Group 62:
GUI interfaced Client server group chat

Matthew James [1]

Other group members:
Andrei Iulian Niculita[2], **Thomas O Ibitoye**[3] **Alexandru I Toia**[4] **Behar Shurbi**[5]

Abstract. This project is a Java client-server group chat featuring a graphical user interface, a database, and Junit testing. The programme enables clients to communicate in real-time via the server. Clients can send messages to all members of the group, send private messages to particular members of the group, update their local list, and browse the local list. The coordinator can delete clients and check to see if everyone is still active.

1 Introduction

This project is a client-server group chat created in Java with a GUI, Database, and Junit testing. The application allows clients to communicate with each other in real time through the server. Clients can send messages to everyone in the group, send private messages to specific clients, update their local list, and view the local list. The coordinator has the ability to remove clients from the application and verify their activity status. The application operates on a client-server architecture, where clients establish connections with the server and communicate with each other indirectly through the server. This allows the server to manage the chat and ensure that all clients receive messages in a timely and efficient manner. The GUI provides a user-friendly interface that allows clients to easily send and receive messages. Messages are displayed in real-time, and the user can scroll through the chat history to view previous messages. The Database is used to store messages, user information, and other data related to the chat. This allows two or more programs to communicate with each other. The Junit testing ensures that the application functions as expected and catches bugs early in development. This feature enables developers to create test cases that verify the performance of various components of the application and ensure that modifications do not disrupt the existing functionality. In this report, we will be reviewing the design and implementation of the client-server group chat including, design patterns, unit tests, and components. We will also analyse and discuss the results of our code, detailing how we achieved modularity, our different forms of fault tolerance (human and computer), and testing. Finally, our limitations and potential areas for improvement will be the topic of our discussion.

[1] School of Computing and Mathematical Sciences, University of Greenwich, London SE10 9LS, UK, email: mj8857d@gre.ac.uk
[2] School of Computing and Mathematical Sciences, University of Greenwich, London SE10 9LS, UK, email: an5225r@gre.ac.uk
[3] School of Computing and Mathematical Sciences, University of Greenwich, London SE10 9LS, UK, email: ti0257t@gre.ac.uk
[4] School of Computing and Mathematical Sciences, University of Greenwich, London SE10 9LS, UK, email: at7931f@greenwich.ac.uk
[5] School of Computing and Mathematical Sciences, University of Greenwich, London SE10 9LS, UK, email: bs4514s@greenwich.ac.uk

2 Design/Implementation

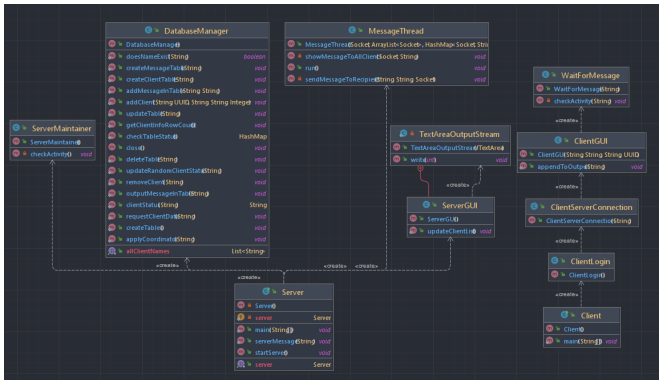

Figure 1. class diagram

During the review and planning phase of our project, we noted that there were three main technical requirements, each with a variety of subtasks needed to meet them. The first requirement is group formation, connection, and communication. The second requirement is group state maintenance, and the third is the coordinator selection process. In this section of the report, we will discuss how all three of these requirements were met, the classes and design patterns used, and how these requirements were tested. Afterwards, we will provide an overview of the entire project.

2.1 Group formation, connection, and communication

This requirement involves enabling a group of clients to connect to a server and communicate with each other without errors. To fulfil this requirement, we divided it into several subtasks. The first task was to create a server for clients to connect to. We accomplished this in our Server class, which opened a server socket on our chosen port and used the localhost as the DNS address. Additionally, we implemented a continuous loop in the server to wait for clients to connect and add their sockets to an Array List. We also made the Server class a singleton to ensure that only one instance of the server runs at

any given time. This was necessary to avoid conflicts with the server socket, which could result in a bind exception, as well as the other classes that the Server creates, such as the database manager, server maintainer, and various message threads created for clients. Now that we have our server, we need a class for the client to connect from. Our client class creates a GUI called "client login" that enables the user to enter a name. This name acts as a unique ID as the program checks in the database through the database manager if the name already exists. When a client has chosen a name, they are sent to a new window that allows them to connect to the server by entering the IP and port of the server. The program ensures that they will enter the correct server information. Once they have entered the correct server information, they will be taken to the final GUI, which is the group chat itself. They will be given a terminal-like screen where they can view messages, a text box in which they can write messages, and a button to send the messages in the text box. Lastly, there is a button to exit the program. The client should be able to exit the program at any stage without causing any errors as the connection only occurs at the last stage. It is only when the server socket accepts the newly created client socket that a server and message thread are created. But the message thread is responsible for sending messages between clients and any commands that require a client's name as input. It is also responsible for deleting client information when a socket exception occurs meaning that a client has been disconnected. When a client socket is created, an output and input stream is created for it. We can capture the output stream and place it in a PrintWriter which converts a string into bytes that can be sent using println to the same socket's input stream, which would be in the client's message thread class that the server creates. This input stream is fed to an InputStreamReader and BufferedReader which turn the bytes back into a new line of string that can be read. The message thread takes the client's message and runs a series of if statements that detect whether the input is for a command or just a broadcast message. If it's the latter, then the message thread will send the message to every client's message table within the database so it can be displayed on their GUI when the time is right. If it's the former, the message thread will act accordingly. For example, to send a private message, the client must enter "/whisper [recipient name] [message]". The message thread will read this string and separate the recipient's name and the message into separate strings and send them to a method that will only send the message to the recipient's message table. In this way, the message thread acts as an observer design pattern as it is designed to observe changes in the state of another object and then react accordingly. As a class that waits for a message, it is essentially acting as an observer that is waiting for a message from another object. Once the message arrives, the class can then update its state or perform some other action based on the contents of the message. If the object, which in this case would be the client, were to disconnect, then the message thread object for that client would also be deleted.

2.2 Group state maintenance

Our group has implemented two classes for group maintenance: the "WaitForMessage" class, which each client creates, and the main class the "ServerMaintainer". The main function of the "ServerMaintainer" class is to check if all connected clients are active. It does this by comparing each client's local list to the server's list. If they match, it means that the client has been responding to all the server's pings via the "update list" button when a client joins or leaves. If a normal client is deemed inactive, a notification will be displayed in the server terminal indicating it. If a coordinator is deemed inactive, their coordinator status will be revoked, and another member will be chosen at random to be the coordinator. That member will then be notified. This class is a form of polling mechanism, which isn't a design pattern but is a common technique used in the industry. A polling mechanism involves repeatedly querying a system or data source at fixed intervals to check for changes or updates. In our case, it repeatedly queries our database for inactive members. The "WaitForMessage" class is a thread that checks a client's message table every five seconds for new messages. It retrieves the message(s) from the table and posts them in the client's GUI.

2.3 Coordinator selection

This coordinated status is given to the first client to join the server. This is done by checking the number of existing members connected to the server via the database. If there is only one client record in the database, then that client becomes the coordinator. We implemented it this way because if there's only one client in the server and that client leaves, then the next client to join will still become the coordinator. If the coordinator quits the program, leaving behind three other clients within the server, the system will detect the socket exception within the message thread. Upon detecting the exception, the system will check if the disconnected client was the coordinator. If this is the case, the database manager will update the status of another client from a client status to a coordinator status. The newly appointed coordinator will then receive a message notifying them of their status update. If some errors were to occur in which a coordinator had their program disconnected abnormally, so they were removed from the database, but their socket connection was not disrupted, the server would be left without a current active coordinator. To combat this, we implemented a small if statement in the server maintainer that ensures if there is no coordinator in the database, out of the list of active members, a random one will be chosen to be a coordinator. Even though the chance is rare, there will be a new coordinator at least every two minutes. If a coordinator disconnects properly by force shutting down the program or using the exit button or typing /logout, then the coordinator status transfer will be automatic.

2.4 Database

Our system incorporates a JDBC connection to access the Client_Server.db database. To facilitate access to the database, we have implemented a database manager class that acts as a proxy for the database. The Proxy design pattern, a structural design pattern, enables an object, known as the proxy, to serve as a replacement for another object, known as the subject, and manage access to it. In our system, the database manager class serves as the proxy while the database is the subject. Access to the database is restricted to the database manager class alone, and all other classes must use a database manager object to access the database. The database manager class comprises various methods containing Data Definition Language (DDL) and Data Manipulation Language (DML) SQL commands. Upon creating a database manager object, a new table called the "client info" table is generated to store all relevant information about each client. As soon as a client object is created, and all the relevant information is obtained from the client and the system, the client is added to the client info table. Additionally, a local list table and a message table are created specifically for the client, which will be manipulated by the database manager over the course of the client's connection. When a client disconnects, the message

table, local list, and record in the client info table are all deleted as they are no longer required or appropriate to retain.

2.5 Junit test

JUnit tests are a crucial tool in software development because they can automate the testing process, catch bugs early, and ensure that existing functionality is not broken. Our team recognized the importance of JUnit tests and created a series of tests to check if the underlying parts of our code were functioning as expected. For instance, we tested whether the server GUI met the requirements we had set, such as being unable to be edited. Additionally, we tested whether clients were being added to our database correctly and whether the row count provided the desired outcome. We focused our testing efforts on the database manager, as a lot of our implementation relied on the accuracy of the database information. By using JUnit tests, we were able to detect mistakes early on that would have otherwise required us to go through the entire process of server and client creation to detect. The tests allowed us to test new methods as soon as they were created within different scenarios, making them an invaluable resource for our team. Overall, the use of JUnit tests improved the quality and reliability of our code, and we highly recommend incorporating them into any software development project.

3 Analysis and Critical Discussion

For this part of the report, it will be easier to break up our implementation into two sections, the client side and the server side. This is because our program as a whole is responsible for running two different applications that run on different instances that cannot see each other, the server side run by the server class on the client side run by the client class. in this part of the report will be discussing in detail the results of each part of the codes execution and the various threads that are run in tandem with each other. We will also discuss modularity, fault tolerance, testing, components and limitation of our implementation.

3.1 Server side

The Server class is the foundation of all server-side code. Upon execution, it creates three classes: the Database Manager, the Server Maintainer, and the Server GUI. The GUI is created first, followed by the instantiation of the server instance, which is used to run the start server method. This method creates the server socket, displays the necessary information for clients to connect, and runs a continuous loop that waits for clients to connect. Originally, the server terminal was used to print all information. However, we altered the ”write” method in the output stream to display print statements in the Server GUI’s terminal instead of the command line terminal. The Server GUI also includes a section that displays current members who have connected to the server. A static method in the Salvage UI is executed at various points on the server and client side, which appends a list of all current members, active or inactive, to the Server GUI’s list. The current coordinator will be designated with the coordinator status next to their name in brackets. The Database Manager class connects to the actual database file upon creation. It then deletes and recreates the main table, ”client_info,” which contains all client information. This ensures that information from prior servers that was not deleted is removed, and auto-incrementation on any tables is reset. The same process of deleting and recreating tables is used for all tables in our database. For example, if a client named Matt is added to the client_info table, the local list and message table are created for them using their name as their unique ID. This method ensures fault tolerance if the server abruptly shuts down and restarts, as the client info, local list, and message tables are reset, preventing any contradictions.

Lastly, the server maintainer class will be executed. This class will be used to check the status of all clients and ensures that the coordinator is always active. It also ensures that there is always a coordinator present in the server and if there isn’t then one is chosen randomly. This is an example of fault tolerance; however, it can only check every two minutes which is a limitation as it could potentially mean that the server can go without coordinator maintaining it for at least two minutes instead of 1 being chosen instantly. This can only happen if a coordinator leaves the server in abrupt manner failing all of the check client status in various socket exceptions.

3.1.1 Modularity

When considering modularity, every class on the server side is modular. All classes within the server side are highly cohesive, meaning they work together to build a well-functioning server. For instance, the server class starts the server, the server maintainer maintains the state of the server, and the Server GUI gives the server a graphical interface and a list of current members. The classes are also low coupling, meaning that they do not depend on one another. Although the server class creates all of the classes, the server does not necessarily need these classes to connect to the server. Similarly, the server maintainer does not necessarily need a server to run; all it needs is access to the database via the database manager. The Server GUI also does not necessarily need the server to be running; it only means that the Server GUI will not have any messages in the GUI or current members connected. Regarding encapsulation, for the sake of the JUnit task, some of our modules had to be made public, such as the start server method in the server class. However, important parts of our classes are kept private. The only exception to this is the database manager, which has all of its methods public and static, as it is used as a proxy for the other modules to access the database. Generally, the parts that are public and static and our modules are only that way because we wish to reuse them in other parts of the code. This is mainly evident in the database manager. However, methods such as the server message in the server class are used to send messages to the server and by extension the Server GUI. While all these classes take something from each other, they are not solely dependent on each other’s execution. This makes it easier to test them in isolation, except for the server maintainer, which needed the server and clients to be connected to be efficiently tested.

3.2 Client side

For client class is the starting part of all the client side code when the client class is an empty class the simple calls the client login class when it is executed. The client login class is a GUI that is responsible for collecting the users name. This name however is very important as will act as the users ID for the local list table and messages table. To prevent conflicts within the database we have to ensure that every client in our server has a unique name therefore, when a user types in the name and click enter the database manager will check if the name entered is already in the client info database, and if it is, a warning message will appear and the user will be prompted to enter a new name. A similar warning will also appear if the user enters no name. Once a user has entered an appropriate name it will

be prompted to connect to the server by entering the IP address and port of the server, this would be done in the client server connection class. We also have to ensure some fault tolerance here as well. if the client enters the incorrect information and try to connect to the server it will result in an unknown host error, therefore we have ensured the use has to enter 127.0.0.1 for the IP address and 8080 for the host. We also throw warning if they don't fill in all the information needed. When the user enters the correct information to connect to the server and presses enter, they will be redirected to the client GUI. Here, they will be able to send and receive messages and enter commands. To do this, the client must enter the message in a text field and press send. When the client enters the client GUI, they will be connected to the server using the information gathered at the beginning. When the server socket accepts the client socket, a message thread object is created for the client. As mentioned earlier in the design and implementation section, along with the socket connection, the client class also creates a print writer output stream. This stream is used to send messages and commands that the client enters to the message thread the server creates via their socket. The message thread then receives this message and, based on its content, will perform different actions. If the client enters "/logout," a socket exception will be thrown, deleting all the client's information from the database, all the socket information in the message threads themselves, and closing their socket and program. If the client input starts with "/whisper," the input is broken down into relevant parts and sent to the send message to recipient method, which checks if the recipient's socket is in the client's name list and sends the message to the recipient's message table. If the recipient does not exist, a message will be sent back to the client stating that the recipient does not exist. This is similar to the remove command, which can only be performed by coordinators. If the coordinator needs to add types in "/remove" followed by the recipient's name, the name will be found within the client list, and the client socket will be closed, ending their program. "/check" is a command that only coordinators can use to check who is active and inactive on the server. A message will be sent to everyone in the server stating which members are inactive using the check table status method in the database manager. If the client simply enters a message with no command, this message will be sent to all other clients via the show message to all clients, which sends messages to all the message tables of every client in the server. There are also commands that the message thread does not interact with, such as "/requestinfo," which is used to display a client's local list, and "update list" button, which is used to update the local list.

There are two more threads to discuss. The threat client class is a class as created when the client connects to the server. this class was originally used when our project was geared more towards a CLI implementation to ensure that the client did not send empty messages, however now it's used to ensure that if this socket is interrupted and the socket belonged to a coordinator, the coordinator status will change hands and ensure client GUI is closed if the server connection is lost. in the database every client has a client status, which can either be coordinator or client. When a coordinator status is given someone else, the current coordinator will be wiped from the client_info table, then another client will be given the coordinator status, then the previous coordinator will be entered back into the client info table as a client and everybody's local list will be updated to reflect the new change. The last and one of the most important parts of the client side implementation is how client receive messages in their GUI. This is done within a separate thread that is created by the client GUI when they connect to the server, this thread simply runs a database manager method that takes the clients name and uses it output all of the clients' messages onto their GUI and the clear the table. This is done every 5 seconds.

3.2.1 Modularity

In terms of modularity, the client-side modules are highly cohesive as each serves a specific function and gathers information for other modules to use. For example, the client login and server connection modules gather information for the client GUI module, the wait for message thread module is used to send messages to the client GUI, The message thread module receives and processes information sent by the client GUI, This module ensures that the coordinator status is transferred and the client GUI is closed if the server connection is lost. Lastly, the client GUI module allows the client to see and send messages. While different modules on the client side rely on information given and sent by other modules, each module exists independently. However, it's accurate to say that every module depends on a connection to the server and database. We believe these client-side modules are highly reusable and testable because of their simplicity. Most of them either require database access or socket connection. The only one that might be hard to test is the client GUI module, as it depends on the message thread module to send messages, which in turn depends on the server class to add a socket and client name list into the message thread.

4 Conclusion

In conclusion, the client-server group chat project described in this introduction is a robust and feature-rich application that utilizes various programming concepts and tools to provide users with a near seamless communication experience. The use of a client-server architecture, GUI, Database, and Junit testing ensures that the application is efficient, user-friendly, and reliable. The project demonstrates the importance of modularity, fault tolerance, and testing in software development, as well as the benefits of using design patterns to improve code quality and maintainability. While the project has achieved many of its goals, there is always room for improvement, and further development and testing can help to identify and address any issues or limitations such as providing quicker times for sending messages, coordinator transfer and message thread modularity. Overall, we believe this project serves as a good example of how software engineering principles can be applied to create reliable applications that meet the needs of end-users.

REFERENCES

(Horstmann, C. (2016) Core Java Volume I - Fundamentals. 10th edn. New Jersey, USA: Prentice Hall.)

(Evans, B.J., Clark, J. and Flanagan, D. (2023) Java in a Nutshell: A Desktop Quick Reference. 8th edn. Sebastopol, CA, USA: O'Reilly.)

(Deitel, P. and Deitel, H. (2012) Java for Programmers. 2nd edn. Boston, MA, USA: Pearson Education, inc.)

(Evans, B.J. and Verburg, M. (2013) The Well-Grounded Java Developer: Vital techniques of Java 7 and polyglot programming. Shelter Island, NY, USA: Manning Publications Co.)

(Boyarsky, J. and Selikoff, S. (2015) OCA: Oracle Certified Associate Java SE 8 Programmer I Study Guide. Indianapolis, Indiana, USA: Sybex.)