# Distributed Systems Job Scheduler Optimising Cost

Jarrod Adair - #45432260

## Introduction

This distributed systems project, focusing on stage 2, is the continuation of our work on the client by building a new scheduling algorithm, made in the Java language that is compatible with the ds-sim server simulator provided [1]. This required us to build a job scheduler client that could schedule jobs via discrete sequences of commands sent over a communication layer, with the goal to dispatch every job in a cost-effective manner to try and beat any of the three base algorithms. I named my algorithm the Holy Grail Algorithm 😊.

## Problem Definition

The scheduling problem that was outlined to us in stage 2 is that when optimising performance based around three objects:

- Minimisation of average turnaround time

- Maximisation of average resource utilization

- Minimisation of total server rental cost

There will be sacrifices in other metrics and as such part of the stage 2 problem is to make a decision on what metric to improve.

In this stage I will be focusing on minimizing the total server rental cost metric through a means of a customized scheduling algorithm to work in conjunction with the vanilla client my team and I built in stage 1. I made the decision to optimise for this specific metric because I felt like this one would be the easiest to reduce, having all server costs provided during runtime. I also hypothesized that with a very efficient algorithm, reducing server cost would also cross over into the two other metrics as reducing the time servers were running would reduce costs as well as higher resource utilization which would allow for less servers to be required.

## Algorithm Description

My scheduling algorithm was designed in a way that is similar to the provided best fit algorithm with some slight modifications using my own implementation of the client. The algorithm can be structured into two parts, being: the initial job scheduler that takes a JOBN command and calculates its fitness based on a variety of variables which will be discussed later; and a job-server migration manager which was designed with the intention to overlook all servers and migrate jobs to other servers when needed, which will also be discussed later.

##### Initial Job Scheduler

This job scheduler is used to handle the JOBN commands primarily and secondarily as a starting point for the later migration scheduler component to work. The job scheduler works by following the pseudocode of the best fit algorithm.

***Input:*** *a job j, a set of servers, S in ascending order of their resource capacity based on core count.*

***Output:*** *a server for j to be scheduled on.*

***Process:*** *search for a server s\*, from the first one to last one in S, that satisfies conditions of 1. Having sufficient resources readily available for j, 2. Having no waiting jobs, and 3. The fitness value\* is the smallest (best-fit).*

*If no s\* found, select the first Active/Booting server with the best-fit\*\* resource capacity regardless of resource availability.*

*\* The fitness value for a given server si is calculated by the core count of si – the core requirement of j. If two servers give the same fitness value, the first one is selected. For instance, if the fitness value of COMP3100/6105: Distributed Systems S1, 2021 second server is 1, any servers after this server, with their fitness value of 1 are not considered for the selection.*

*\*\* The best-fit resource capacity is determined based on the core count while the capacity of other resources (memory and disk) is still sufficient. In particular, when no s\* is found in the main process, those Active/Booting servers have one or more running*

My version of this algorithm slightly differs from this and follows the pseudocode of:

*Loop through all available servers from a GETS Capable command*

*Calculate the absolute value of the current loop’s available server minus the current scheduling job’s CPU core requirement and compare it with the current max difference (diff) found. (Absolute value (abs) is calculated such that a value closer to 0 means a server is completely utilised).*

*If abs is less that max diff then calculate if the server has any waiting jobs and if it is less that the best current server’s waiting jobs then that server now becomes the next best server.*

*After looped through every server, return the best stored server for that job to be scheduled on.*

I found that this algorithm worked very well compared to the original best fit algorithm when improving the cost metric. As an initial starting point, I was able to drop the cost of server operations by a couple hundred dollars alone. This coupled with the job migration component to balance out the wait time and turnaround time as well as further improve the total cost metric would be a good algorithm design.

##### Job-Server Migration Manager

This component of the new algorithm was uniquely designed in a way to handle the LSTJ and MIGJ commands. This component wasn’t fully completed but mostly all main functionality was implemented and built from scratch, working as follows.

Firstly, the client would send off a lSTJ command to every server that was available from the previous GETS command performed when initially scheduling the job to a server. This command was multipart and its handler code was designed in a way to loop through every data command sent back from the server to return all data in one return call, adding all the data to an Array list as it progressed.

Once completed, it would then loop through all jobs and check if any were lone jobs on a server; migrating those jobs that were onto the best server it could find by performing another similar loop on all the available servers and calculating their fitness based on a mixture of server cores and jobs scheduled already.

In the end the job migrator wasn’t fully completed and as such was disabled for the submitted algorithm; as was built inefficiently and was super slow due to the multilevel nested loops. Additionally, any attempt I made to patch together a working version of the desired algorithm would either perform worse when I enabled it or very similarly but with heavy overhead extending the time to complete a single configuration.

##### Sample Configuration

For a sample configuration, I ran my client on the ds-sample-config01.xml file provided in the ds-sim repo [1] provided to us as it was small enough to display but large enough to actually give the algorithm a change to work.

ds-jobs.xml

<jobs>

    <type name="short" minRunTime="1" maxRunTime="300" populationRate="60" />

    <type name="medium" minRunTime="301" maxRunTime="1800" populationRate="30" />

    <type name="long" minRunTime="1801" maxRunTime="100000" populationRate="10" />

    <job id="0" type="medium" submitTime="37" estRunTime="653" cores="3" memory="700" disk="3800" />

    <job id="1" type="medium" submitTime="60" estRunTime="2025" cores="2" memory="1500" disk="2900" />

    <job id="2" type="medium" submitTime="96" estRunTime="343" cores="2" memory="1500" disk="2100" />

    <job id="3" type="medium" submitTime="101" estRunTime="380" cores="2" memory="900" disk="2500" />

    <job id="4" type="short" submitTime="137" estRunTime="111" cores="1" memory="100" disk="2000" />

    <job id="5" type="short" submitTime="156" estRunTime="8" cores="3" memory="2700" disk="2600" />

    <job id="6" type="medium" submitTime="198" estRunTime="1074" cores="4" memory="4000" disk="7600" />

    <job id="7" type="medium" submitTime="225" estRunTime="442" cores="2" memory="500" disk="2100" />

    <job id="8" type="medium" submitTime="249" estRunTime="926" cores="1" memory="100" disk="800" />

    <job id="9" type="medium" submitTime="308" estRunTime="2010" cores="2" memory="600" disk="1500" />

</jobs>

ds-system.xml

<system>

    <servers>

        <server type="juju" limit="2" bootupTime="60" hourlyRate="0.20" coreCount="2" memory="4000" disk="16000" />

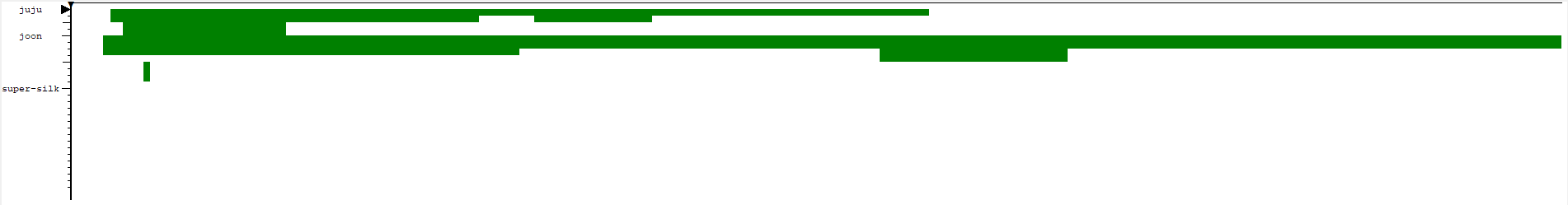
        <server type="joon" limit="2" bootupTime="60" hourlyRate="0.40" coreCount="4" memory="16000" disk="64000" />

        <server type="super-silk" limit="1" bootupTime="80" hourlyRate="0.80" coreCount="16" memory="64000" disk="512000" />

    </servers>

</system>

ds-viz visualisation



## Implementation

My algorithm was implemented in Java Development Kit version 14 which is the one of the latest versions of the JDK version also being used in stage 1 of the project. Specifically, it was developed using the Adopt JDK, which is an open-source implementation of the Java development kit. I used Microsoft’s Visual Studio Code to develop the entirety of the algorithm and further client. The client was tested on Ubuntu 20.04 LTS in a VirtualBox virtual machine.

I specifically chose Java 15 not just because it was good practice as it’s becoming the new standard, overtaking java 8, but because of several language features that are simply not available on previous versions of Java. A notable example is the enhanced switch statement used heavily throughout the project, being available since Java 14. This enhanced switch statement was only one of the many new Java standard library improvements with other components such as TCP socket connections, XML parsing and I/O handling all included in the standard library; allowing for no third-party libraries to be included in the project.

Continuing with the implementation, I will be discussing files that received significant changes from stage 1 of the client as well as the new class files that were created to fulfill the assignment specifications of stage 2.

#### HolyGrailAlgorithmHandler.java

This file contained the entirety of the behaviour of my new scheduling algorithm, using the original algorithm designed in stage 1 as a template to build up from. In this file it contained all the basic cases from a switch statement to interface with the ds-server. One of the main changes in this file is in the LSTJ case, where I used a combination of nested for loops and if statements to check conditions as well as multiple ArrayLists of custom types such as: listedJobsStructure and avaliableServersStructure. These class’ were used as object structures to help storing data in one place about jobs and servers respectively, containing only a few variables and a custom constructor for each. Further in this LSTJ case I use additional custom data structures such as Server and job to store the migration target server and job objects.

In this file I also have the server fitness function chooseProritiseRunningServers that is used in the initial job scheduling component to determine and return a server object to schedule a job too. In this function I again use a Server class variable and a foreach loop to perform the calculations.

Mostly the rest of the file is a copy of the original DefaultEventLoopProtocolHandler file that interfaces in the overall general client structure using the Actions class to represent OnEnterState behaviour and OnReceieveMessage behaviours to specify how to handle each message from the server. This is the main functionality of the client with the rest of the class’ and files acting as boilerplate or support class’ to all the files listed above.

#### ClientMain.java

Refactored the entire MultiPart command system, moving it out of the algorithm code and into the client code as it was common for any and all algorithms running on the client using DS-Sim server. I added another next action intent enumerator called MULTIPART and was a collection of all code to dependently handle all GETS and LSTJ commands to return their results. Internally, it contained a switch statement to determine the case of the command and then loop over all data sent from the server until it received a specific “.” Command to represent the end of the data. With the LSTJ command it was a slightly trickier case but worked almost the same but with a lot more jumping around the stack.

#### Action.java

The change in this file was relatively small and only included an additional action overload constructor to deal with multipart commands such as GETS and LSTJ more efficiently.

## Evaluation

The ds-sim evaluation of my algorithm was performed using the demoS2Final script provided and the setup sequence was as follows: “javac ClientMain.java”, the test was setup using the configuration “./demoS2Final “java Client -a HolyGrail” -o co -n -c S2DemoConfigs/”. This test aimed at comparing my algorithm against the baseline algorithms: ATL, FF, BF and WF and measured the averages of all the test scripts for each of the three testable metrics.

Results (averages of metrics):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Metric | ATL | FF | BF | WF | Holy Grail Algo (Mine) |
| Turnaround Time (TT) | 226351.22 | 1474.39 | 1475.56 | 5869.94 | 5745.50 |
| Resource Utilisation (RU) | 100 | 63.83 | 61.61 | 70.86 | 63.60 |
| Total Rental Cost (CO) | 405.00 | 643.29 | 642.28 | 655.72 | 592.84 |

Final Results: 10/10

Pros: My algorithm dealt with the intended outcome of reducing server cost very well and with heal to a certain standard up to my initial hypothesis that the other metrics would either

## Conclusion

In conclusion the algorithm wasn’t finished in time for submission and because of that hurt the potential performance of it as well as marks acquired by the automated tests. Due to time constrains and other factors such as (hard to code in the client from stage 1 and not enough time to refactor it entirely, which would have needed to be done), only the first half of the two components of my new algorithm were completed. In future I could have resolved these issues with ultimately more time and have produced an algorithm that would have successfully passed all tests and worked as intended.

## References

GitHub Repo: https://github.com/jarrod10/DS-Sim-Client.git

[1] Y. Lee, Y. Kim, and J. King, ‘ds-sim: A Distributed Systems Simulator User Guide’. [Online]. Available: https://github.com/distsys-MQ/ds-sim.