MESSAGING SYSTEM APPLICATION

YAGOL XU CHEN, JUNJIE MA, JAMAL GWARADA, NATASHA BENHAMIN, JESTING BIJU
COMP-1549: Advanced Programming

University Of Greenwich

Old Royal Naval College, Park Row, London SE10 9LS

United Kingdom

Key words: Messaging app design, messenger system, database for messenger.

***Abstract* –** This report analyses how the messaging application is planned, developed, and tested. There is also a discussion on the design patterns implemented. Additionally, it evaluates the performance of the application with different network conditions and the overall messaging experience. The report presents the strengths and weaknesses of the application based on the findings of the testing and ease of use.

I. Introduction

In today's world, network distributed systems are essential for facilitating communication and collaboration among individuals and groups. It enables communication and collaboration among people regardless of their location. With the rapid increase of the internet and advancements in technology, messaging applications have become an essential tool for society. In this project we are implementing an application that allows group-based client server communication. This coursework's objective is to create a system that facilitates communication among group members. To meet this goal, we developed a system that allows members to send and receive messages by using specific commands. The aim is that everyone in the group should be able to send private messages or broadcast messages to every member.

The app is a messaging application designed to send and receive text messages between two or more users. To design and implement the messaging application Java and SQL was used. Additionally, public and private messages can be sent to any connected or non-connected user. There is a coordinator that manages the group chat by adding or removing the user from it. On the other hand, the database is a stateful system that maintains information about all the members and information used by the application. This includes information such as user profiles, available groups, chat history, and message delivery status.

II. Design/ Implementation

The design and implementation of the application will be discussed based on the **Figure 1** (See appendix).

The messaging system is divided into two primary components, the client and the server, which communicate with each other using a client-server architecture. This approach enables the client to request services from the server and receive responses without other clients interfering with the system's performance or security (Michael Siegel, 2003). The client and server classes are separated and only depend on each other to send data through sockets. This connection uses a Transmission Control Protocol (TCP) for a reliable two-way communication between the two systems. To start the communication, a three-way handshake is required. Then, the application can exchange data at the same time (Goralski, 2017).

The "Chat Client" class on the client-side is responsible for managing the input and output of the client. The mediator design pattern is used to separate the dependency between the reading and writing functions. To get a better separation of concerns the "Reader" and "Writer" class are created. This design pattern encapsulates the objects and distributes the work by creating them on a separate thread (Erich Gamma, 1995). It avoids conflicts from processing incoming messages or writing to the server simultaneously.

Once we have successfully connected the client to a specific address, the next step is to establish a server with a specific port and IP address to receive incoming connections. The "Server" class is responsible for accepting incoming connections. Then, it delegates the task of handling new users to the "User Thread" class, which operates on a separate thread to enable concurrent connections (Pankaj, 2022). The "Server" class is capable of reading and writing incoming messages. With the implementation of these classes, we have a fully

functional client-server architecture prepared to support additional features.

The current system is capable of having a bidirectional message exchange, but the server cannot process the messages. To solve this, an application programming interface (API) that interacts with the data sent by the user is used to get data from a database. Additionally, the API works as a façade, it provides a simplified interface for users to interact with the system (Erich Gamma, 1995). This structural pattern allows a more simple access to the system's functionality without requiring the user to have any knowledge of the complexity of the backend. By using an API we can hide and handle the interactions by establishing a set of commands.

The "Database Proxy" class is a key component of the implementation. It is responsible for communicating the "User Thread" with the database. Since it needs to Create, Read, Update, or Delete (CRUD) data in the database, it sends a request to the "Database Proxy". Then the message is processed into a database query. To ensure that there is only one instance of the “Database Proxy” class, we can implement the Singleton pattern. By doing so, it prevents multiple instances accessing the database simultaneously by creating only one instance. Without this feature, it could result in data inconsistencies and other issues. Additionally, by using the "synchronized" keyword, we can control multiple threads accessing the global database resource, ensuring that they do not interfere with each other (Oracle, 2022). Moreover, the Proxy design pattern controls the access of the original object. For example, in a group chat, only the coordinator should be allowed to add or remove users from the group. With the Proxy pattern, before adding or removing a user, the validity of the command is checked to ensure that it is coming from the coordinator. This provides more security and control over the system, ensuring that only authorized users can use specific commands (Ross Harmes, 2008).

The database is responsible for storing data. This includes user information, group chat details, message data, and message states. By efficiently querying and updating this information, the database ensures that the system remains consistent and scalable.

Lastly, the “Coordinator” class is the main handler for all the group chats in the system. If there is an inactive coordinator via the “iterator()” function the coordinator is changed. It implements the Observer design pattern to create a subscription mechanism for incoming users (McDonough, 2017). It keeps a HashSet to track the connected users and remove them when they disconnect. This approach avoids the need for adding timestamps to the database, reducing its complexity. In addition, the Chain of Responsibility design pattern can be used to handle different scenarios on the algorithm “iterator()”. This pattern allows the algorithm to handle multiple cases in a sequential manner and stops if a function doesn't need to be executed. For instance, the algorithm will check if there are active users or if all the coordinators are active.

III. Analysis and Critical Discussion

All the components created in the system such as the client, server, database, and coordinator, are designed to separate the modules to perform different tasks. Then, these components are connected with each other through design patterns or inheritance. This allows each module to be developed and tested separately, without affecting the functionality of the overall system. Additionally, the use of design patterns, such as the mediator, singleton, observer, and chain of responsibility, helps to separate different modules within the system separating the responsibilities of each class. This way, it develops a more flexible approach to maintain, update, and scale the system when needed.

To ensure fault tolerance, we identified potential failure points in the code, such as the API class. For example, we implemented error handling measures for operations like string splitting and indexing. By detecting and handling these errors, it ensures that the server will not crash in an event of unexpected error. Additionally, with the JUnit testing, we were able to catch the remaining errors to solve them in the development process.

The API, Coordinator and Database Proxy class were tested to verify the functionality and behavior of each class.

The API testing handled various operations related to group chat and user management. The tests were designed to cover a wide range of scenarios, including positive and negative cases. The test cases were selectively chosen to ensure a good evaluation of the test. This includes features such as creating and modifying group chats, obtaining user and group information, and managing user interactions in group chats. By having a good separation of classes, it was able to isolate the functionality of the “Coordinator” class and ensure that the methods work as expected.

The database was tested to ensure that it can efficiently manage user information, group chat details, message data, and message states. The tests included inserting and retrieving data from the database, as well as testing its scalability and efficiency for handling large amounts of data.

While the implemented application has many features, there are some limitations. Firstly, it relies on the user skill in using the commands to send and receive data. An intuitive user interface could simplify the interaction between the client and server. Additionally, the current database may not be able to perform well with many users or groups. As the number of users or groups grows, the database's performance may decline significantly. To solve this issue, a solution would be to implement a distributed database system to handle the information across multiple servers (Stribny, 20220) . By scaling horizontally, it would allow a higher workload and fault tolerance system. Furthermore, a caching method in the client and server side could be used to reduce the workload of the server. Each time a query is run, the database needs to retrieve the whole data. Using caching would reduce the computing power needed by only returning the latest information.

IV. Conclusions

Our application has been designed to meet the requirements of a group-based client server communication, allowing users to send private and broadcast messages. The report has a detailed description of the applications design and implementation, including the client side and the server side. By separating the components and responsibilities with the API, Database Proxy and Coordinator class; the architecture of the code is more robust. The strengths and weaknesses of our project have also been evaluated based on the findings within our testing and the ease of use. Overall, the report provides a broad overview of the development and performance of the project, offering insights into the design and implementation of a network distributed messaging application.

V. References

Erich Gamma, R. H. R. J. J. V., 1995. *Design Patterns, Elements of Reusable Object-Oriented Software.* [Online]
Available at:
https://www.csie.ntu.edu.tw/~b98208001/Design%20Patterns%20%20Elements%20of%20Reusable%20Object-Oriented%20Software.pdf

Goralski, W., 2017. *The Illustrated Network: How TCP/IP Works in a Modern Network.* [Online]
Available at:
https://books.google.co.uk/books?hl=en&lr=&id=0IzfDQAAQBAJ&oi=fnd&pg=PP1&dq=how+tcp+works&ots=4-O90eKhv5&sig=BwI4cgiqbN3yuiZX_A2uqUb2bAM&redir_esc=y#v=onepage&q&f=false

McDonough, J. E., 2017. *Observer Design Pattern.* [Online]
Available at:
https://www.researchgate.net/publication/318150145_Observer_Design_Pattern

Michael Siegel, S. M. E. S., 2003. *Context interchange in a client-server architecture.* [Online]
Available at:
https://www.sciencedirect.com/science/article/abs/pii/0164121294900442

Oracle, 2022. *Synchronized Methods.* [Online]
Available at:
https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html

Pankaj, 2022. *Java Socket Programming - Socket Server, Client example.* [Online]
Available at:
https://www.digitalocean.com/community/tutorials/java-socket-programming-server-client

Ross Harmes, D. D., 2008. *The Proxy Pattern.* [Online]
Available at:
https://www.researchgate.net/publication/319724200_The_Proxy_Pattern

Stribny, P., 20220. *Scaling relational SQL databases.* [Online]
Available at:
https://stribny.name/blog/2020/07/scaling-relational-sql-databases/

VI. Appendix

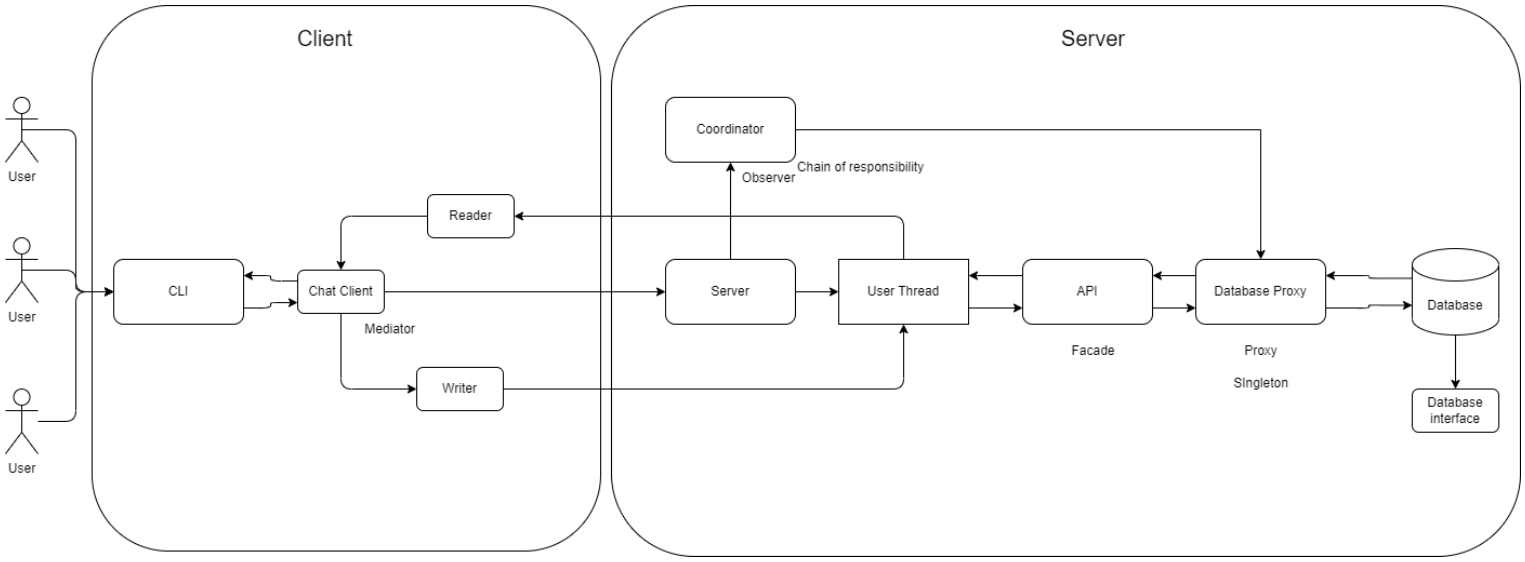

Figure 1