COMP1549 – Advanced Programming

Abstract

The project was designed and implemented to allow a network distributed system for group-based client-server communication catering to both Command-Line Interface (CLI) and Graphical User Interface (GUI) interactions. Through utilizing design patterns, implementing JUnit tests, including fault tolerance mechanisms, and integrating with a Version Control System, our system is scalable, reliable, and maintainable. The project's key results include seamless communication between multiple clients and a central server, dynamic assignment of coordinator roles, uninterrupted communication (in the case of member departures), and great user interaction through CLI and GUI interfaces.

Introduction

The project was designed in two parts, the first centers on client communication, and the second focuses on server communication The decision to separate classes was to ensure that the project achieves modularity and separation of concerns; is scalable and flexible; the code is reusable and testable; as well as adhering to the Single Responsibility Principle (SRP) by using encapsulation. This is to ensure that our design remains scalable, reliable, and maintainable.

Git repository URL:
https://mv5742c@dev.azure.com/mv5742c/COMP1549%20Coursework/_git/COMP1549%20Coursework

Design/Implementation

To achieve modularity and separate concerns, the communication processes were divided into distinct components, such as "Client," "ClientMessagingHandler," "Server," and "ServerMessagingHelper," which allowed each class to focus on specific aspects of functionality.

For instance, the "Client" class connects the client to the chosen IP address, the "ClientConnectionHandler" manages connection and checks if it has been established and initializes the input and output streams for communication and the handling of client messages by creating the instance of the "ClientMessagingHandler" class, while the "ClientMessagingHandler" handles message reception and transmission, differentiates the types of messages (for example if it was a broadcast or direct message).When looking at the "ClientUtils" class, it provides the menu and keeps track of time for the client side. Finally, the "CoordinatorHandler" checks every 20 seconds to check the active group member details.

Looking at the server side of the project, the "Server" class detects incoming connections and creates threads to handle client requests, while the "ServerMessagingHelper" helps forward and route messages within the server side, and sends group details coordinates messaging, and updates clients about who the current coordinator is. The "ClientHandler" class sends requests to gather group details and coordinates who the next client is, if the current coordinator leaves the server, then it assigns a new coordinator. The "ClientServerConnectionHelper" class handles the disconnection of a client by closing the socket object from the UserDetails object from the server and closes it, and then removes the UserDetails object for the specified user ID from the client_details map. In the "ServerUtils" class it checks what type of message is being sent, adds client details to a Hash Map, and prints the date and time and client id. Finally, for the "UserDetails" class it is used to indicate if the client is a coordinator or not.

The classes from both the server and client side were created to be scalable and flexible, the modular design was a key aspect of our development, as it allowed for easy scalability, as new features or improvements could be added to individual components without impacting the entire system. For example, additional functionality like data validation was incorporated at a later stage, which can be seen in the "ClientMessagingHandler" class which checks if the message is being sent to the right client. While separating concerns allowed for flexibility in making changes or updates to specific functionalities without disrupting the overall architecture. For example, the ability to modify message routing logic in "ServerMessagingHelper" class without affecting other components.

To achieve code reusability and testability, each component was designed to be reused across distinct parts of the system or even in other future projects. For example, the "ClientMessagingHandler" class (if needed) can be reused for handling messages in other client applications. While separating communication logic into distinct classes was

vital to improve testability, as individual components could be unit-tested in isolation ensuring that each part of the system behaved as expected and this overall facilitated easier debugging and maintenance, this implementation can be seen by the unit testing that has been done for this project. Finally, to improve encapsulation and follow industry best practices, the communication responsibilities were divided into separate classes. Each class encapsulates its state and behaviors, which adheres to object-oriented design principles, while also following best practices such as the Single Responsibility Principle (SRP), ensuring that each class has a single, well-defined responsibility, in the case of the "ClientHandler" class, this can be seen because it has a well-defined single responsibility of assigning a coordinator.

Following the project guidelines carefully, all the implementation requirements were achieved to a high degree. The requirement to have a unique ID was achieved by creating a HashMap ("client_details") which is initially empty in the "Server" class, this class stores information about connected clients, and the key for each entry in this HashMap is an Integer representing a unique client ID, ensuring that each client has a distinct ID within the project. The requirement to assign coordinator can be seen when a new client connects, "ServerUtils.add_client_details" method (in the "Server" class) checks if the "client_details" HashMap is empty. If it is, then the new client becomes the coordinator, and their ID is stored in the "current_coordinator" variable (in the "Server" class). The newly assigned coordinator is informed of their role using "ServerMessagingHelper.send_new_coordinator_info" (in "ServerMessagingHelper" class).

Another requirement is that the client should be able to request group details such as the coordinator's ID and the client details. This was implemented in the "ServerMessagingHelper.redirect_to_correct_message_routine" method in the class "ServerMessagingHelper", where the client is able to request the information by sending a message "act-grp-details", then the request is taken by the server and it calls "ServerMessagingHelper.send_coordinator_info" method to send a message indicating the current coordinator's ID to the client, and a formatted string list containing the ID, port, coordinator status, about teach client. Another requirement was that clients were able to privately message each other, and to broadcast messages, this is handled based on their format, so messages starting with "brm-" are treated as broadcast messages and are sent to all clients using "ServerMessagingHelper.send_to_all_clients" method (in ServerMessagingHelper" class), whereas the messages starting with "dm-" are treated as direct messages and are sent to a specific recipient using the same method.

To handle the requirement for a client leaving, the class "ClientServerConnectionHelper"' is used, specifically "ClientServerConnectionHelper.disconnect_routine" method. This method handles the closing of the socket which removes the user details from the server). requirement to automatically assign a new coordinator has also been achieved. As seen in the "Server" class in the method "ServerMessagingHelper.redirect_to_correct_message_routine", when the current coordinator leaves, it checks if the user being disconnected is the coordinator by verifying the "is_coordinator" flag in the "UserDetails" object retrieved from the "client_details" HashMap. It then iterates through all connected clients (skipping the current user), and reassigns the first client as the coordinator, which is done by removing them from the "client_details" HashMap, setting the "is_coordinator" to true, and then updating the current_coordinator variable to store the new coordinator's ID.

The requirement for uninterrupted communication is also achieved as the server continues even when the clients disconnect and does not disrupt communication for the remaining members. Lastly, the requirement for providing a record of communication is achieved as the "ServerUtils.console_output" method in ("ServerUtils" class) logs messages sent by members.

In conclusion, both client and server communication were built with an overarching goal to enhance modularity, scalability, flexibility, code reusability, testability, and encapsulation, and to adhere to a robust and maintainable client-server communication system. As the objective was to fulfil the requirements, the objective was met to a high degree. Fault tolerance was applied throughout the code, and unit testing was completed to ensure that the program was executed as intended. The use of Version Control System was applied, as it guaranteed the code changes were tracked and guaranteed good communication throughout the project.

Analysis and Critical Discussion

Throughout the code, the implementation of fault tolerance has been embedded, and when executing unit tests to each of the classes to check for errors, adjustments were made as necessary. This was to ensure that service complied "with the specification in spite of faults" An example of this is for the class "ClientServerConnectionHelper" class, we used

the Arrange-Act-Assert pattern to assess that the code works as intended by performing a unit test with “test_disconnect_routine” method. The user_id was given, and the client_details map was created, then the disconnect_routine method was used. Finally, checks were made to ensure that the client was disconnected successfully and there were no issues.

Within the design, although the code was optimized to achieve modularity, some improvements could have been made to further optimize the code. An example of this is through breaking the classes and the methods within them into more getters’ and setters' components, which would help break down the problem further into different sub-problems. This would have allowed the project to be separated further to solve the problem independently, therefore allowing modular understandability, and continuity for future developments. However, this would have increased the complexity of the program and would have made it difficult to keep track of the different classes and methods. Furthermore, this could lead to an overhead performance as each module function call may add to the overall execution time of the project. Therefore, the system was balanced to include modularity but also to reduce the complexity and the performance overhead.

Reviewing the coordinator client in the project, a minor improvement that could be improved on is the visual appeal of the Command Line Interface output, as currently if the coordinator does not send messages, they are bombarded with messages checking on the active users, and this means that the CLI output may be visually unappealing, an improvement for this may require a separate CLI for the coordinator’s messages and a separate CLI for the periodic checks. However, as the benefit of this was only visual and as the implementation of this may unnecessarily hinder the client’s ease of use, the consensus was to keep the system as it is.

Furthermore, the implementation of exception handling could have been further improved. Currently the exception handling generally only prints stack traces, these may not be useful when clients face errors while using the messaging server and could be further improved by logging more meaningful information to handle the error. An improvement that should also be made is with IP/Port inputs, currently it is hardcoded and only allows specific ports to be used so that the program does not cause an exception. This would need to be adjusted if multiple instances of the server were required.

Although the tests are functioning, and are modular, these can be further modularized to check every method within the class and could inspect each aspect of the class further. This may improve the overall code, as it would mean that the code may be more readable and identifying the issues within the code may be easier.

Finally, another improvement that could have been made is with messaging, currently the clients can directly message themselves. This may not be a major issue, but it was not the design aspect that was originally planned for. And if the client messages a user that does not exist, it does not inform them of this. Which although is a minor issue, this could mean that on the rare occasions that the client mistakenly sends a message to a client that does not exist it is not reported by the system.

Conclusion

In conclusion, the project aimed to develop a network distributed system for group-based client-server communication, that catered to Command-Line Interface interactions. Significant requirements were met, such as seamless communication between multiple clients and a central server, dynamic assignment of coordinator roles, fault tolerance mechanisms for uninterrupted communication, and enhanced user interaction through CLI interfaces. The design is modular, scalable, flexible, reusable, testable, and encapsulated, all aligning with industry best practices and principles such as the Single Responsibility Principle (SRP). While the project requirements have successfully been met to a high degree, there are areas for further improvement, such as further optimizing code complexity and enhancing exception handling. Throughout the project, fault mechanisms were embedded, thorough unit testing was conducted, and Version Control Systems were used for effective collaboration and code management. By addressing the areas for improvement the code can be used as a solid foundation for future developments and adhering to industry standards.