Multi-Client Communication System Report

Hadil Bati, Yuhaï Namugunga, Anum Sheikh, David Morgan and Mohamed Saif Meeran

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

Abstract— We have developed a multi-user chat application using Socket Programming and implements multi-threads which work simultaneously. The Client threads are initialised and stored within a thread List Array List.

Keywords— client, server, server threads, coordinator, message

Introduction

The aim of the project follows the creation of a Command Line Interface (CLI) based networked distributed system, which allows for group-based client-server communication. The way in which the group-based communication system would work, follows the requirements that joining members would enter their unique IDs, Port number that the Server runs on, as well as an IP address. These would confirm as login essentials to allow the users to connect within the communication server. Following requirements that we implemented included all user IDs are unique and therefore the same user ID cannot be used twice. Furthermore, the first connected member would be assigned as coordinator, however if they choose to leave, the role is reassigned to the second member within the user list and all current members are alerted that the coordinator has been updated. To create the foundation of the program, we created several classes such as Client, Server, as well as ClientThreads and ServerThreads which are inherited from the client and server. Within these classes, we used Socket programming and different Array Lists to establish the Server Sockets that the clients would connect to, and all their stored threads. Within this report, we breakdown the stages that were encountered when designing and implementing our program features, as well as all analysed testing and fault-tolerances we took into consideration, when faced with the results of running the program to ensure an optimised version of the system.

I. Design/Implementation

A. Creating Server/Client:

The first step to create the multi-client chat system, was to create the Server class. The Server was initialised by creating the ServerSocket with the assigned Port number and IP address which the client will use to connect. The reason why we assigned a specific Port and IP address is due to the requirement that they need to be the same for all clients as a form of error handling in development. We further initialised two array lists, one is the “IDArrayList” that is used to store the list of userIDs that would be used to connect to the Server. The ArrayList was the best data structure to use, due to it being dynamic, which allows it to be resized. This was essential because clients can join and leave at any moment. Additionally, the elements are accessed by index, meaning that if a specific client exits, their ID can be erased from the ArrayList by finding it's index without having to loop through to find it like how a linked list would function.

When the server class is run, it displays the message “Waiting for Client” until the client socket is accepted. The Server then uses the BufferedOutputStream to print out statements requesting the client to input their ID, Port Number and their IP Address. All inputs are read with the use of an InputStreamReader. The combination of an InputStreamReader and the BufferedOutputStream are coefficient when used together, as the BufferedReader read the text from the InputStream and therefore buffering the characters so to provide for the optimised reading of all characters, lines, or arrays.

We used Telnet which is a protocol used on a Local-Area-Network to establish bidirectional communication via a terminal, therefore being ideal for this application as we are using the CLI. The Server class has a series of while loops that are used to ensure that the client’s inputs fit the criteria required by the Server, and if they do not fit the requirements, then the Client would be notified of that and further asked to enter new inputs fitting the necessary parameters. If the Client enters an ID, that has been used by another client, there is a measure of error handling to prevent them from being duplicated. Furthermore, if a user enters the wrong Port or IP address, they are required to try again.

Within the Client class, we have instantiated the Socket class and initialize the connection to the Server bypassing the required IP address, the specified Port as well a unique ID. There is a level of error handling for those functions. The Client class has an imported Scanner that is used to get an input from the user, followed by sending their data to the Server using a PrintWriter object.

Furthermore, we created a ConcurrentHashmap within the Client class for the activeClients. This is because the ConcurrentHashmap is thread-safe, meaning that it allows for multiple threads to operate on a single object without any further complications. This aspect was key within the implementation of our system, because when the user/Client connects to the Server, their client thread is started, therefore with a multi-threaded chat system, the client threads are required to operate together simultaneously without any complication.

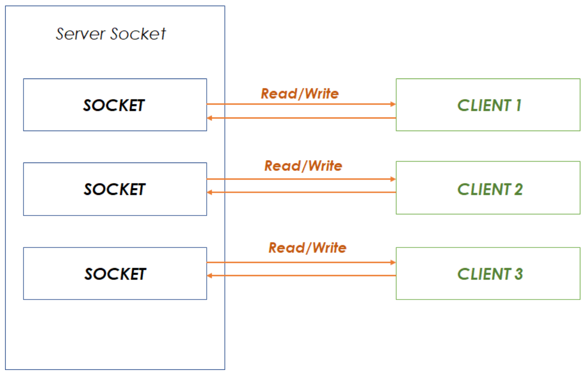

Figure 1: Socket Programming

B. ClientThreads and ServerThreads:

Through the process of creating the required program, one of the most essential implementations was the communication of multiple clients. This was achieved by firstly creating inherited classes of the Client and Server, of which we named ClientThreads and ServerThreads.

ClientThreads class was created for the purpose of listening to the responses from the Server, without getting blocked when reading from the Scanner. With the BufferedReader input, we can get information from the clients. This allows for simultaneous processes to occur. The basing reason why the ClientThreads can run these simultaneous processes is due to the initialising of it as ‘implements Runnable’. This causes the instances of the Client class to be executed within the ClientThreads.

The ServerThreads class is instantiated and started from within the Server class, further initialising it by using ‘extends’ as it therefore creates ServerThreads as an inherited class from Server. With the instantiated ServerThreads thread, we found that the most efficient way for implementing a multi-client chat, is to create an ArrayList and add the started client threads to it. Whilst paying attention to concurrency, when the users connect to the Server, their thread is started and added within the Threads ArrayList. This conclusively allows the multiple clients to send messages back and forth, as all the open threads are grouped together.

We created a while loop with a coordinator variable called “coord”, to store the ID of the first element i.e., the element at index 0 in the ArrayList using “IDArrayList.get(0)”. This is because at first, the coordinator will be the first client to fully connect, and their ID will be outputted to all new members when they first connect. Another variable, “firstcoord” only stores the first coordinator and is not updated if the coordinator leaves. Using this .get method also means that if the first coordinator loses connection, their ID will be deleted from the array, so the next coordinator is set to the next ID within the ArrayList, who is now at Index 0.

For broadcasting the messaging, we created a method within ServerThreads named ‘PrintToALLClients’ which included a for loop that executes the ThreadList repeatedly and states that the output to read the user ID and their outputString. This method is then called within a while(true) loop to execute and allow the users to send however many outputStrings they want, which will be read across all client prompts consistently.

Partial to the while(true) loop, we have initialised various if statements to allow multiple executions and implementations to occur. One of which is the normal termination by inputting the word ‘exit’. We created an if statement that entails that if the user inputs ‘exit’ and is the coordinator, then their application would be terminated, they would be removed from the IDArrayList. Furthermore,we included a print line within the if statement so that when these functions take place, they would also receive a broadcasted message declaring who the new Coordinator is.

From within that while loop, to make sure that all users who are not a coordinator can still exit, and the same functions would take place, we further nested else if statements which function if the user only enters the normal termination word “exit”. Otherwise, if they do not normally terminate their application, then the users can continue messaging and communicating, which the Server will receive. Instead of creating separate if statements for these important functionalities, we initialised that while(true) loop with all these nested if statements, as the process of these functions are required to consistently repeat. To avoid having to manually create these functions in a repeat cycle, the most efficient way to implement all these functions is to store them all within that while loop which therefore will continue to cycle and occur whenever any users connect, disconnect or communicate.

The group-state maintenance was efficiently moderated as we implemented the messaging to be received within the Server as a recording of all occurrences, as well as an included timestamp of every input. The environment that we utilised in the process of developing this application, was the CLI. We found that the most efficient way of establishing all our implementations and functions was to utilise the Command Prompt as it has the power, speed and the ability to accomplish the vast array of tasks that are required of it. We also implemented various GUI windows for the purpose of connection at the end, however as we neared the end of all our implementations, we found that the continuation of using the Command Line is more advantageous for our system.

G.R.A.S.P were essential design patterns that were utilised throughout the implementation of our multi-user chat system. G.R.A.S.P is used to help decide what responsibilities would be assigned to which objects and classes. One of the responsibilities we had designed was the PrintToALLClients method which acts as the Creator, as it has an aggregation associated with the Client threads within the method. Additionally, another example of the Creator is the Coordinator object which can be instantiated within the ServerThreads class. The ThreadList ArrayList further acts as the information expert, as it knows about all the Client threads which are started and stored within it. This assigns the responsibility of knowing all the Client threads that are started, to the threadList Array List.

There is another design pattern which we have established within our program, which is Low Coupling. The usefulness of Low Coupling is that having minimized the dependency therefore makes the system maintainable, efficient, and also code reusable. We established the Low Coupling within ClientThreads class which implements from the Client class, as well as the ServerThreads which extends from the Server class. This means that there is aggregation associated within the inherited classes. Polymorphism as a design pattern, is the inherited functionality of a class or instance. We have established Polymorphism within the clientSocket, IDArrayList and the readID which we inherited from the Server class and into ServerThreads class to instantiate in other functions. For example, the readID which was instantiated, was used within the if statements of normal termination and having the removal of the user, therefore allowing it multiple behaviours.

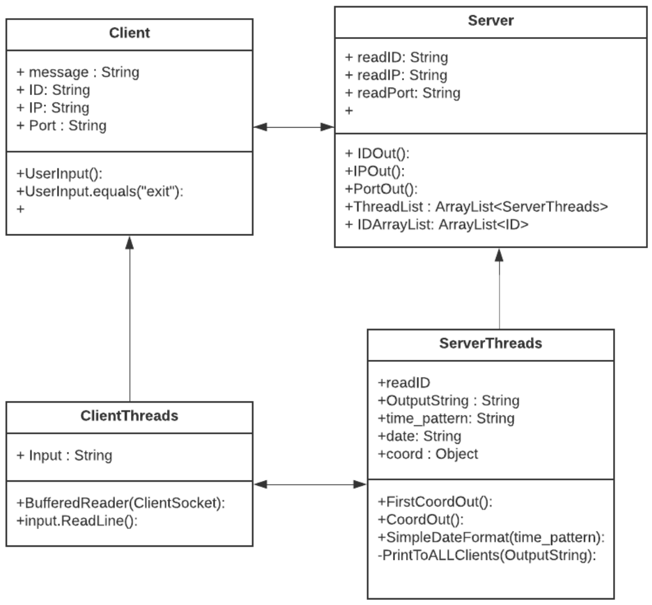

Figure 2: UML of application classes.

II. Analysis and Critical Discussion

We have achieved modularity by separating the different functions of our chat application into different classes that execute and perform different implementations. The Server class solely deals with the storing and retrieving of input from the client when they first connect to the server. The Server Threads class allows simultaneous multiple connections to run, this is achieved by having multi-threads and the clients being able to chat to one another. The Server Threads class is also unique to each Client connecting as it contains methods of reading the user's messages and writing their messages to the CLI. The Client class is an independent occurrence as each user connects. It allows for the reading of input from the user interface so that it can be sent back to the Server to verify and the constant reading of the user input from the Client class for the messaging function. Client Threads is created for the purpose of listening to the responses from the Server, without getting blocked when reading from the Scanner. With the BufferedReader input, we can get information from the clients. This allows for simultaneous processes to occur. (Gyawali, A. 2020)

As each class performs an individual component to our chat application, they are connected and executed using our standardised interface of the Command-Line. Within each class there are connections made so that the classes can communicate with each other. An example of this communication is the 'BufferedReader' and

‘PrintWriter’ that are made with matching variable names between two classes such as the Server class and the Client class. This communication method allows the server to retrieve information from the client, where the client retrieves the inputs from the user. Another example of communication between classes that we have utilised is the ‘extension’ method. The Server Threads class is an extension of the Server class as it needs to use the same variables stored in the Server class. Another vital instance of the communication being allowed between the Server and the Client being connected is seen in Client Threads. Client Threads allows the Client to not be blocked by the Server due to simultaneous reading and writing occuring between the Server Class and the Client Class. We have allowed the connection by implementing a ‘runnable’ interface in Client Threads so that it is not blocked.

Fault tolerance is fundamental to accomplish as the chat application should not run into any errors if the user has inputted invalid input not recognised by the Server or the user has abnormally terminated from the application. The fault tolerances cause errors and each of these errors must be dealt with effectively.

We have dealt with fault tolerance by adding exceptions when a specific fault tolerance occurs. The first fault tolerance is when a user connecting to the chat application enters an ID that has already been used by another user already actively running the application. To deal with this issue we have enabled a ‘while’ loop to constantly ask the user for a different ID so that we can protect and uphold the uniqueness of the ID’s in our ArrayList. This ensures that we can differentiate between the different clients when they use the chat feature, when they are the coordinator, and when they disconnect from the chat application.

The second and third fault tolerance are tied together and the way that we have dealt with it is when the user initially wants to set up a connection to the server to gain access to the chat application. We have enabled the server to have a specific IP address(Nam Ha Minh, 2019) and Port number so that when a user wants to connect, they must input the corresponding IP address and Port number associated with the chat application server. If they do not input the associated IP address and Port number the user will be told that they have given invalid input and will be asked to retry instead of an error occurring. The user will only be able to connect when they have a unique ID and the correct IP address and Port number are given.

The fourth and fifth fault tolerances are tied with the coordinator user. This deals with when a coordinator leaves abnormally or validly (as stated by the ‘exit’ command the user can input which is recognised to disconnect that specified user). These fault tolerances are dealt with in the ‘Server Threads’ class, this is shown by if the user abnormally/forcefully disconnects by pressing ‘X’ on the chat window, it will be seen as the user input returning as null and our abnormal termination code will execute. The code checks if the user who has abnormally disconnected is the coordinator, if so, our code recognises that. We remove them from our IDArrayList which stores all our active client IDs, we print to the active client that this user has left, we retrieve the new coordinator which is seen as the new client ID at index 0 in our IDArrayList and broadcast that to our active clients. If a coordinator has validly asked to exit the chat application by inputting the string ‘exit’ into the message reader, our alternative fault tolerance statement will execute. The server validly recognises that the user would like to leave, we remove the specific coordinator user from our IDArrayList, broadcast to all active clients that the user has left, and due to the user being the coordinator a new coordinator will be assigned by retrieving the new client at index 0 and assigning them to be the new coordinator, this change is then broadcasted to all active clients.

The sixth and seventh fault tolerance has to do with a regular client user abnormally or validly wanting to exit from the chat application. If a user abnormally/forcefully disconnects by pressing ‘X’ on the chat window, it will be seen as the user input returning as null and our abnormal termination code will execute. The code checks if the user who has abnormally disconnected is the coordinator, if not, our code recognises that. We remove them from our IDArrayList which stores all our active client IDs and we print to the active client that this user has left and to update their personal list of members. If a user has validly asked to exit the chat application by inputting the string ‘exit’ into the message reader, our alternative fault tolerance statement will execute. The server validly recognises that the user would like to leave, we remove the specific user from our IDArrayList and broadcast to all active clients that the user has left.

The eighth fault tolerance is to not allow the simultaneous running of clients wanting to connect to the chat application. When a user is in the process of connecting to the server, another user can not simultaneously want to connect to the server too. This is essential as we need to ensure that each user has a unique ID and if we had the possibility of two or more clients connecting at the same time it could result in duplicate IDs being made in our IDArrayList.

The last fault tolerance concerns the constant scanner being run from the client, as well as the constant writing from the server to the client and to their user CLI of the messages. This fault tolerance is taken care of by not allowing one to override the other as the class Client Threads implements a runnable interface which allows the two simultaneous actions to occur.

We realised that some of our implementation choices had some limitations. An example of these was the client timer feature. This was because ServerThreads extends from Server, therefore we couldn't have the Timer class also extend from Server Threads without causing issues. The reason for this was that this is known as multiple inheritance. However Java does not allow multiple inheritance to occur to avoid the ambiguity caused by it, ie. being interpreted in more than one way.

Additionally, another limitation is that there is currently only one way to connect to the server which is through the server class. This is because opening up the CLI before running Server does not allow for a successful connection. We have implemented a group chat feature however if clients would like to private message somebody who has connected, this is a feature that we could implement in the future. Finally, we encountered a limitation with JUnit testing. We tested the Port Input to be 7000, but we could not compare the Client Input for the PortNumber to be 7000, because the input would only exist after the Server is run. Therefore, it was not possible to test the expected input using JUnit.

We carried out our own test cases to ensure that any valid or invalid input given by the user will be dealt with accordingly. This is shown by the fault tolerances that we have handled. i.e, we have inputted invalid strings and integers for the Port Number and IP address to test our while loop. When a user inputs an ID that has already been used in our IDArrayList, it will constantly ask the user to provide an alternative ID. Lastly, we have tested different user input for the messaging function as the user will only terminate their session if they input 'exit', otherwise they can also leave abnormally which is also dealt with.

Conclusion

Conclusively, the aim of the project follows the creation of a CLI based networked distributed system, which allows for multi-client communication. We found that we achieved that through several different implementations such as the IDArrayList which stores all the readID and the threadList that maintains all started client threads. We developed different functions within the inherited ServerThreads class, such as the Coordinator being assigned to the first user added to the IDArrayList, and the normal termination process, which entails that the user would input "exit" and they would be removed from the IDArrayList and their process terminated. These functions are built within the main method of the class. Similarly, to the way in which we have the normal termination process, we also created an abnormal termination feature, which allows the user to forcibly click on the X button, and their thread would still be terminated, allowing the other users to still be informed of their leave. We commenced in several measures of fault tolerance as they are fundamental in producing a system with high modularity and efficiency. They prevent disruptions from arising in any single point of failure, eliminating any weaknesses within the developed program. Despite having taken consideration of fault-tolerance and creating a system that avoids any errors from occurring, there is still space for improvement in future developments. One of which is the possibility of incorporating private messaging service within this multi-user chat system.

Appendix

References:

[1] Nam Ha Minh (2019). *Java Socket Server Examples (TCP/IP).* [online] Codejava.net. Available at: https://www.codejava.net/java-se/networking/java-socket-server-examples-tcp-ip

[2] Amit Gyawali (2020). *Multi-Client Chat Server using Sockets and Threads in Java.* [online] Available at: https://gyawaliamit.medium.com/multi-client-chat-server-using-sockets-and-threads-in-java-2d0b64cad4a7

[3] FullStackMastery (2017). *1.3. Creating the Server Worker to Handle Multiple Connections* [online] Availableat:https://fullstackmastery.com/page/14/13-creating-server-worker-handle-multiple-connections

[4] Abhay Redkar (2018) *Java Socket Programming.* [online] Available at:https://youtube.com/playlist?list=PLoW9ZoLJX39Xcdaa4Dn5WLREHbIolbji4