**Project** **Title**: Job Scheduler for Distributed Systems

**GitHub Repository:** <https://github.com/SamuelRyu/COMP3100_ds_client>

**Group** **Members**: Jonathan Teo (45430020), Rehan Jayasuriya (45377251), Samuel Ryu (45377553)

**Introduction**:

The Job Scheduler for Distributed Systems oversaw a simple simulation between client and server. The simulation establishes a socket connection between the client and server, implementing a handshake protocol and requests and schedules jobs to be completed. The aim of the project was to create a client-side solution that receives jobs from the server and schedules them to the largest available server.

**System Overview**:

This program focuses on the client side of a simulated two-way exchange between a server and a client machine to produce a streamlined job scheduler. The program involves an initial handshake exchange to establish a connection, an XML parser to parse the details of the different server types that is sent from the server, and finally a scheduling loop that can automatically retrieve each job and schedule the job to the largest core count server.

**Design:**

This section entails the different design decisions that were made throughout the development of the program.

Reading and writing the messages was created in a helper function in the main body, this was because read and writing was essential function that would be continuously used. This refactors the code such that only a single line is required to perform read and writing to the server.

The handshake is a series of exchanged messages between client and server which is characterised as the client initialising the connection by sending “HELO” to which the server replies “OK”, the client then sends “AUTH” with their name attached and the server replies with “OK” and the server list. These exchanges of messages uses the read and write helper functions described above.

The XML parser is a helper function that reads the server list file that the server sends after being authenticated. Our team decided that using XML parser over using the GETS command to be more suited for this current solution as it was simpler due to the XML file only displaying one server for each type which made it easier to find the largest server. This however is subject to change in upcoming implementations which might involve more advance server allocation.

The scheduling of jobs required a method of continuously advising the server that the client is ready to handle events, and when the server sends the code, it is handling the events accordingly to designated code. These event includes jobs that need to be scheduled, or telling the state of a job.

**Implementation:**

Sending and reading messages was a basic and essential functionality of the client side program which was required to communicate to the server. Send Message utilises the DataOutputStream object to wrap socket connection’s output stream such that it can send messages to the server using the “.write()” method given by DataOutputStream. The server expects messages containing bytes rather than strings, so our solution involves the String class “.getBytes()” method to convert the string into a byte array. Reading utilises DataInputStream and likewise, wraps the socket connection’s input stream such that “.read()” method can be used. This method returns bytes and requires an initialised byte array of fixed length to append the data into. The read method reads up until the length of the byte array, which may cause it to end before the input stream is actually closed which resultantly cuts the message abruptly. This is solved by using the socket method “.available()” which returns the current number of bytes in the input stream. This number is used when initialising the byte array define the size correctly with an additionally increased length for reassurance. To read each byte, a for loop is used to cast each byte to character appended to a String variable which returns the final message. Both the send and read functions have been adjusted such that it uses new line as Java has occasional issues with distinguishing the end of each message. This was done through adding “\n” to the end of the String in read, and adding int 10 to the end of the byte array before sending. In both the send and read functions, a try and catch case was required since an IOException maybe thrown.

XML Parser

* WRITE LOW LEVEL INDEPTH FEATURES OF THE XML PARSER HERE

“ Xml parser takes the doc, retrieves the length of the file by checking for the server tag, it will instantiate a list of nodes that contain this tag. Once retrieved, it will go through each server and retrieve three key pieces of information, server name, server core count and server limit. The core of this parser is in the implementation of a document builder which allows the structuring of an XML file. The next core component is in the initialisation and mapping of the nodelist. Without mapping the file, it would be difficult to access each server and retrieve the key pieces of information Refer to NodeList servernodelist = doc.getElementsByTagName("server"); DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance(); DocumentBuilder db = dbf.newDocumentBuilder(); “ - WHAT JONO WROTE

The job scheduler is a while loop that continuously sends “REDY” to inform the server that the client is ready to perform the next task. Once the “REDY” has been sent, a reply in the form of a code is sent. The while loop condition is met when the server does not send the code “NONE” which signifies that there are no jobs left to be done. Another code is “JOBN” and the jobID which indicates that a job is ready to be scheduled, which the appropriate response is to schedule the job by replying with “SCHD” with the jobID serverType and serverID. The jobID is obtained using a String class method which is “.split(“\\s+”)”. It differentiates each word by using space as a separator, and places it into a String array where each index corresponds to each word. The jobID is obtained by accessing the third index. The XMLFileParser function returns a String type variable which holds the name of the largest server. The server then will display either “OK” or an “ERR” for ERROR which means another server should be used. Our implementation uses the next available largest server by incrementing the ID and keeping the serverType. When all the jobs are complete then the server will send “NONE” and the scheduling is complete, and the client can then send “QUIT” and close the communication.

**References**: XML parser: https://www.tutorialspoint.com/java\_xml/java\_dom\_parse\_document.htm