COMP3100

**Simulated Distributed System Job Scheduler**

Cheap-Fit resource utilisation algorithm

horizontal line

|  |  |
| --- | --- |
| **Name** | **Student ID** |
| Max Williams | 45200556 |

# Introduction

This project (Stage 2) is focusing on a very relevant subject which is creating and analysing scheduling algorithms by strict performance metrics. Job scheduling revolves around assigning a specific job to a server that has the required resources to complete the work. Many algorithms exist to do this that have different positives and negatives such as First-fit which is fast but inefficient, Best-fit which is memory efficient but slow and Worst-fit which is slow but can put small jobs in the fragmentation gaps on the server.

A scheduler is usually created to load balance resources efficiently, allow multiple users to share resources and to achieve a target quality of service. Scheduling is fundamental to computation as it is a key concept of multitasking.

The overall goal of Stage 2 is to code a new Scheduling algorithm that will optimise one or more of the provided metrics and to create a report that will outline the algorithm, justify why I chose to create it how I did, clearly indicate the performance objectives, and compare the results to the three baseline algorithms (FF, BF, WF). This algorithm must also work with any provided simulation configuration.

# Problem Definition

For Stage 2, we will be creating our own custom algorithm that must optimise one of more of the following objectives:

* Minimise average turnaround time
* Maximise average resource utilisation
* Minimise total server rental cost

These objectives are often conflicting so therefore, optimising in one area may lead to sacrificing performance in another metric.

The two metrics I focused my algorithm on was to **minimise the total server rental cost** and to **maximise average resource utilisation**. These two metrics came at the cost of having a increase in average turnaround time.

I chose to create my algorithm in this way as many individuals and companies prefer to keeps costs (overhead) as low as possible even if it results in a longer turn-around time. This algorithm would also be useful if the jobs were not on a strict time-schedule to be allocated such as making system backups.

# Algorithm Description

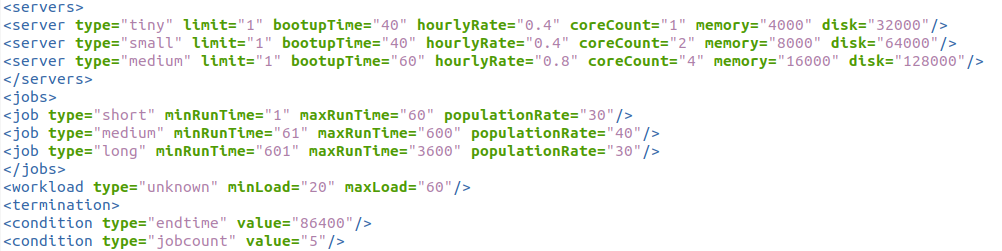
**Configuration file used:** ds-config01—wk9.xml

Figure A- Configuration file

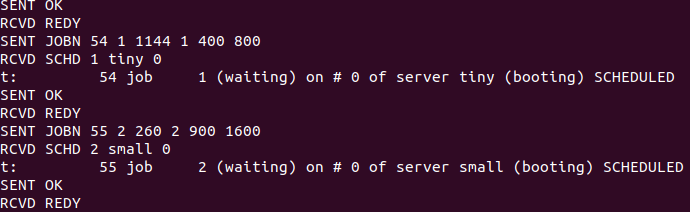
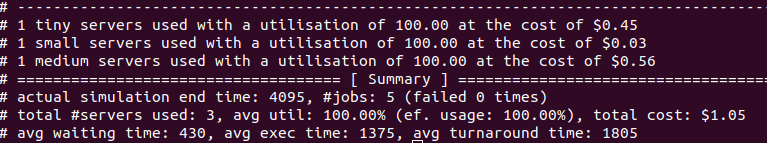


Figure B- Example Scheduling

Figure C- Final Schedule Results

**Description & Discussion:**

My algorithm works off a very simple principle: If the job requires **X** number of cores, **Y** amount of memory and **Z** amount of disk then it will be sent to a server that has the same **X** cores as long as the jobs **Y** (mem) and **Z** (disk) are lower than the servers **Y** and **Z**. If the job does not fit the above specifications it will be compared to the next largest server.

In Figure B you can see that JOBN 1 was assigned to the server “tiny” as that server has the same Core-count and larger memory and disk than the job requires (Figure A).

I went with this algorithm design because I noticed that the default “all-to-largest” algorithm was often the cheapest, but it could be even cheaper if I assigned jobs to smaller servers that had same amount of cores, mem and disk.

This overall results in a **cheap total cost** and **high average utilisation** at the cost of high turnaround time.

# Implementation Details

My algorithm uses multiple different data structures to achieve the goals. I first started by initalising all the variables and java utilities. I then made the constructors for reading the files and opening the connection. I then created functions for communicating to and from the server and client (Figure 1). I also created two functions that parses the job and server information (Figure 2) and splits it so I can compare the relevant data. readSysInfo2() reads all the server information from the ds-system.xml and stores in in a Linked list. Finally my Client() function (Figure 3) goes through the authentication process and compares the jobs with the servers. Finally a job is scheduled when a match is found.

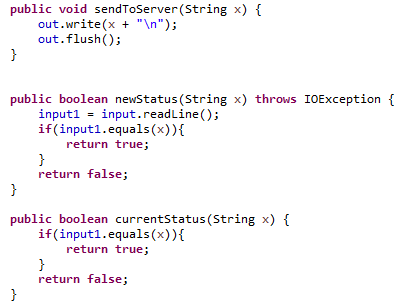
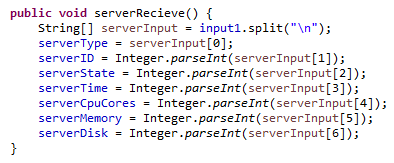


Figure 1

Figure 2



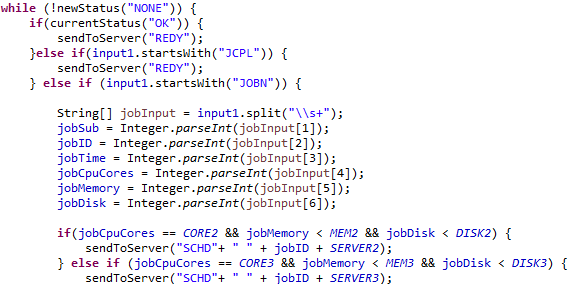
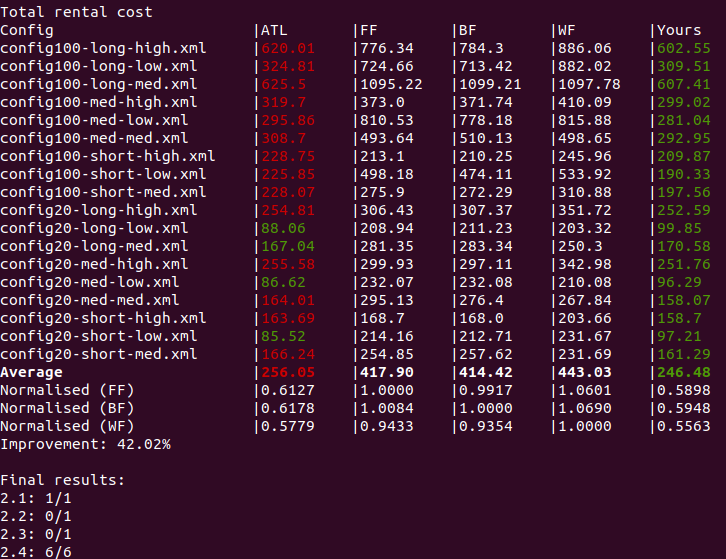


Figure 3

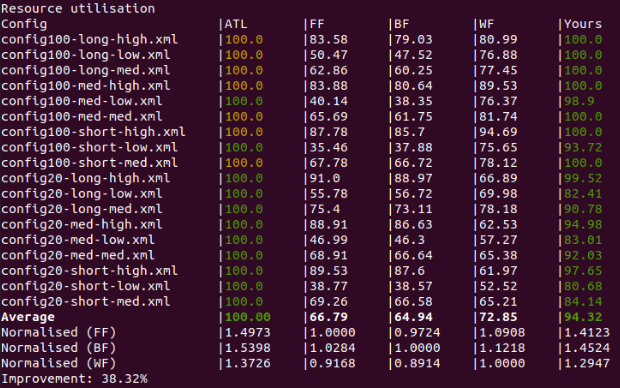
# Evaluation

**Simulation setup:**

* Compile Client
* Run ./test\_results “java Client” -o co -n -c other (Figure 2.1)
* Run ./test\_results “java Client” -o ru -n -c other (Figure 2.2)

**Comparisons:**

As seen in Figure 2.1 my algorithm outperforms all 3 baseline algorithms by having a lower total rental cost. This resulted in an improvement of 42.02%. Also in Figure 2.2 it can be seen that my average resource utilisation outperforms the 3 baseline algorithms by 38.32%.

Figure 2.1 

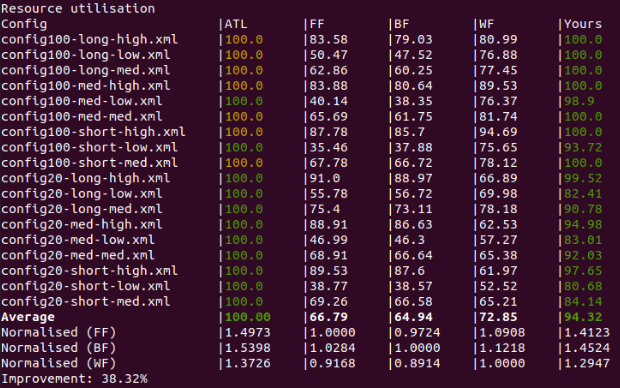
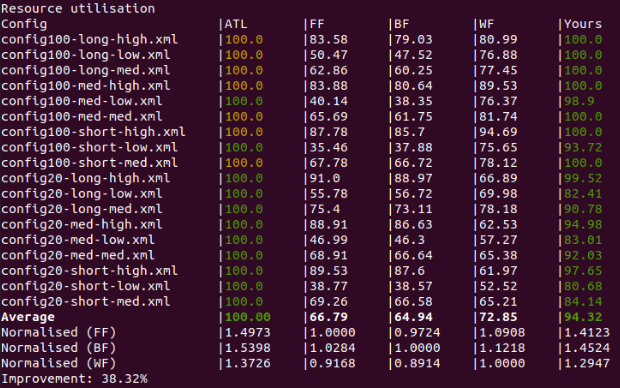
When running my algorithm with just the normal configuration files (e.g. ds-config-01.xml) it also shows a noticeable reduction in total rental cost at the expense of higher turnaround time.

Figure 2.2 

**Pro’s:**

* Low cost
* High utilisation
* Code runs quickly with no errors

**Con’s:**

* High turnaround time
* Custom capability selection instead of using GETS

**Conclusion:** Overall, my algorithm achieves the problem statement and optimises 2 performance metrics (total rental cost & maximising utilisation). However, this was done at the cost of a greatly increased turnaround time which may stop individuals from using this scheduling algorithm when a more balanced algorithm such as best-fit is available. I found it interesting that by starting up fewer servers it greatly reduced the overall cost which is why all-to-largest is sometimes cheaper than the other algorithms.

# References

**GitHub Repository:** https://github.com/maxiebaddie/Comp3100Assignment2