privacy concerns, complex architecture, not suitable for learning socket programming fundamentals.
- **Open Source Platforms:**
  - *Examples:* Mattermost, Rocket.Chat (often self-hosted alternatives to Slack), Matrix (federated protocol) with clients like Element, XMPP (protocol) with various servers (e.g., Openfire, Prosody) and clients (e.g., Gajim, Pidgin).
  - *Pros:* Often self-hostable (more control), customizable, potentially free, adhere to open standards (XMPP, Matrix).
  - *Cons:* Can be complex to set up and manage, codebase might be large and difficult to grasp for beginners, may use higher-level frameworks or different languages/protocols.
- **Specific Libraries/Frameworks:**
  - *Examples:* Netty (asynchronous networking framework), Java NIO (non-blocking I/O), WebSockets (via libraries like Java-WebSocket or frameworks like Spring), Java Message Service (JMS) (for messaging middleware).
  - *Pros:* Provide powerful abstractions, improve performance and scalability (NIO, Netty), standardized APIs (JMS, WebSockets).
  - *Cons:* Abstract away the direct socket handling, adding a layer of complexity that might hinder learning the basics. Require learning the specific framework/API.
- **Simple Socket/GUI Examples (Online Tutorials/Repositories):**
  - *Examples:* Various GitHub repositories, blog posts, tutorial sites (e.g., GeeksforGeeks, Baeldung, Stack Overflow snippets).
  - *Pros:* Often directly address the problem, use basic sockets and sometimes Swing/JavaFX. Can be good starting points.

- *Cons:* Vary hugely in quality, completeness, and correctness. May lack features (e.g., proper multi-client handling, GUI updates), contain bugs, have poor error handling, or lack clear explanations. Few combine sockets and a decent Swing GUI robustly.

This project differentiates itself by aiming for a sweet spot: more complete and robust than typical basic tutorial snippets, but far simpler and more focused on core Java SE concepts (Sockets, Threads, Swing) than commercial or complex open-source platforms or high-level frameworks.

2.3. Bibliometric analysis

A formal bibliometric analysis for this specific project (a relatively standard educational software development task) is generally not performed unless it's part of academic research comparing different implementation techniques or analyzing trends in network programming education. However, if one were to conduct such an analysis, it would involve:

1. **Defining Scope:** Focusing on academic papers, conference proceedings, and potentially technical reports related to network programming, socket programming in Java, GUI development with Swing, and chat application design.
2. **Keyword Search:** Using databases like IEEE Xplore, ACM Digital Library, Scopus, Google Scholar with keywords such as: "Java socket programming," "client-server application," "Java Swing GUI," "network programming education," "chat application architecture," "multi-threaded server Java."
3. **Data Collection:** Gathering metadata for relevant publications: authors, publication year, journal/conference, citations, abstracts, keywords.
4. **Analysis:**
    - **Publication Trends:** Analyzing the number of publications over time to see peaks in interest (e.g., likely higher interest in basic socket programming in the late 90s/early 2000s, potentially less focus in recent academic research compared to newer technologies like WebSockets or gRPC).
    - **Citation Analysis:** Identifying highly cited papers, which might represent seminal works on socket programming techniques, threading models for servers, or GUI design patterns.
    - **Keyword Co-occurrence:** Analyzing which keywords frequently appear together (e.g., "socket" and "thread," "Swing" and "event handling") to understand common technological pairings and research topics.
    - **Author/Institution Analysis:** Identifying key researchers or institutions contributing to this area.
    - **Technology Evolution:** Tracking how the focus shifts from basic sockets/threads to NIO, asynchronous frameworks (Netty), or different communication paradigms over the years.

Expected Findings (Hypothetical): Such an analysis would likely confirm that while foundational, basic TCP socket programming with a thread-per-client model is a well-established technique, often covered in introductory networking courses. Research focus has largely shifted to scalability (NIO, asynchronous models), security, different protocols (WebSockets, QUIC), and framework-based development. Swing, while still functional, might show declining relevance in new research compared to web or mobile UI technologies or JavaFX. However, the combination remains relevant for educational purposes due to its direct illustration of core concepts.

2.4. Review

This section reviews the core technologies and concepts underpinning the project:

- **Java Sockets (`java.net` package):**
  - **Concept:** Sockets are endpoints for network communication. Java provides classes to abstract the underlying network protocols (primarily TCP and UDP).
  - **TCP (Transmission Control Protocol):** Used for this chat application. It's a connection-oriented, reliable, stream-based protocol. It guarantees that data arrives in order and without errors (or notifies if errors occur).
  - **`ServerSocket`:** Used on the server side. It listens on a specific network port for incoming connection requests from clients. The `accept()` method blocks until a client connects, then returns a `Socket` object representing that specific connection.
  - **`Socket`:** Used by both the client (to initiate a connection to a server) and the server (as returned by `accept()` to communicate with a specific client). It provides access to input and output streams for sending and receiving data.
  - **`InputStream` / `OutputStream`:** Obtained from a `Socket` object (`getInputStream()`, `getOutputStream()`). Used to read byte streams from and write byte streams to the network connection. Often wrapped in higher-level readers/writers (like `BufferedReader`, `PrintWriter`, `ObjectInputStream`, `ObjectOutputStream`) for easier handling of text or objects.
- **Java Swing (`javax.swing` package):**
  - **Concept:** A platform-independent GUI toolkit for Java. It provides a rich set of components (buttons, text areas, lists, menus, etc.) for building desktop application interfaces.
  - **Components:** `JFrame` (top-level window), `JPanel` (container for other components), `JTextArea` (multi-line text display), `JTextField` (single-line text input), `JButton` (clickable button), `JList` (display a list of items), `JScrollPane` (adds scrollbars to components like `JTextArea` or `JList`).
  - **Layout Managers:** Control how components are arranged within containers (e.g., `BorderLayout`, `FlowLayout`, `GridLayout`, `GridBagLayout`). Essential for creating organized and resizeable GUIs.
  - **Event Handling:** Swing is event-driven. User actions (clicking buttons, typing text) generate events. `ActionListener`, `KeyListener`, `MouseListener`, etc., are interfaces used to implement code that responds to these events.
  - **Concurrency (Event Dispatch Thread - EDT):** Swing components are generally *not* thread-safe. All interactions with Swing components (creating, modifying, querying) *must* happen on a single, special thread called the Event Dispatch Thread (EDT). Network operations or other long-running tasks must be performed on separate background threads. To update the GUI from a background thread (like the one receiving network messages), `SwingUtilities.invokeLater()` or `SwingUtilities.invokeAndWait()` must be used to schedule the GUI update code for execution on the EDT. Failure to do this can lead to graphical glitches, freezes, or race conditions.
- **Java Threads (`java.lang.Thread`, `java.lang.Runnable`):**
  - **Concept:** Allow concurrent execution of different parts of a program. Essential for network servers to handle multiple clients simultaneously without blocking each other. Also used in clients to handle network I/O separately from the GUI thread.

- **Server Model (Thread-per-Client):** A common (though not the most scalable) approach. The main server thread listens for connections (`ServerSocket.accept()`). When a client connects, the server creates a new `Thread` (often by passing a `Runnable` client handler object) dedicated to managing communication with that specific client. This allows the main thread to immediately go back to listening for new connections.
- **Client Model:** The main thread typically handles the GUI and user input (running on the EDT). A separate background thread is needed to continuously listen for incoming messages from the server, preventing the GUI from freezing while waiting for network data. When this background thread receives a message, it uses `SwingUtilities.invokeLater()` to update the `JTextArea` or other GUI components on the EDT.

2.5. Problem Definition (Refined)

Based on the review, the problem is refined as: To design, implement, and document a functional, multi-user, real-time chat application featuring a server and multiple clients, utilizing core Java technologies. Specifically, the application will employ Java TCP Sockets (`java.net`) for reliable client-server network communication and Java Swing (`javax.swing`) for creating a graphical user interface on the client-side. The server must handle multiple simultaneous client connections concurrently using a thread-per-client model. The client GUI must provide basic chat functionalities (displaying messages, user list, input field, send button) and handle network reception without blocking the user interface, correctly using Swing's concurrency model (`SwingUtilities.invokeLater`). The project serves as a practical, educational example demonstrating the integration of these core Java technologies for networked applications.

2.6. Goals/Objectives

The primary goals and objectives of this project are:

1. **Develop a Functional Server:** Create a Java server application capable of:
    - Listening for incoming TCP connections on a specified port.
    - Accepting multiple client connections concurrently.
    - Managing connections using a separate thread for each client.
    - Receiving messages from any client.
    - Broadcasting received messages to all other connected clients.
    - Maintaining and distributing a list of currently connected users.
    - Handling client disconnections gracefully.
2. **Develop a Functional Client:** Create a Java Swing client application capable of:
    - Connecting to the server at a specified IP address and port.
    - Providing a graphical user interface with:
        - A display area for incoming messages.
        - An input field for typing messages.
        - A button or action to send messages.
        - A display area for the list of connected users.
    - Sending user-typed messages to the server.
    - Receiving broadcasted messages and user list updates from the server asynchronously (without freezing the GUI).
    - Displaying received messages and updating the user list in the GUI correctly (using the EDT).

    - Allowing the user to disconnect cleanly.
3. **Ensure Basic Robustness:** Implement basic error handling for common issues like connection failures, server unavailability, and abrupt client disconnections.
4. **Demonstrate Core Concepts:** Clearly illustrate the use of `ServerSocket`, `Socket`, `InputStream`/`OutputStream`, Threads for concurrency, Swing components, Layout Managers, Event Handling, and Swing's threading model (`invokeLater`).
5. **Provide Clear Documentation:** Document the code with comments and produce a comprehensive report detailing the design, implementation, and results.
6. **Maintain Simplicity:** Avoid external libraries (beyond core Java SE) and overly complex features to keep the focus on the fundamental technologies.

These objectives define the scope and success criteria for the project.

CHAPTER 3. PROPOSED METHODOLOGY

This chapter details the step-by-step methodology used to develop the Java Socket and Swing chat application, covering requirement analysis, system design, development phases, testing strategies, deployment considerations, and maintenance plans, along with illustrative code snippets.

3.1 Requirement Analysis and Planning

Before coding, the specific requirements were defined:

- **Functional Requirements:**
    - **FR1: Server Connection:** The server must listen on a user-specified (or default) port.
    - **FR2: Client Connection:** Clients must be able to connect to the server using its IP address and port.
    - **FR3: User Identification:** Upon connection, the client should prompt the user for a username, which is sent to the server. Usernames should ideally be unique (though strict enforcement might be a V2 feature).
    - **FR4: Message Sending:** Clients must be able to type messages into an input field and send them to the server.
    - **FR5: Message Broadcasting:** The server must receive messages from one client and broadcast them to all *other* currently connected clients. (Optionally: broadcast to self as well for confirmation).
    - **FR6: Message Display:** Clients must display received broadcast messages in a chronological scrolling text area.
    - **FR7: User List:** The server must maintain a list of connected usernames.
    - **FR8: User List Display:** Clients must receive and display the current list of connected users, updating it as users join or leave.
    - **FR9: Client Disconnection:** Clients must have a way to cleanly disconnect (e.g., closing the window, a disconnect button).
    - **FR10: Server Handling Disconnection:** The server must detect client disconnections (graceful or abrupt) and remove the user from the list, notifying other clients.
- **Non-Functional Requirements:**
    - **NFR1: Usability:** The client GUI should be simple and intuitive to use for basic chatting.

    - **NFR2: Concurrency:** The server must handle multiple clients simultaneously without significant delays in message broadcasting for a small number of users (e.g., < 20).
    - **NFR3: Responsiveness:** The client GUI must remain responsive during network activity (message receiving).
    - **NFR4: Platform:** The application should run on any OS with a compatible Java Runtime Environment (JRE) installed.
    - **NFR5: Simplicity:** The implementation should rely primarily on core Java SE libraries (`java.net`, `javax.swing`, `java.lang`).
    - **NFR6: Basic Error Handling:** The application should handle common errors like invalid server address, server not running, or unexpected disconnections without crashing completely, providing informative messages where possible.
- **Planning:** The development was planned following the task breakdown in Chapter 1.3, prioritizing server setup, then client connection and GUI, followed by message handling and user list synchronization.

3.2 System Design and Architecture

- **Architecture:** A classic Client-Server architecture was chosen.
    - **Server:** A single Java application acting as the central hub. It listens for connections, manages clients, and relays messages.
    - **Client:** Multiple instances of a Java Swing application, each representing a user. Clients connect to the server to send and receive messages.
- **Communication Protocol:** A simple text-based protocol over TCP was designed. Messages exchanged between client and server are strings with a prefix indicating the message type.
    - `CONNECT <username>`: Sent by client upon connection.
    - `MESSAGE <sender_username> <message_body>`: Sent by server to clients to broadcast a chat message. (Alternatively, client sends `SEND <message_body>` and server adds sender info before broadcasting). Let's use the latter for simplicity on the client side. Client sends `MESSAGE <message_body>`. Server receives, identifies sender from the socket/handler, and broadcasts `MESSAGE <sender_username> <message_body>`.
    - `USERLIST <user1>,<user2>,...`: Sent by server to clients to update the user list.
    - `JOINED <username>`: Sent by server to all clients when a new user connects.
    - `LEFT <username>`: Sent by server to all clients when a user disconnects.
    - `DISCONNECT`: Sent by client when gracefully disconnecting.
- **Server Design:**
    - **`Server` Class:** Contains the `main` method. Creates a `ServerSocket`. Enters an infinite loop calling `accept()`. For each new `Socket` returned, it creates a `ClientHandler` thread and starts it.
    - **`ClientHandler` Class (implements `Runnable`):** Each instance handles one client.
        - Holds the client `Socket`, `BufferedReader` (for input), `PrintWriter` (for output).
        - Stores the client's username.
        - Contains the `run()` method: Reads the initial username. Adds user to a shared list (synchronized access). Broadcasts join notification and initial user list. Enters a loop reading messages from the client.
        - If a message is received, broadcasts it to other clients via the `Server` class.
        - If `DISCONNECT` is received or an `IOException` occurs (client disconnected), it cleans up resources, removes the user from the shared list (synchronized), broadcasts a leave notification and updated user list, and terminates the thread.

    - **Shared State:** The `Server` class (or a dedicated manager class) needs to maintain a thread-safe collection (e.g., `Collections.synchronizedSet` or `ConcurrentHashMap`) of active `ClientHandler` instances or their `PrintWriter` objects to facilitate broadcasting. Methods for adding/removing users and broadcasting messages must use synchronization (`synchronized` blocks/methods) to prevent race conditions.
- **Client Design:**
    - **`Client` Class:** Contains the `main` method. Prompts for server IP/port and username (can be GUI dialogs). Creates the `ClientGUI`. Attempts to establish a `Socket` connection. If successful, creates input/output streams and starts a message-receiving thread. Passes necessary objects (streams, username) to the GUI.
    - **`ClientGUI` Class (extends `JFrame`):** Builds the Swing interface (using `JTextArea`, `JTextField`, `JButton`, `JList`, `JScrollPane`, `JPanel`, Layout Managers).
        - Stores references to the `PrintWriter` (for sending) and potentially the `Socket` or main `Client` object.
        - Attaches `ActionListener` to the send button and input field. When triggered, it reads text from `JTextField`, sends it to the server via `PrintWriter`, and clears the input field.
        - Provides methods like `displayMessage(String msg)` and `updateUserList(String[] users)` that will be called by the receiving thread. These methods *must* use `SwingUtilities.invokeLater` to append text to `JTextArea` or update the `JList` model safely on the EDT.
        - Handles window closing event to send a `DISCONNECT` message to the server before exiting.
    - **`MessageReceiver` Class (implements `Runnable`):** Runs on a separate thread.
        - Holds the `BufferedReader` for the client's socket and a reference to the `ClientGUI`.
        - Enters a loop reading lines from the server (`readLine()`).
        - Parses incoming messages based on the protocol prefixes (`MESSAGE`, `USERLIST`, `JOINED`, `LEFT`).
        - Calls the appropriate `ClientGUI` methods (using `SwingUtilities.invokeLater`) to update the display.
        - Handles `IOException` (server disconnection) by informing the user (e.g., displaying a message on the GUI) and potentially disabling input/send functionality.

3.3 Development

The development followed the design, implementing classes and methods step-by-step.

- **Server Implementation:**
    - Created `Server.java` with `ServerSocket` setup and the main `accept()` loop.
    - Used a `Set<PrintWriter>` or similar thread-safe collection to store output streams of connected clients for broadcasting. Used `Collections.synchronizedSet` initially for simplicity.
    - Implemented `ClientHandler.java` (`Runnable`).
    - Used `BufferedReader` (wrapping `InputStreamReader` wrapping `socket.getInputStream()`) and `PrintWriter` (wrapping `socket.getOutputStream()`, with `autoFlush` set to `true`) for text communication.

    - Implemented the `run()` method logic: read username, add to shared list, broadcast join/userlist, enter message read loop.
    - Implemented broadcasting methods in the `Server` class (or accessible to `ClientHandler`), ensuring they iterated over the synchronized collection of writers. Included checks to avoid sending messages back to the original sender if desired.
    - Added `try-catch` blocks for `IOException` during I/O operations and connection acceptance.
    - Added `finally` blocks in `ClientHandler` to ensure resources (socket, streams) are closed and the user is removed from the shared list upon disconnection or error. Used synchronized blocks for adding/removing users and broadcasting.
- **Client Implementation:**
    - Created `Client.java` for connection logic and launching the GUI.
    - Created `ClientGUI.java` extending `JFrame`. Used `BorderLayout` as the main layout. Placed `JTextArea` (in a `JScrollPane`) in the center, `JTextField` and `JButton` in a `JPanel` at the bottom (`SOUTH`), and `JList` (in a `JScrollPane`) on the right (`EAST`).
    - Implemented `ActionListener` for the send button and `JTextField`. The action reads the text, sends `MESSAGE <text>` using the `PrintWriter`, and clears the field.
    - Implemented `MessageReceiver.java` (`Runnable`). The `run()` method continuously reads lines from the `BufferedReader`.
    - Implemented parsing logic in `MessageReceiver` to handle different message prefixes from the server.
    - Crucially, all calls from `MessageReceiver` that modify Swing components (e.g., `textArea.append()`, updating `JList` model) were wrapped in `SwingUtilities.invokeLater(() -> { ... });`.
    - Used a `DefaultListModel` for the `JList` to allow easy dynamic updates.
    - Added a window listener (`WindowAdapter`) to the `JFrame` to detect the window closing event (`windowClosing`). In this event handler, send the `DISCONNECT` message to the server and then call `System.exit(0)`.
    - Added basic error dialogs (`JOptionPane`) for connection failures or server disconnections.

3.4 Testing

A multi-stage testing approach was used:

1. **Unit Testing (Conceptual):** While formal JUnit tests weren't mandated for this scope, the concept was applied by testing individual components mentally or with simple drivers. For example, testing the protocol parsing logic with sample strings.
2. **Server Testing:**
    - Ran the server application.
    - Used a simple tool like `telnet` or `nc` (netcat) to connect to the server port.
    - Verified that the server accepted the connection.
    - Sent mock protocol messages (`CONNECT`, etc.) via telnet/nc to check server responses.
    - Checked server console output for logs/errors.
3. **Client GUI Testing:**
    - Ran the client application *without* connecting to a server.
    - Checked the layout, component responsiveness (typing, button clicks).
    - Manually called GUI update methods (like `displayMessage`) to ensure they worked visually (though this doesn't test the threading aspect).

4. **Integration Testing:**
    - Ran the server application.
    - Ran one instance of the client application. Verified connection, username prompt, initial user list display. Sent messages and checked if they appeared (often echoed back by simple server logic initially). Verified disconnection.
    - Ran *multiple* instances of the client application (e.g., 3-4 clients).
    - Checked if all clients connect successfully and receive the updated user list.
    - Sent messages from one client and verified they appear on *all other* clients but not the sender (or on all including sender, depending on design choice).
    - Checked if user list updates correctly when a client joins or leaves.
    - Tested abrupt disconnection (e.g., killing a client process) to see if the server handles it and updates other clients.
    - Tested connection refusal (server not running) and server crashing/stopping during a session.
5. **Usability Testing:** Performed basic walkthroughs of the client GUI to ensure the flow was logical and controls were accessible.
6. **Concurrency Testing:** Focused on sending messages rapidly from multiple clients and observing if messages were interleaved correctly and if the server/clients remained stable. Looked for signs of race conditions (e.g., incorrect user lists, mangled messages – though less likely with simple text broadcasting and proper synchronization).

Bugs found during testing (e.g., GUI freezing, incorrect user list updates, exceptions on disconnection) were iteratively fixed by refining thread synchronization, ensuring correct use of `SwingUtilities.invokeLater`, and adding more robust error handling.

3.5 Deployment

Deployment for this type of application is typically straightforward:

1. **Prerequisites:** Ensure the target machines (server and clients) have a compatible Java Runtime Environment (JRE) installed.
2. **Packaging:** Compile the Java source files (`.java`) into bytecode (`.class`). Package the server classes and client classes into separate executable JAR files.
    - This involves creating Manifest files (`MANIFEST.MF`) within the JARs specifying the `Main-Class` attribute (e.g., `Main-Class: com.chat.server.Server` for the server JAR, `Main-Class: com.chat.client.Client` for the client JAR).
    - Build tools like Maven or Gradle can automate this, but it can also be done using the `jar` command-line tool included with the JDK.
3. **Distribution:** Distribute the server JAR (`server-chat.jar`) to the machine designated as the server and the client JAR (`client-chat.jar`) to all user machines.
4. **Execution:**
    - **Server:** Open a terminal or command prompt on the server machine, navigate to the directory containing the JAR, and run: `java -jar server-chat.jar [port]` (port argument optional if a default is coded).
    - **Client:** Open a terminal or command prompt on a client machine, navigate to the directory containing the JAR, and run: `java -jar client-chat.jar`. The client application will then typically prompt for the server's IP address and port.

3.6 Maintenance and Updates

Post-deployment, maintenance involves:

- **Bug Fixing:** Addressing any issues reported by users or discovered later (e.g., unexpected disconnections, GUI glitches under specific conditions, minor protocol mismatches).
- **Feature Enhancements:** Based on feedback or evolving needs, future updates could include features listed in Chapter 5.2 (e.g., private messages, file transfer, encryption). This would involve repeating the requirement-design-develop-test cycle for the new features.
- **JRE Compatibility:** Ensuring the application remains compatible with newer JRE versions, although core Java Sockets and Swing are relatively stable APIs.
- **Documentation Updates:** Keeping documentation (code comments, user guides if any) updated with changes.

3.7 Code

Below are illustrative, simplified code snippets demonstrating key parts of the implementation. *Note: These are conceptual snippets and omit imports, full error handling, and potentially some helper methods for brevity.*

Server.java (Main loop and broadcast):

Java

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.concurrent.*;

public class Server {
    private int port;
    private Set<ClientHandler> clientHandlers = new CopyOnWriteArraySet<>();
// Thread-safe set
    private Set<String> userNames = new CopyOnWriteArraySet<>();

    public Server(int port) {
        this.port = port;
    }

    public void start() {
        System.out.println("Chat Server started on port: " + port);
        try (ServerSocket serverSocket = new ServerSocket(port)) {
            while (true) {
                try {
                    Socket socket = serverSocket.accept(); // Wait for client
connection
                    System.out.println("New connection from " +
socket.getRemoteSocketAddress());
                    ClientHandler clientThread = new ClientHandler(socket,
this);
                    clientHandlers.add(clientThread);
                    new Thread(clientThread).start();
                } catch (IOException e) {
                    System.err.println("Error accepting client connection: "
+ e.getMessage());
                }
```

```
            }
        } catch (IOException e) {
            System.err.println("Could not listen on port " + port + ": " +
e.getMessage());
        }
    }

    // Broadcast message to all clients (or all except sender)
    void broadcast(String message, ClientHandler excludeUser) {
        for (ClientHandler client : clientHandlers) {
            if (client != excludeUser) {
                client.sendMessage(message);
            }
        }
    }

    // Add user and notify others
    void addUser(String userName, ClientHandler handler) {
        if (userNames.add(userName)) {
             broadcast("JOINED " + userName, null); // Notify all
             updateUserLists(); // Send updated list to everyone
        } else {
             // Handle duplicate username - maybe reject connection or ask
for new name
             handler.sendMessage("ERROR Username " + userName + " already
taken.");
             removeUser(null, handler); // Disconnect this handler
        }
    }

    // Remove user and notify others
    void removeUser(String userName, ClientHandler handler) {
        boolean removed = clientHandlers.remove(handler);
        if (userName != null) { // If username was successfully set before
disconnect
             userNames.remove(userName);
             if (removed) {
                 broadcast("LEFT " + userName, null); // Notify remaining
users
                 updateUserLists(); // Send updated list
             }
        }
        try { handler.getSocket().close(); } catch (IOException e) {} //
Ensure socket is closed
        System.out.println("User disconnected: " + (userName != null ?
userName : "Unknown"));
    }

    // Send updated user list to all clients
    void updateUserLists() {
        if (!userNames.isEmpty()) {
            String userListMsg = "USERLIST " + String.join(",", userNames);
            broadcast(userListMsg, null);
        } else {
             broadcast("USERLIST ", null); // Send empty list if no users
        }
    }
```

```
     Set<String> getUserNames() {
        return this.userNames;
     }

    public static void main(String[] args) {
        int port = 5000; // Default port
        if (args.length > 0) {
            try {
                port = Integer.parseInt(args[0]);
            } catch (NumberFormatException e) {
                System.err.println("Invalid port number. Using default
5000.");
            }
        }
        Server server = new Server(port);
        server.start();
    }
}
```

ClientHandler.java (Server-side thread for one client):

Java

```
import java.io.*;
import java.net.*;

class ClientHandler implements Runnable {
    private Socket socket;
    private Server server;
    private PrintWriter writer;
    private BufferedReader reader;
    private String userName;

    public ClientHandler(Socket socket, Server server) {
        this.socket = socket;
        this.server = server;
        try {
            InputStream input = socket.getInputStream();
            reader = new BufferedReader(new InputStreamReader(input));
            OutputStream output = socket.getOutputStream();
            writer = new PrintWriter(output, true); // true for autoFlush
        } catch (IOException e) {
            System.err.println("Error creating client handler streams: " +
e.getMessage());
            try { socket.close(); } catch (IOException ex) {}
        }
    }

    public void run() {
        try {
            // First message should be CONNECT <username> (Simplified: just
read username)
            // In a real app, add protocol check: String line =
reader.readLine(); if (line.startsWith("CONNECT ")) ...
            this.userName = reader.readLine(); // Simplistic: assume first
line is username
```

```
            if (this.userName == null || this.userName.trim().isEmpty() ||
this.userName.contains(",")) {
                 System.err.println("Invalid username received.");
                 sendMessage("ERROR Invalid username.");
                 server.removeUser(null, this); // Disconnect invalid user
                 return;
            }

            server.addUser(this.userName, this); // Add user, broadcast join
& userlist

            String serverMessage;
            String clientMessage;

            while ((clientMessage = reader.readLine()) != null) {
                if ("DISCONNECT".equals(clientMessage)) {
                    break; // Graceful disconnect
                }
                // Assume any other line is a message to broadcast
                serverMessage = "MESSAGE " + userName + ": " + clientMessage;
                server.broadcast(serverMessage, this); // Broadcast to others
                // Optional: send confirmation back to sender
                // sendMessage("MESSAGE You: " + clientMessage);
            }

        } catch (IOException e) {
            // Handle abrupt disconnection
            System.err.println("Error handling client " + (userName != null ?
userName : "") + ": " + e.getMessage());
        } finally {
            server.removeUser(userName, this); // Ensures cleanup happens
        }
    }

    void sendMessage(String message) {
         if (writer != null) {
             writer.println(message);
         }
    }

    Socket getSocket() { return socket; }
    String getUserName() { return userName; }
}
```

ClientGUI.java (Relevant Swing parts and updating):

Java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class ClientGUI extends JFrame {
    private JTextArea messageArea;
    private JTextField inputField;
    private JButton sendButton;
```

```
    private JList<String> userList;
    private DefaultListModel<String> userListModel;
    private PrintWriter writer;
    private Client clientApp; // Reference back to main client logic if
needed

    public ClientGUI(PrintWriter writer, Client clientApp) {
        super("Simple Chat Client");
        this.writer = writer;
        this.clientApp = clientApp;

        // --- Component Initialization ---
        messageArea = new JTextArea();
        messageArea.setEditable(false);
        JScrollPane messageScrollPane = new JScrollPane(messageArea);

        inputField = new JTextField();
        sendButton = new JButton("Send");

        userListModel = new DefaultListModel<>();
        userList = new JList<>(userListModel);
        JScrollPane userListScrollPane = new JScrollPane(userList);
        userListScrollPane.setPreferredSize(new Dimension(100, 0)); // Set
preferred width

        // --- Layout ---
        JPanel bottomPanel = new JPanel(new BorderLayout());
        bottomPanel.add(inputField, BorderLayout.CENTER);
        bottomPanel.add(sendButton, BorderLayout.EAST);

        setLayout(new BorderLayout(5, 5)); // Gaps between components
        add(messageScrollPane, BorderLayout.CENTER);
        add(userListScrollPane, BorderLayout.EAST);
        add(bottomPanel, BorderLayout.SOUTH);

        // --- Event Listeners ---
        ActionListener sendListener = e -> sendMessage();
        sendButton.addActionListener(sendListener);
        inputField.addActionListener(sendListener); // Send on Enter key

        // Window Closing Event
        addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                if (writer != null) {
                    writer.println("DISCONNECT");
                }
                clientApp.shutdown(); // Call a method in Client to close
socket etc.
            }
        });

        // --- Final Setup ---
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);    //    Changed    to
EXIT_ON_CLOSE assuming shutdown handles disconnect
        setSize(600, 400);
        setLocationRelativeTo(null); // Center window
```

```
        // setVisible(true); // Visibility should be set after connection
attempt in Client.java
    }

    private void sendMessage() {
        String text = inputField.getText();
        if (text != null && !text.trim().isEmpty() && writer != null) {
            writer.println(text); // Server adds protocol/sender info
            inputField.setText(""); // Clear input field
        }
    }

    // Method called by MessageReceiver thread
    public void displayMessage(String message) {
        // Ensure GUI update happens on the Event Dispatch Thread
        SwingUtilities.invokeLater(() -> {
            messageArea.append(message + "\n");
            // Auto-scroll to bottom

messageArea.setCaretPosition(messageArea.getDocument().getLength());
        });
    }

    // Method called by MessageReceiver thread
    public void updateUserList(String[] users) {
        // Ensure GUI update happens on the Event Dispatch Thread
        SwingUtilities.invokeLater(() -> {
            userListModel.clear();
            for (String user : users) {
                if (!user.trim().isEmpty()) {
                    userListModel.addElement(user);
                }
            }
        });
    }

     // Method to show connection error
     public void showConnectionError(String message) {
          SwingUtilities.invokeLater(() -> {
              JOptionPane.showMessageDialog(this, message, "Connection Error",
JOptionPane.ERROR_MESSAGE);
              inputField.setEnabled(false); // Disable input if disconnected
              sendButton.setEnabled(false);
          });
     }
}
```

MessageReceiver.java (Client-side listening thread):

Java

```
import java.io.*;
import javax.swing.SwingUtilities; // Needed here conceptually

class MessageReceiver implements Runnable {
    private BufferedReader reader;
```

```
    private ClientGUI gui;
    private boolean running = true;

    public MessageReceiver(BufferedReader reader, ClientGUI gui) {
        this.reader = reader;
        this.gui = gui;
    }

    public void run() {
        String serverLine;
        try {
            while (running && (serverLine = reader.readLine()) != null) {
                System.out.println("DEBUG: Received: " + serverLine); // For
debugging
                if (serverLine.startsWith("MESSAGE ")) {
                    gui.displayMessage(serverLine.substring(8));   //   Remove
"MESSAGE " prefix
                } else if (serverLine.startsWith("USERLIST ")) {
                    String userListData = serverLine.substring(9);
                    String[] users = userListData.split(",");
                    gui.updateUserList(users);
                } else if (serverLine.startsWith("JOINED ")) {
                     gui.displayMessage("[System] " + serverLine.substring(7)
+ " has joined.");
                     // User list will be updated separately via USERLIST
command
                } else if (serverLine.startsWith("LEFT ")) {
                     gui.displayMessage("[System] " + serverLine.substring(5)
+ " has left.");
                     // User list will be updated separately via USERLIST
command
                } else if (serverLine.startsWith("ERROR ")) {
                    gui.showConnectionError("Server      Error:      "      +
serverLine.substring(6));
                     // Decide if client should disconnect based on error
                } else {
                     // Handle other potential system messages or ignore
                     gui.displayMessage("[Server] " + serverLine);
                }
            }
        } catch (IOException e) {
            if (running) { // Avoid showing error if we intentionally stopped
                 System.err.println("Connection   lost   to   server:   "   +
e.getMessage());
                 gui.showConnectionError("Connection lost to server.");
            }
        } finally {
             stop(); // Ensure running flag is false
             // Optional: attempt cleanup or notify Client class
             System.out.println("MessageReceiver thread stopped.");
        }
    }

     public void stop() {
         this.running = false;
         // Closing the reader/socket from another thread can be tricky.
```

```
        // Often, the main client class handles closing the socket, which
will cause the
        // reader.readLine() in this thread to throw an IOException, ending
the loop.
     }
}
```

CHAPTER 4. RESULTS ANALYSIS AND VALIDATION

This chapter evaluates the developed chat application based on usability, functionality, performance, security, stability, and how well it meets the initial requirements and goals.

4.1 Usability Assessment

The usability of the client application was assessed through informal walkthroughs and usage during testing.

- **Interface Clarity:** The Swing GUI, using standard components like `JTextArea`, `JTextField`, `JButton`, and `JList`, provides a familiar and generally understandable interface for anyone who has used basic desktop applications or other chat clients. The layout (messages central, input at bottom, user list on the side) is conventional and intuitive.
- **Ease of Use:**
    - Connecting: Requires manually entering the server IP and port, which is simple for technical users but could be improved with discovery mechanisms or saving profiles for non-technical users. The username prompt is straightforward.
    - Chatting: Typing in the input field and pressing Enter or clicking "Send" is standard and easy. Reading messages in the scrolling text area is also conventional.
    - User List: The list clearly shows who is currently online.
    - Disconnecting: Closing the window is a natural way to disconnect.
- **Feedback:** The application provides immediate visual feedback: sent messages appear (implicitly, by clearing the input field), received messages appear in the main area, and the user list updates dynamically. Error messages (e.g., connection failed, server disconnected) are displayed using `JOptionPane` dialogs, providing clear information about problems.
- **Potential Improvements:**
    - Visual styling is basic (default Swing Look and Feel). Themes or custom styling could improve aesthetics.
    - No indication if a message is successfully sent/delivered (only that it was passed to the output stream).
    - The user list could show more status information (e.g., idle, typing).
    - Handling long messages or messages with unusual characters was not extensively tested. Word wrapping in the `JTextArea` is enabled by default, which helps.

Overall, for its intended purpose as a basic, functional chat application, the usability is acceptable. It meets **NFR1 (Usability)** at a fundamental level.

4.2 Functionality Evaluation

The application's core features were tested against the functional requirements defined in section 3.1.

- **FR1/FR2 (Server/Client Connection):** Successful. The server listens on the specified port, and clients can connect using the correct IP/port. Connection failures (wrong IP, server down) are handled with error messages.
- **FR3 (User Identification):** Successful. Clients send a username upon connection, which the server uses for identification in messages and the user list. Basic validation (non-empty) was added. Duplicate username handling was implemented to reject the connection.
- **FR4 (Message Sending):** Successful. Clients can type and send messages via the input field/send button.
- **FR5 (Message Broadcasting):** Successful. The server correctly receives messages and broadcasts them to all *other* connected clients, prepending the sender's username.
- **FR6 (Message Display):** Successful. Clients display incoming messages chronologically in the `JTextArea`. The use of `SwingUtilities.invokeLater` ensures this happens correctly without GUI freezes. Auto-scrolling keeps the latest message visible.
- **FR7/FR8 (User List Maintenance & Display):** Successful. The server maintains an accurate list of connected users. It sends the full list upon connection (`USERLIST`) and sends notifications (`JOINED`, `LEFT`) along with updated lists when users connect or disconnect. Clients parse these messages and update the `JList` correctly via the EDT.
- **FR9/FR10 (Disconnection Handling):** Mostly Successful.
    - Graceful disconnect (closing client window) sends a `DISCONNECT` message, which the server handles correctly, removing the user and notifying others.
    - Abrupt disconnect (e.g., killing client process) causes an `IOException` on the server-side `ClientHandler` thread. The `finally` block ensures the user is removed, and notifications are sent. This works reliably.
    - Server shutdown: Clients detect this via `IOException` in the `MessageReceiver` loop and display an error message.

All core functional requirements were met.

4.3 Performance assessment

Performance was assessed qualitatively during multi-client testing on a local network.

- **Latency:** Message delivery between clients on a local area network (LAN) felt instantaneous. No noticeable delay was observed with a small number (<10) of clients. Latency over a wider network (Internet) would depend heavily on network conditions but is expected to be reasonable for text messages due to TCP's efficiency for such data.
- **Throughput:** The application is designed for human-typed text messages, not high-volume data transfer. Throughput is more than adequate for this purpose. Flooding the chat with rapid automated messages could potentially saturate the server or network connection, but this is outside normal usage.
- **Resource Usage (Server):**
    - *CPU:* Idle CPU usage is negligible. When clients connect and send messages, CPU usage increases due to thread creation, I/O operations, and string manipulations. With the thread-per-client model, CPU usage scales linearly (roughly) with the number of active clients and message rate. For a few dozen clients, usage should remain low on modern

hardware. However, scaling to hundreds or thousands of clients would become inefficient due to context switching overhead.
    - *Memory:* Each client connection consumes resources: a `Socket`, associated streams, a `Thread` (stack space), and objects for the `ClientHandler`. Memory usage grows linearly with the number of clients. For a small number of clients, this is acceptable.
- **Resource Usage (Client):**
    - *CPU:* Idle CPU usage is very low. Spikes occur when sending/receiving messages or updating the GUI, but these are brief. The background thread for receiving adds minimal constant overhead.
    - *Memory:* Swing applications have a certain baseline memory footprint. The chat client's memory usage is relatively stable, growing slightly with the number of messages stored in the `JTextArea` and users in the `JList`. It remains well within acceptable limits for a desktop application.
- **Scalability:** The chosen **thread-per-client** model on the server is the primary performance bottleneck for scalability (**NFR2 Concurrency**). While simple to implement, it doesn't scale well to a very large number of concurrent users due to thread overhead. For the intended scope (educational, small groups), performance is adequate. For higher scalability, non-blocking I/O (Java NIO) or asynchronous frameworks (Netty) would be necessary.

Performance meets the requirements for a small-scale chat application.

4.4 Security and Stability

- **Security:** This is the weakest aspect of this basic implementation.
    - **No Encryption:** All communication (usernames, messages) is sent as plain text over the network. Anyone sniffing the network traffic (e.g., on the same Wi-Fi) can read the entire conversation. This is a major security vulnerability for any sensitive communication. Implementing SSL/TLS using `SSLSocket` and `SSLServerSocket` would be necessary for secure communication.
    - **No Authentication:** Usernames are taken at face value. There is no password mechanism or verification. Anyone can connect with any username (subject to basic uniqueness check).
    - **Denial of Service (DoS):** The server is vulnerable to simple DoS attacks. Rapidly opening and closing connections could exhaust server resources. Sending very large messages could consume memory or bandwidth. No rate limiting or input validation beyond basic checks is implemented.
    - **Protocol Vulnerabilities:** The simple text protocol could be exploited if not handled carefully (e.g., injecting fake server commands if parsing is weak, though the current design limits this).
- **Stability:**
    - **Error Handling:** Basic `try-catch-finally` blocks are used to handle `IOExceptions` from network operations and socket closures. This prevents the server `ClientHandler` threads or the client `MessageReceiver` thread from crashing the entire application on typical network errors or disconnections. The use of `finally` blocks ensures resources are released.
    - **Concurrency Issues:** Proper synchronization (`synchronized` blocks/collections on server, `SwingUtilities.invokeLater` on client) was used to prevent common concurrency issues like race conditions during user list updates or GUI modifications from background threads. This contributes significantly to stability.

- **Resource Leaks:** `Socket` and stream resources are explicitly closed in `finally` blocks, preventing resource leaks on disconnection.
- **Unhandled Exceptions:** While common IO errors are handled, other unexpected runtime exceptions might still occur, potentially terminating a `ClientHandler` thread or causing issues. More comprehensive exception logging and handling could improve robustness.

The application demonstrates reasonable stability for basic operation (**NFR6 Basic Error Handling**). However, the lack of security features makes it unsuitable for use in untrusted networks or for sensitive communication without significant enhancements.

4.5 Comparison with Requirements and Goals

Let's map the results back to the initial goals (Section 2.6):

1. **Develop a Functional Server: Achieved.** Server listens, accepts multiple clients via threads, receives/broadcasts messages, manages user list, handles disconnections.
2. **Develop a Functional Client: Achieved.** Client connects, provides GUI (message area, input, send, user list), sends messages, receives/displays messages and user list updates asynchronously using `invokeLater`. Clean disconnect implemented.
3. **Ensure Basic Robustness: Partially Achieved.** Handles common network errors and disconnections gracefully, preventing crashes. Stability is decent. However, robustness against malicious input or edge cases could be improved.
4. **Demonstrate Core Concepts: Achieved.** The code clearly demonstrates `ServerSocket`/`Socket` usage, thread-per-client model, basic text protocol, Swing GUI construction (`JFrame`, `JTextArea`, etc.), event handling (`ActionListener`), and crucial Swing threading (`invokeLater`).
5. **Provide Clear Documentation: Achieved.** Code includes comments, and this report provides comprehensive documentation.
6. **Maintain Simplicity: Achieved.** Relies solely on core Java SE libraries as intended.

The project successfully met most of its primary goals, particularly those related to functionality and demonstrating the core Java technologies. The main area where it falls short is robustness/security, which was anticipated given the focus on fundamental concepts over building a production-ready system. The performance is adequate for the intended scale.

CHAPTER 5. CONCLUSION AND FUTURE WORK

This final chapter summarizes the project's outcomes and suggests potential avenues for future development and enhancement.

5.1 Conclusion

This project successfully designed, developed, and tested a multi-user chat application using fundamental Java technologies: Java Sockets for client-server networking and Java Swing for the graphical user interface. The application fulfills the core requirement of enabling real-time text communication between multiple users connected to a central server.

The key achievements include:

- A functional client-server architecture implementation.
- Successful use of TCP sockets (`ServerSocket`, `Socket`) for reliable stream-based communication.
- Implementation of a thread-per-client model on the server to handle concurrent users.
- Development of a responsive and functional Swing-based client GUI.
- Correct handling of Swing concurrency using `SwingUtilities.invokeLater` to update the GUI from the network-listening thread, preventing UI freezes.
- Implementation of a simple text-based protocol for commands and messages.
- Dynamic updating of user lists across all clients.
- Basic error handling for common network issues and disconnections.

The project serves as a valuable educational tool, clearly demonstrating the integration of networking, multithreading, and GUI programming concepts within the Java ecosystem, using only core libraries. It successfully meets the goal of providing a simpler, understandable alternative to complex commercial chat platforms or high-level frameworks for learning or small-group use.

However, the project also highlights the limitations of this basic approach. The thread-per-client model poses scalability challenges for a large number of users. Most significantly, the lack of any encryption or robust authentication makes the application insecure for use over public or untrusted networks.

In conclusion, the application meets its defined objectives as a functional demonstration and learning tool, providing a solid foundation upon which more advanced features could be built.

5.2 Future work

While the current application is functional for basic chat, numerous enhancements could be implemented to improve its features, security, scalability, and usability:

1. **Security Enhancements:**
    - **Encryption:** Implement TLS/SSL using `javax.net.ssl.SSLServerSocket` and `javax.net.ssl.SSLSocket` to encrypt all communication between clients and the server, protecting against eavesdropping. This is the most critical improvement needed for real-world use.
    - **Authentication:** Add a user registration and login system (e.g., username/password). The server would need to store credentials (securely, e.g., hashed passwords) and verify them upon connection.
    - **Input Validation:** Implement stricter validation on the server for usernames and messages to prevent potential injection attacks or protocol manipulation.
2. **Feature Additions:**
    - **Private Messaging:** Allow users to select another user from the list and send a direct message that only the recipient sees. This requires protocol extensions and server logic to route messages specifically.
    - **Chat Rooms/Channels:** Extend the server to support multiple chat rooms, allowing users to join specific topics or groups. Clients would need UI elements to select/switch rooms.

- **File Transfer:** Implement functionality to send and receive files between clients, likely involving separate data streams or chunking mechanisms.
- **User Status:** Show user statuses (Online, Away, Idle, Typing...) in the user list.
- **Persistent Chat History:** Save chat messages to a file or database on the server and potentially load history when a client connects.
- **Timestamping:** Add timestamps to displayed messages.
- **Server Administration:** Create commands or a separate interface for server administration (e.g., kicking/banning users, broadcasting system messages).

3. **Scalability and Performance Improvements:**
   - **Non-Blocking I/O (NIO):** Replace the thread-per-client model with Java NIO (`java.nio` package). This uses selectors and channels to manage many connections with fewer threads, significantly improving scalability.
   - **Thread Pool:** Instead of creating a new thread for every client, use a fixed-size thread pool (`java.util.concurrent.ExecutorService`) to manage client handling tasks. This reduces thread creation overhead and limits resource consumption.
   - **Asynchronous Frameworks:** Utilize libraries like Netty or Vert.x, which provide high-performance, asynchronous networking capabilities, abstracting away much of the low-level NIO complexity.
4. **GUI Enhancements:**
   - **Look and Feel:** Allow users to change the Swing Look and Feel or apply custom themes for better aesthetics.
   - **Rich Text:** Support basic text formatting (bold, italics), emojis, or inline images.
   - **Notifications:** Implement desktop notifications for new messages when the application window is not focused.
   - **UI Framework:** Consider migrating the GUI to JavaFX, which is the modern recommended UI toolkit for Java desktop applications, offering more features and better styling capabilities than Swing.
5. **Refinement:**
   - **Protocol Definition:** Formalize the text protocol, perhaps using JSON or another structured format for messages to make parsing more robust and extensible.
   - **Configuration:** Use configuration files (e.g., `.properties`) for server port, default username, etc., instead of hardcoding or relying solely on command-line arguments.
   - **Build/Dependency Management:** Introduce Maven or Gradle for easier building, packaging, and managing potential future dependencies.

Implementing these future work items would transform the basic chat application into a more robust, secure, and feature-rich communication tool, while also providing further learning opportunities in advanced Java development concepts.