Efficient Cloud Resource Allocator for Distributed Systems

GROUP 55: https://github.com/breydenlandbergstudent/ds-sim-job-scheduler-dispatcher

1. **Amir Lingcoran (45419302)**
2. **Breyden Landberg (45357528)**
3. **Tanay Bhaskar Gandhi (45533296)**

Introduction - Breyden Landberg

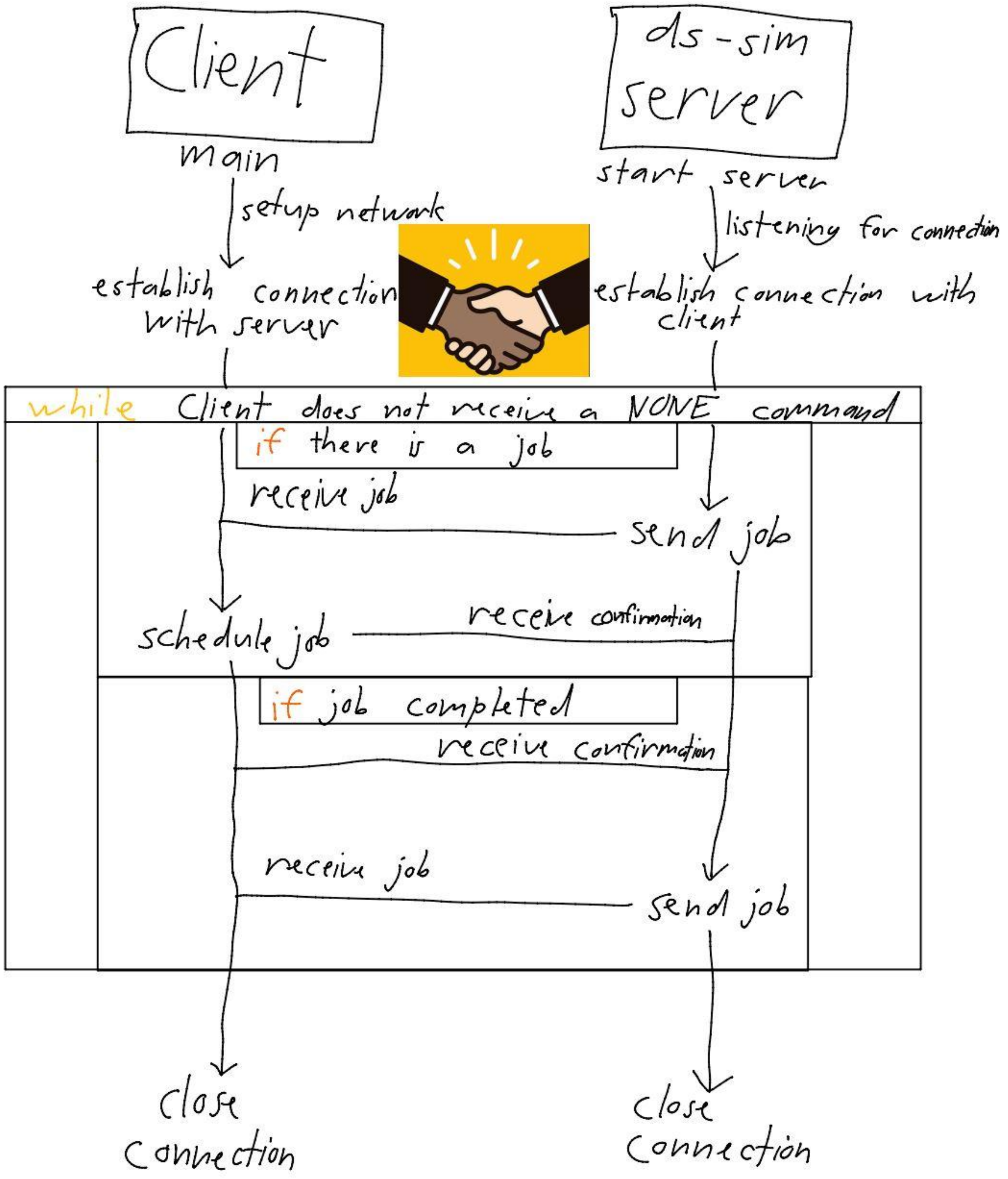
This document serves as the report for Stage 1 of the COMP3100 assessment - the group portion. Stage 1 of the project aims to design a job scheduler and dispatcher for (simulated) distributed systems. This is created collaboratively, with students working together in a group. This scheduler-dispatcher must be an efficient resource allocator for cloud servers. Best effort has been made to adhere to provided stage 1 specifications.

System Overview

The ds-sim server is intended to simulate real-time distributed systems, where client activities ("jobs") must be effectively scheduled and dispatched among a number of servers (cloud resources). The server to which a job is scheduled depends on a number of variables, including the server capabilities/size and the job workload. In Stage 1 of the project specifically, jobs must be scheduled to the first available of the largest server. The client (which simulates a scheduler-dispatcher just as the ds-sim server simulates a real distributed system pipeline) continuously loops through the jobs the server outputs until it receives the command that there are no more jobs. The connection will then be closed between the server and the client.

The client contains the hard logic for determining the server to schedule to, scheduling and receiving jobs, and maintaining connection with the ds-sim server.

The figure below outlines the interaction workflow between the client and server.



Design - Tanay Bhaskar Gandhi

The client-side simulator is designed with philosophy that the scheduler should schedule jobs to the largest available server. The scheduler has been designed to achieve maximum performance at the most appropriate costs. The client side scheduler has been designed, with three major functional components.

The main constraint faced was as the path of the XML file supplied by ds-server after authentication would change on a different demo environment. The same problem was faced in authentication, where the username would change. Hence to overcome this issue, we used System.getProperty() inbuilt method supplied by Java to get the System name and the path of the XML file. In addition, our previous implementation required messages sent to the Server to contain new line characters while running the testS1.sh. However, the latest demo script (BETA) failed the test cases due to the addition of the new line characters. Hence, the design was reverted back to an earlier version where messages did not contain the new line character.

The first component of the scheduler is the establishing handshake with the ds-sim server. In this part, the client opens a socket server and connects with the server and authenticates itself as GROUP 55.

The job of the second component in the design is to read the xml file supplied by the server. Principles of Object-oriented programming are used here. Every single server read from the xml file is used to create a server object and added to a Server List. Then we find the largest server in the list based on their core count.

Third functional component is scheduling of the jobs. This component keeps on scheduling jobs to the largest Server until there are no jobs left.

There were two main problems that were faced with the scheduler. Since the client and server communicate with each other in bytes, we converted the bytes to char array, and then converted them to string, so we can respond to the server and read the messages they were sending us.

The second problem was that the loop stopped after receiving 'JCPL' command from server, so we added an condition to handle that, where we send 'REDY' to server to get the nest available job.

Implementation - Amir Lingcoran

Client.java

All recurring variables are declared/initialised as global attributes of Client for code reusability and readability.

- hostname ("localhost")
- serverPort (50000)
- InputStreamReader din – provides a channel for receiving messages sent by the Server
- DataOutputStream dout – provides a channel for sending messages to the Server
- byteBuffer – to store messages, converted into bytes, to be sent to the Server
- charBuffer – to store messages sent by the Server in the form of a char array
- stringBuffer – to store converted String messages (from charBuffer)
- fieldBuffer – to store values obtained from the stringBuffer
- serverList – a List of Server objects read from an XML file
- DSsystemXML – a File object that stores an XML file from a specified directory
- COMMANDS: HELO, OK, AUTH, REDY, JOBN, JCPL, SCHD, NONE, QUIT

A Socket object is initialised using the hostname and serverPort to establish a connection between the Client and the Server. Subsequently, after the input and output streams have been initialised, the handshake phase begins.

Using the getBytes() method, the String HELO is converted into bytes and is stored into the byteBuffer. This converted message is then written into dout (DataOutputStream) and sent to the Server by calling the write() and flush() method. Once the Server receives the message, it replies with the message OK. The Client follows up with a message containing the AUTH command and username which the Server accepts before sending another OK back to the Client. The Java in-built method System.getProperty() is used to obtain the system name. This exchange completes the client-server handshake. In addition, the Server informs the Client of an XML file that consists of the system's information (servers). The file path of the .xml file is dynamically obtained using System.getProperty() to pass the XML file location to the File object (DSsystemXML).

Client makes a call to its class method readXML() which converts the information from the ds-system.xml file (DSsystemXML) into a NodeList where each node/element represents a specific type of server. Each type of server has a certain limit which specifies how many servers of that type exist within the system. A nested for loop is used to iterate through all servers:

- Outer loop iterates through all types of servers
- Inner loop iterates through the number of existing servers for each type

Each server from the NodeList is stored into a Server object and added to the Client's serverList. Once the readXML() call finishes, getLargestServer() method is called to identify and store the server with the highest core count, memory, and disk into a Server object largestServer. All jobs received by the Client are sent and scheduled to the said server, though such implementation generally results in higher scheduling costs.

The scheduling phase starts with the Client sending a REDY message to the Server. This informs the Server that the Client is ready to receive the first Job (if any). The Server will send the job data to the Client via the InputStreamReader which converts the message in bytes to a char array when the read() function is called. This array of characters is stored in the charBuffer. The charBuffer is converted into String using the String.valueOf() every time the loop iterates and is stored in the stringBuffer..

The loop will only terminate when the stringBuffer contains the character sequence NONE. Otherwise, if the stringBuffer contains JOBN (normal Job), the String is split into individual values, separated by spaces, using the split() function. This results in an array of String objects, stored in the fieldBuffer, that were once merged as a single String. As soon as the fieldBuffer obtains the data of the Job sent by the Server, a new Job instance is created with the fieldBuffer passed in as argument for the constructor. Each item in the fieldBuffer array is then assigned to the Job instances' fields by index order. Fields of the Job class are specifically organised to follow the order by which values appear in the Job String sent by the Server.

The Client schedules the newly created Job containing the fieldBuffer values to the largest server by sending the SCHD command, along with the Job's ID and the largest server's type and ID, as a single String. This String is converted into bytes and is stored in the byteBuffer, ready to be sent to the Server via the output stream. After the Job has been successfully scheduled, the Client sends REDY again, clears the charBuffer, and calls read() from the input stream to obtain the next Job. However, there are cases when a scheduling on a server completes while the loop is running. This sends a JCPL (job completion) message to the Client via the input stream. So, it is possible that the Client receives JCPL instead of the next Job after it sends REDY. To handle this, an else-if conditional branch for JCPL is added to resend REDY to the Server, clear the charBuffer, and attempt to get the next unscheduled Job. The whole scheduling process is repeated while there are still Jobs to be scheduled until the stringBuffer contains NONE.

Once all jobs have been scheduled, the Client sends the command QUIT to the Server which terminates the connection between the two. All streams and sockets are closed.

Server.java

The Server class consists of 8 fields/attributes:

- id – int
- type – String
- limit – int

- bootUpTime – int
- hourlyRate – float
- core – int
- memory – int
- disk – int

Server constructor takes 8 arguments which match the class' fields exactly. These arguments are passed and assigned to the Server instances' fields by order of declaration.

Job.java

The Job class consists of 7 fields/attributes:

- type – String
- submitTime – Integer
- id – Integer
- estRunTime – Integer
- core – Integer
- memory – Integer
- disk – Integer

Job constructor takes a String array argument (fieldBuffer) and assigns each array element to the class fields by order of declaration. Moreover, all String array elements, except type, are converted to integers using the parseInt method.

Given how messages sent by the server are handled and stored by the Client (character buffer has a constant length of 100), the 7th element of the String array would always consist of whitespace. This is resolved by using the trim() method on the last element of the array before it is converted to Integer.

The Job class also consists of a printFields() method which prints all fields of a Job instance on the Client-side terminal. This is mainly used for debugging.