COMP1549 Advanced Programming Group-based client-server communication portal with GUI

Bhavesh Nankani, Pranathi Tulasi, Rishikumar Patel, Yash Patel, Hemant Dayal

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

Abstract. This report demonstrates and documents the development of a multi-user client-server communication platform built using Java. It provides a detailed breakdown of the design of the application, its implementation; including justification for the design of all the components, both the networking side and the GUI side. The report also discusses the various programming principles applied, the tests carried out to check for efficiency and the challenges faced while developing and testing the solution. It presents an critical analysis of the features implemented and limitations and the results of the tests, as well as the reflective conclusions drawn during the entire development process and lessons to take forward for future developments.

Keywords: *Communication portal, client-server, Graphical User Interface, Java Network API*

I Introduction

This project is about building a client-server communication system where multiple users can connect to a centralized server and communicate with each other. This distributed network is accessible through a GUI in which clients can login, select their server and send/receive messages. In essence, it is a convoluted version of most messaging/discussion platforms which allow multi-user communication across devices, with just its core features. The aim of this report is to provide a detailed breakdown of the development of this project throughout all its stages, and the features implemented along with testing measures followed. We aim to analyse and justify the fundamental programming concepts applied and reflect on the shortcomings of our project in a critical manner to help improve and learn for future use. `https://dev.azure.com/bn9640z/COMP1549_CW/_git/COMP1549_CW`

II Design/Implementation

This section will be exploring the design choices of the solution we produced, its technical components and the application of programming principles like GRASP and class design. The platform is essentially a distributed network with a centralized server over which multiple clients can connect and communicate by sending and receiving messages. The two main parts of this are the server and client-side implementation.

II.1 Server-side

The server's responsibility is to handle incoming client connections and establish a portal that enables communication between multiple clients. The socket communication is implemented using Java's networking API which helps establish a ServerSocket object that runs on a specified port number and accepts incoming client requests. When a client connects to the server, an instance of the Socket object is created and a TCP connection is established. Then, a new instance of the ClientHandler class is created which also extends the Tread class that enables it to handle each client's actions separately on their own thread.

Once a client is connected to the server, the ClientHandler methods allows them to send messages through the input stream and reads them using the processInput() method which broadcasts the message according to whether it's public or private. The messages are displayed through the output stream using the PrintWriter object. This process is carried out every time a new client requests a connection, and each client is given a thread on which all of their activities are carried out. Indifferent to the number of clients, the threads all run concurrently, executing each process of that specific client.

To sum up, the server-side implementation uses a Socket object to establish a TCP connection between clients and the server. A new instance of the ClientHandler class is created every time a client joins the server which uses multi-threading to carry out each client's requests which ensures the I/O stream is not interrupted. This server design focuses on concurrency and robustness to allow it efficiently manage multiple client connections and communication tasks as well as maintaining the state of the server. [2]

II.2 Client-side

The client-side implements the communication part between users and provides a UI for the application. It utilizes sockets for network communication and implements methods for connecting to the server, sending messages and requesting user lists. For the GUI components and event handling, it utilizes Swing.

The constructor methods takes in the username and port number and creates a Socket object using these parameters. Once the Socket object is created, the input and output streams are initialized and the individual client can use them to send and receive messages using the BufferedReader and PrintWriter classes. The users can interact with each other through the GUI and event listeners are attached to buttons to trigger actions such as sending messages and refreshing user lists. For example, the Send button is attached to an ActionEvent, sendMessageAction which gets triggered when the button is pressed and the text in the input field is obtained using the getText() method and then the PrintWriter object is used to display the text on the JTextArea through the output stream.

The client-side design incorporates asynchronous communication to prevent blocking the UI thread during background network operations by using the invokeLater() method. Network operations such as sending and receiving messages are performed in separate threads to maintain responsiveness of the UI. We have also implemented appropriate error handling methods and providing feedback messages to confirm successful operations. Error handling is done mostly using the TryExcept method and displaying error messages and feedback messages include such as successful client connections. Overall, the client-side design provides a user-friendly interface while effectively managing network communication. [2]

II.3 Classes and Features Implemented

To build the application, we made use of several well-organized classes to break the problem down into smaller, easily-manageable chunks. These classes are divided into two groups; server-side and client-side. The server-side classes consist of an entry point into the server component, the actual implementation and a GUI interface. The client-side classes are more intricate with additional classes for the various features implemented in the application, along with the entry point and GUI interface. A detailed analysis about each class follows:

II.3.1 ***ServerMain***

The ServerMain class is the entry point of the server implementation; it is a way to start and stop the server application and handle command line arguments for port configuration. It consists of a main() method which takes the command line argument to specify which port to run the server on, and has a default port number of 8080 if no specific port is selected.

To initialize the server, a Server object is created with the previously selected port number and a message is displayed indicating that the server is starting on the specified port. The server is then started using the start() method. This class also contains a shutdown hook which ensures that the server is not stopped abruptly when the application is terminated. It first displays a message to indicate that the server is shutting down and then calls the stop() method. An exception handler is also implemented to catch exceptions during server initialization and display the appropriate error message.

II.3.2 ***Server***

This class contains the main implementation of the server and uses Java's Socket API to create a socket-which is a combination of an IP address and a port number-to enable communication between devices over a distributed network. The class uses a ServerSocket object to accept incoming connections from clients and a threadPool to manage concurrent connections. It also uses a JTextArea to display server logs. The initialization of the server begins by the creation of a ServerSocket object and once the connection is accepted, it creates an instance of the ClientHandler object in a separate thread to handle the communication with that particular client. The isUsernameTaken() checks if a username is unique and if it's already taken by another user, the connection fails to get established.

Upon connecting to the server, the first user is chosen to be the coordinator by the setCoordinator() method and if the coordinator leaves the server, the next user in the user list is reassigned to be the coordinator by the reassignCoordinator() method. The broadcastMessage() is used to display messages sent by clients and there is an automatic shutdown method for the threadPool after 60 seconds of inactivity. This class also implements the Kick feature through the kickUser() method which disconnects that specific user from the server; however, this can only be done by the coordinator. Other methods in this class deal with notifying when a new client connects (notifyNewClientConnection) and the private messaging feature between clients(sendPrivateMessage).

II.3.3 ***ServerGUI***

This class defines the GUI of the server and extends JFrame. The GUI components are imported from Swing and AWT. It contains a start/stop button, a portSpinner component to input the port number and a logTextArea to display server logs. The constructor method, initializeUI() sets up the initial GUI, with all the mentioned components and the appendLog() method updates the server log area whenever a change takes place, like the start of a server, or the joining of a new client, etc.

II.3.4 ***ClientMain***

The ClientMain class serves as the entry point of the client-side implementation and also contains the Swing GUI initialization to manage some GUI components but this initialization occurs on the EDT. The main purpose of this class is to establish a connection with the server. Similar to the ServerMain class, it takes arguments for port number and server IP which are set to 8080 and localhost respectively. Once the socket is set up, and this application is run, a dialog box prompts the client for username, port number and server IP and the input is validated upon entering. After the input is validated, it initializes the client-side components and attempts to connect to the server using the entered port number and it the connection fails, an error message is displayed and it exits the application.

II.3.5 ***ClientHandler***

The ClientHandler class is responsible for handling communication between the server and an individual client. It extends the Thread class, enabling it to run parallel to other threads. The constructor method is initialized by a Socket which represents the client's connection to the server. The run() method reads the username and notifies the server about the new client as well as displaying a welcome message. It then enters a loop to read messages from the clients and classifies private messages if they are preceded by an @username and delivers it to the mentioned user, or broadcasts it to everyone. This class also contains several helper methods to send welcome messages, notifications about newly-joined clients and input processing.

II.3.6 ***ChatClientUI***

This class implements all the GUI components of the application and contains methods that provide functionality to the application. It has various instance variables for buttons and text areas as well as variables that manage network and client-related operations. The initializeUI() method sets up the main frame and its layout; it also configures the UI components like messageArea, userList and buttons. The sendMessageAction() method handles the sending of messages, displaying them in the message area and clearing the input field after sending the message. Another method in this class, requestUserInfo() is called when a user selects the "View Information" option after right-clicking on a user's name to know their IP address and port number; it sends a message to the server to request information about the selected user. There are several other helper methods than enable network management and display messages. [3]

II.3.7 ***ClientUserManager***

The ClientUserManager class is responsible for file handling operations related to usernames, making it easier to manage user data and output it when requested. The class contains a variable filename which stores the path to the text file containing the usernames. It also has a readUsernames() method which is responsible for reading the user input for username and adding it to a list which is then fed into the text file using the addUsername() method. Additionally, there is a removeUsername() method to remove specified usernames from the file.

II.3.8 ***ClientNetworkManager***

This class is responsible for handling the socket connection with the server and ensuring that the client can have seamless communication within the application. It also deals with updating the UI when messages are being sent and received. As soon as it is instantiated, it attempts to connect to the server and initializes the serverIP, serverPort and userName variables. The tryToConnect() method attempts to establish a socket connection and then initializes the input and output streams for communication. It then starts a separate thread for listening to messages from the server and the received messages are processed by the startListening() method which checks for prefixes in case of them being private. This class also contains helper methods to check for active clients and sending the usernames to the server through the output stream.

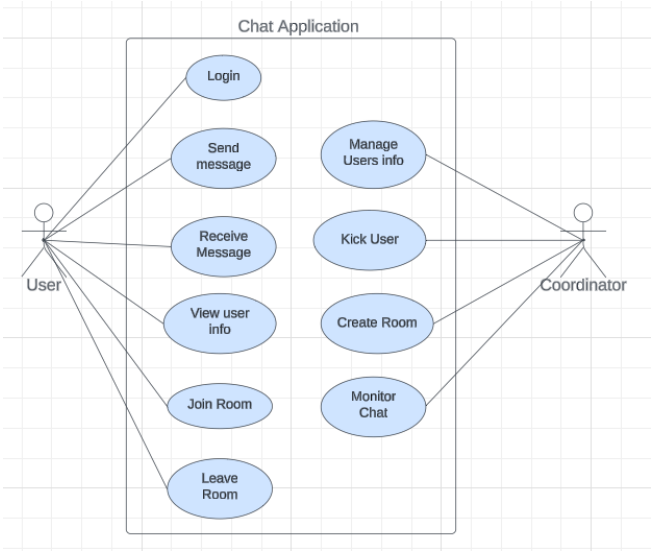
II.4 Software design

The class diagram shows the relationship between the important classes, their attributes and methods. The Server and ClientHandler class have a one-to-many relationship as one server is used to host multiple clients and they have an Aggregation relationship because a server can exist without any clients but the ClientHandler object only has meaning in the context of a server. The same applies for the ServerGUI class as it wouldn't exist without a Server.

ChatClientUI and ClientNetworkManager/ClientUserManager have a Dependency relationship as ClientUI depends on the other two classes to display messages and keep track of user information. ClientHandler and ClientNetworkManager/ClientUI/ClientUserManager have a Composition relationship as the ClientHandler manages these functionalities and they wouldn't exist on their own. When a ClientHandler object is destroyed (i.e. client disconnects), its associated ClientNetworkManager, ClientUI, and ClientUserManager objects are destroyed as well. [1]

The Use Case Diagram below shows the two actors, User and Coordinator and their functions (use cases). The User's use cases are Login, Send/Receive Message, View User Info, Join/Leave Room. These actions represent the actions a user can perform .

The Coordinator can Manage Users' Info, Kick User, Create Room and Monitor Chat. The Coordinator has all the privileges that a User has, along with these additional ones. Both the User and Coordinator share a binary association with the system, meaning that they both interact with it. The system boundary, Chat Application, represents the entire system and encompasses the use cases.



II.5 GRASP and Design Patterns

GRASP stands for General Responsibility Assignment Software Patterns and it is a set of guidelines for assigning responsibilities to objects/classes. It helps developers understand objects and responsibilities from the problem domain and defines a blue print for those objects and how they interact with each other. [5]

1. **Creator:** The Creator pattern assigns responsibility for creating instances of classes to a class that contains the most information required to create the instance. In our application, ServerMain creates instances of the Server class and ClientMain creates instances of ChatClientUI, ClientUserManager and ClientNetworkManager.
2. **Controller:** The Controller pattern assigns responsibility for handling system events to a class that represents the overall system. In the ServerMain class, its main() method acts as a controller for starting the server. Similarly, in the ClientMain class, its main(0 method acts as a controller for initializing the client. Aditionally, in the ChatClientUI class, its sendMessageAction() and refreshUserListAction() methods handle user input and UI interactions.
3. **High Cohesion:** The High Cohesion pattern aims to keep related responsibilities and behaviors within a single class. The ClientHandler class encapsulates functionality related to managing client connections and communication and ClientUserManager manages all the functionality related to managing usernames.
4. **Low Coupling:** The Low Coupling pattern aims to minimize the dependencies between classes or components in a system, reducing the impact of changes. The Server class communicates with instances of ClientHandler without tightly coupling with individual client connection.
5. **Information Expert:** The Information Expert pattern assigns responsibility to the class that has the most information required to fulfill that responsibility. The ClientUserManager class manages user-related actions like reading and adding usernames to the text file and ClientNetworkManager manages network-related operations like connecting to the server and maintaining communication.
6. **Indirection:** The Indirection pattern introduces an intermediary or intermediate class or interface to decouple two elements in a system, providing low coupling. The Server class uses instances of the ClientHandler class to indirectly establish communication with individual clients.
7. **Model-View-Controller:** The code follows the MVC pattern where the Model represents the logic (classes like Server and ClientHandler), View represents the UI components (ServerGUI) and Controller acts as the connection between the two (classes like ClientNetworkManager and ChatClientUI).
8. **Singleton:** The ClientUserManager class follows the Singleton pattern as only one instance of the user manager is created throughout the application, as there is only one list of all the users.
9. **Observer:** This pattern can be seen in the asynchronous communication between the network manager and UI components. The UI components observe changes in the network status and look out for incoming messages, updating the interface accordingly.

III Analysis and Critical Discussion

This section is focused on the JUnit test cases carried out, the expected outcomes and their results. It also discusses the limitations of the proposed solution and areas to improve on.

III.1 Testing

To test our code, we used a combination of unit testing and mocking to test the functionality of the different features implemented. During unit testing, we carried out multiple JUnit tests to test each individual component of the code. Some of these methods include testUserLogin(), testBroadcastMessage(), testConnection() which are designed to test the login portal, message broadcast and client-server connection respectively.

Mocking is a technique used in testing to create mock objects that mimic the behaviour of real objects. We used the Mockito library to implement this for the Server and ClientHandler classes to test the code in isolation without having to rely on external dependencies.

This testing methodology helps identify defects in the early development process and provides quick feedback on code. It also ensures thorough testing of the code's functionality while maintaining isolation and helps create repeatable test scenarios. [4]

Test No.	Test case	Aim	Input	Expected output	Outcome
1	testGuiDisplayed	This test ensures that the GUI of the client is properly displayed.		Chat client GUI displayed correctly.	Passed
2	testConnection	This test verifies if the connection to the server is successful.		Client is successfully connected to the server.	Passed
3	testUserLogin	This test checks if the user can login successfully .		Client is successfully connected and gets a welcome message.	Passed
4	testUserDisconnection	This test checks if the user can be removed from the server successfully.		Client is successfully disconnected from the server.	Passed
5	testBroadcastMessage	This test checks that the message is broadcasted to all users.	Hello everyone	Hello everyone	Passed
6	testSendPrivateMessage	This test checks that a private message is sent correctly to the specified user only.	@bhavesh Hello	Message is only displayed in the sender and recipient's GUI.	Failed
7	testChatHistory	This test is for verifying that the chat history contains the correct number of messages and that they are all present in the server logs as well as the GUI.		All messages are correctly displayed and present in the chat history.	Passed

III.2 Limitations

The solution that we developed implements most of the required features, however, it has some limitations that should be discussed. The first limitation is that users cannot have the same username, and if a new user tries to connect to the server with an existing username, it cannot establish a connection. Another limitation is that when the client-side fails to establish a connection with the server, the code terminates abruptly without any error message which means the user isn't informed of what went wrong. This can be considered an error handling limitation. To improve the solution, we can focus on ensuring a seamless user experience through the GUI and appropriate error handling methods.

IV Conclusions

To conclude, the development of this application was challenging but presented us with numerous learning opportunities and helped us gain a strong understanding about networking in Java and how to implement a client-server application. It helped us learn about Java sockets, network APIs and various useful GUI functions that can help us build a user-friendly platform.

We believe that this report has successfully encompassed the crucial milestones of the development process, the design and implementation of the actual code and a collection of test cases that were carried out. We hope that it highlights our thought-process and skill that went into building the application as well as our knowledge about Java and programming constructs which helped us break the problem down into smaller chunks to solve it in units. All the group members contributed in coming up with the solution, working on separate parts of the code, demonstrating parallel programming and atomicity. We also demonstrated modularity in the code through the implementation of all the features, that exist as separate methods but work together as the application takes shape.

However, our solution may have limitations due to time constraints and gaps in knowledge. We aim to take the knowledge gained through this project to address and overcome these limitations to improve usability and efficiency of the group-based client-server project and also apply it to future projects.

ACKNOWLEDGEMENTS

I would like to thank our tutors, Dr. Taimoor Khan and Dr. Markus Wolf for their guidance and support in the development process and also suggesting improvements to make our project better.

REFERENCES

[1] GitHub - kunz398/Multiple-Chat-Clients-RMI-Java: making a chat application using Java RMI(java remote method invocation) — github.com. https://github.com/kunz398/Multiple-Chat-Clients-RMI-Java. [Accessed 29-03-2024].

[2] Java Networking - GeeksforGeeks — geeksforgeeks.org. https://www.geeksforgeeks.org/java-networking/?ref=gcse. [Accessed 25-03-2024].

[3] lpdwww.epfl.ch. http://lpdwww.epfl.ch/upload/documents/ids/exercises/Ex3/Ex3.pdf. [Accessed 29-03-2024].

[4] Richard H Carver and Kuo-Chung Tai, *Modern multithreading: implementing, testing, and debugging multithreaded Java and C++/Pthreads/Win32 programs*, John Wiley & Sons, 2005.

[5] James William Cooper, 'Java design patterns: a tutorial', (2000).