Project report: Stage 1
COMP3100 Distributed Systems, S1, 2023
SID: 46598634, Name: Justin Bedwany
Github repository link: https://github.com/jbedwany/COMP3100-Assignment

1 Introduction

This project is the creation of a client using the Java programming language for participation in a simulated distributed computing system. The client connects to a provided distributed systems job server, receives information about the available participating servers, then is issued jobs for which it must choose an appropriate server with which the job is then scheduled. The client will utilise the Largest-Round-Robin (LRR) format for scheduling jobs, where the 'largest' (i.e. highest number of cores) type of participant server is identified, then each available server of that type is issued a job in a round-robin fashion.

This project forms Stage 1 of a larger project. The overall goal of both stages is to develop a functional client which implements multiple scheduling algorithms, to be compared with a number of baseline algorithms such as First Fit, Best Fit, and Worst Fit. In Stage 1, the primary goal is to develop the baseline client which successfully connects to the `ds-sim` server provided here [], correctly parses server information, and schedules jobs identically to the reference `ds-sim` client (when operating in LRR mode).

2 System Overview

The client interacts with the `ds-sim` server. The interaction can be summarised by 3 stages:

1. Connection & Preamble
2. Job Generation & Scheduling
3. Termination

The below diagram indicates each step of the process. Connection, Authentication, and Parsing Server Info form Stage 1, the loop between Receive Job and Schedule Job form Stage 2, and Quit If None Available forms Stage 3.

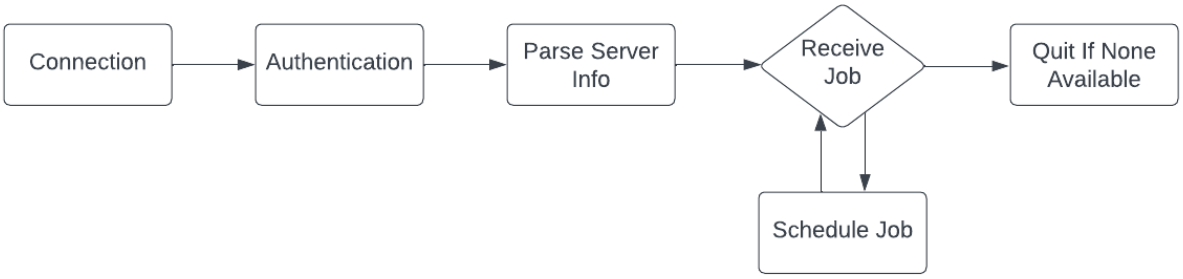

Figure 1: Process Outline

The jobs generated and scheduled are entirely independent of each other — i.e. no job depends on another job being completed first. Due to the guarantee within the specification that no job will require more resources than those of the largest server type, no capability checking is performed prior to issuing each job to each server.

3 Design

3.1 Philosophy

The major aspect of design philosophy is a focus on simplicity for users and in code architecture. Aside from a number of runtime arguments, namely the server host address, the port, and the user to authenticate as, the person running the simulation does not need to interact with the client for it to function. Programmatically, the code developed utilises the DRY (Don't Repeat Yourself) principle by compartmentalising and reusing code where appropriate in order to improve comprehension and maintenance.

3.2 Considerations, Limitations, & Assumptions

Future readability and ease of extension has been the chief consideration in this project, as it forms Stage 1 of a two stage project, and will therefore be built upon in future. Documentation is being created alongside the project in order to facilitate consistency in direction as well as ensure minimal operating details are lost.
The project is limited in that it currently only uses the LRR format for scheduling jobs; whenever any other format is required, this implementation of the client will not be suitable. Additionally, jobs are not scheduled in parallel — any given server that may be able to handle multiple jobs simultaneously are only ever given one at a time.
It has been assumed that no jobs generated require more resources than those of the largest server provided. This is a condition on which the `ds-sim` server operates and therefore will always be true for the purposes of each simulation, but must be considered when drawing real-world conclusions from simulation data.

3.3 Server Simulator

The server simulator functions by reading a configuration file which defines properties of servers and jobs that will be dynamically generated. Once the client connects and completes the preamble, the server will generate a job to be scheduled or information regarding the job queue or server statuses. This will repeat for the duration of the simulation.

3.4 Client Simulator

The client simulator functions by connecting to the server simulator, receiving information about the available servers, then receiving information about the current job to be scheduled. It then makes a scheduling decision based on the available servers and their capacities. Afterwards, it requests a new job to process, and the cycle repeats until the server responds with 'NONE', indicating there are no more jobs to be completed. The simulation then ends.

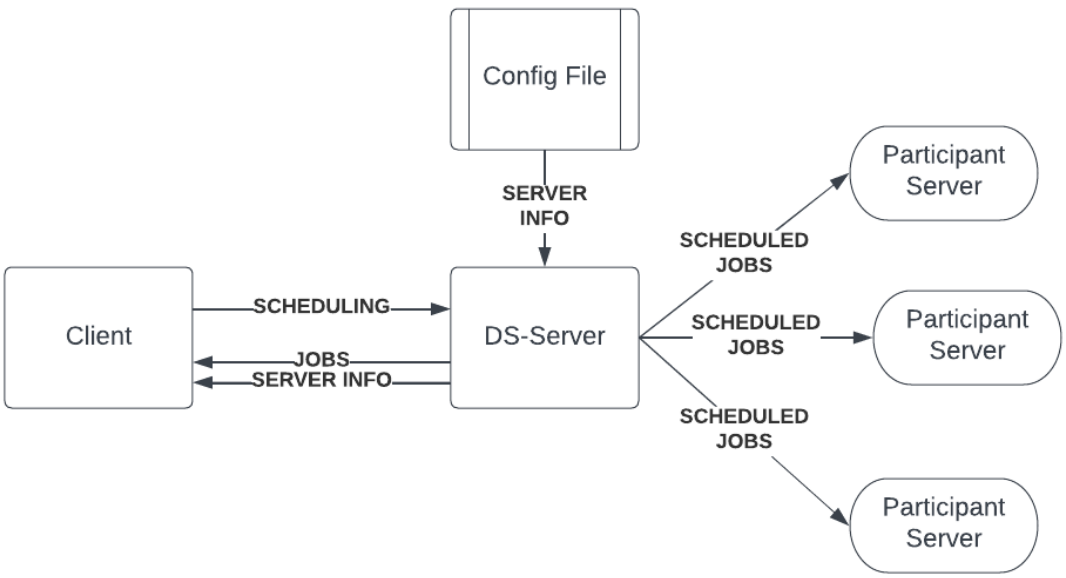

Figure 2: System Structure

4 Implementation

A number of key decisions have been made as part of this implementation. Structurally, the code is separated into the following sections:

1. Input Validation
2. Connection to Server
3. Identification of Largest Server Type
4. Scheduling of Job
5. Connection Termination
6. Helper Functions

It also makes use of two custom-defined classes, Job and Server.

4.1 Object-Oriented Programming

Java is an Object-Oriented Programming language, which means it is designed to handle complex data structures through the definition of classes [1]. This is relevant for this project, because each job and server provided by the `ds-sim` server contains multiple data fields, and in the case of the servers, multiple data types (strings e.g. server type and integers e.g. server ID). There are a number of ways to handle this, such as multi-dimensional arrays (e.g. one dimension for each job, and one dimension for each field within each job), however being a strongly typed language, arrays containing multiple datatypes are not possible — which is an issue for keeping track of the servers involved.
Instead, by defining a class each for Job and Server, we can cleanly unite all the relevant fields within a single object for both jobs and servers, and we get a bonus of being able to include as a built-in function of those objects for parsing the strings received from the `ds-sim` server declaring each job or server. This allows us to keep the main program clean and easy to read, both through modularisation of code functions as well as simplifying data structures in use, but also provides scalability options — for example, when implementing an alternative method for determining which server to schedule which job on, a comparison function can be trivially included within the job or server class definition to allow comparing jobs requirements with server capabilities.

4.2 Job Scheduling

The client schedules jobs based on a Largest Round-Robin format. After connecting to the `ds-sim` server, it retrieves the list of available servers using the `GETS All` command. It parses the results of this command in order to determine the available servers, and identifies the largest server based on the amount of cores each has. The provided server list does not need to be sorted in order of server size to accurately determine the largest server type. After the largest server type is identified, it creates a list of them, then as each JOBN command is received from the `ds-sim` server, it iterates through this list with each server having a turn completing a job, then repeats from the start. This continues until the `ds-sim` server issues the `NONE` command, indicating there are no jobs left to be processed. It then terminates the connection.

4.3 Helper Functions

A number of helper functions have been created in order to keep the main logic of the program clean and easy to read. As message sending and receiving is done in the same way repeatedly throughout the program, including message logging, they have been converted into functions.
As the methodology of finding the largest server is somewhat cumbersome, it has been separated into its own helper function. First, it issues and receives the data. It uses array data structures throughout instead of ArrayLists, which have convenient built-in functionality of being able to be resized (as compared to Arrays which cannot be, and must have new arrays of the correct size redeclared). This would have been used to potentially simply remove the elements of the servers list that were not needed (i.e. not of the largest type) and retain the same data structure, thereby reducing overhead. However, the implementation of ArrayLists is such that appending an element to the list is done by creating a new array of the correct size and moving the elements across with the new element at the end. As numerous elements are added to the array iteratively, it is therefore much more efficient to simply create a new array containing only the largest servers.

References

[1] Sun Microsystems, *The Java Language Environment*, 1999.