Project report: Stage 2
COMP3100 Distributed Systems, S2, 2022
SID: 4481 6936, Name: Jacob Temperley

1 Introduction

This projects aim is to create a load balancing algorithm that meets a set of performance objectives that are derived from a set of predefined baseline algorithms. We will first define load-balancing, then outline the algorithms to compare against, and the performance metrics.

Load balancing is the process of assigning a series of jobs to a set of servers that con process them. Different algorithms can be used to assign the jobs to servers, and the characteristics of the jobs combined with the algorithm usually determines how effective different algorithms are, i.e. it is very rare that any single algorithm is 'better' than another in all cases.

We will be comparing the new algorithm to four existing algorithms: first-capable, first-fit, best-fit, and worst-fit. These algorithms are relatively simple, first-capable for example chooses the first server which is capable of running the current job regardless of whether it is already running a job. Understanding how these algorithms work can give us a starting point for building a new algorithm, and understanding what they don't do can help us create better algorithms.

There are three performance metrics we can use to compare the effectiveness of our algorithms: average turnaround time, average resource utilisation, and total cost. Increasing the performance in one of these usually results in decreasing the performance in another, so will focus on decreasing the average turnaround time: the time between submission of a job, and the completion of that job.

2 Problem definition

If we have to sacrifice one performance metric to increase another, we have to make a decision on which we will focus on. We can rule out utilisation because of the way it is calculated; the baseline algorithms do not migrate jobs and servers are not simulating failures (at least I don't think they are), which means the utilisation will always be 1, and we can't have more than the maximum utilisation. That leaves turnaround time or cost.

Both average turnaround time and cost are valid focus' when creating a new algorithm. Decreasing total cost would mean we have to know the individual cost of each server, then prioritise by cost. Unfortunately cost is only available through the xml configuration files that usually are specified when running ds-server, and without know the names of the files, becomes impossible. This leaves only turnaround time for our focus.

3 Algorithm description

How can we decrease the average turnaround time? Examining the baseline algorithms, it becomes clear that they all have one characteristic in common: they never migrate scheduled jobs to use resources that have been made available post-schedule. An example of this is depicted in figure 1, where we can see server 2 is unused for the majority of the simulation because the algorithm doesn't identify that a resource has become available. Clearly job 3 could have been executed on server 2 in parallel with job 1 as in figure 2, but the order of scheduling prevents this. Having servers that sit idly by while jobs are waiting is a huge waste of time that increases the average waiting time. To achieve this effect with the simulator, we can listen for a JCPL message which notifies us a job has completed and a server might be idle. We can then use the MIGJ command to move a waiting job to an idle server.

What happens to the cost when we use a different server? Cost is calculated by multiplying the runtime by the server cost/second. If server 1 and 2 are of the same type, then regardless of where we place job 3 (behind job 1 or 2), the cost of running in either scenario does not change. Only when server 1 and 2 have different running cost will the total cost change. Using this principle, we can decrease the average turnaround time.

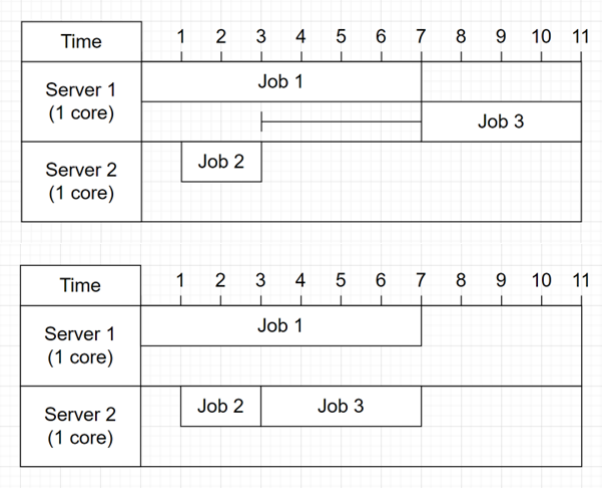

Figure 1: An example of a scenario where migrating a waiting job (job 3) to an available server (server 2) decreases overall turnaround time.

4 Implementation

The implementation for the check algorithm is relatively simple. The pseudo-code is as follows:

1. Receive a job.
2. Search for the first available server.
3. If a server is available.
   (a) Assign a job to the server
4. Else
   (a) Assign the job to the first capable server.
5. If a server becomes idle through completing a job
   (a) Find a waiting job that fits on the server.
   (b) Migrate the waiting job to the idle server.

The class diagram for the algorithm implementation is in figure 3. The Client and Check classes are where the bulk of the functionality takes place; Client handles general ds-sim related messaging and program structure, Check contains the algorithm specific instructions like how to make scheduling decisions. The other classes are used as data structures to simplify ds-sim interaction, including Job, Server, JobState, and JobComplete. Generally, Client directly messages the simulation, receives messages in return, processes the information into data structures, and passes the data to a Scheduler.

5 Results & discussion

The Check algorithm excels in turnaround time (figure 3), primarily through its use of job migration which allows jobs scheduling to adapt based on job completion. It can also adapt to realised running times, i.e. the algorithm can migrate jobs when the actual running time of a job is shorter or longer than the estimated runtime. Compared to the baseline algorithms which 'fire-and-forget', it seems obvious that considering the dynamic runtime environment will improve the turnaround time.

Comparing Check's average utilisation (figure), we can see that its performance (63.71%) is on-par with the baseline algorithms (FF = 63.3%, BF = 60.72%, WF = 68.79%, FC = 94.37%). This shows that our idea theory about negligent cost differences mostly works. There are hypothetical cases which increase the total cost; If a job completes on a 4-core server, searches for a waiting job, and finds a job which requires a single core, then 3 cores will be wasted where more jobs could have potentially run in parallel. Also, upgrading a job

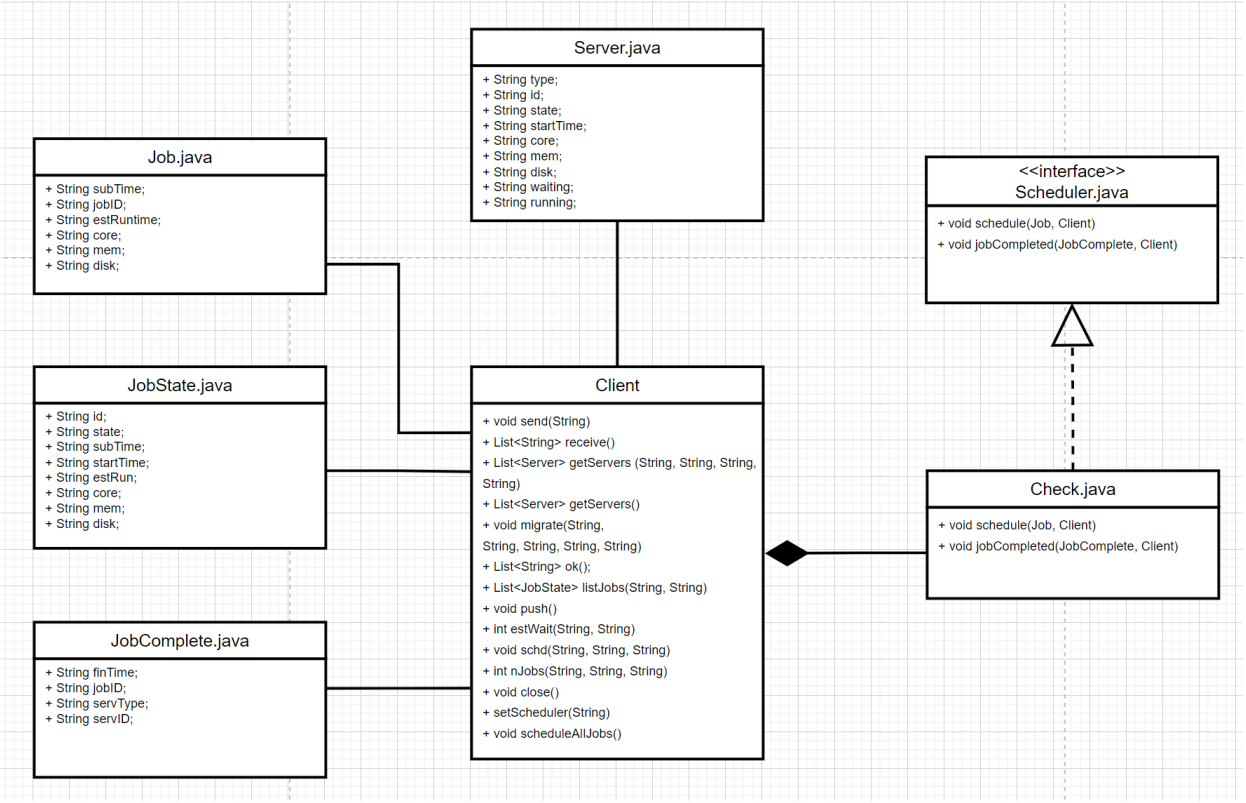

Figure 2: The class diagram for the Client, Scheduler algorithm, and data structures used.

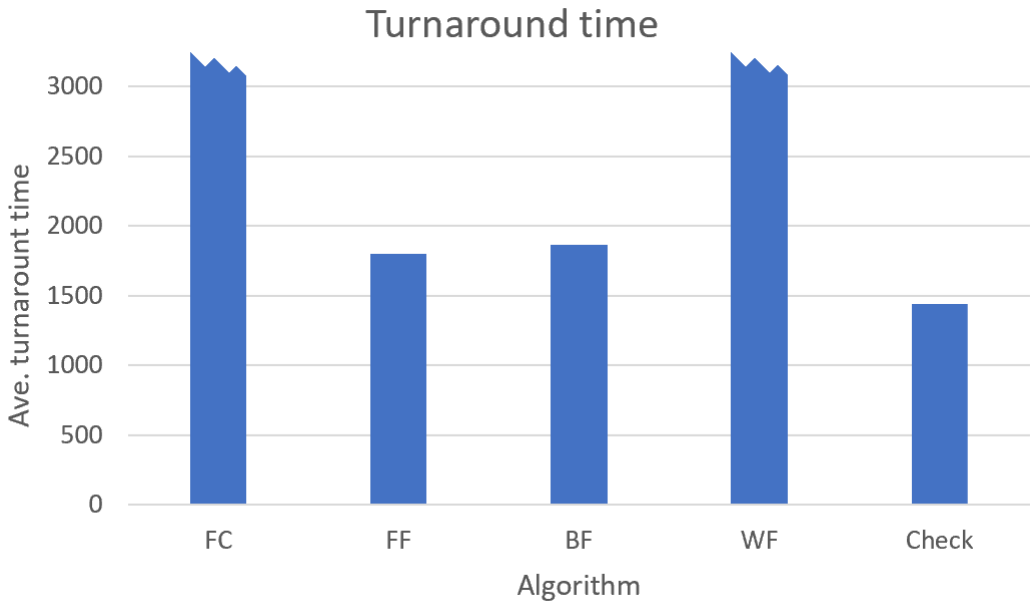

Figure 3: The comparison of turnaround time (seconds) for baseline algorithms and the new Check algorithm.

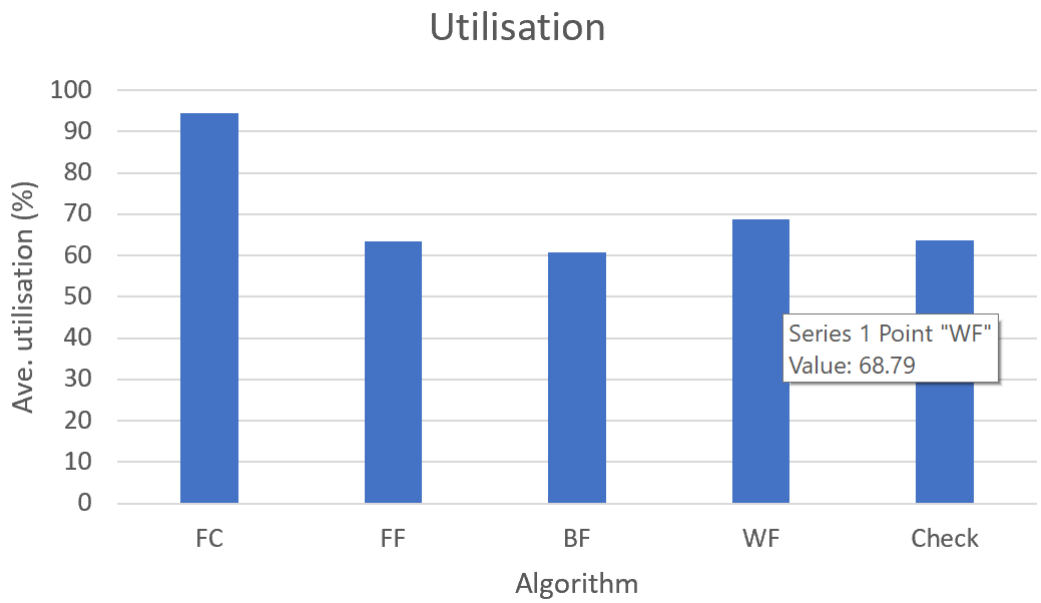

Figure 4: The comparison of utilisation (percentage) for baseline algorithms and the new Check algorithm.

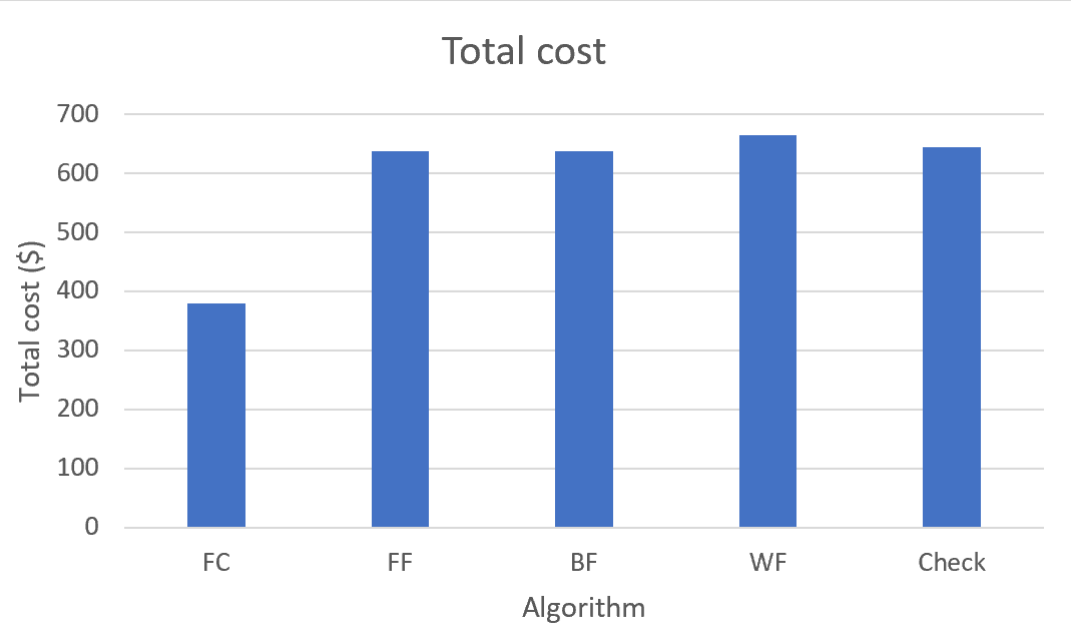

Figure 5: The comparison of total cost ($) for baseline algorithms and the new Check algorithm.

that would have run on a low cost server (but is currently unavailable) to a high cost (but available) server, will increase the cost of running the job and consequently the total cost.

Finally, cost is roughly on par with FF, BF, and WF, but all of these are almost double FC (figure). If we consider that FC will only assign jobs to server types that have an ID of 0 (because IDs greater than 0 will never be the first discovered in a first capable search), then we can convince ourselves that a set of servers with multiples of each server type will only utilise a small fraction of the servers which minimises wasted server time. This goes hand-in-hand with high utilisation (FC = 94.37%) which stands out from the rest of the baseline algorithms.

The cost resource wastage that potentially occurs could be countered by: finding multiple waiting jobs to run on large servers, using a best fit algorithm in tandem with Check to place the best job on completion, or both of the mentioned. If we do both, then we can place a best fitting job on an idle server and fill up leftover space with more concurrent jobs.

6 Github link

https://github.com/jetemperley/COMP3100ass