Cloud Job Scheduler & Discrete Event Simulation

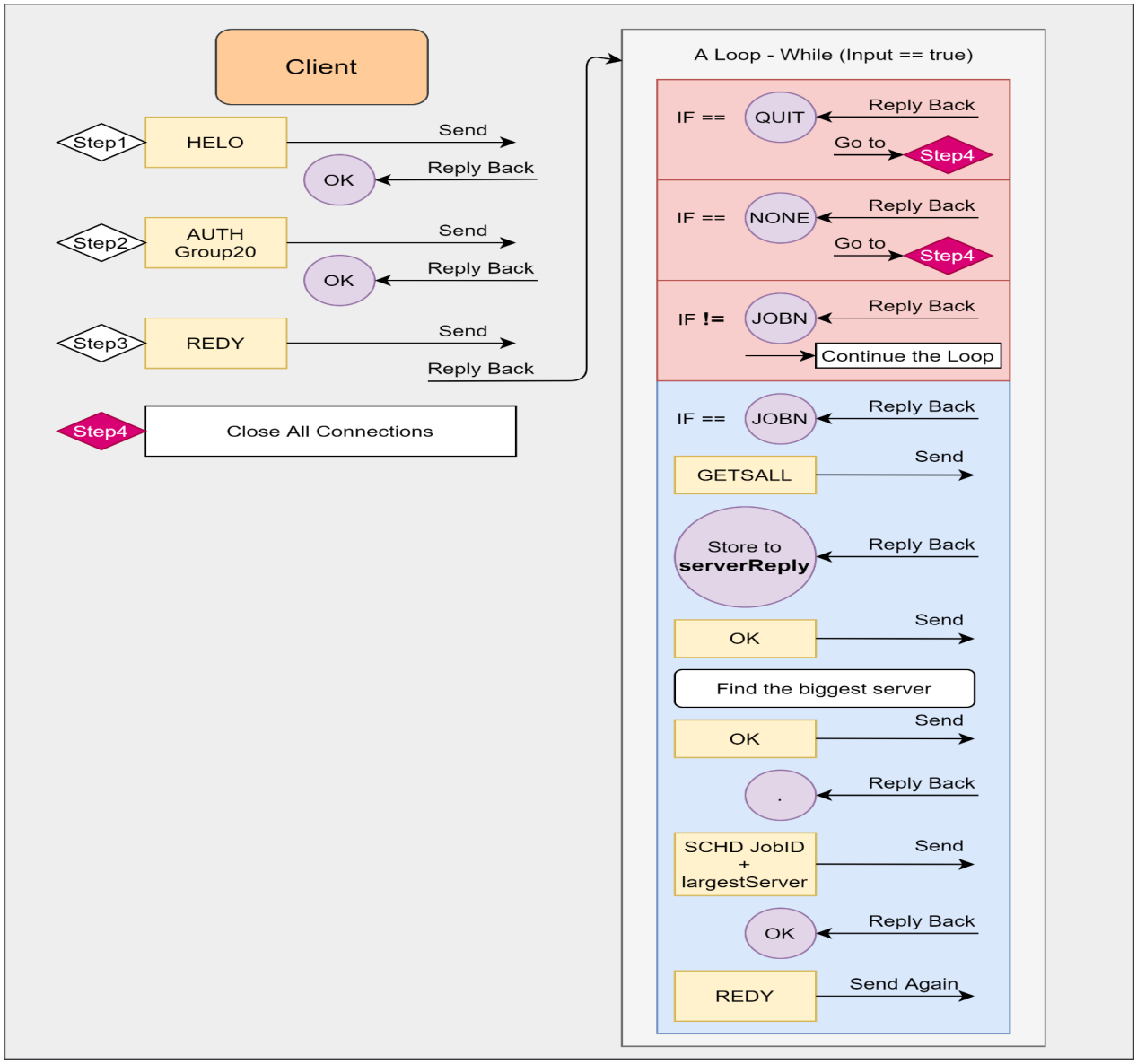
Hayden Chan (44817002)
Dac Minh Quang Nguyen (45506698)
Huu Cuong Nguyen (44491158)

Introduction

This project is to introduce the basic concept of a distributed system, also known as distributed computing. This system has multiple components placed in several machines such as computers or any devices that have local memory and are able to process multitasking. Those components communicate and interact with each other via a network to achieve designed tasks. A Distributed System is also able to scale up to millions of servers to meet a demand on the explosion of technology nowadays such as Google Data Centres, AWS clouds.

In the approach to the Distributed System, our first task for Stage 1 is to design a simple system that is able to dispatch jobs to an appropriate server which, in this case, is the biggest possible server. This report will take into account System Overview, Design and Implementation to accomplish the goal.

System Overview



Design

The project has a straightforward task of finding the biggest server and assigning jobs to the server, a loop has been taken into account to find the biggest server and then assign needed jobs to the biggest one. At first, establishing a connection between client and server had some troubles due to the wrong input port which took hours to fix. The design (e.g sending, receiving, handling values and algorithm for job dispatcher) does

not encounter that many difficulties, however, receiving value sizes brings confusion in Java Language causing "**NumberFormatException**" which has to be processed separately in order to get the right value. Another constraint is about tests Simulation provided under week 6. Because we are all new to this unit and the Demo made by the tutor during workshop time is not probably correct led to the misunderstanding that the code is wrong, it took a few days to realize and fix it.

After finishing handshake steps which are discussed in more detail in the Implementation part, we send REDY to be able to read .xml files which are a list of jobs that need to be dispatched. We start a loop to continuously assign jobs to the biggest server until it says NONE jobs to be dispatched or QUIT. If it is a job that is needed to assign, JOBN should be at the head, we send GETSALL to get the server details which are read in .xml file again and start to find the biggest coreCount over those servers. After that, assign a job to the server by sending SCHD + JobID + biggest server. The above description is a general picture of the functionalities of each simulator, this will be concentrated more in the Implementation part.

Implementation

For the implementation, first of all, the network, input and output data streams libraries need to be imported for the connection between client and server, and for reading outputs and writing inputs from and to the server respectively. Socket, data input and output streams declarations and initializations are a must in this type of communication as the socket is one endpoint of a two-way communication link between two programs, which, in this case, are client and server, running on the network. The socket mechanism provides a means of inter-process communication by setting up named contact points between which the communication occurs. Basically, the server creates a socket, attaches to it a network port and IP address, then waits for the client to contact it. The client then creates a socket and attempts to connect to the server socket at the created port and IP address. When the connection is established, transfer of data begins, where data input and output streams are required to process the data exchange between them.

What's next, the client says "HELO" to the server to confirm if the server is actually listening to it to start the data exchange. If what the server received from the client is indeed a valid message, the server will confirm "OK" for the client to be sure that it can start on the authorization and transfer of data. Otherwise, the server will send an "ERR" message to the client to indicate that the message is invalid or in the wrong format. After the authorization is done with the reply as "OK" from the server, now it is ready to schedule jobs (if any) by processing through the loops until there are no ones left.

The loop now starts: If what the client received from the server is indeed "NONE" or "QUIT", indicating that there are no jobs left or the server wants to terminate the connection (communication) respectively, the client simply sends "QUIT" to it to stop the communication and close all connections and socket.

Otherwise, if it is not actually a real job (starting with "JOBN") that needs to be scheduled, it simply sends "REDY" to the server to just say that "it is not actually a job that needs to be scheduled, just give me the next one, and I am ready for it". Then, the server will send the next piece of information (probably a job or not) and the client will continue processing this piece of information from the beginning of the loop.

If it is now indeed a job with information like submit time, job id, estimated run time, core, memory, disk, then the client knows that it will have to schedule the job with its job id (each job has a specific and different job id). To start with, the client "GETS All" from the server to obtain information on all servers regardless of their state and including Inactive and Unavailable. What the client receives in response to "GETS All" is the data with a number of records and the maximum length of one record in bytes including spaces between fields. To really get a full list of records and their information, the client will need to read in the outputs from the server in (number of records x maximum length of one record) bytes. Now, a full list of records with their information is obtained. This list of records is an array of strings separated by new line characters. In this list, each record is another array of strings separated by spaces with information like server name, limit, bootup time, hourly rate, core count, memory, disk. We will process through each record to get the name of the largest server, which is the one with the highest core

count located at the string index 4 of each record. After attaining the biggest server name, the client will need to dispatch jobs to server-id 0 of this biggest one by calling the command SCHD along with the job id and largest server with the id 0. After the job has been scheduled, the server will return “OK” as a confirmation of the successful scheduling. Then, the client is ready for the next job by telling the server that it is “REDY”. Now, it will return to the beginning of the loop and process another piece of information.

After all the jobs have been scheduled, the client and server end the communication by sending each other “QUIT” messages and then close all the connections and socket. The communication process ends here.

This project is a rather big and complicated one, so each member has his own responsibility to fulfil the requirements. More specifically, I (Huu Cuong Nguyen) have handled the codes (with all comments and proper indentation and naming) for client-server connection (communication), data exchange, and jobs dispatching (scheduling), and bugs fixing (encountered during the execution of the program) and the implementation part of the system (in this report). On the other hand, Dac Minh Quang Nguyen assists in the coding part, reviewing the code, testing it, locating the bugs and fixing them, and the introduction, system overview and design (in this report). In the meantime, Hayden Chan has literally done nothing.

REFERENCES

https://www.javatpoint.com/socket-programming
https://www.geeksforgeeks.org/socket-in-computer-network/

Project Git Repository

https://github.com/DacNguyen234/Comp3100_project