COMP3100

Simulated Distributed System Job Scheduler

"Vanilla" client-side simulator

Group Members

Name	Student ID
Max Williams	45200556
Joshua Barnett	45253005

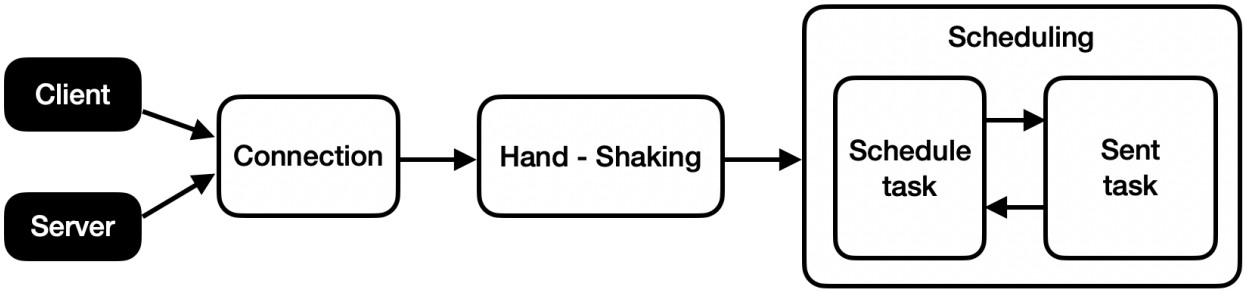
Introduction

The main goal of this assignment is to create a client-side simulator that schedules jobs that are communicated via the given "ds-server" server side protocol. The jobs are scheduled to the server of largest type which is dictated as the server with the most CPU cores and ID of 0. The client must also initially form a connection with the server and translate the received byte messages into strings. The final output of the simulation must be identical to the provided reference implementation and work for any configuration files. This assignment overall gives a basic example of how a distributed system can work and will be further developed in stage 2 when we have to program more elaborate scheduling algorithms.

System Overview

The following figure describes the relationship between the client and the server. The client-side simulator creates a connection between itself and the server, and then proceeds to hand-shaking. This occurs for the server-simulator to authenticate a user, once complete the system information is provided, and the client is ready to begin scheduling the tasks. When tasks exist

for the client to give, then it is scheduled to the server and provided to the largest server available.



Design

The system design of the solution effectively satisfies the scheduling of tasks to the server. It does this by first specifying the specific connection details at port 50,000, which is hardcoded into the system.

After this the system then begins with handshaking, which sends the message “HELO” and receives back a message from the server. The Client would then send through their details specified as AUTH, which is their system username. Once the server accepts this, it sends a xml file which contains all of the servers ranked by their CPU capacity.

Once this step is completed, the client will send an “REDY” message, meaning that it is ready to receive information to schedule to the servers. The server will send back a variety of messages, all of which will complete different actions. The most important is “JO”, which specifies that the server has a job to be scheduled. The other messages are handled by the client, which will either result in the re-sending “OK”, which is a specification required by the server after the receiving of data. Or, “QUIT”, if there is a server error or no jobs exist.

Implementation

Group Member	Completed Tasks
Max Williams	Client.java: Initial code layout, Public Client(), sendMsg, getMsg, void exit(), main(), final code comments and testing with script. Document subsections: Introduction, Implementation, references
Joshua Barnett	Client.java: readSysInfo(), code debugging and testing, final code with new line adjustments for testing. Document subsections: System overview, Design

Constructor: Utilised sockets and DataInputStream/ DataOutputStream to be able to connect to the ds-server and read/transmit messages.

sendMsg & getMsg: The 2 functions sendMsg and getMsg allow the Client and Server to communicate by making conversions between string objects and byte arrays. The java.net library is utilised to stream and extract data.

readSysInfo: This function parses the inputfile (ds-system.xml) and then stores all servers in a list. Utilising Java's getElementsByTagName we can target specific parts such as "server". We then look at the target types corecount attribute and return the target server and ID as a string object. This server variable will be utilised in main when scheduling jobs.

```
private void readSysInfo() {
    try {
        File inputFile = new File(xmlPath);
                    DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance
                    DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
                    Document doc = dBuilder.parse(inputFile);
        doc.getDocumentElement().normalize();
        NodeList nList = doc.getElementsByTagName("server");
        int max = 0;
        for (int i = 0; i < nList.getLength(); i++) {
            Node n = nList.item(i);
            Element e = (Element) n;
            int cCount = Integer.parseInt(e.getAttribute("coreCount"));
            if (cCount > max) {
                SERVER = e.getAttribute("type");
                max = cCount;
            }
        }
        SERVER = " " + SERVER + " 0";
    } catch (Exception i) {
        System.out.println(i);
    }
```

readSysInfo() code from Client.java

Exit: This exit function gets called in main when either the Client experiences an error or when the server has sent a quit message to the Client. It closes the socket connection and streams.

Main: Initially, "Client.java" was coded to utilise the "GETS All" command, however, after some discussion between members we realised we didn't need to request resource information for this stage as we are just sending every job to the largest server (which we have already identified in the readSysInfo() function) so we could just use the SCHD command. This main() function first substrings the incoming messages as it decides how to respond depending on the message received. It then follows the responses dictated by the provided ds-client documentation. It finally schedules jobs by calling the SCHD + jobID + SERVER. The SERVER variable was created in the readSysInfo() function.

```
public static void main(String[] args) {
    Client c = new Client("localhost", 50000);
    c.sendMsg(HELO);
    c.getMsg();
    c.sendMsg(AUTH);
    c.readSysInfo();
    while(true) {
            RCVD = c.getMsg();
            CMD = RCVD.substring(0,2);
            if(CMD.equals(sOK)) c.sendMsg(REDY);
            else if (CMD.equals(NONE)) c.sendMsg(QUIT);
            else if (CMD.equals(sQUIT)) {
                    c.exit();
                    break;
                    }
            else if (CMD.equals(DATA)) c.sendMsg(OK);
            else if (CMD.equals(JCPL)) c.sendMsg(REDY);
            else if (CMD.equals(ERR)) c.sendMsg(QUIT);
            else if (CMD.equals(JOBN)) {
             c.sendMsg(SCHD + jobID + SERVER);
             jobID++;
             }
```

Main() code from Client.java

Testing

We included a “server newline = true” statement to deal with the test scripts requiring newLine being enabled. We have this set to false for now to pass the test script and normal job scheduling but in stage 2 we will be using newLine instead of just “.read”.

References

Github Repository:

https://github.com/maxiebaddie/COMP3100-Group40