COMP1549: Advanced Programming Report

Chisel Chavush and Rakshit Arora
COMP1549: Advanced Programming
University of Greenwich
Old Royal Naval College
United Kingdom

Abstract-

This project focuses on implementing a Command Line Interface (CLI) based networked distributed system for group-based client-server communication. The system allows new members to join by providing unique IDs, server IP addresses and port, and their own IP addresses with optional listening ports. The first member becomes the coordinator and maintains the state of the group members. New members receive existing member details from the server and notify the group upon joining. If the coordinator is unresponsive, any other active member takes on the role. The system enables private and broadcast messaging through the server, and members can quit using a simple command.

The implementation is in manual operation modes. In the manual mode, the program simulates the tasks with user input. The project is demonstrating modularity using design patterns, JUnit-based testing, fault tolerance, and component-based development to showcase strong programming principles and practices.

Key words: *(Coordinator, CLI, client-server communication, network-distributed system)*

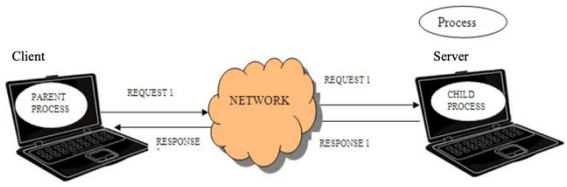

Figure 1:Client-Server Communication System

I. Introduction

In today's climate of fast-moving change and interconnectedness, effective and reliable group-based communication systems are becoming more important for a few applications, such as collaborative projects, remote teams, and massive networks (*Maata,*et al., 2017). In order to comply with programming principles and practices, this project will create a networked distributed system based on a Command-Line Interface (CLI) that allows group-based client-server communication and provides a stable and scalable platform for a variety of use cases such as interdepartmental communication systems.

By making it simple to manage members, and employ message capabilities, the offered communication system will guarantee that the communication process is both secure and efficient, ensuring the chat stays active.(*Sawant, et al., 2013*) New members can easily join to the group by providing Unique IDs, the server IP address and port with the socket connecting through the client's listening ports. The system will automatically designate the first client as the coordinator, who will be in charge of keeping track of the group members' statuses and managing the on-boarding of any new participants.

The project has been written in manual operation mode, to accommodate different user preferences and use cases. In the manual mode, client must input messages to share with broadcast. Also, any client can chat privately. Through this, any client can easily access the list of active members. coordinator has option to manage the entire client lists.

The project was created with strong programming principles and practices that include modularity using design patterns, allowing a clear separation of concerns and simpler code maintenance, to provide a well-structured, maintainable, and stable implementation.

JUnit-based testing guarantees the quality and stability of the application; fault tolerance, enhancing the system's resilience in case of unexpected issues or failures; and component-based development, promoting its reusability and adaptability. The project aims to deliver a comprehensive and efficient group-based communication system that can be seamlessly integrated into various applications and environments by incorporating these principles.

II. Design/implementation

Combining design patterns and concepts has been utilized in chat server implementation to enable concurrency, resource sharing, and separation of responsibilities, resulting in a reliable and effective solution. Several client connections can be managed concurrently using the "Thread-per-Connection" pattern, with a different instance of the "ClientThread" class handling each connection. Using this method, the server can grow in accordance with the number of clients connected, resulting in streamlined communication and responsiveness.

The Singleton pattern is applied to the Server class, which makes sure that only one instance is produced for the duration of the program. This design decision makes it easier for all client threads to share resources, such as a list of clients who are currently active and the coordinator (Figure 3:Coordinator Function). Moreover, the Observer pattern is utilized to keep clients updated about events occurring in the chat, such as users joining or leaving, and messages being broadcasted (Figure 2: Broadcast Messaging). The broadcast() method in the Server class serves as a notification mechanism that efficiently informs all connected clients about the latest events, excluding the sender when specified.

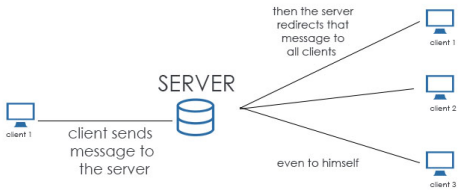

Figure 2: Broadcast Messaging

The Factory pattern is employed to create instances of the Scheduled Executor Service class, responsible for scheduling tasks to be executed in the future.

"Executors.newSingleThreadScheduledExecutor()" method acts as a factory method, generating a new "ScheduledExecutorService" instance with a single worker thread. This approach simplifies the management of scheduled tasks while providing a clean and maintainable code structure.

Encapsulation is a fundamental principle used throughout the implementation to separate the responsibilities of the Server and ClientThread classes. The Server class manages connected clients and the coordinator, while the ClientThread class handles communication with individual clients. By maintaining a clear separation of concerns, the implementation remains modular, making it easier to maintain, extend, and debug.

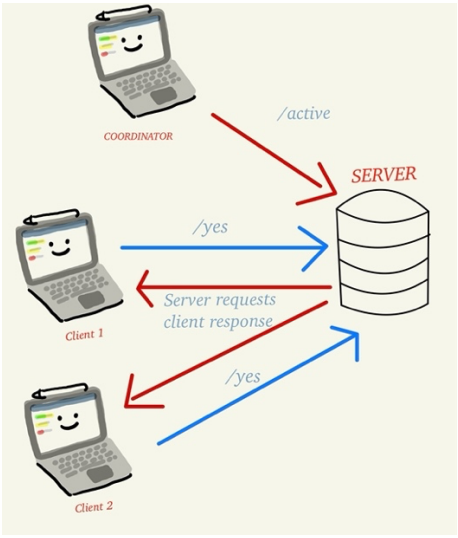

Figure 3:Coordinator Function

To summarize, the development and deployment of this chat server make effective use of a mix of design patterns and methodologies to create a capable, adaptable, and easily maintainable system. By combining these approaches, the server can serve multiple clients at once, providing instant updates, and maintaining a comprehensible code organization that can be easily understood and adjusted as needed (*Harold, et all., 2004*).

The Client class's implementation adheres to a procedural style, with the primary function is to process user input, server connections, and message transmission. The main function is responsible for managing resources such as sockets and input/output streams that can further be enhanced by isolating server connections and resource handling through a separate class.

The "ServerListener" class implements the Runnable interface and is operating on an independent thread to monitor incoming server messages that in turn allows the processing of incoming messages as the user engages with the terminal.

JUnit testing plays a pivotal role in software development, contributing to the assurance of accuracy, dependability, and maintainability of an application. The primary rationale behind employing JUnit tests is the early detection and resolution of potential issues and defects within the code, thereby preventing their escalation into more severe complications. Moreover, JUnit tests function as a form of documentation, elucidating the anticipated behavior and functionality of the application's constituent components.

Another noteworthy advantage of JUnit tests lies in their ability to facilitate code refactoring and modifications. By providing a safety mechanism, these tests ascertain the preservation of the codebase's integrity following any alterations. Ultimately, JUnit testing fosters the creation of high-quality software by endorsing a methodical, test-driven strategy that continuously validates the functionality of individual components throughout the development life cycle (*Xue, et all., 2009*).

The client class JUnit test focuses on “ServerListener” class. The test ensures that the ServerListener is able to receive and correctly process messages from the server. The test class, “ServerListenerTest”, follows a structured approach for testing. It uses the @BeforeEach and @AfterEach annotations to set up and tear down the necessary resources for each test, such as the server socket and the executor service. This approach ensures that each test runs in isolation with a clean environment, making the tests more reliable and easier to maintain.

In the “testServerListener” method, the test creates a client socket to connect to a local server and then submits a task to the executor service to simulate the server's behavior. The task listens for incoming connections, accepts a connection, and sends a predefined message ("Hello, Client!") to the connected client. Next, the test creates an instance of the ServerListener class with the client socket, starts a new thread for the server listener, and reads the received message from the client socket's input stream. The test asserts that the received message is equal to the expected message, ensuring that the ServerListener is functioning correctly.

Same as client class, server class also has few JUnit tests. The first test, “ServerTestCoordinator”, checks whether the server assigns the role of coordinator to the first client that connects. In the setUp() method, a server socket is created, and an executor service is started. The server listens for incoming connections, assigns a coordinator if one does not exist, and welcomes regular clients otherwise. The test asserts that the first client becomes the coordinator, and the second client is welcomed as a regular client. This test ensures that the server assigns the coordinator role correctly and maintains the necessary functionality for client coordination.

The second test, “ServerTestId”, verifies that the server enforces unique usernames for connected clients. During the setUp() method, the server listens for incoming connections and checks whether the provided username is already in use. If the username is unique, the client is welcomed; otherwise, the client is prompted to choose a different username. The test asserts that the second client with the same

username is rejected and receives the appropriate response. This test ensures that the server correctly handles and enforces unique usernames for connected clients.

The third test, “ServerTest”, examines whether clients can successfully connect to the server using an IP address and port. The test creates a server socket and a client socket using the same IP address and port.

The “assertNotNull()” assertion checks whether the created client socket is not null, confirming a successful connection. This test verifies that clients can connect to the server using the IP address and port, ensuring proper server functionality and accessibility. These tests cover important aspects of the server, such as coordinator assignment, unique client IDs, and client connectivity, ensuring the server operates as expected.

III. Analysis and Critical Discussion

The built CLI-based networked distributed system for group-based client-server communication was designed to handle a wide range of use cases while being efficient, scalable, and secure. The system successfully manages resources, connections, and concurrent processes while retaining a modular and manageable code structure by incorporating design patterns such as the Singleton, Observer, and Factory patterns.

The project's usage of JUnit-based testing assists in the early discovery of possible faults as well as the guarantee of the application's quality and stability. This methodical approach to software development encourages continuous validation of functionality throughout the project's life cycle, ensuring the system's reliability and maintainability.

Some elements might be improved further. The client class implementation takes a procedural approach, and refactoring the code to encapsulate the server connection and manage resources in a separate class could improve the system's modularity and maintainability. Furthermore, the testing suite could be expanded to include more functionalities and edge cases, resulting in a more thorough validation of the system's capabilities.

Despite its strengths, the implemented CLI-based networked distributed system has some limitations that should be acknowledged. The coordinator-based approach, while effective in managing group members and maintaining the state of the group, may present challenges in cases of high group turnover or frequent coordinator unresponsiveness. Developing a more sophisticated mechanism for coordinator assignment and management could improve the system's resilience and adaptability to varying group dynamics.

The use of manual operation mode in the project, while allowing for flexibility in accommodating different user preferences, may also introduce potential inefficiencies or human errors in the system. Automating certain tasks or integrating an intelligent user interface could enhance the system's usability and performance.

Furthermore, the security and privacy issues that client-server communication systems naturally raise are not explicitly addressed by the current implementation. Encryption, authentication, and access control mechanisms could be implemented to greatly increase system security and safeguard users' private data.

As a result, the CLI-based networked distributed system for group-based client-server communication exhibits a solid and effective solution that makes use of sound programming principles and practises. To ensure the creation of a comprehensive, secure, and adaptable communication system that can be seamlessly integrated into a variety of applications and environments, it is crucial to be aware of and address its limitations.

IV. Conclusions

In conclusion, the CLI-based networked distributed system for group-based client-server communication provides an effective and scalable solution for various applications and use cases. By adhering to strong programming principles and practices, such as modularity using design patterns, JUnit-based testing, fault tolerance, and component-based development, the project showcases a well-structured and reliable implementation.

The system is designed to accommodate diverse user preferences through its manual operation mode and supports efficient communication with features such as private and broadcast messaging, dynamic coordinator management, and seamless group membership updates. However, it is crucial to acknowledge the limitations identified in the analysis and critical discussion sections, such as the procedural approach in the client class implementation, potential inefficiencies in manual mode, and the need for enhanced security and privacy measures.

Future work on the project could focus on addressing these limitations by refactoring the client class implementation to enhance modularity, introducing automation or intelligent user interfaces to improve system performance, and incorporating encryption and authentication mechanisms to ensure secure communication. By continually refining and expanding the system's capabilities, it can evolve into a comprehensive and adaptable communication platform suitable for a wide range of applications and environments.

The CLI-based networked distributed system for group-based client-server communication ultimately shows the potential to develop effective, dependable, and scalable communication solutions by abiding by sound programming principles and practices. The project provides a valuable foundation for future developments in the area of networked distributed systems as technology develops and there is an increasing need for efficient communication systems.

Acknowledgement

We would like to express our special thanks of gratitude to our lecturers Dr. M. Taimoor Khan, Dr. Markus Wolf and who gave us the golden opportunity to Dr. Harry Triantafyllidis work on socket programming and advanced feature of Java.

References

Calvert, K.L. and Donahoo, M.J., 2011. TCP/IP sockets in Java: practical guide for programmers. Morgan Kaufmann.

Harold, E.R., 2004. Java network programming. " O'Reilly Media, Inc.".

Kalita, L., 2014. Socket programming. International Journal of Computer Science and Information Technologies.

Maata, R.L.R., Cordova, R., Sudramurthy, B. and Halibas, A., 2017, December. Design and implementation of client-server based application using socket programming in a distributed computing environment. In 2017 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC). IEEE.

Sawant, A.A. and Meshram, B., 2013. Network programming in Java using Socket. Google Scholar.

Xue, M. and Zhu, C., 2009, May. The socket programming and software design for communication based on client/server. In 2009 Pacific-Asia Conference on Circuits, Communications and Systems. IEEE.