**Distributed Systems Job Scheduler – Stage 3**

Bradley Anderson – 45229678

**Introduction**

In the past 2 stages of this project, the aim was to create a job scheduler that could receive jobs from clients and scheduling them to appropriate servers that can handle the jobs based on core count. The system would be completely autonomous and would run through a job queue until completion and prints a report detailing utilization of servers and costs to run the scheduled jobs.

Stage 1 involved sending all incoming jobs to the largest server that was available. Stage 2 involved the introduction of 3 distinct scheduling algorithms. First Fit took the list of available servers and scheduled the incoming job to the first available one, regardless of core count and other requirements. Best Fit finds the closest possible match between the required cores for the job and the available cores amongst all servers and schedules the job to the best fitting server. Finally, Worst Fit takes the server with the largest core count difference between it and the job to be scheduled and places the job on that server.

This time around in Stage 3, the job scheduling algorithm needs to be refined beyond that of the 3 baseline algorithms from the 2nd stage. This will involve either modifying any of the 3 existing algorithms or creating a new algorithm from scratch. The aim of creating this enhanced algorithm is to achieve optimization in one or more aspects of the job scheduling process. Weather that be time, cost, server utilization or a combination of the three, this enhanced algorithm should aim to achieve a better result in some of these aspects than what the baseline algorithms are able to accomplish.

**Problem Definition**

As stated before, the objective of this stage is to improve upon the baseline algorithms in some aspect, the three baseline algorithms First Fit, Worst Fit and Best Fit all have advantages but disadvantages as the same time in certain areas. For example the First Fit will take the first available server, minimizing turnover but potentially increasing cost depending on how the server list was read in and sorted. Best Fit aims to fit jobs with servers that very closely fit their requirements, this has the advantage of saving money but may increase turnover time as the client needs to search for the best fitting server for the job. Worst Fit has the worst cost effectiveness as it automatically sends every job to the server with the biggest core disparity. This could potentially have the effect of increasing turnover time since jobs all queued to only the largest servers may incur a waiting period before being handled by the server. First Fit takes the first suitable server and assigns the job to that, this has the benefit of fast turnover since only available servers are scheduled to.

With these algorithms and their benefits and downsides in mind, there are two proposed solution algorithms that aim to improve upon these three in one area or another. These two algorithms are Cheap Fit, which aims to further reduce costs by fine-tuning scheduling decisions to also include memory and disk requirements at the cost of turnover time and client-side runtime and Fast Fit, which equally distributes all jobs across the largest server type in an attempt to decrease turnover time at the expense of server running costs.

**Algorithm 1: Cheap Fit**

Cheap Fit is an algorithm designed to further improve Best Fit’s capabilities of closely fitting a job based on it’s core count requirements by also considering memory and disk space as well. For example where Best Fit would only take the server with the least difference between its core count and that of the job’s, Cheap Fit would go one step further and check if another server with the same core count has a lower difference in memory and disk space it will use that one over the previous server.

A screenshot of a cell phone

Description automatically generated

Figure - Output from ds-config-s3.1 for Cheap Fit

This sample configuration comes from ds-config-s3-1, which has 2 small servers with 1 core and 2 medium servers with 2 cores.

Cheap Fit works as such:

1. RESC Avail is called to see what available servers there are
2. IF RESC Avail comes back empty, RESC All is used instead.
3. The result from RESC Avail/All is placed into an arrayList and sorted
4. The last item in the list is taken and stored as the current smallest server.
5. The list is traversed to find the smallest core count server in the list that still fits the job’s requirements. If so, then move on to check the memory and then disk space. If all three turn out to be smaller, then that server is set as the new smallest server. If RESC Avail is used, skip to Step 7.
6. IF RESC All was used, the required core count for the job is compared to the current server’s original core count from ogServers, if the server has the required cores to handle the job then it we move onto Step 7.
7. Once the traversal of the server list is done the smallest server is then used as the one the job is scheduled to.

The idea behind this is that further considering memory and disk size to find a smaller server, it will save costs at the expense of turnaround time.

**Algorithm 2: Fast Fit**

A screenshot of a social media post

Description automatically generatedFast Fit is an algorithm that aims to speed up Worst Fit’s turnover time by distributing jobs across the largest kind of server, rather than scheduling all the jobs on only one of the largest servers. Distributing jobs across all the largest servers will increase costs dramatically but should reduce turnover time as queues for each server will be shorter than that of queuing all jobs on a single, albeit large server.

Figure - Output from ds-config-s3.1 for Fast Fit

Just like before, this config comes from ds-config-s3-1.

Fast Fit works as such:

1. The list of servers is put into an arrayList and sorted at the start of execution. Servers are sorted by smallest core count to largest core count. This list is not updated beyond this point.
2. The last server in the list has its name taken and stored in a separate string variable.
3. The list of servers is traversed through backwards to count the number of servers with that same server name as the last server. This is to assess how many of that kind of server there are.
4. A modular counter is set to indicate how far backwards in the list you’d need to go to start off with the first of the largest kind of server.
5. The list of servers is then traversed starting from the position where the first of the largest kind of servers are in the list.
6. Each new incoming job is scheduled to the next server onwards in the list until it reaches the end of the list, at which point the loop restarts from the first of the largest kind of server and the cycle begins anew.

The list of servers is only ever taken once, so there’s no regard for if the server is available to do the job or not. The only thing that matters is that there is a relative even distribution of jobs across all the largest kind of server. This is to hopefully reduce the queue times from that of worst fit by spreading jobs out, at the expense of increased costs.

**Implementation**

Each algorithm is implemented as a single Client class with each algorithm called depending on the input arguments given to the client. Each important function and data structure are kept within this client class. These are as follows:

**inputArrayString**: A string array which stores each response that is read from the server.

**BFFServer**: A string array which stores all the information about the most ideal server that a job is to be scheduled to for the selected algorithm.

**ogServers**: An ArrayList of Strings that stores the list of the initial states of all the servers used by the current server configuration. This list is sorted from smallest core count to largest and is only ever updated at the start of execution.

**allServers**: An ArrayList of Strings that stores the updated list of all servers returned by a RESC All call. This list is updated to reflect any servers that may be busy or that might have freed up by the time RESC All is called again.

**serverList**: An ArrayList of Strings that stores the updated list of servers from RESC Avail. If the list is empty, a RESC All call is then used in it’s place for scheduling jobs.

**sortType**: A String that stores the 2nd argument in the input string for which kind of scheduling algorithm the program will be run with. This string will either be ‘cf’ for Cheap Fit or ‘fastfit’ for Fast Fit. Depending on what sortType is depends on whether allServers or serverList is updated and used in the scheduling process.

**sortList**: A function that’s used to sort a server list that’s passed to it in ascending order, going from smallest number of cores, memory and disk space to the largest in each respective area. This function is called on ogServers once to get it in it’s ordered state and every time allServers and serverList is updated to keep them in sorted order.

**cheapFit & fastFit**: Are both scheduling algorithms that have been described above and make use of all the data types seen above to store and sort server lists to find the best possible server depending on their objectives.

**Evaluation of Cheap Fit**

The objective of the Cheap Fit algorithm was to be cheaper and more resource efficient or at least equal that of Best Fit at the expense of turnover time. While it achieved this goal in a few test cases the overall result was that Cheap Fit ended up performing worse than that of Best Fit.

All 3 figures come from ds-config-s3-1, Cheap Fit in this scenario performs better all-around than all other algorithms in all aspects. Cheap Fit performs 20% better than Best Fit in cost and by over 90% in turnover time despite having stricter scheduling criteria. However this is one of the best case scenarios for this algorithm, where there aren’t many servers or a large amount of each server type.

These figures come from simple\_config2, a much larger file which shows one of the many test cases where the Cheap Fit fails to meet it’s expected outcome of being superior to that of Best Fit. It’s cost is increased by 2.5%, with a very slight increase in wait time and a 1% flat increase in server utilization. This configuration shares servers with simple\_config3, however this configuration has a much higher amount of larger scale jobs which take longer to complete.

In a larger example, these figures come from simple\_config3, These figures show the intended outcome from this algorithm, to decrease costs (in this case by around 7% of that of Best Fit while having a slightly longer turnaround time (24 ticks versus 20 for Best Fit) and in this instance a higher server utilization. This configuration features 5 server types with 20 of each, so there’s a good variance in how many servers there are to schedule jobs to.

The results from the 16 test cases show that out of 16, 6 of them allow Cheap Fit to achieve an equal or better result cost wise than Best Fit. For the remaining test cases, the cost increase was observed to be anywhere from 2% to 10% higher, however unlike what was predicted, most of the time the total turnover time was observed to be equal to or faster than that of Best Fit in all but one of the cases. (The exception being config\_simple3 with an average 24 tick wait as opposed to 20 for Best Fit). The Cheap Fit algorithm, while designed to end up being cheaper, for the most part hasn’t met this goal. This can be attributed to the fact that core count, memory and disk sizes don’t necessarily increase costs. To add to this, the algorithm doesn’t take into account server availability time, if a server is available in 5 ticks but doesn’t show up on RESC Avail, the algorithm will choose a server that does have available cores but is far larger in order to fit the job. This can have the effect of driving up costs if the larger server ends up being more expensive but will have the opposite effect if using it turns out to be cheaper. The slightly higher cost may be justifiable if the aim is to achieve higher server utilization, as a 2% to 10% increase may not be that severe for a large company that wishes to get the most out of each server. However the algorithm hasn’t managed to universally improve over that of Best Fit, for which it hasn’t met it’s goal.

**Evaluation: Fast Fit**

Fast Fit is a different story to that of Cheap Fit. Fast Fit’s objective was to decrease turnover of that of Worst Fit, overall this objective hasn’t been successful. While it wasn’t successful in its aim, it has unexpectedly had superior results in costs for certain test cases, even over that of Cheap Fit or Best Fit.

These figures are from config\_simple7, As can be seen, the turnover time far exceeds that of the other algorithms despite the distributed job load. However, in certain cases such as this, the cost ends up being about 25% less costly than that of any other algorithm. This may be one of the cases where the largest kind of server is cheaper than that of one of the smaller, more fitting servers for each job.

Figures from ds-config-s3-1, the only test case where Fast Fit has bested Worst Fit in terms of turnover time. This is most likely since this is the smallest test case with very simple jobs and small servers. By spreading jobs out amongst the 2 servers, the algorithm saves time by distributing over 2 servers rather than one.

These figures come from sample\_config3. This data is representative of what a vast majority of the test cases are like for the Fast Fit algorithm, often times it ends up being more costly, utilizes less resources and has a vastly higher turnaround time due to a waiting time that’s in the hundreds, if not thousands of times larger.

Looking at the test results out of the 16 test cases there were only 6 cases where the turnover time was like that or better than the Worst Fit algorithm, only one of which was a noticeable improvement. Since the only aim of this algorithm was to improve waiting times, thus improve turnover times and it hasn’t been able to accomplish that with any consistency bar one test case. The anomalies seen where cost was reduced was seen in 3 out of 16 test cases, this is most likely due to the issues with Cheap Fit where larger servers don’t necessarily mean more expensive. The longer waiting times may be due to the sheer number of jobs assigned to every server, however this doesn’t explain how Worst Fit is able to place a large queue of jobs on only the largest server and still have a far superior turnaround time than Fast Fit is able to achieve. Overall the way that Fast Fit handles job scheduling makes it hard to justify ever using it in any scenario. The algorithm doesn’t decrease turnover time over that of worst fit and ends up costing more since it’s using more larger scale servers. The algorithm hasn’t achieved it’s goals overall, and the cases where it does improve upon other algorithms are fringe cases when the actual algorithm wouldn’t hold up in more realistic scenarios.

**Conclusion**

Overall, the findings from test cases demonstrate that while each algorithm can perform in certain scenarios, there is no universal fix across all configurations that these algorithms can achieve their intended goals. With Cheap Fit, implementing a way to look at the price of each server as a measure of cost over time would allow for cheaper scheduling decisions, as well as a look at the time left on busy servers to gauge weather waiting to schedule on a server that’s about to become available will is a smarter option. For Fast Fit, using the RESC Type command on the largest type of server returns information about that kind of server. Using this instead of traversing the ogServers list to find the amount of the largest kind of server. Another change would be to consider other server types and distribute jobs similarly to Best Fit, only jobs that closely match server requirements would be scheduled but every time a server of a certain type gets filled, the next one over gets scheduled to. For example if a job comes in needing a medium server, medium 0 would get scheduled to, if another job comes in needing the same kind of server medium 1 would be up next. However if the next job that comes in needs a large server, large 0 would be scheduled to as it’s the first server of its kind.

These changes would be made to further improve these algorithms to potentially make them applicable to other server configurations and make them more proficient at improving the aspect that each algorithm is aiming for, cost in exchange for turnaround time in Cheap Fit’s case and turnaround time in exchange for cost in Fast Fit’s case

**References**

Github Repository: <https://github.com/AussiePanda/Comp_335-Job_Scheduler/tree/Best-Fit-Bradley-Anderson-45229678>