Project Report: Stage 1
COMP3100 Distributed Systems, S1, 2022
SID: 45956987, Name: Dylan Vongsouvanh

1 Introduction

This report aims to outline the goals of this project involving ds-simulators. The goals defined will particularly be reflective of Stage 1 of the project. The project (i.e., ds-sim, source code, report, etc.) can be accessed via GitHub [1].

The main goal of Stage 1 is to develop and implement a client-side simulator for the ds-sim which will connect to the server-side sim, receive jobs, and will schedule future jobs. The jobs that will be scheduled will be based on LRR (Largest-Round-Robin) which will schedule the largest job available based on the number of cores. If more than one job is found then the first one will be used.

This report will contain the following:

- System Overview
- Design
- Implementation

The System Overview will describe how the system both server-side and client-side will work outlining the process of running jobs.

Design will look in-depth at what the client-side simulator will look like, mapping out the functions within whilst looking into considerations and constraints of the project.

Finally, implementation will explore what technologies, techniques, software libraries and data structures will be applied to create the client-side simulator and how they are implemented.

2 System Overview

This section will describe the workflow of the ds-sim showing the steps taken to connect to the server-side simulator and how to dispatch jobs. Firstly, the server must be running before the client can connect using sockets running on localhost and port 50000. Once connected the client will begin communicating with the server and will begin a Handshake Protocol starting by sending “HELO” as a greeting to which the server will reply with “OK” once received. The client will then send an authentication request using “AUTH [username]” and as the server does not have proper authentication measures it will again reply with “OK” and will welcome the user to the system and will grant access to “ds-system.xml”.

The client will then read through “ds-system.xml” then send “REDY” to signal the server that it is ready to receive the next job. The server will reply with the next job written as JOBN [submitTime] [jobID] [estRuntime] [core] [memory] [disk]. An example is JOBN 37 0 653 3 700 3800. The client will then run the GETS command either using “GETS All” or “GETS Capable [job information]” which the server will send the number of available jobs. The client will then send “OK” and will receive a list of the jobs from the server followed by another “OK” from the client. The server will then send “.” to signify that there is no more information to send. The client then can begin scheduling a job using the command SCHD [jobID] [serverType] [serverID]. An example is SCHD 0 joon 0. The server will then tell the client that the job has been scheduled.

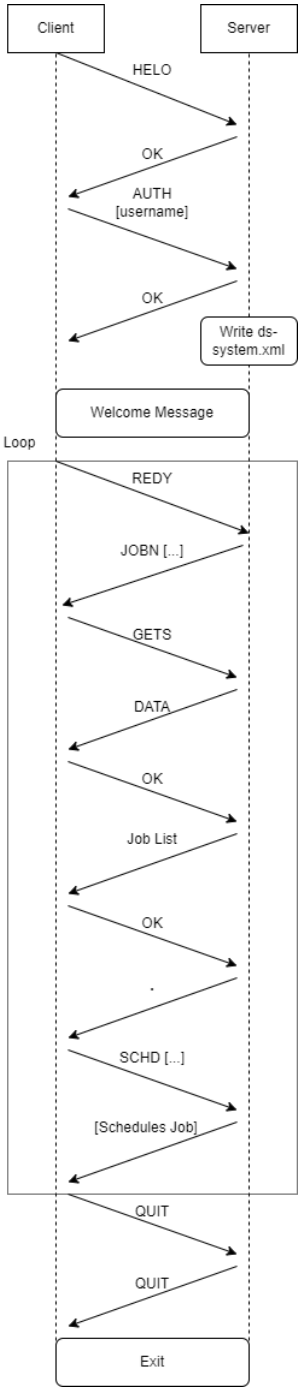


Figure 1: ds-sim workflow diagram

3 Design

In this section, the functions of the client-side simulator will be explored focusing on connecting to the server, handshake protocol, and receiving and scheduling jobs.

3.1 Connecting and Communicating with the Server

To connect to the server, the client will need to create a TCP socket and initialise it to connect to an IP address (localhost) and on port 50000. To enable communication between server and client input and output streams will be used to read the data being sent on both ends. BufferedReader will be used to read the data being sent from the server while DataOutputStream will be used to send the data being sent by the client. A conditional statement will be written to end connection to the server once there are no more inputs/data being received from the client or the server receives “QUIT”.

3.2 The Handshake Protocol

When connecting the the server, the client will need to complete a handshake protocol to prior to scheduling jobs. The client will begin by writing the message “HELO” which would then be converted into bytes in which the server will respond by sending bytes that when read will produce “OK”. This process will be repeated for authentication writing the message “AUTH [username]” instead.

3.3 Receiving Jobs and Server Information

To receive jobs and the server information, the BufferedReader and DataOutputStream will mainly be used just like during the Handshake Protocol to make requests to the server. When the correct commands (“REDY” and “GETS”) are inputted by the client and received by the server, the server will send the client the next available job and the server information. A loop will be used when gathering the server information to allow for the BufferedReader can read multiple lines that the server sends. The DataOutputStream will then be used to confirm the data sent by the server before the client can begin scheduling jobs.

3.4 Reading xml Files

Before the client can schedule jobs, it needs the server information from “ds-server.xml” which stores a list of servers. The client will do this by reading and extracting the data from the file. This function will include parsing the data in the file and creating arraylists to store the data. A DOM parser will be used to extract the data [2]. The file path for the xml file also needs to be defined for the parser to work. As the parser is reading the data, it will temporarily save each element into a list. A loop will be used to store all the server available for the client to run the job on. A server class will also be implemented to initialise the data types of each element in the arraylist. This class will also simultaneously store the data extracted by the parser into the arraylist.

3.5 Scheduling Jobs

To schedule jobs, the DataOutputStream will once again be used to run the SCHD command alongside specific elements from the extracted data and if correct, the server will schedule the next job. To receive the correct information for the command, a sorting function will be implemented to sort the arraylists and return the server with the largest amount of cores. After the job has been scheduled, the client can choose whether to continue the loop and receive/schedule the next job or quit the simulator.

4 Implementation

4.1 Technologies

Visual Studio Code [3] was used to create and edit the Client code in Java. VirtualBox [4] was used to run Linux distribution, Ubuntu [5], for the purpose of creating distributed systems (i.e., client and server ds-sim). Ubuntu’s terminal was also used to compile the client-side simulator and to run tests with the server.

4.2 Data Structures

The data structures used included arraylists to store the extracted information relating to server information from the parser into a list which then could be called later when required.

4.3 Software Libraries

Software libraries were imported for various functions for the client side simulator. Java.net was imported for the use of sockets while Java.io was used for the BufferedReader and DataOutputStream. Another library was org.w3c.dom which was used to allow for the client to read xml files whilst javax.xml.parsers, specifically DocumentBuilder, DocumentBuilderFactory, and ParserConfigurationException were imported to to facilitate the use of parsers. Finally, java.util was imported to create both lists and arraylists to store the parsed data. org.xml.sax.SAXException was imported to pass exceptions when reading XML files.

```
import java.net.*;
import java.io.*;

import java.util.ArrayList;
import java.util.List;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;
```

Figure 2: Software Libraries

4.4 Function Implementation

To connect to the server an empty socket was created and then initialised to IP address “localhost” and on port 50000. To enable communication between the client and server, a variable for BufferedReader and DataOutputStream was set and initialised to the socket.

```
class Client {
    public static void main (String args[]) {
        Socket s = null; //Create new socket
        try{
            int port = 50000; //Sets the server port to 50000
            s = new Socket("localhost", port); //Initialises the socket
            BufferedReader in = new BufferedReader(new InputStreamReader(s.getInputStream())); //Reads the data from server
            DataOutputStream out = new DataOutputStream(s.getOutputStream()); //Writes output to server
```

Figure 3: Connecting to Server

For the Handshake Protocol out.write(([Message])).getBytes()); was used to send and encode messages to the server while readLine() was used to read the messages being received from the server. System.out.println() was simply used to print to console what the server responds with. As previously mentioned, the server does not actually have proper authentication so inputting anything after “AUTH” would result in the same outcome.

```
            //Begin Handshake Protocol
            out.write(("HELO\n").getBytes()); //Client sends "HELO" to server
            String helo = in.readLine();
            System.out.println("Received: " + helo); //Client receives "HELO" success
            out.write(("AUTH Dylan\n").getBytes()); //Authentication
            String auth = in.readLine();
            System.out.println("Received: " + auth); //Client receives Authentication success
            //End Handshake Protocol
```

Figure 4: Handshake Protocol

The three lines were used again to begin receiving and scheduling jobs but with their respective inputs in place. However, a loop was used to read the server’s response to the clients request for server information as multiple lines needed to be read.

Exceptions were implemented in case the client were to encounter any errors during running.

```
        catch (UnknownHostException e){ //IP of server is incorrect or cannot be connected to
            System.out.println("Host:"+e.getMessage());
        }
        catch (EOFException e){ //End of File
            System.out.println("EOF:"+e.getMessage());
        }
        catch (IOException e){ //An error has occurred
            System.out.println("IO:"+e.getMessage());
        }
```

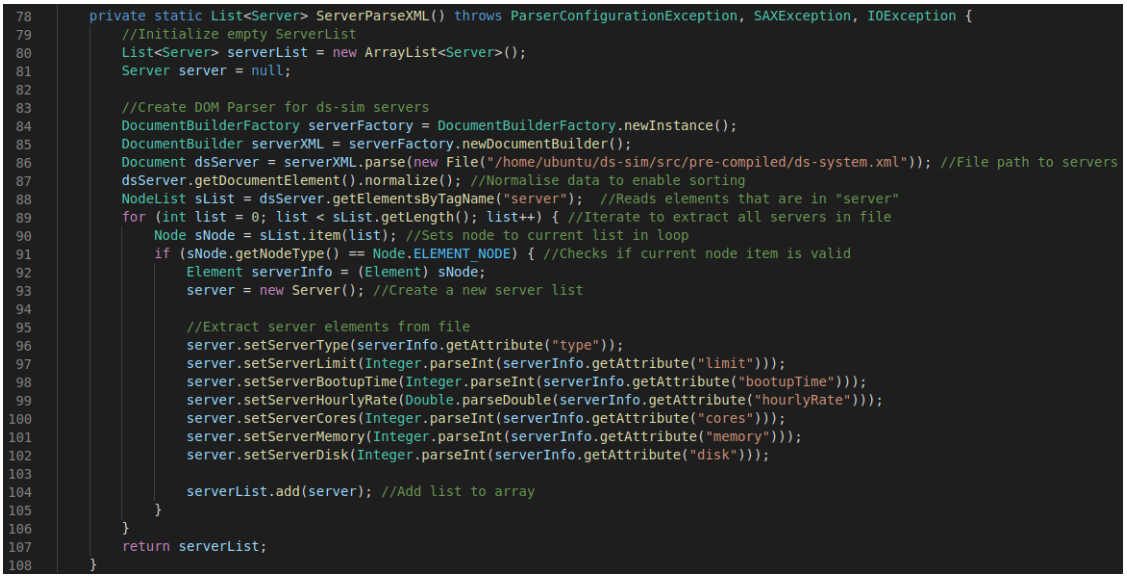
Figure 5: Throwing Exceptions

An if conditional was implemented to exit the simulator when the client inputs ”QUIT”. An exception was also thrown in case any issues arose when disconnecting.

```
62          if(s!= "QUIT") try {
63              s.close(); //Ends connection to server if client sends "QUIT"
64          }
65          catch (IOException e){
66              System.out.println("EXIT:"+e.getMessage());
67          }
```

Figure 6: Disconnecting from the Server

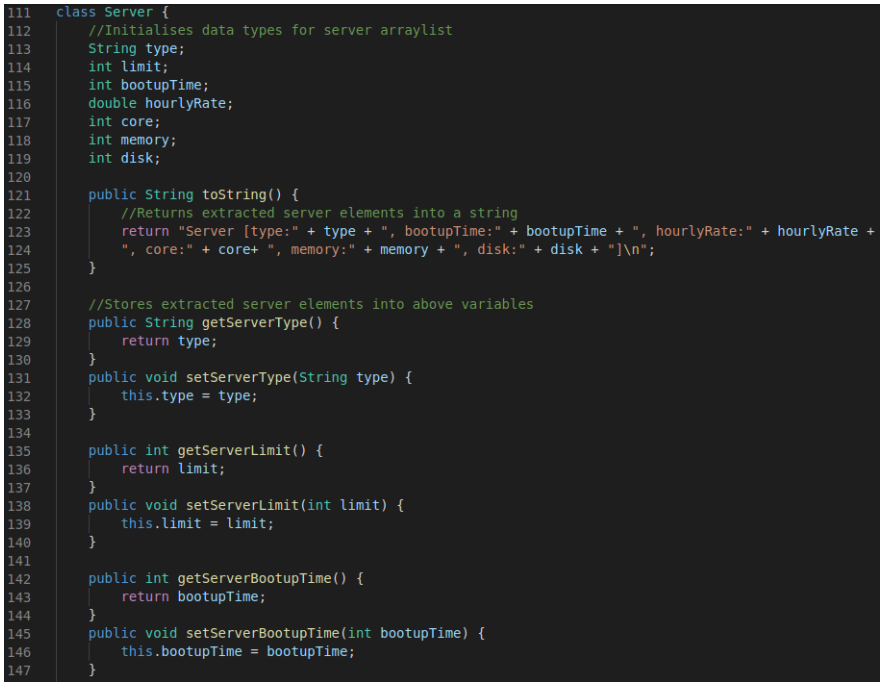
The parser class will firstly create and initialise an empty arraylist to store the extracted data. New instances of DocumentBuilder will be created to allow for the parser to read documents. A file path will be directed to “ds-system.xml” which contains the server information. The data then will be normalised to allow for sorting for later. A loop will be used to go through every list found in the file and as the loop is iterating, all elements of the list will be stored into a variable which then will be added to the arraylist.



```
78      private static List<Server> ServerParseXML() throws ParserConfigurationException, SAXException, IOException {
79          //Initialize empty ServerList
80          List<Server> serverList = new ArrayList<Server>();
81          Server server = null;
82
83          //Create DOM Parser for ds-sim servers
84          DocumentBuilderFactory serverFactory = DocumentBuilderFactory.newInstance();
85          DocumentBuilder serverXML = serverFactory.newDocumentBuilder();
86          Document dsServer = serverXML.parse(new File("/home/ubuntu/ds-sim/src/pre-compiled/ds-system.xml")); //File path to servers
87          dsServer.getDocumentElement().normalize(); //Normalise data to enable sorting
88          NodeList sList = dsServer.getElementsByTagName("server");  //Reads elements that are in "server"
89          for (int list = 0; list < sList.getLength(); list++) { //Iterate to extract all servers in file
90              Node sNode = sList.item(list); //Sets node to current list in loop
91              if (sNode.getNodeType() == Node.ELEMENT_NODE) { //Checks if current node item is valid
92                  Element serverInfo = (Element) sNode;
93                  server = new Server(); //Create a new server list
94
95                  //Extract server elements from file
96                  server.setServerType(serverInfo.getAttribute("type"));
97                  server.setServerLimit(Integer.parseInt(serverInfo.getAttribute("limit")));
98                  server.setServerBootupTime(Integer.parseInt(serverInfo.getAttribute("bootupTime")));
99                  server.setServerHourlyRate(Double.parseDouble(serverInfo.getAttribute("hourlyRate")));
100                 server.setServerCores(Integer.parseInt(serverInfo.getAttribute("cores")));
101                 server.setServerMemory(Integer.parseInt(serverInfo.getAttribute("memory")));
102                 server.setServerDisk(Integer.parseInt(serverInfo.getAttribute("disk")));
103
104                 serverList.add(server); //Add list to array
105             }
106         }
107         return serverList;
108     }
```

Figure 7: Parser

A server class will be implemented to initialise the elements for the arraylist to the proper data types. This class also saves the extracted data from the file to the appropriate data type for easy access when sorting the array.



```
111 class Server {
112     //Initialises data types for server arraylist
113     String type;
114     int limit;
115     int bootupTime;
116     double hourlyRate;
117     int core;
118     int memory;
119     int disk;
120
121     public String toString() {
122         //Returns extracted server elements into a string
123         return "Server [type:" + type + ", bootupTime:" + bootupTime + ", hourlyRate:" + hourlyRate +
124         ", core:" + core+ ", memory:" + memory + ", disk:" + disk + "]\n";
125     }
126
127     //Stores extracted server elements into above variables
128     public String getServerType() {
129         return type;
130     }
131     public void setServerType(String type) {
132         this.type = type;
133     }
134
135     public int getServerLimit() {
136         return limit;
137     }
138     public void setServerLimit(int limit) {
139         this.limit = limit;
140     }
141
142     public int getServerBootupTime() {
143         return bootupTime;
144     }
145     public void setServerBootupTime(int bootupTime) {
146         this.bootupTime = bootupTime;
147     }
```

Figure 8: Server Class

A sorter will also be implemented to sort through the arraylists of servers and will return the largest server based on the number of cores. If multiple lists are found after sorting, then the first list will be returned.

References

[1] D. Vongsouvanh, “Github Repository.” https://github.com/Dylan-vong/COMP3100Project.

[2] Oracle, “Class documentbuilderfactory.” https://docs.oracle.com/javase/7/docs/api/javax/xml/parsers/DocumentBuilderFactory.html.

[3] Microsoft, “Visual Studio Code.” https://code.visualstudio.com/.

[4] Oracle, “VM VirtualBox.” https://www.virtualbox.org/.

[5] Canonical Ltd, “Ubuntu.” https://ubuntu.com/.