Job Scheduler for Distributed Systems

Francisco Butturini (45437408)

Nathan Shaheen (45867194)

Nipun Shrestha (45549192)

I. INTRODUCTION

Distributed systems are made up of many separate systems and vary in scale, being made up of a few systems to thousands across the globe. Due to the vast nature of a distributed system, they often are heterogeneous in terms of processor size and memory and require some form of management to allocate processes/requests sent to them [1]. The purpose of a job scheduler is to fill this role of a 'manager' and to distribute jobs to servers, creating an acceptable user experience.

This project aims to simulate a Client-Server model job scheduler for a distributed system and provide basic functionalities such as sending/receiving, scheduling and dispatching jobs. A server-side application has been provided which will be used to simulate a distributed system and all of its servers, while a client-side application must be developed to interact with it. Throughout two stages features will be built to complete this project.

Stage 1 implements a connection/disconnection protocol as defined within the provided documentation, additionally a job scheduler and dispatcher will be implemented. The job scheduler (as of stage 1) will only attempt to schedule jobs to the largest server and will be extended upon in stage 2 to have more complete functionalities

II. SYSTEM OVERVIEW

The job scheduling system is designed to run on an x86 Ubuntu Linux platform as a simple language-independent simulation application. It comprises of two components, the client-side and server-side which made up a client-server model. Both communicate through a predefined protocol to provide an accurate simulation of a discrete-event simulator.

A. Server-Side

The server-side is designed to simulate the features of a distributed system and provide an interface for the client-side to interact with. Features implemented within the server-side include user job submissions and server job executions. Both features allow the client-side to register a connection and schedule jobs that have been generated by the server-side to be executed within the generated servers.

Through a predefined command interface the server-side responds to requests and commands to schedule jobs on each of its servers.

B. Client-Side

The client-side is designed to act as a job scheduler and interact with the generated jobs/servers within the server-side, algorithms are then utilized to determined where a job is scheduled.

Once a connection has been established the client-side has the ability to get and schedule jobs through a predefined command interface. Each job is scheduled according to a predefined algorithm. Once execution is complete an additional job can be scheduled, or the connection can be terminated.

Throughout execution the client-side is able to request information on the status of the servers and schedule job accordingly.

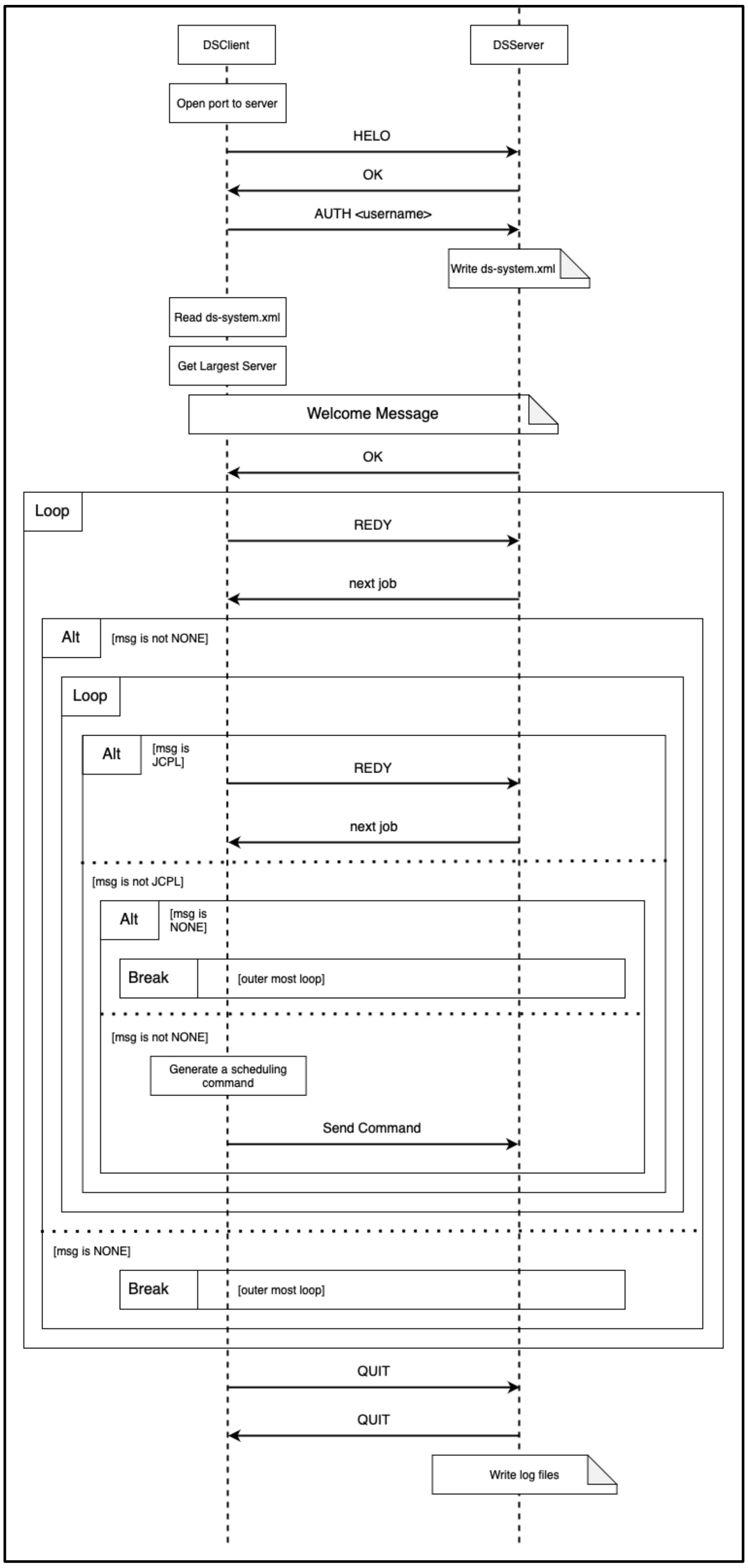

Figure 1 – Sequence Diagram of the Simulator

III. DESIGN

The design of the job scheduler required many elements to be taken into consideration such as, requirements and constraints. An iterative design philosophy was implemented to allow us to adapt and accommodate changes as we progressed within the project.

A. *Considerations*

During the design process interaction between applications as well as the user had to be considered. Elements such as what was required to be sent between the client and server, how a connection was established and what arguments were required for the simulation had to be considered. Similarly, the interaction between the user and application included considerations on what was required of them to preform (i.e. entering a username) and in what manner the output was displayed within the console. The path for the XML file that was to be read (ds-system.xml) by our client also had to be considered in the design process.

B. *Constraints*

Constraints for the project were derived from the project specifications and included physical, technical and time limits, as follows:

- Java is the required language.
- Ubuntu Linux is the required platform.
- The project must be complete within 7.5 weeks.
- Work must be carried out in groups of 2-3 people.
- The application must use the predefined communication protocol.

C. *Functionalities*

The job scheduler required two main functionalities, being, connecting/disconnecting to the server and receiving/scheduling jobs on the server. With both cases requiring interaction with the server the method of communication had to be considered and as per the constraints the predefined protocol was utilized.

Connecting to the server required the use of Java libraries so that input could be read, and an output could be transmitted. Disconnection and receiving jobs utilized the same libraries as above.

Scheduling jobs to the server required the implementation of an algorithm to determine where the job will be sent. Stage 1 only required the job to be sent to the largest server, i.e. the server with the highest CPU core count.

D. *Process*

An iterative design process was implemented to help divide the problem into smaller subproblems that were incrementally achieved, building upon the previous subproblem.

During preliminary problem analysis it was identified that the problem was comprised of three subproblems, being, Connection/disconnection, job scheduling and the scheduling algorithm. Each subproblem was solved in the above stated order and once each were completed, they were tested against sample input/outputs.

This process proved effective as it allowed the overarching problem to be solved in phases which coincided with the content that had to be learnt week by week.

IV. IMPLEMENTATION

A. Connection and Disconnection

Nathan Shaheen was tasked with the implementation of connection/disconnection functionalities and well as carrying out the initial handshake. Implementation included the use of standard Java libraries that interacted with the server through the predefined communication protocol.

Socket Programming was utilized within this implementation to allow communication to and from the server. Java classes such as Socket, BufferedReader and DataOutputStream from standard Java libraries such as Java.io and Java.net were utilized to create a connection.

Socket was used to open an endpoint the client and server to communicate, which in our case to place on the localhost on port 50000. BufferedReader allowed incoming data to be read from the socket and its readLine() function provided this functionality. Each time a message has been received, a new line would be read and depending on what was received, a message was sent back. DataOutputStream provided sending functionality via the write() function. Due to the nature of the server-side being written in C, a new line character (\n) had to be appended to the end of each sent message and the converted into bytes.

Try and Catch were used throughout the main and auxiliary functions in case an IOException was thrown by BufferedReader or DataOutputStream due to any errors during the reading/writing process.

The auxiliary function handshake() perform the initial setup between the client and server by following the predefined communication protocol. This function is where the username is parsed by the java function System.getProperty("user.name").

B. Job Scheduling and Algorithms

The implementation of job scheduling and the algorithm of finding the largest server was done by Nipun Shrestha. For finding the largest server the XML file ds-system.xml was parsed using the DOM parser API.

The DOM parser parses the entire XML file and creates a DOM object in the memory. As all the element in the DOM is a node, a nodelist of the element with the tag-name “server” was created. This nodelist consisted of all the information of the server in the XML file. The “type” attribute of the element which is the name of the server and “coreCount” attribute of the element is stored in a 2D array.

After getting all the information of the servers from the ds-system.xml, the largest server was found by iterating through the 2D array. Here, the largest server is the server with the highest core count (server_max). The auxiliary function getServerList() returns the 2D arrays of all the servers and its core count listed in the ds-system.xml file. After finding the largest server, the scheduling of all the job is then sent to the server.

C. JCPL Mitigation and Execution Flow

Francisco Butturini oversaw the flow of execution in the program, once the connection was established between the DS-Server and our client via handshake protocol that Nathan wrote. Once data was properly and efficiently stored using Nipun's data structures of 2D arrays, Francisco was then able to design and implement the flow of execution between our client and the ds-server. Essentially we have one main loop which will iterate until we receive NONE from the server signalling we have reached the end of the scheduled jobs and it is time to disconnect as per Nathans implementation of the simulation protocol.

Inside of this main loop we have how the client handles messages from the server, initially after entering the loop we create a string which reads in a line from the server, we then check if the string that was read in is a JCPL string (More on this later). After checking it isnt a JCPL string we can assume for stage 1 that it is a JOBN string. To which we split the job string into an array called job_info which takes the initial string we read in and separates it whenever there is a space. This way we can access parts of the job string very easily using indexing such as job_info[2], Job scheduling is then handled by Nipun's previously instantiated values of server_max and server_id and the string is then concatenated with the SCHD string and job_info[2] which contains the job number that needs to be scheduled. Following the Job scheduling string we then write it out to the server using the out.write method, to then which we also write REDY to signal to the server that we are ready to receive the next job.

Before we iterate over the main loop again, we read in the next line and update the main loops msg string which will check if we have received NONE from the server, if so then we exit the loop and follow disconnection procedures. In our main loop we have another nested loop which upon receiving a string from the server longer than 4 characters and is a JCPL string will just reply with REDY to acknowledge the job completion, this was designed to be a loop since occasionally the JCPL strings are received one after another. Lastly after exiting this JCPL loop we check if the last read line from the server is NONE and if that is the case, we break from the loop and exit from the main loop as if we have reached the end of the main loop.

V. REFERENCES

GitHub Repository: https://github.com/Rex781/COMP3100-Stage1.git

[1] A. S. Tanenbaum, Distributed Systems: Principles and Paradigms, Harlow, Essex, England: Pearson Education Limited, 2014.