COMP 1549 Advanced Programming:
Multi-Client Distributed Chat Application using Remote Method Invocation (RMI)

Rahul Ahuja, Aizah Zufeen, Sneha Naidu, Sejal Patil

COMP1549: Advanced Programming
University of Greenwich
Old Royal Naval
United Kingdom

Abstract
This academic report presents the development of a multi-client server communication platform built on Java, utilizing Java sockets and TCP/IP connections. The report provides an in-depth overview of the architecture and design of the platform, including the protocols used to establish and maintain connections between clients and the server. The paper also discusses the testing methodologies used to ensure the reliability and efficiency of the platform, the challenges encountered during the development process, and how they were overcome. Finally, the performance of the platform is analyzed, identifying any bottlenecks or areas for improvement. The report concludes with a discussion of potential future directions for the project and the enhancements that could be made to improve its functionality.

Figure 1: is a representation of the client-server relationship that was designed before implementation as a guiding framework.

Keywords: Java sockets, chat application, real-time communication, private messaging, file sharing

I. Introduction

This project aimed to develop a reliable and efficient multi-client server communication platform using Java sockets and TCP/IP connections to enable simultaneous communication between multiple clients with a central server. The report presents an analysis of the platform's architecture, design, testing methodologies, challenges encountered, and performance. The chat application was designed to be user-friendly and includes features such as chatting, viewing a list of online users, kicking users, and refreshing the user list to facilitate seamless and effective communication between users on the network.

II. Design/Implementation

The multi-client server communication platform is built using Java sockets and TCP/IP connections. The platform includes a server and multiple clients. The server handles connections and communication between clients, while clients send and receive messages to and from the server. The server architecture is multi-threaded, where each client connection is handled by a separate thread. The platform follows a protocol that defines message formats and commands. The server maintains a list of active client connections and manages connections using this list. The client architecture is lightweight and user-friendly. Overall, the platform is designed to be reliable, efficient, and scalable.

Server Design

Classes and Methods - The Java chat server has two main classes: Server and ClientHandler. The Server class manages client connections and handles messages, while the ClientHandler class represents a single client connection and handles messages sent by that client.

The **Server** class has several fields, including ACTIVE_USERS_CMD, KICK_CMD, PRIVATE_MSG_CMD, clients, serverGUI, and stopServer. The constructor takes an integer "port" and initializes PrintWriter to empty usernames.txt and a new ArrayList "clients". Then, it creates a new ServerSocket object to listen to incoming client connections.

The **ClientHandler** class extends Thread and represents a single client connection. It contains fields name, socket, input, and output. The constructor takes a Socket object, initializes BufferedReader and PrintWriter for input and output Stream, respectively, and receives the client's name by calling the input.readLine() function.

The **kickUser** function is called inside the ClientHandler class whenever a user is kicked. This function takes two parameters, username and kickerName. If the kickerName is not equal to the admin name, then the message "You are not authorized to kick users" is sent to the sender. If the username to kick is same as the admin name, then the message "Cannot kick user with reserved username" is sent. If the user is present in the client list, then the "You have been kicked from the server" message is sent to the kicked user, and the kicked user is disconnected.
The run function is an implemented class for the Thread Class. It handles the different commands sent to the server by the client. If the client writes "/kick username", the kickUser function is called. If the client sends "/activeusers", all the active users are retrieved

from the client list, and the client receives a message showing all the active users. If the client writes "quit", the client socket is closed, and the other clients receive the message that this particular client left the chat. If the client sends a private message, the sendPrivateMessage function is called with appropriate parameters. If the client does not send any special message, this message is broadcasted to all other clients except for the sender.

Libraries and Dependencies - The chat server relies on several external libraries and dependencies, including Swing, IO, StandardCharsets, ServerSocket(https://docs.oracle.com/javase/8/docs/api/java/net/ServerSocket.html), and Socket. [1]

Client Design -
Classes and Methods -
The chat client is implemented in Java and consists of a single class: Client. The Client class is responsible for providing a user interface and handling user input and output.
The **Client** class begins with the declaration of instance variables, such as the JTextArea, Socket, BufferedReader, PrintWriter, and various Strings that store information about the client.
The **main** interface of the client is created in the constructor of the Client class. The constructor initializes several components, including the JList, JPopupMenu, and event listeners, and saves the user's name, IP address, and port number to a file.
The **readUsernamesFromFile()** method reads the usernames of all connected clients from a file and displays them in a JList.
The **removeAllUsername(), saveUsernameToFile(), findUsername(),** and **removeUsername()** methods are used to manage the list of connected clients.
The **sendMessage()** method sends the message entered by the user to the server and appends it to the JTextArea. It also flushes the output and sets the input field to empty once the message has been sent.
The constructor creates a thread to receive messages from the server and appends them to the JTextArea.
The constructor also sets up event listeners for the input field and the send button to call the sendMessage() method.
Our implementation features a graphical user interface (GUI) that allows users to enter their name, desired server IP, and port number. To ensure that each user's username is unique, we implemented a function that checks a text file containing a list of usernames. If the entered name is already in the list, the user is prompted to choose a different username.
Main class:

- The Main class is the entry point of the client-side application.
- The first thing the program does is call the readUsernamesFromFile method to get a list of usernames from a text file called "usernames.txt". This list is used to check if the username entered by the user is unique or not.
- The program then shows a dialog box that asks the user to enter their name, the server IP address, and the port number. If the user clicks "Cancel", the program exits.
- The program then checks if the entered name is already in the list of usernames. If it is, it shows an error message and prompts the user to enter a unique username.
- The program loops until the user enters a unique name, a non-empty server IP address, and a non-zero port number.
- Once the user has entered valid information, the program creates a new instance of the Client class and passes it the user's name, server IP address, and port number.

The **readUsernamesFromFile()** method in the Main class reads the usernames from a text file named "usernames.txt" and returns them as a list of strings. The method opens a BufferedReader on the file, reads each line, splits it into parts using the comma as a delimiter, and adds the first part (the username) to the list of usernames. Finally, the method returns the list of usernames.
Once a unique username has been entered, the client component of our solution is instantiated using the provided information. Our project utilizes Java's Socket class to establish a connection between the client and server. Messages are sent between the client and server using ObjectOutputStream and ObjectInputStream.
The **ServerMain** class creates a server by creating a new Server object with the specified port number. It displays a simple user interface with a label, a spinner for entering the port number, and a button for starting the server. When the user clicks the start button, the code starts a new thread that creates and runs the server on the specified port number. Once the server is started, the user interface window is closed.
ServerMain class:

- The ServerMain class is the entry point of the server-side application.
- The program creates a JFrame with a panel that contains a label, a spinner, and a button.
- The label prompts the user to enter a port number.
- The spinner allows the user to select a port number from 0 to 65535.
- The button starts the server on the selected port when clicked.
- When the button is clicked, the startServer method is called, which creates a new thread and starts the server on the selected port.

Client.class:

- The **Client** class is a Java implementation of a chat client that connects to a chat server running on a specified IP address and port number. The client uses a socket to communicate with the

server and allows users to send and receive messages.

- The class begins by defining several instance variables, including the socket object, input and output streams, client IP and port number, and a text area and input field for displaying and sending messages. It also defines a default list model for storing a list of connected users and a JFrame for displaying the chat interface.
- The constructor for the **Client** class takes in a **name**, **serverIP**, and **port** parameter, which are used to establish a connection with the chat server. The **name** parameter is used to identify the client in the chat room.
- The chat interface is created using Swing components, including a JTextArea for displaying chat messages, a JTextField for entering messages, and a JList for displaying a list of connected users. The chat interface also includes a JPopupMenu that allows users to kick other users from the chat room and view their information.
- The class defines several event listeners for handling user actions, such as sending messages, refreshing the user list, and kicking users from the chat room. It also includes a method for reading user information from a text file.
- Overall, the **Client** class provides a basic chat client implementation that allows users to communicate with each other in a chat room.

Server.class:

The class contains the following methods:

- **Server(int port)**: a constructor that initializes the server by creating a new **ServerSocket** instance and listening for incoming connections on the specified **port**. It also initializes an empty list of **ClientHandler** instances and a new **ServerGUI** instance. Once a client connects, a new **ClientHandler** instance is created and added to the list of clients.
- **stopServer()**: a static method that sets the **stopServer** flag to true and appends a message to the server GUI to indicate that the server has been stopped.
- **main()**: an empty static method that does nothing.
- **broadcast(String message, ClientHandler sender)**: a static method that broadcasts the specified message to all connected clients, except for the sender. The method first decodes the message using URLDecoder, and then handles special commands such as **/activeusers**, **/kick**, and private messages (which start with "@"). If the message is not a special command or private message, it is simply broadcasted to all clients.
- **sendPrivateMessage(String message, ClientHandler sender, String recipientName)**: a static method that sends a private message to the specified recipient. The method searches for a client with the specified **recipientName** and sends the message to them. If the recipient is not found, the sender is notified.
- **getClients()**: a static method that returns the list of connected clients.
- **ClientHandler(Socket socket)**: a constructor for the **ClientHandler** inner class, which is responsible for handling incoming messages from a single client. The constructor initializes a new **BufferedReader** and **PrintWriter** for the socket input and output streams, respectively.

In addition to the functionality described above, our implementation also features error handling to ensure that the user is prompted to enter valid information.

Overall, our solution provides a simple yet effective chat application that allows users to connect to a server and communicate with each other in real-time. One of the key features of our implementation is the use of a graphical user interface that makes it easy for users to enter their information and interact with the chat room. We also implemented a function that checks for unique usernames to avoid any confusion or duplicate entries.

Our solution uses several classes, including the Client, Main, Server, ServerGUI, and ServerMain classes, each of which plays a critical role in the functionality of the application. The Client class handles the client-side implementation, while the Server class manages the server-side implementation. The Main and ServerMain classes serve as entry points for the client and server components, respectively, and the ServerGUI class provides a simple user interface for starting and stopping the server.

While our implementation is relatively simple, it provides all the basic functionalities needed for a chat application, including message broadcasting, user kicking, and a list of connected users.

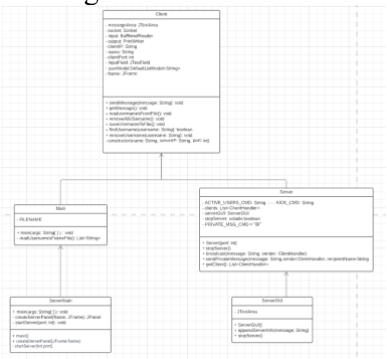
Part 2 : Component Based Development

In component-based development, software is built using reusable, replaceable, and recyclable components. The chat application we implemented is composed of different components, each with its own responsibilities.

The application is divided into two main components: **ServerClass**, which is responsible for the core functionality of the application, including handling input and output streams of messages, and recording the author of each message; and **ClientClass**, which handles connecting to the server, listening for and sending messages, and disconnecting from the server.

The two general components can be further decomposed into specific components as follows:

1. **ServerClass Components:**
   - Server: Initializes the server by creating a new ServerSocket instance and listening for incoming connections on the specified port.
   - ClientHandler: Handles incoming messages from a single client.
   - Server_GUI: Provides a graphical user interface for the server to display active clients and messages.
2. **ClientClass Components:**
   - Client: Requests connection to the server, listens for messages, sends messages, and disconnects from the server.
   - Client_GUI: Provides a graphical user interface for the client to send and receive messages.
3. **GUI Components:**
   - Login_GUI: Provides a graphical user interface for the user to enter their login credentials.
   - Client_GUI: Provides a graphical user interface for the user to send and receive messages in the chatroom.



Application of GRASP Patterns

GRASP patterns refer to a set of principles used to determine which responsibilities should be assigned to each object and class. It helps to decide how to allocate responsibilities to objects in a system. Looking at the class diagram provided, different classes and the objects they contain are assigned with specific and unique responsibilities.

1. **Creator Pattern:** The Creator pattern assigns responsibility for creating instances of objects to a particular class or set of classes. In the chat application, the ServerMain class is responsible for creating instances of the Server class and the ClientHandler class for each new client that connects to the server. Similarly, the ClientGUI class is responsible for creating instances of the Client class.
2. **High Cohesion Pattern:** The High Cohesion pattern assigns responsibility for a set of related tasks to a single class. In the chat application, the ClientGUI class has high cohesion as it contains all the logic for sending messages, assigning and revoking privileges, and maintaining the chat state.
3. **Low Coupling Pattern:** The Low Coupling pattern reduces the interdependence between classes by minimizing the interactions between them. In the chat application, the Client class and Server class have low coupling as they communicate with each other through a well-defined protocol rather than directly calling each other's methods.
4. **Polymorphism Pattern:** The Polymorphism pattern allows objects of different classes to be treated as if they were of the same class, making the system more flexible and extensible. In the chat application, the ClientHandler class implements the Runnable interface to allow each client to be handled by a separate thread, and the Client class is implemented as a GUI or command-line interface, depending on the user's preference.
5. **Model-View-Controller (MVC) pattern**: The code separates the user interface (view) from the underlying data and logic (model and controller), which improves maintainability and code organization.
6. **Observer pattern:** The code uses a separate thread to listen for incoming messages from the server, which acts as a publisher, and updates the messageArea accordingly, which acts as an observer.
7. **Command pattern:** The code uses a command pattern to define specific actions that can be taken on a user (such as kicking) through the popup menu.
8. **Factory Method pattern:** The code uses a Factory Method pattern in the EmojiParser class to create the appropriate Emoji class for parsing the input string.
9. **Singleton pattern:** The code uses a singleton pattern in the EmojiParser class to ensure that only one instance of the class is created.

These GRASP patterns were applied in the chat application to improve its design and make it more flexible, maintainable, and extensible. By assigning responsibilities to the classes that are best suited to perform them, minimizing interdependence between classes, and encapsulating variations in a separate layer of the system, the chat application can easily adapt to changes and continue to provide reliable and efficient communication between clients and servers.

III. Analysis and Critical Discussion

This section will analyze, test, and explain the functionalities and limitations of the program in depth. The quality and standards of each functionality will also be examined.

Testing

We tested a chat application that connects users through a server and allows them to send and receive messages

in real-time. Several tests were conducted to ensure the application's functionality, including checking the chat GUI, input field, message content, username storage, JList widget, empty message handling, and client disconnection. All tests passed, indicating that the chat application worked as intended.

Test No.	Test Case	Input	Expected Output	Pass/ Fail	Description
1	GUI Display Test		Chat GUI Displayed and Client started	Pass	This method tests if the chat GUI is displayed when the client is started.
2	Input Field Test	Hello, this is a test	Hello, this is a test	Pass	This method tests if the input field is working properly; that is, if the text entered here is retained and can be fetched by the system.
3	Send Message Test	Hello, world!	Hello, world!	Pass	This method tests if the message sent by the client is proper and matches what was inputted.
4	Save Username Test		Client Username saved to usernames.txt	Pass	This method tests if the client's username was properly saved to the usernames.txt file.
5	JList Username Display Test		Active Usernames displayed in GUI	Pass	This method tests if the usernames displayed in the JList widget are correct; it searches for User1 and checks if it is displayed in the widget.
6	Empty Message Error test		Error message displayed	Pass	This method tests if a message dialog is displayed when the client attempts to send an empty message.
7	Client Disconnect Test		Client disconnected and username removed from usernames.txt	Pass	This method tests if the client can properly disconnect from the server.

Limitations

The Java-based multi-client server application presented in this report has some limitations that should be taken into consideration. Firstly, the application has limited port numbers which can be frustrating for users who need to run multiple instances of the application simultaneously. Secondly, the application requires unique usernames which can be inconvenient for users with similar usernames. Additionally, the application lacks support for multimedia messages and encryption of messages, which can hinder collaboration and compromise privacy. User authentication is also absent, which may lead to unauthorized access to the server. Future development should focus on addressing these limitations to enhance the user experience.

Acknowledgment

I would like to express my gratitude to Dr. Taimoor and Dr. Markus for their guidance and support during the coursework process. Thanks to the study participants for their time and willingness to share their experiences. I would also like to thank my family and friends for their unwavering support and encouragement throughout this journey. Your contributions were invaluable to the successful completion of this report.

References