# COMP3100 Stage 2

Cory Macdonald, 45445672

## Introduction

Stage 2 involves implementing a distributed system scheduler with the goal of either reducing turnaround time, increasing resource utilisation, or reducing rental cost relative to the baseline algorithms (FF, BF and WF). Additionally, turnaround time should be lower than that of ATL.

In particular, this instance of the project is intended to reduce turnaround time.

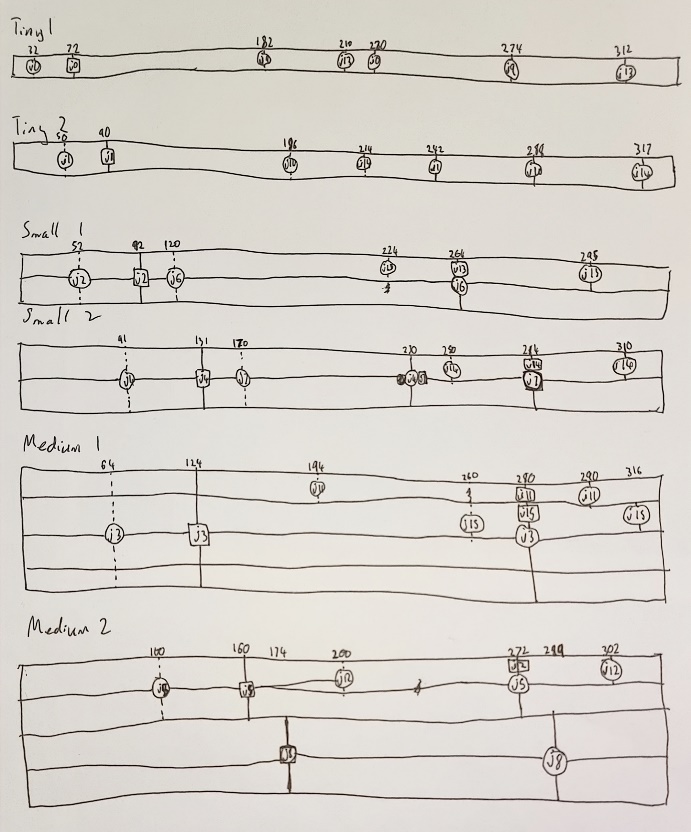
## Problem Definition

The problem here lies in finding the server that will most quickly complete a job after its’ submission. In theory, that means the server either has cores remaining on which the job can be run, or that it has the least jobs waiting in its’ queue. The function being implemented should do this by generating a value that is weighted towards the number of waiting jobs in the queue, so as to primarily find servers with few waiting jobs, reducing turnaround time.

## Algorithm Description

My implementation of the algorithm works by finding the capable (determined via data stored during the initial GETS All and a subsequent filter based on the jobs’ requirements) server with the minimum fitness value. Said fitness value is calculated as (8 \* the number of waiting jobs) + the number of running jobs + (the number of server cores – the number of cores required for the job). The number of server cores is stored as above, and the number of waiting and running jobs is found via two CNTJ commands per server as the algorithm iterates through all the options. The idea is that the server is determined primarily by the number of waiting jobs, but if there are only very few waiting jobs, or several servers with the same number of waiting jobs, the server can then be narrowed down by the one that is running the least jobs and that the current job will most appropriately fit onto.

An example scenario is below:



## Implementation

The algorithm stores information on the servers received from GETS All in an ArrayList of a custom Server class, which is provided a String in the constructor, and stores the server type and ID, as well as the cores, memory, and disk of the server based on said String. This class also provides functions to retrieve this information and to check if it can run a certain job provided to it, as well as some functions for testing or that were dropped during development.

Jobs are stored in another ArrayList, of which is similarly populated by a custom Job class that can be provided a String as a constructor to store the ID and requirements for core, memory and disk. It also has an equals function and other variables that were dropped during development.

The algorithm will repeatedly get jobs from the server, filter the compatible servers with that job, then pass the job and compatible servers to the selectServer function, which simply runs turnaroundTime by default. This iterates through the provided servers to find the one with the minimum fitness value, then the type and ID of that server are returned in a custom Tuple class that stores a String and an int. The job is then scheduled using this String and int, and this continues until NONE is sent instead of a job. JCPL messages are effectively ignored.

Additionally, there are functions to simplify sending and receiving messages, including a receive function that checks that the server sends a specific, expected message.

## Evaluation

The algorithm was tested using the sample configs provided, and the test\_results script provided.

As for results, this algorithm provided a 28.98% improvement in turnaround time, 5.43% improvement in resource utilisation and a 3.15% improvement in rental cost. It worked out to be better than WF in terms of turnaround time, but worse than either FF or BF, better than FF and BF for resource utilisation, but slightly worse than WF, and better than all 3 in rental cost. This was not the intended result of the algorithm, so I tried changing the fitness function to add server cores / job cores instead of server cores – job cores, with the idea that this would favour servers in which there were few running jobs and the highest likelihood that there were cores remaining where a small job could be added, therefore reducing turnaround time by using extra cores to get more jobs done concurrently. This did improve turnaround time to ~32% improvement, but decreased resource utilisation and rental cost by a small margin that was enough to potentially reduce the marks from 2.4, therefore I decided to roll back the change.

The pros of the algorithm are that it results in better resource utilisation by favouring smaller servers that fit a job better, but a con is that this increases turnaround time by not using other servers with spare room to run the job.

## Conclusion

Overall, my algorithm is not an improvement when it comes to turnaround time but does better in resource utilisation and rental cost than most of the baseline algorithms. Having drawn the execution diagram above, I realise now that this is at least partly as a result of the algorithm favouring servers with jobs running and no cores free over those with jobs running and cores free. As a result, this increases resource utilisation, because jobs get placed first on the smaller servers, which will therefore be running something more often.

## References

<https://github.com/Megukaphii/COMP3100-Project>

### Installation

Simply copy + paste the Client.java file and classes folder, run javac Client.java, then the client can be run with java Client.java.