COMP3100 - Stage 1 Report

Cloud-Based Job Scheduler for Distributed Systems

Group 22 Details

Name	Student ID
Christoper Stedman	45609888
Rhiannon Luo	-
Ronaldo Wenas	45609950

1. Introduction

The aim of this project is to develop a job scheduler which functions as a client of the ds-server provided within the ds-sim project. The scheduler must connect to the server to receive jobs, select the ideal server from the cluster for the job and then communicate the selection to the server for processing. Together, these programs will simulate the scheduling of jobs over a distributed system for educational purposes.

For stage 1 of the project, the scheduler is required to connect to ds-server using a socket and establish an ongoing connection by following the 3-way handshake protocol. Through this channel, the scheduler will receive jobs to be completed and communicate the server selected to perform it. Server information is read from the file *ds-system.xml*, and server allocation for all jobs is performed by the *allToLargest* function which selects the server with the greatest number of CPUs.

2. System Overview

The system consists of two simulators. A client-side simulator and a server-side simulator. The client-side simulator acts as a job scheduler while the server-side simulator simulates the users and job execution.

The client and the server will both attempt to make a handshake connection. The server will create an xml file to read for the client about the server information (server type and its properties). The Client will first read the servers from the xml file and store it’s content. After it will determine the server with the most CPU’s.
Generally, the server (user) generates a job and submits it to the client-side simulator. The client then responds by making a scheduling decision and sends it to the server and the server runs the job based on the scheduling decision. The client and server will continue to have this interaction until the server has no more jobs to execute and will close the connection. If an error occurs - The connection will be closed abruptly. The client has many methods on decisions to make based on what the server has given it.

3. Design

For detailed information on component functionality, please see section 4.3 (Component Functionality).

3.1. Philosophy

Our design of the job scheduler is based upon the single responsibility principle. This approach aims to decouple program functionality by encapsulating logic into independent modules to provide greater flexibility and minimal code duplication. To further facilitate future developments, the classes *Job* and *ServerTypes* were created to improve code readability and simplify the storage and access of data.

While the simplicity of the stage 1 requirements may not have required the complexity of this modularised approach, our group decided the development of a versatile code base was important for the development of future stages. Using this approach, new modules can easily utilise existing functions and can be seamlessly integrated into the project.

3.2 Considerations

Identifying largest server

The *allToLargest* algorithm is responsible for allocating the server with the greatest number of CPUs to each job. This server type can be identified by comparing the *coreCount* variable of each server. However, as the servers within the cluster do not change over time and job information does not alter the algorithm, our group determined it was unnecessary to compute the largest server after it had been identified. For this reason, our implementation stores the largest server information in the *largestServer* variable by calling the setLargestServer function once the server xml data has been parsed. This improves execution efficiency.

Storing server messages

The job messages received from the server contain several points of data which may be useful to scheduling the job. While the stage 1 requirements only the implementation of the *allToLargest* algorithm, future stages will require additional job information when selecting a server. Therefore, received job data has been stored in a Job class to simplify data access in future stages.

3.3. Constraints

Accessing server information:

Server information could be accessed by parsing the *ds-system.xml* file output by the *ds-server* or through a *GETS ALL* request to the server. To reduce communication and latency between the server and client, our team implemented xml parsing to read the *ds-system.xml* file. Server type information is contained within tags of the name *“server”.*

Pre-defined commands

The *ds-server* recognises a pre-defined set of commands, each of which can only be used as specific and in response to specific messages, as outlined in the *ds-sim user-guide* within the *ds-sim* project. The scheduler must abide by these protocol constraints to successfully communicate with the server. In particular, the 3-way handshake protocol must be followed correctly to establish a connection.

Variable message lengths
The messages received from the *ds-server* vary, and as such they are not all the same length. The scheduler must be able to read the byte stream as it is received and recognise the end of each transmission. To achieve this without defining a set buffer, bytes are read individually from the socket and appended to the message using the StringBuilder object.

4. Implementation

4.1. Technologies

The client was developed using the Java programming language. Using in-built libraries, a socket connection is established between the client and server, allowing for a two-way communication channel between them. When the server boots, it writes server information to the file *ds-system.xml*. From this file, XML parsing is used to read server details into the client.

4.2. Software Libraries & Techniques

Socket Connection:

Library: java.net.*

This package contains the Socket class which allows for a socket connection to be established between a client and server.

To create the socket, the IP address (127.0.0.1) and port (50000) of the server is passed to the constructor of the socket. The input and output streams of the socket are then passed to the objects responsible for reading/writing to the channel.

I/O Streams:

Library: java.io.*

This package contains the BufferedReader class which was used for reading from the socket, and the DataOutputStream class which was used for writing to the socket. The corresponding I/O stream from the socket is passed to the constructor for each of these objects. The I/O streams both operate by reading/writing individual bytes at a time, so new line characters are not required.

This package also contains the File object which is used to read a file into the system so it can be accessed. The *ds-system.xml* file is read into the program using this object.

XML Parsing:

Libraries: javax.xml.parsers.*
org.w3c.dom.*

The javax package contains objects which are used for parsing the XML file into a structured object so its contents can be easily accessed.

The org.wc3c package contains data structures for storing retrieved XML content in a usable format. Server information within the XML file is accessed by finding all tags with the name "*server",* which is then stored in a NodeList object. Each Server is then converted to a Node object and then converted to an element and stored in an ArrayList of ServerTypes.

ServerTypes Storage - ArrayList:

Library: java.util.ArrayList;

Parsed server information is passed to the ServerTypes constructor to create a new ServerType object. This class was implemented to provide a simple way of interacting with server data, as all information can be stored against variables within the object. These ServerTypes objects are then stored in a global ArrayList variable so the data can be easily accessed from anywhere within the Client object.

4.3. Component Functionality

Function & Developer	Purpose
Client class Christopher Stedman	The control centre of the project. Responsible for establishing a connection with the server, parsing the server information from an XML file and managing all I/O.
Job class Christopher Stedman	Stores all information related to a new job.
ServerTypes class Christopher Stedman	Stores all information related to a type of server.Implemented to provide an easy way to interact with Server data.
Main Christopher Stedman	Creates a new client object and begins executing client code.
Run Christopher Stedman	The core of the application - Controls execution - Initiates handshake protocol - Initiates reading of *ds-system.xml* - Initiates setting of largest server - Runs the main loop in which scheduling occurs. Within this loop, messages are read from the server and the action to be taken is executed
connectionHandshake Christopher Stedman	Performs the handshake protocol with the server. Contains error handling.
writeToSocket Christopher Stedman	Takes a String as a parameter and writes it to the socket.
readFromSocket Christopher Stedman	Reads bytes from the socket and stores in a StringBuilder object. When the transmission terminates the message is returned as a string.
checkResponse Christopher Stedman	Performs error checking - The message received from the server is expected.

getJob Christopher Stedman	Calls *readFromSocket*, splits the returned string using a space character as a delimiter and returns the resulting array.
determineAction Christopher Stedman	Receives an array as a parameter (a message from the server which has been split) and executes the action to be taken based on the message command. *For stage 1, this includes only the commands JOBN (a new job to be allocated) and NONE (terminate the connection.*
allocateJobToServer Christopher Stedman	Used to determine which algorithm is to be used when allocating a server to a job. *For stage 1, this function simply calls the allToLargest function. However, this will be used in future stages to call specific algorithms based on the requirements of the job.*
allToLargest Christopher Stedman	Utilises the largestServer variable (already defined) when writing to the socket to allocate the server to the job. *largestServer is defined using the setLargestServer function when server information is parsed. This is performed outside of allToLargest function as it is unnecessary to calculate the largest server multiple times.*
setServerList Ronaldo Wenas & Christopher Stedman	Reads the file containing information on the server cluster (*ds-system.xml*), creates *ServerType* objects for each server type and stores them in the global ArrayList *serverList*.
setLargestServer Christopher Stedman	Compares the *coreCount* value of all servers in *serverList* and assigns the server with the greatest number of cores to the *largestServer* variable.
closeConnection Christopher Stedman	Closes the socket and I/O streams. *If the QUIT command received from the server the close connection protocol is followed.* *If closed due to an error, the socket and I/O streams are just closed.*

5. References

Github

https://github.com/ChrisStedman/COMP3100_ds-client

Google Docs

https://docs.google.com/document/d/1Sb4_1zk7Y4lSDeYDZ9-TlXTSQAnYgO97nvYfLIL87K4/edit?usp=sharing