Distributed Systems Client-Server allToLargest Job Scheduler

1. Team Participants

Member	Student ID	Contribution	Sections Completed
Joshua Devine	45238278	33.33%	Team Leader, Data Handling, Introduction, System Overview, Design
Aidan Ho	45256128	33.33%	New ServerInfo Class, Introduction, Implementation
Anthony Halteh	45650225	33.33%	Handshake, Implementation

2. Introduction

Distributed systems are a collection of autonomous computing elements that communicate with each other and coordinate activities in an orderly manner, with the aim of emerging as a coherent system of end-users. This project involves building a program which schedules jobs to specified servers in a simulated distributed system environment. The simulation is based on the TCP client-server architecture where the client program connects to the server side to receive the corresponding data from the byte stream. The server side of this project has been provided to the team which will be listening for any potential connections whereas, the client side of the project will be built by the team to initialise a connection to the server and respond accordingly to the messages specified by the distributed systems simulation protocol.

2.1 Scope

Stage 1 specifications have led to the assumption that the goal of this project is to develop a client-side program which establishes a connection to the server and follows the procedures highlighted in the ds-sim protocol. Specifically, the client should connect and ensure the robustness of the connection through initial handshake between the client and server, if successful the client is ready for its purpose of receiving jobs and scheduling them accordingly. Stage 1 emphasises that the scheduling of the jobs should be in accordance with the allToLargest algorithm where all the jobs sent to the client should be directed to the first largest server type. Hence, the largest focus of the client-side program will be the functionality in accordance with this algorithm. Determining the largest server type is defined through the severe with the largest amount of CPU cores.

3. System Overview

The client-server ds-sim is initiated by the user when the client connects to the server. This occurs as the server will be listening for any potential connections on a specified port number, for the server this can be set at a minimum of 49152 or a maximum of 65535 however the default value is 50000. Once this initial connection is made between the client socket and the server socket on the default port value the communication of messages between the client and the server can begin. As stated, the client must follow the specifications regarding correct message forms as stated in the ds-sim user guide thus the client must follow the handshake protocol before the client can receive jobs to schedule. Once completed the client can either send a GETS All message or parse the data from the system info to read all the data about the servers and then determine which one is the largest. The client will then proceed to a loop that will constantly listen for jobs that need to be scheduled or any other messages that the client will handle. This will happen until a NONE message is received by the client indicating no more jobs to be scheduled thus highlighting the client and server can both quit from the simulation as the all the necessary information has been sent.

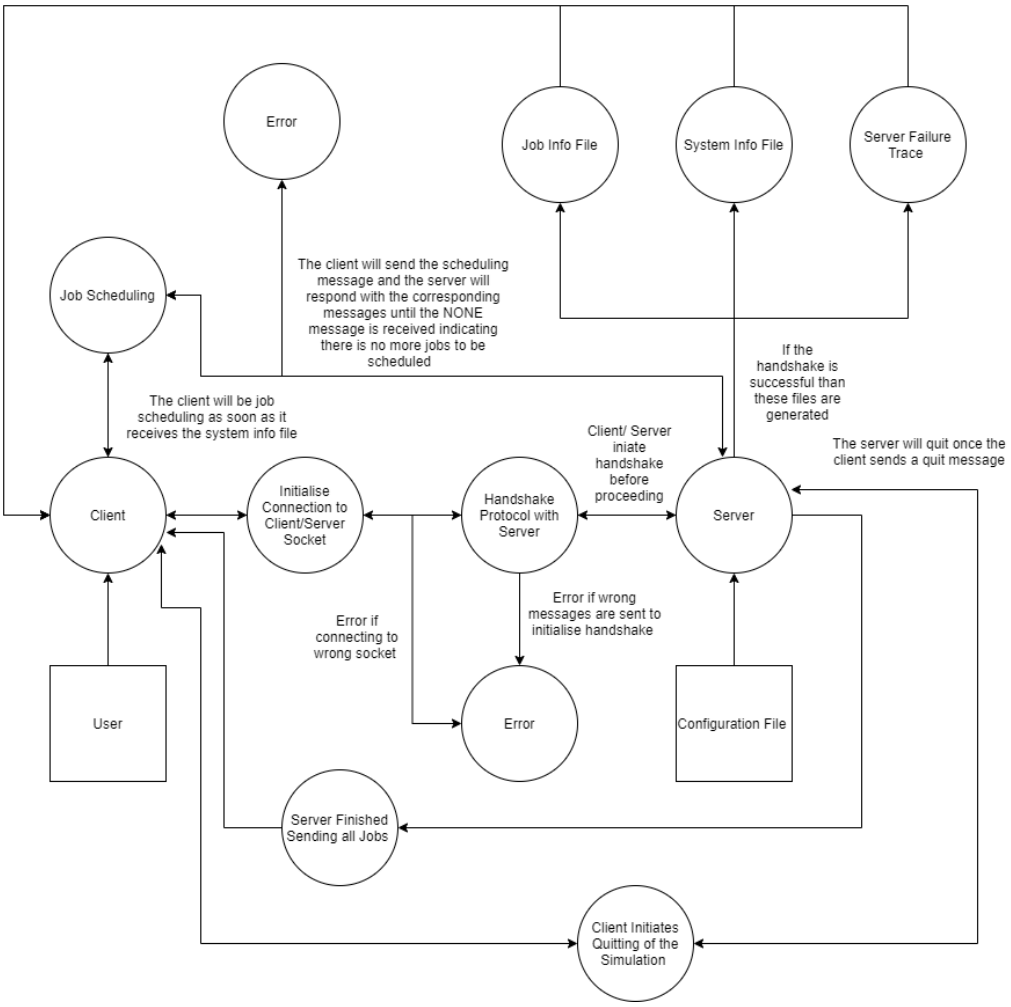

Figure 3.1 Data Flow Diagram of Client-Server DS-sim

4. Design

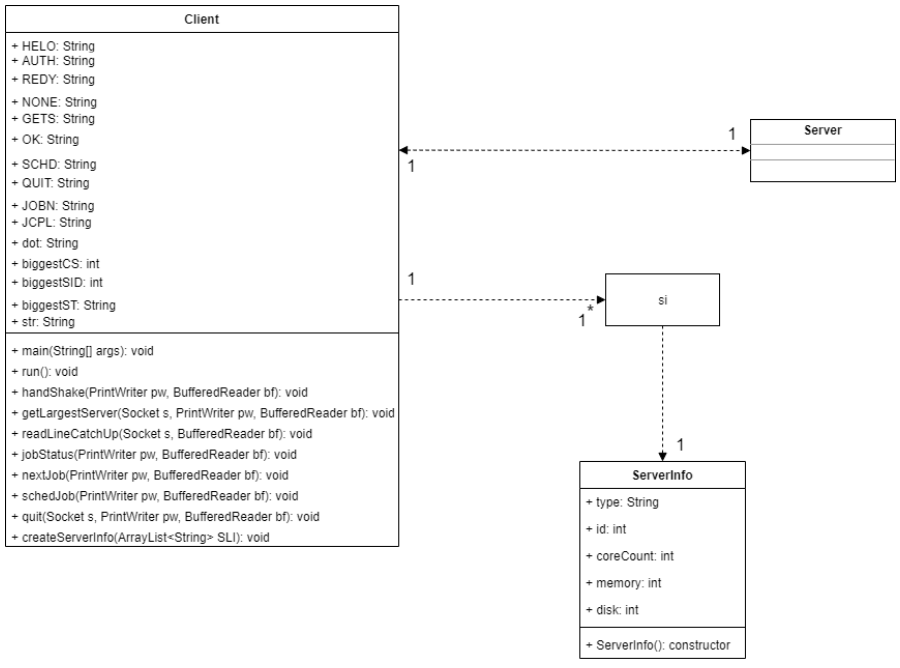

Figure 4.1 UML Class Diagram of Client-Server DS-sim Design

4.1 Design Philosophy

The design philosophy behind the team's approach to the project was to incrementally break down the project into achievable steps rather complete separate aspects all at once. This approach tied into the agile management methodology of using a Trello Kanban board to organise each aspect of the project. This was an effective design philosophy as it allowed the development of each function with a clear direction and indication towards the final stages of the project. The team developed each function until the program satisfied the requirements of stage 1.

4.2 Considerations and Constraints

A. Conflicting Language Types

The specifications indicate that the client-side of the project must be programmed in the java language however the server-side of this project has been programmed in the C language. This influenced the design approach as it became clear in the early development stages that reading and sending messages to and from the server, they would need to account for new line characters ("\n"). The solution for this problem was the connection establishing stage of the client was designed to incorporate a BufferedReader which puts a wrapper around the input stream allowing for the messages to be read by the client and then utilises the readLine function to read each line from the server. The PrintWriter object was the team's solution to sending the messages to the server as it allowed for the client to send data on the sockets output stream. The println function was called on the PrintWriter object to send the necessary messages from the client to the server and it did not require any added new line characters or to invoke an additional function getBytes which would have been required if the client program had used a DataOutputStream object.

B. Coherent Coding

A requirement of this assessment was to develop the client-side as a team. This provided some coherency issues as the coding style of each team member and approach to some problems was different. This raised issues when a review of each member's progress was done whenever they had finished a component of the project. The functions or parts they would be working on would be incompatible with other members work due to differing styles or approaches thus resulting in crucial time being devoted to re-working the code to be incorporated correctly. An instance of this was prevalent during the early stages of the project when 2 members of the team were using DataInputStream/DataOutputStream objects to read and write messages rather than using than the InputStreamReader/PrintWriter objects. After discovering the problems associated with the new line characters when reading and writing a message using DataInputStream/DataOutputStream objects it was determined rather work around this issue just use the other members succinct method of using InputStreamReader/PrintWriter objects. The time wasted on this coherency problem led to the key use of GitHub as it provided a single working point which allowed for each member to work towards the same file and build solutions which would be more in sync with the existing developments.

4.3 Functionalities

4.3.1 Initial Functions

The main method of the client program invokes the run method which essentially starts the program. The run method is a function which initialises the Socket object to the default address and port of the server, the InputStreamReader object which reads the inputStream of the Socket, the BufferedReader object which puts a wrapper around the InputStreamReader object allowing for the data from the server to be read/translated properly and a PrintWriter object which allows the client to message the server. These initialised class objects are essential for the whole project as they establish the connection with the server and allow for communication between the client and the server. The run function calls upon the other methods which are crucial to the functionality of this program. The run function also holds the overarching while loop where the client listens for jobs or other programmed messages.

4.3.2 Handshake Protocol

The handshake protocol is one of the most essential functions of the client-server simulation as it allows for the client and the server to ensure that they are properly connected to one and another before the server begins creating the system info file that allows the client to know all the server information and then is ready to begin scheduling jobs. The handshake method begins with HELO message being sent from the client to the server, then the client must authenticate themselves with the server which is done with a userName variable that uses a System function to get the username of the machine. Once authenticated the server will receive a REDY

message to indicate the handshake is complete and the client is ready to schedule jobs, the server will then send its first job.

4.3.3 Calling Gets/ Getting Largest Server

This function is crucial for getting all the server information and fulfilling the algorithm requirements for stage 1 which is all jobs to the largest server. Firstly, the method getLargestServer is called which is responsible for sending the GETS All message to the server and then receiving all the servers of the configuration file. The while loop is meant to allow the new reply string to hold each server and then be added to the ArrayList SLI of Strings so that the data can be accessed in the next function. Strings were the format choice as it was evident that the server messages were all Strings. Once all the individual servers were added to ArrayList the next function createServerInfo was called which took the parameter of the ArrayList SLI. A new ArrayList of the teams ServerInfo class serverHold was created as well as a temporary array of Strings SLIHold. The initial for loop was created to loop through the SLI list with the purpose of creating new ServerInfo objects to match the total amount of servers in SLI. SLIHold will hold and update with each split server String from SLI which is split on white spaces "\\s+" so that each SeverInfo object si can have its attribute assigned to the corresponding part of the server String from SLI i.e. SLIHold[4] will hold the cores from the servers, this will be converted to an Integer and then assigned to si.coreCount field. These si objects are then added to the ArrayList serverHold so that it can be looped through whilst checking each si objects coreCount is greater than a prior coreCount. If true then the variables biggestCS, biggestSID and biggestST will be updated to hold the largest values.

4.3.4 Catching up readLine

The catching up function is crucial to ensuring that the correct data is being read by the readLine. It was noticed with the Gets All message that when the server sent back all the server information after a dot was sent to indicate all information has been sent which was expected, however after the first dot it was noticed after brute forcing a few readLines that there were more dots and blank spaces between the dots. After careful examination it was determined that the dots were a result of the server responding for each server it sent i.e. 5 servers 5 dots and the blank spaces between were incrementing each time it would send a server i.e. 2 servers would be 2 dots thus 1 blank space between the 1st and 2nd dot or for 5 servers there would be 5 dots thus 4 blank spaces between the 4th and 5th dot, meaning in total for 5 servers there would be 5 dots + 10 blank spaces = 15 lines of useless information. This was a large problem as it was noticed that it created a delay on what was being read as the readLine was backlogged with 15 lines of useless messages and thus preventing jobs from being scheduled properly. Hence the solution to this problem was to have a separate string read the server information from the GETS message and the main global String str hold the initial job from the first REDY once entered to the while loop the first job is scheduled which results in an ok message being added to the end of the stack of backlogged messages. This detail is crucial as it is the component that breaks the readLineCatchUp function out of its loop as the while loop is checking for any dots or empty spaces and reading lines until the OK message.

4.3.5 The Client Listening for any Data from the Server

The while loop inside the run function is largely responsible for handling the variety messages sent too and back from the server related to stage 1 i.e. scheduling, status messages, next jobs. The schedJob function is responsible for scheduling the jobs to the largest server. It will check if the str String is holding a JOBN message, if true a temporary String array hold will hold the spilt on white space "\\s+" JOBN String. The job ID String in the array is then converted to an Integer and assigned to the variable int jbID so that it can be used to schedule along with biggestST and biggestSID variables as required from the user guide for the SCHD message. The nextJob function will check after the job has been scheduled the server should send back OK message, once received the function will send REDY to indicate its ready for the next job. Once jobs have been completed JCPL messages will begin to be sent from the server to highlight the status of jobs i.e. if they've been completed. The function jobStatus handles this by sending a REDY to receive the next message from the server. Once all messages have been sent and all jobs completed a NONE message is sent to indicate no more jobs to be schedule. The client responds by breaking out of loop into the quit function where the client will send a QUIT message to the server to indicate the end of simulation and then the client will close the Socket s.

5. Implementation

When implementing our design, we wanted to create separate functions that unanimously worked together due to the various commands we had to run within the job scheduler. This presented our code in a more organised fashion which could be easily traceable due to the clear understanding of what was happening in each function. We first stored most of our global strings and integers into a class at the beginning. Our run() function is our main function where we stored various variables such as our socket, InputStreamReader and BufferedReader. It was also where we ran our other functions that initiated the job scheduler, held server information and our handshake. Our handshake function is set up to initiate the client-server simulation. The client sends out messages to the server so that it can start running jobs. In our Client we use the GETS command where we receive all the information about the servers. Knowing all the information about the server, we create a function

where we store this list in an Array as well as find the server with the largest core. The getLargestServer function calls upon the command GETS where the server will send out a long string of servers in the XML file. With this we create an ArrayList to store 3 variables, the type, id and coreCount. We call upon another function named createServerInfo to help us find the largest server and store that information. We as well created another class called ServerInfo and a constructor to help store the Arraylist of the server information. In createServerInfo we call upon this class in ArrayList<ServerInfo> to create a temporary array to hold the information. Creating a for loop in storing the server information then we create another for loop to find the biggest server. We then place three variables, the type, id, and core into a placeholder. Once we are given a job, we go into the function schedJob. This is where we schedule the job and recall the server information that we had stored in our createServerInfo. The nextJob function is a checker once we have scheduled a job, we need to schedule the next job. This function is placed inside a while loop in run() until there are no more jobs to be scheduled. The nextJob function checks if the server sends a message "OK" if so then the client will send out a ready statement for the next job. The function readLineCatchUp helps the client catch back up to the server. There is a delay in the client reading the server because of the GETS command. The server will send a dot to signify that there is no more information to be sent. The function will see if there is a dot being sent from the server. If there is no dot then the function will not do anything but if there is the function will tell the client to read the string and move on to the next function. The jobStatus function is called upon when the server sends a JCPL message. JCPL is asking the client to provide information on the most recent job. If the server sends a JCPL message the client can just send a REDY message back to see if there are any more jobs to be completed. If not, then if will keep sending JCPL messages until all the jobs are scheduled. After all the jobs are schedule, the server will send a message NONE, meaning that there are no more jobs left. Due to the splitting of the project based upon functionality this allowed for ease in assigning tasks. Each member did as follow:

Anthony was responsible for organising the handshake protocol as well as the two functions nextJob and jobStatus which are both mentioned above. In this situation, the handshake protocol between the client-server acts as an authentication process. The client must identify itself to the server before being able to proceed. As illustrated in the handShake function, after sending HELO to the server, the client sends an authentication message before receiving any information from the server regarding servers or jobs. Joshua was responsible for initiating the GETS command and creating an Arraylist to store the information. This section of the program is of importance as it allows the for the storage of all server information sent by the server. This then gets placed into an ArrayList which also creates placement arrays to hold different values that were sent within the server information. This aids in Joshua's other responsibility of scheduling jobs and gathering the jobID. Joshua was also responsible for handling a bug that created from GETS that created a lot of delay on readLine. The function readLineCatchUp uses that natural running of the system to put an OK message at the end of the delay stack. Aidan was responsible for assigning the server info fields into variables by accessing the ServerInfo class. Joshua had placed a temporary array for Aidan to access the data from the Server Information Strings so that we can access the fields we need to schedule the jobs as well to find the largest server. Aidan was also tasked with creating a loop to determine the largest server with Joshua helping work out the algorithm. This was then stored in an array along with the server type and array as a variable to schedule jobs with ease.

The java.net.* package is allowed for the use of the Socket class. The Socket class is used to establish a connection between the server and the client. In this program, a connection is made by passing the server address and port number ("127.0.0.1" and "50000" respectively). Java.io* is used for "system input and output through data streams". Meaning that by using java.io* the client reads and writes to the server. This is achieved by using PrintWriter and BufferReader. Importing Java.util.ArrayList allowed for the use of ArrayLists, which is used in the functions createServerInfo and getLargestServer. The ArrayLists were used to store the different servers Strings the server sent, as mentioned above. ArrayLists were the preference for data storage as it provides a dynamic form of storage. This means that we can easily adjust the ArrayList without specifying any initial size.

6. References

[1]J. Devine and A. Ho, "JDC0DE/COMP3100-Group-52", GitHub, 2021. [Online]. Available: https://github.com/JDC0DE/COMP3100-Group-52. [Accessed: 07- Mar- 2021].

[2]Y. Lee, Y. Kim and J. King, ds-sim: A Distributed Systems Simulator User Guide. Sydney: Macquarie University, Department of Science and Engineering, 2021, pp. 1-40.