Cloud Job Scheduler

Scheduling in Distributed Systems

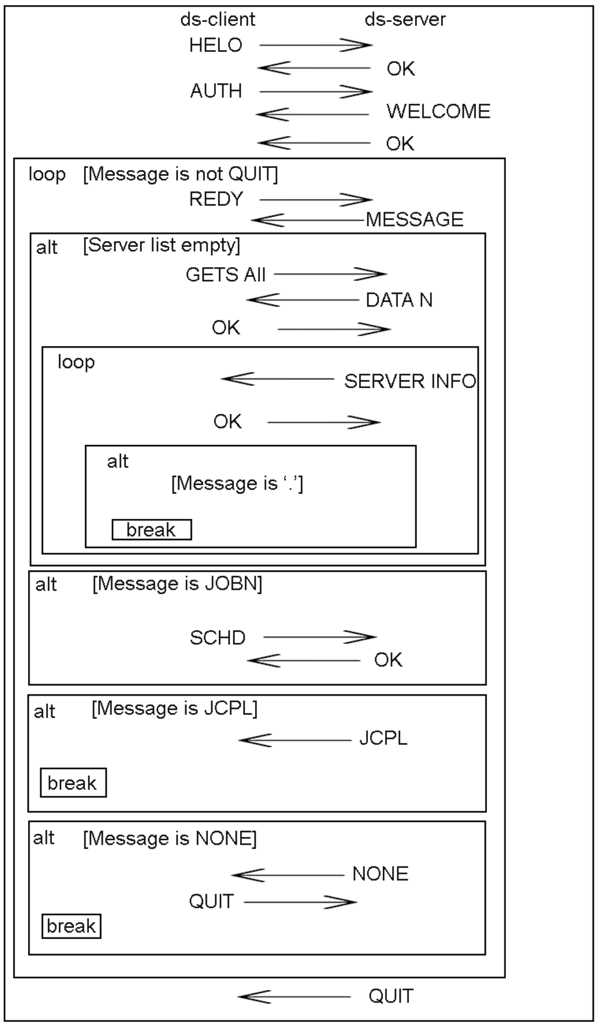
COMP3100 - Distributed Systems

Group Members:	Joshua Brooks	43603467
	Robinson Le	45948852

Introduction:

This project is to aim to create a client that will serve as a job scheduler for a cloud server which will follow a set communication protocol. The server side of this project has been provided and it simulates various scenarios of the collection of servers and jobs to be scheduled.

For Stage 1, our 'vanilla' version of a client which has been named Cloud Job Scheduler, has the purpose of scheduling all the jobs to the first largest server which is to be determined by the core count. At this stage the client will have the IP address and default port number hard written, to allow for the basic framework of the communication to be understood and programmed. It has also been specified that we are to use a specific method to gain the list of the servers and we are not worried about error handling at this moment.



System Overview:

The Cloud Job Scheduler is a simple job request and scheduler for cloud based distributed systems. It has been currently designed to quickly request jobs from the server and then provide scheduling of those jobs back to the server. It's current algorithm for scheduling jobs is sending all jobs to the first largest server. The Cloud Job Scheduler has also been programmed to find the largest server at this stage of the project. This is meant to be in use with a discrete-event simulation with the server application 'ds-sim'.

The general scheme of the communication process is shown in the figure.

The server that works with the Cloud Job Scheduler has been provided to us, which has been designed by Young Choon Lee, Young Ki Kim and Jayden King. This server is called ds-sim and is an open-source, language-independent and configurable distributed system simulator. [2] This server will simulate events such as job creation and submission, forwarding the scheduling information to the servers.

Design:

The design we went for was one that keeps track of multiple different states that are similar to the commands that are received from the server. A switch statement is used, and it has multiple cases which are corresponding to the different states. Within the cases, the commands that will be sent through the output stream are handled. Some more complex cases, such as the one that receives the server information, or the scheduling of jobs require additional functions to be completed.

Some considerations that we managed were that this is a simplified version of this job scheduler and is only required to send each job to the largest of the servers in the list. This allowed us to not worry about the server's utilisation or specifications or details about each job, such as its required memory or storage space. Although we do split up the servers and their details into an array, we only ever use their Server-type, ID and core count.
We used a switch with the use of a state that represents the various stages of the communication protocol. These switch cases will determine what is sent to the server and what code is to be executed. These states include: 'Initial', 'Authorisation', 'Authorised' 'Ready', 'SysInfoPrep', 'SysInfoReading', 'JobScheduling', 'Quitting' and 'Default'.

The 'Initial' state is the starting state when the Cloud Job Scheduler is run, and this will send 'HELO' and proceed to change the state to 'Authorisation' and break from this switch. 'Authorisation' will then get the system's username and send the 'AUTH' message with the username we just received from the system, it will then proceed to the next state 'Authorised'.

This state will send 'REDY' to the server to indicate that the client (Cloud Job Scheduler) is prepared to schedule the jobs. After this it will change the current state to 'Ready'. In the 'Ready' state it will proceed to go through numerous checks in order to supply the correct message at any given time. These checks will include: checking if we already have received the list of servers, where if we have not received it, the client will then change the state to 'SysInfoPrep' and send the message 'GETS All' to the server. If we already have the list of servers, the program will check if the message sent from the server is 'NONE' the client will proceed to begin the process to stop the communication with the server. If the message was not 'NONE'. It will then check if the message was 'JOBN', if it was, it will change the state to 'JobScheduling'. If it was not any of these previous cases it will ask the server for a new job to be scheduled and proceed with 'JobScheduling'. After all of that it will proceed to store the 'JOBN' message as a local copy named jobString.

In the 'SysInfoPrep' state, it prepares the client for the 'SysInfoReading' state by getting the number of servers currently existing on the simulation by using the 'getServerCount()' method. The 'SysInfoReading' state will then take the number of servers generated from 'SysInfoPrep' and then use the 'readSystemList()' function to then read the entire list of servers and their details.

The ‘JobScheduling’ state will check if the message sent by the server had the job to be scheduled, if it was, it will then proceed to change the current job stored in the client as a local copy. If the message was instead ‘NONE’, it signals to the client we are done and can begin the quitting process. If it was not that but the message was ‘JCPL’ and the last message we sent was ‘REDY’ we will break out of the switch and change the state back to ‘JobScheduling’

The ‘Quitting’ state indicates we are going to stop the communication with the server which is done by changing the state to ‘QUIT’, sending the ‘QUIT’ message and calls the ‘quitCommunication()’ function to close off the connections.

The ‘Default’ state is present to catch any errors that might occur, and simply printing out “Error has occurred” and closing the connection.

Within the Cloud Job Scheduler, we have employed the use of various helper functions to assist our main program in executing its roles and responsibilities. These functions include ‘getServerCount()’, ‘readSystemList()’, ‘quitCommunication()’, ‘getLargestServer()’ and ‘getJobID()’. The ‘getServerCount()’ function will extract the number of servers that are existing in the current simulation. The function ‘readSystemList()’ will loop the required number of times based on the result from ‘getServerCount()’ extracted, in order to read through the response the server sends, which is the server details and specifications. ‘readSystemList()’ then proceeds to split the message which should be the server details and stores those details into an array. The ‘quitCommunication()’ function reduces the amount of redundant code by having the process to close the connections in one place. ‘getLargestServer()’ is the function tasked with comparing the servers we stored from ‘readSystemList()’ with their CPU core counts and finding the server with the largest amount of CPU cores. Lastly the ‘getJobID()’ function has the responsibility of finding the job ID from the ‘JOBN’ received from the server.

Implementation:

The overall technique to facilitate the communication between the Cloud Job Scheduler and the Server is the use of a switch-based control scheme, with the usage of states to determine the position within the communication protocol. The use of loops to allow the reuse of code and scalability with various situations, as well as various conditionals to ensure the correct actions are executed during run time.

These states determine which case to execute that will change the message sent to the server based on what Server should expect. This is determined by checking the messages sent by the Server and what actions have been taken at any given time. The states are changed after each case has been executed to signal that the program is ready for the next step in the communication process.

A switch-based design compliments well with the while loop use, as it allows for a specific branch of code to be executed on each loop. This branch is chosen by the state that it is currently in, this allows the client to send the expected message on each loop and ensures that errors will not occur during run-time. This is further enforced by its use of break statements that guarantee that this case is the only one chosen during a loop. By using a switch, it also allows for more readability of the code as the indentation and conciseness is more consistent compared to an if else design.

The use of a loop allows the reuse of code and to ensure that communication is maintained between the server and the Cloud Job Scheduler throughout the job scheduling duties the client has. A while loop has been used to continually send messages to the server while still connected, to maintain speed and consistency with the communication protocol. There have also been the uses of for loops to cycle through the various data structures we have used to store information such as the lists of servers and several messages sent from the server like the job to be scheduled (JOBN).

Even with a switch-based control and the use of states to determine where the Cloud Job Scheduler is within the protocol, there is still the need for various conditional branches to ensure the correct action is executed during run-time to maintain smooth communication between the server and the client. This includes the check of whether the message received from the server contained JCPL, in order to know that we must ask for a new job to be scheduled.

Within this client we have designed and written, we have used multiple libraries which are:

- java.net.Socket;
- java.io.DataInputStream;
- java.io.BufferedReader;
- java.io.DataOutputStream;
- java.io.InputStreamReader;
- java.io.IOException;
- java.util.ArrayList;

The socket class has been used to allow the connection between the server and client by opening a specified port when writing the code, in our case it has been specified to use the port ‘50000’. It will allow the program to then communicate with the server and proceed with reading and writing messages.

After the connection has been created, the sending and receiving of messages is accomplished with the use of the java.io libraries. These libraries contain classes that deal with taking in data or sending out data and preparing a buffer for each event. Where the IOException would allow the client to handle errors relating to the other IO classes we have used. These errors could have included failed or interrupted IO operations, such as during reading a message from the server. In this Cloud Job Scheduler, we have chosen to ignore all of these IOException events with a throw statement.

The data structures we have used within the Cloud Job Scheduler, include the use of arrays and array lists. We have used arrays to store certain messages such as the server information and job information that have been split up by the regex “ “, this allows us to quickly access the specified information we need such as the number of servers in total, the core count of each server and the job ID. To keep each of these servers and their information together, we used an arraylist which had a type of string array.

There has been a collaborative effort in implementing various methods to ensure correct communication with the ‘HELO’, ‘AUTH’, basic ‘REDY’ functionality, ‘GETS All’, handling the ‘DATA’ message, and basic printing of the server list and proceeding with the QUIT process. The member Robinson has been in charge of providing automation to the existing code base and implementing various helper methods that are to be used for the job scheduling. The member Joshua has had the responsibility of handling the providing a basic framework of the code so it can be built upon and fully implementing the job scheduling aspect of this program. It is also his task to ensure that clean code is maintained by checking over Robinson’s work. Robinson also has the responsibility to clean up some parts of the code when he is tasked to improve on the functionality of it.
With the responsibilities listed above, it has allowed our group to efficiently work on this project with a cycling nature of work.

References: (IEEE)

[1] J. Brookes, R. Le (2021), “COMP3100 - Project”, March 2021 [Online]. Available: https://github.com/RobbieLe/COMP3100-Project

[2] Y.G. Lee, Y.K. Kim, J. King, COMP3100. Documentation, “ds-sim_user-guide”
Faculty of Engineering and Science, Macquarie University,
Sydney, NSW, Feb. 18th 2021.