Project report: Stage 2
COMP3100 Distributed Systems, S2, 2022
SID: 44849354, Name: Michael Merrick

1 Introduction

In this report, I aim to explain the Stage 2 objectives of this project, as well as show the process I went through to reach those objectives. The stage 2 objectives were to design and implement a scheduling algorithm that optimises average turnaround time. In particular, the scheduling algorithm, coded as part of my client, should schedule jobs to servers aiming to minimise the average turnaround time without sacrificing too much of other performance metrics, such as resource utilisation and server rental cost. The project was designed and constructed over a five-week period, both in and out of class, and can be found on GitHub here [1].
The rest of this report is organised as follows. Section 2 gives a high-level overview of the system and its workflow; Section 3 will give a brief description of the design philosophy involved, and Section 4 will give an indepth explanation of the implementation specific information.

2 Problem Definition

As noted in the introduction, the first objective of this stage was to minimize average turnaround time. This would be evaluated by comparison to several other algorithms, such as First Fit(FF), Best Fit(BF), Worst Fit(WF), lowest round robin(LRR) and First Capable(FC). The secondary objective was to fulfil the first objective without sacrificing too much of other performance metrics. This would be evaluated by comparing other metrics such as end time, avg/ waiting time, total cost, avg utilization, and efficient usage. My function was coded to meet this requirement by prioritising servers that were ready to fulfil jobs first, then prioritising servers with no jobs running, then finally choosing by priority of lowest estimated run time of remaining jobs. This fulfilled the primary objective quite well; however, it was met with limited success in pursuit of the secondary objective. As will be demonstrated below, pursuing the first objective too heavily impeded success at the second objective metrics.

3 Algorithm Description

Here I will provide a simple example scheduling scenario to give an example of how my scheduling algorithm works; however, note that in the server table below I have not included bootupTime or hourlyRate, as those are not included in algorithm decisions. The jobs are of somewhat odd dimensions; however, this is to more clearly demonstrate the decision making process for the algorithm.

JobID	estRuntime	Cores	Memory	Disk
0	320	2	2000	8000
1	320	2	2000	8000
2	320	2	2000	8000
3	320	2	2000	8000
4	640	4	8000	32000
5	640	4	8000	32000
6	1280	4	8000	64000
7	1280	4	8000	64000
8	1280	4	8000	64000

Table 1: Example jobs.

Server	Number	Cores	Memory	Disk
Small	3	2	4000	16000
Medium	2	4	16000	64000
Large	2	8	32000	256000

Table 2: Example servers.

In this example, my scheduling algorithm would take the first JOBN (JOBN 0 0 320 2 2000 8000) and request a GETS Avail for a job of that dimensions, which would return all servers, beginning with SMALL 0 and Ending with LARGE 0. It would then iterate through this list, and find that SMALL 0 has 0 jobs scheduled on it and immediately break the iteration, scheduling it on SMALL 0. Job IDs 1 would be assigned to SMALL 1, Job ID 2 to SMALL 2, Job ID 3 to MEDIUM 0, and so on, visualised below:

Job	Assigned Server	Total Est Runtime
0	SMALL 0	320
1	SMALL 1	320
2	SMALL 2	320
3	MEDIUM 0	320
4	MEDIUM 1	640
5	LARGE 0	640
6	LARGE 1	1280
7	LARGE 0	1920
8	LARGE 1	2560

Table 3: Example Schedule.

Once it hits job ID 7, it has no more servers with 0 jobs to place it on, so the GETS avail will have returned two servers: LARGE 0 and LARGE 1. LARGE 0 has been assigned job 5, with 640 total runtime, so it's total runtime will be lower than LARGE 1, which has been assigned job 6, with 1280 total runtime. As a result, Job 7 will be placed on LARGE 0, taking the overall total runtime to 1920. Job 8 will then be placed on the lower EST runtime of the two servers: LARGE 1.

4 Implementation Details

I primarily implemented this on top of stage 1 code; there were two large shifts in how the algorithm processed data, however. First, I moved from using GETS Capable to using GETS Avail instead, with GETS Capable only being executed in case of there being 0 results from a GET Avail command. The returned server list from either GETS command would be run through my function lstj(), which sent a request for LSTJ each server listed as per the documentation in the form LTSJ [server type] [server id]. It would then check the number of jobs returned by the LSTJ request for the server, and would iterate through each accordingly adding up the estimated run time for each server. It would then calculate if the total estimated run time was lower than the current lowest estimated run time to record in the iteration, and if so, save the server type and ID for later SCHD assignment as per usual.

5 Evaluation

For my simulation I used the ds-sample-config files 04 and 05 included in the ds-sim repository. 05 is a smaller size configuration with 500 jobs, whereas 04 is a larger configuration with 3000. I chose these two configuration to capture the results with both large and small sample size job loads.

5.1 Raw Data

Algorithm	End Time	Avg. Wait Time	Avg. Turn time	Total Cost	avg. Util.	Eff. Usage
LRR	142191	3740	8517	$906.89	80.30%	82.80%
WF	208461	4978	9755	$882.17	93.50%	93.62%
FF	141951	7	4784	$489.38	63.26%	62.24%
BF	141951	7	4784	$434.26	64.19%	64.56%
FC	1019997	227419	232196	$253.90	99.99%	100%
MC	141041	7	4784	$489.38	63.26%	62.24%

Table 4: Example of metrics when run on ds-sample-config05.

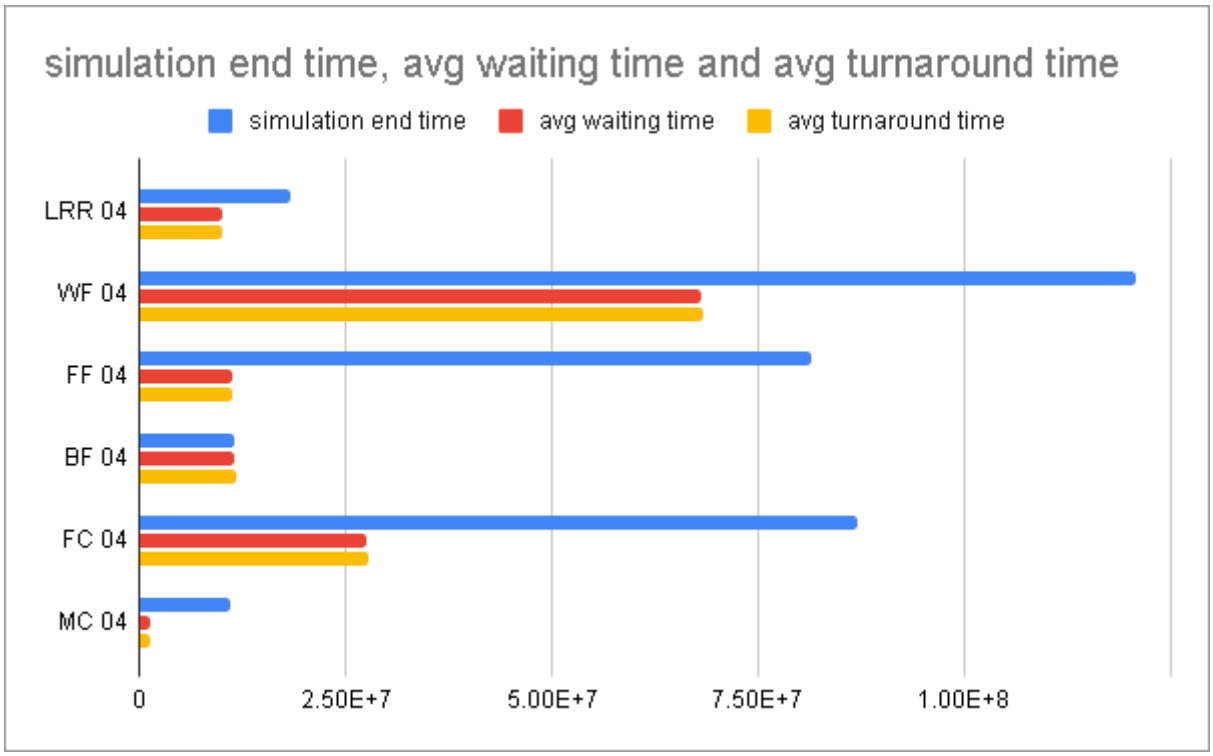
As can be seen in the above table, MC(myClient) performs about as well as FF and BF in regards to runaround time, beating them out slightly in terms of end time, and being slightly more expensive. The avg. Util. and Eff. usage of all three are comparatively low, highlighting a possible issue with MC.

Algorithm	End Time	Avg. Wait Time	Avg. Turn time	Total Cost	avg. Util.	Eff. Usage
LRR	1837195	10015818	10112779	$1,129,039.88	100%	100%
WF	120664880	68141486	68238447	$914,935.88	99.50%	100%
FF	81523986	11242606	11339567	$684,172.88	92.85%	95.49%
BF	11500334	11599334	11696295	$677,852.12	92.32%	95.66%
FC	87052778	27570403	27667364	$673.967.88	100%	100%
MC	11002144	1297872	1394833	$681,701.19	92.99%	95.67%

Table 5: Example of metrics when run on ds-sample-config04.

This table demonstrates that on a larger scale, MC suffers from some cost, util and Eff usage issues, although its Turnover time is performing well.

5.2 Graphs



Here is a graph, demonstrating how strongly MC performs in these three metrics.

5.3 Pros

As the graph demonstrates, MC performs strongly in turnover time, avg waiting time, and overall simulation end time.

5.4 Cons

However, it is clear that the utilization rate and effective usage rate are quite poor, suffering extremely low rates and consequently driving the total cost upwards as a result. Larger sample sizes (too large to fit in this report) reveal that this effect becomes of noticeably larger magnitude and greatly impacts the algorithms ability to fulfill its secondary objectives.

6 Conclusion

While myClient performs strongly in fulfilling its primary objective, I neglected to provide sufficient support for the algorithm to meet its secondary objective by focusing too much on a single metric. For future development I would ideally incorporate an additional decision making layer into the algorithm's process, wherein it prioritises servers that have already been spun up and are idle over inactive servers. I would also ideally allow it to decide in a more nuanced manner where it is prudent to stack multiple jobs on a single server that has

capacity rather than spread them out across every empty server. I was unable to properly implement these ideas within the required timeframe, however, and the project's quality suffered for it.

References

[1] M. Merrick, "Stage 1." https://github.com/myrmecoleonn/COMP3100-Assignment1, 2022.