Project report: Stage 1
COMP3100 Distributed Systems, S2, 2022
SID: 44849354, Name: Michael Merrick

1 Introduction

In this report, I aim to explain the Stage 1 objectives of this project, as well as show the process I went through to reach those objectives. The stage 1 objectives were to design and implement a plain version of a client-side simulator that acted as a simple job dispatcher, including a simulation protocol and a scheduler. The project was designed and constructed over a four-week period, both in and out of class, and can be found on GitHub here [1].
The rest of this report is organised as follows. Section 2 gives a high-level overview of the system and its workflow; Section 3 will give a brief description of the design philosophy involved, and Section 4 will give an indepth explanation of the implementation specific information.

2 System Overview

While the project focus is the ds-client file which I worked on, some overview of the ds-server file is helpful to understand the context in which it operates. The server essentially operates by opening a websocket for the client to plug into, before allowing for the exchange of messages between server and client before completion and termination of those processes. The intent is to simulate server and job allocation data flow in a manner similar to that of other distributed systems. In order to simulate a wide variety of configurations, the server has access to a user specified .XML file which contains information on servers and jobs for the server to simulate. The server will log all scheduling events including server failure and recovery and place them into three categorised log files. In this particular implementation, the objective of the project was to select the largest server type available and assign them jobs round-robin style - for example, if there were 8 servers available, and two of them were the largest type, then the first job would be assigned to largest server 1, the second to largest server 2, the third to largest server 1, and so on. This is referred to elsewhere in project files as Largest Round Robin - LRR.

Pairing	Client	Server
1	HELO	OK
2	AUTH	OK
3	REDY	JOBN
4	GETS	DATA
5	SCHD	JCPL
6	REDY	JOBN
...	...	...
7	REDY	NONE
8	QUIT	QUIT

Table 1: Server/Client loop messages example.

3 Design

In this section, I will outline briefly in which manner I approached this project, first focusing on the design philosophy before discussing the considerations that came up and constraints I worked under.

3.1 Design Philosophy

While initially I chose to forego the use of functions(and classes), it quickly became clear that the project was complex enough to warrant dividing processes that would be repeated frequently, such as sending and receiving messages, into functions. This expanded out into encompassing a general ”message handling loop”, so that the main program was relatively simple to view. I discuss the pros and cons of this relatively function-heavy approach further in Section 4.

I also made use of global variables; while generally considered poor programming practice, it is relatively easy to implement and function without much additional effort or complexity, which I also discuss in Section 4.

3.2 Considerations and Constraints

The most notable constraint I encountered while working on the project was the aversion to importing additional packages; while this was not a significant roadblock, as most features that this project required were fairly simplistic, it should be noted that there are several implementations of tasks that could be performed more cleanly with additional package use. Aside from this, the order in which messages were sent and jobs were scheduled was quite strict, due to the task requiring an exact replica of the logs created by the reference program; any methods of completing the task objectives which required extra messages(or possibly did it in fewer, somehow) would not suffice. Additionally, while the ds-server is perfectly capable of calculating cost and efficiency statistics, the objective was to perform the task following ONLY the largest round robin method - any other methods, which utilised other servers than the largest, were unacceptable. All of these were fairly fundamental constraints to be considered during project implementation, and depending on Stage 2's requirements, may cause later implementation issues.

4 Implementation

I used no additional packages beyond java.net and java.io, which were both given as part of the starting project. I attempted to keep global variables to a minimum, and I made a significant effort to modularise my code past a certain point and breaking down the communication processes greatly enhanced code readability made it much easier to see where I was in the response-request chain at any one point. It made troubleshooting during development vastly easier. I also tried to ensure that my main function only called a few functions itself, and to split function calls across functions if needed, to ensure that the main function loop didn't become too complex.

4.1 Global Variables

In this section I list the unique global variables and their purposes; this excludes the Socket, BufferedReader and DataOutputStream objects that are likely to be present in any implementation of this stage.

4.1.1 Global Variable information

String[] server_list was a string array containing the strings of every server sent in response to a GETS All message. String biggestServer_type contained the largest found server type, and int biggestServer_num contained the number of these largest found server types. I used int serverCounter to keep track of which server out of these number of servers I was currently scheduling a job on, and String job to keep track of which job was being scheduled. Finally, int dataGet was used to keep track of whether I had recieved the server data for GETS All(seen in JOBN Handling below.)

4.1.2 Global Variable reflection

Upon reflection, many of these variables could and should be moved elsewhere; there are more elegant ways of only running a function once than having a global variable set to 1, like I did with dataGet. I could have created a biggestServer object that had two properties, num and type, and then assigned the largest server number and type to that object instead. There are also more elegant ways to iterate through a range then using serverCounter as a global iterator, but I chose to leave them as is as it was helpful to understand my coding processes and improve upon them in the future.

4.2 Modularisation

The following subheading categorise and explain each set of functions I used in stage 1 to achieve the stage's objectives.

4.2.1 Handshaking, Basic communication, and Termination

These functions served as the basic core of most interactions between the server and client. The function sendRequest(String request) simply wrote to the DataOutputStream object, and the function rcvRequest() calls readline on the BufferedReader output and at most point in the project printed the output to console

for development purposes. The functions handshake() and closeConnection() simply contained the set of instructions for initialising and terminating connections between the server and client to unclutter the code.

4.2.2 Response Handling

The function handleResponse(String response) was used as the core of the program and was run by itself in a loop, with rcvRequest being used as the argument. It would receive the request from the server, split it into a string array, and then determine what the server message wanted and how to respond to it by scanning the first entry of the string array(or first word of the message). It has responses for the five key words for this stage of the project: ”JOBN”, ”DATA”, ”JCPL”, ”NONE”, and ”OK”. ”JOBN” was handled by jobnHandle(String[] response), ”DATA” by dataHandle(String[] response), and ”NONE” by closeConnection().
The messages ”OK” and ”JCPL” were handled by the function with a sendRequest(”REDY”) call, rather than a specific function.

First Word	Handling function	Subfunction/argument
JOBN	jobnHandle	schdJob
DATA	dataHandle	getBiggest
JCPL	sendRequest	”REDY”
NONE	sendRequest	”REDY”
OK	closeConnection	N/A

Table 2: Overview of response handling.

4.2.3 DATA Handling

Once dataHandle was called, it would check the second object in the String[] response array to identify the number of servers that the ”GETS All” request had returned during the first execution of the jobnHandle function. It would contique to recieve data that many times, and add every entry to a string array, that it would then feed into the function getBiggest(String[] list), which would identify the largest server type and how many of that largest server type there were, which would assign those to the relevant global variables biggestServer_type and biggestServer_num.

4.2.4 JOBN Handling

There was an interesting design question in that the first message sent to the client after handshaking would always be a JOBN message, before the client had any information about servers. Due to the need to adhere to the sample client precisely, this JOBN message basically needed to be held ”in stasis”, so to speak, so as to not require an extra message exchange. As a result, I used a dataGet global variable to execute a GETS All request on the first execution jobnHandle and no other time after saving the JOBN message, and every time after that to call the schdJob(String job) which constructed an SCHD message from client to the server, iterating through each of the largest servers before looping back to the first by referencing the serverCounter global variable.

4.2.5 Function reflection

As mentioned in the first heading of this section, The modularisation of this project was immensely helpful in aiding code readability, an attribute which proved to be very important as I progressed through development. I think there were some functions that may have been too large; dataHandle and getBiggest could have been split into subfunctions, but overall I was satisfied and considered my implementation of all functions acceptable.

References

[1] M. Merrick, “Stage 1.” https://github.com/myrmecoleonn/COMP3100-Assignment1, 2022.