Cost Reduction Distributed Systems Client-Server

1. Participants

Aidan Ho	**45256128**

2. Introduction

Distributed systems are a collection of autonomous computing elements that communicate with each other and coordinate activities in an orderly manner, with the aim of emerging as a coherent system of end-users. This project involves building a program which schedules jobs to specified servers in a simulated distributed system environment. The simulation is based on the TCP client-server architecture where the client program connects to the server side to receive the corresponding data from the byte stream. There are several performance objectives and constraints when jobs are scheduled in the distributed system. Such as execution time, turnaround time, resource utilisation and the costs of execution. The client side of the project will tackle one or more of these constraints to improve better optimisation of the scheduling.

2.1 Scope

Stage 2 specifications have led to the assumption that the goal of this project is to develop an algorithm to better optimise the performance of scheduling jobs in the distributed system. Specifically, the client should optimise one or more objectives such as minimise the average turnaround time, maximising the average resource utilisation or minimising the total server rental cost. However, these objectives are conflicting performance objectives, the optimisation of one objective may lead to sacrificing the optimisation of another objective. For example, maximising resource utilisation by employing more resources may end up resulting in higher costs. Therefore, the objective function of the algorithm will have to have a defined purpose of which objective it optimises.

3. Problem

When looking at the objective function, we can see that there is a correlation between each objective and how optimising certain objectives could disadvantage another. By choosing to optimise the cost of execution there is little to no effect or negative effects for turnaround time and utilisation. The total server rental cost is made up by adding the cost of all servers that has handled jobs. Each server has a specified hourly rate where servers with less resources cost less and servers with more resources cost more. The hourly rate of a server is then multiplied by the turnaround time in hourly form. To reduce the total server costs the turnaround time must be reduced for the jobs that are scheduled as having a higher turnaround time would increase the cost. Also, to reduce cost there is a need to maximise the number of servers used when the algorithm needs to schedule more jobs. This is to ensure that servers are not backlogged or minimise the number of jobs that are backlogged. Backlogging a server would mean that the job would have to wait until there is enough resources in a server before it can run, which can increase the turnaround time for a job. By maximising the amount of servers that a job can handle we can cut the amount of jobs that are backlogged resulting in improving the turnaround time for jobs and lowering the total server cost. In improving the cost of execution, we can maximise the number of servers used while as well looking to minimise the turnaround time in each job.

4. Algorithm Description

The cost reduction Algorithm is an algorithm that deals with allocating the smallest available free partition that meets the requirement of the requesting process. The algorithm when implemented in a job scheduler for distributed systems works by undertaking four steps: each time a job is called upon the command GETS Capable is stored into a server array which will then be sorted into ascending order by the number of cores. It then searches for the first available server that can handle the job while it is in its booting, active or idle state. If there are no available cores in either of those three states it will find the first inactive server that can handle the job. The final step is if there are no available cores in any servers for the job to handle, it will look for the

least amount of waiting jobs in a server and schedule the job there. This is to ensure that when jobs are backlog it does not go into one server but evenly distributed reducing the turnaround time.

4.1 Scheduling Scenario

In this scheduling Scenario it uses the sample configuration “ds-sample-config01” and below the scheduling results after it runs through the cost reduction algorithm. The sample configuration has 5 servers; Juju 0 and Juju 1 which holds 2 cores, Joon 0 and Joon 1 which holds 4 cores and Super-silk 0 which holds 16 cores. When scheduling the first job since all servers are inactive it schedules the first capable job being Joon 0 with 4 available cores. Coming to the second job it first scans the booting server Joon 0 to see if it has the available cores. Since the core required for second job is 2 and the first job required 3 cores, Joon 0 does not have enough cores required for the job. The algorithm will then look for capable inactive servers with available jobs and schedule the first one. The fifth job requires one core, it first looks over three servers that are in the state of booting and active in the order of Juju1, Joon 0 and Joon 1. Juju 1 is looked at first but is not scheduled due to having zero cores available. The second scan looked at Joon 0 which is booting has 1 core available, since the current job only requires one core it breaks out of the algorithm and schedules the job in Joon 0. In Job 8 the number of cores needed is 2. It schedules into Joon 1 since it is the first server in Active state that has enough cores to run the job. Previously the two jobs before required more cores than 2 which then schedule the jobs in Super-silk. The last two jobs are schedule in super-silk as all the remaining servers do not have enough core to handle the jobs and super-silk still has enough cores to handle the jobs.

RCVD SCHD 0 joon 0	RCVD SCHD 1 juju 0	RCVD SCHD 2 juju 1	RCVD SCHD 3 joon 1
RCVD SCHD 4 joon 0	RCVD SCHD 5 super-silk 0	RCVD SCHD 6 super-silk 0	
RCVD SCHD 7 joon 1	RCVD SCHD 8 super-silk 0	RCVD SCHD 9 super-silk 0	

5. Implementation

When implementing the scheduling algorithm, I created separate functions that unanimously worked together due to the various commands that had to run with the job scheduler. This presented the code in a more organised fashion which could be easily traceable due to the clear understanding of what was happening in each function. First storing most of our global strings, integers, and servers into a class at the beginning.

The run() function is the main function where we stored various variables such as our socket, InputStreamReader, DataInputStream, DataOutputStream and BufferReader. It was also where I ran other functions that initiate the job scheduler, the handshake through the function setup() and job information. The setup() function is a handshake to initiate the client-server simulation. The client sends out messages to the server through the function send() and the server sends back a message through the function readBuff() to start running jobs.

When a job is sent, I split the message of the Job and then using a switch case I read the first index that the server sent about the job to determine what to do. If the job data sends back JOBN, it first stores the job id with a global integer globalJobId. It then goes into function command_get that calls the command GETS Capable where we receive information about the servers that are only capable at running the job. This is determined by the core, memory, and disk resources that the job requires the server to have. Command GETS will send out a long string of servers in the XML file where function command_get will store the strings in a list of all capable servers in an Array. After the information of capable servers is stored in an array, we sort the list in ascending order by the number of calls through Collection.sort(). The method Collection.sort() calls the function sortServer() a compare method. It compares Server a and Server b with a.core – b.core which returns a negative integer causing the list to be sorted in ascending order. Then returning back inside the switch case in function run() we call the function bestfit() which helps returns the server that I want to schedule.

Function bestfit() is the scheduling algorithm that helps reduce the total cost of the rental server. It loops through each server with if conditions to determine which server to return to schedule. Once a server is returned it calls the last function command_schd to schedule the job with the returned server from function

bestfit(). Function command_schd schedules the server by calling the command SCHD with the job id that stored as a global integer, the server type and sever id.

If the server sends us a message JCPL it will break out of the switch case and the client will send back a message REDY again to get the next job. After all jobs are scheduled and completed the server will send back a message QUIT where it will go into a function called quit(). The function quit() will close the socket, DataOutputStream and BufferReader then will exist the system and closing the distributed system.

The java.net.* package is allowed for the use of the Socket class. The Socket class is used to establish a connection between the server and the client. In this program, a connection is made by passing the server address and port number ("127.0.0.1" and "50000" respectively). Java.io* was used as it allowed for inputs and outputs of data through the datastream. Meaning that by using java.io* the client reads and writes to the server. This is achieved by using BufferReader. Importing Java.util.ArrayList allowed for the use of ArrayLists, which is used in the function ServerList. The ArrayLists were used to store the different servers Strings the server sent, as mentioned above. ArrayLists were the preference for data storage as it provides a dynamic form of storage. This means that we can easily adjust the ArrayList without specifying any initial size. Importing Java.util.collection allowed us to use the Collection.sort() method. It is used to sort the elements present in the specified list which would be the ServerList of collection in ascending order.

6. Evaluation

Sample Configuration

Config100-long-med/Config100-med-med/Config100-short-med

Server Type	Limit	BootupTime	HourlyRate	Core	Memory	Disk
Tiny	20	60	0.1	1	1000	4000
Small	20	60	0.2	2	4000	16000
Medium	20	60	0.4	4	16000	64000
Large	20	60	0.8	8	32000	256000
xLarge	20	60	1.6	16	64000	512000

Config20-long-med/Config20-med-med/Config20-short-med

Server Type	Limit	BootupTime	HourlyRate	Core	Memory	Disk
Tiny	4	60	0.1	1	1000	4000
Small	4	60	0.2	2	4000	16000
Medium	4	60	0.4	4	16000	64000
Large	4	60	0.8	8	32000	256000
xLarge	4	60	1.6	16	64000	512000

Total Rental Cost

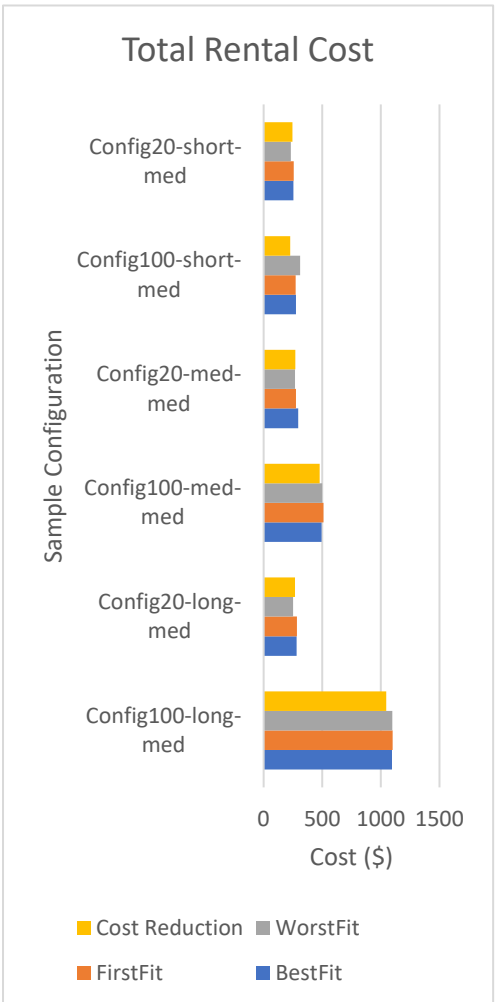
Sample Configuration	BestFit	FirstFit	WorstFit	Cost Reduction
Config100-long-med	1095.22	1099.21	1097.78	1045.55
Config20-long-med	281.35	283.34	250.3	267.65
Config100-med-med	493.64	510.13	498.65	478.91
Config20-med-med	295.13	276.40	267.84	268.96
Config100-short-med	275.90	272.29	310.88	226.75
Config20-short-med	254.85	257.62	231.69	246.57

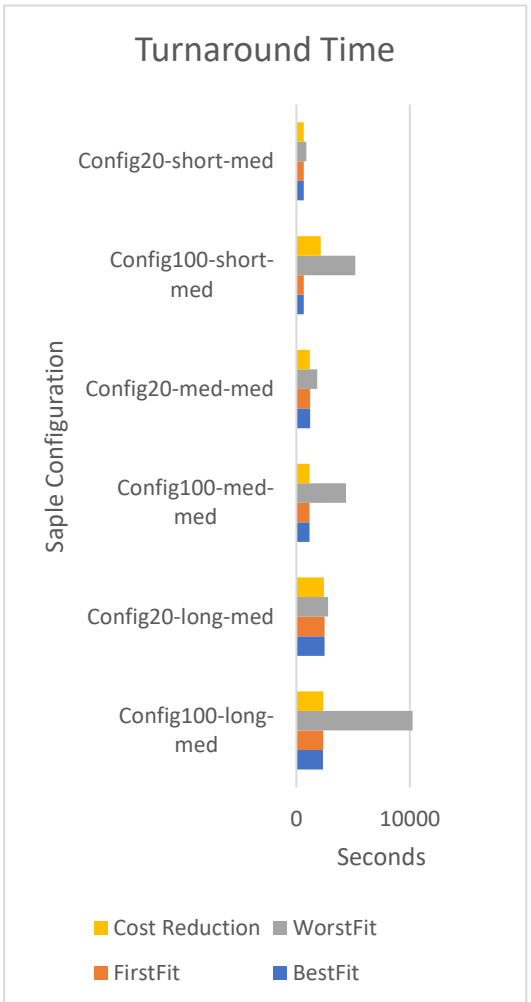
Turnaround Time

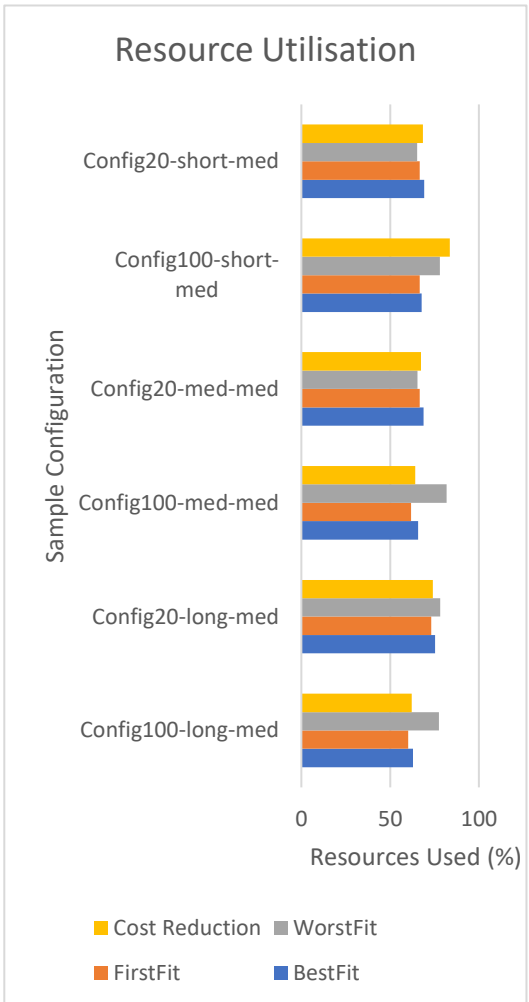
Sample Configuration	BestFit	FirstFit	WorstFit	Cost Reduction
Config100-long-med	2356	2362	10244	2358
Config20-long-med	2491	2485	2803	2417
Config100-med-med	1153	1154	4387	1154
Config20-med-med	1205	1205	1829	1176
Config100-short-med	645	644	5197	2134
Config20-short-med	649	649	878	641

Resource Utilisation

Sample Configuration	BestFit	FirstFit	WorstFit	Cost Reduction
Config100-long-med	62.86	60.25	77.45	62.1
Config20-long-med	75.40	73.11	78.18	74.01







Config100-med-med	65.69	61.75	81.74	64.22
Config20-med-med	68.91	66.64	65.38	67.46
Config100-short-med	67.78	66.72	78.12	83.54
Config20-short-med	69.26	66.58	65.21	68.42

Discussion

When looking at the Cost Reduction Algorithm, it is key to note that there are a few advantages and disadvantages to implementing this algorithm. The apparent of the advantages is that compared to the BestFit and FirstFit algorithm, Cost Reduction is cheaper due to the algorithm finding the first server it can used on 3 different conditions. Unlike FirstFit that only takes the first capable server and BestFit that allocates the smallest available free server that meets the requirement of the job. They both do not account for the size of the job compared to the size of the server. This means jobs are backlogged and frequently must wait for the sever to have the available core to complete the backlogged job. The cost Reduction algorithm accounts for backlog jobs ensuring that if there are jobs that need to be backlogged it spreads out the waiting jobs to reduce turnaround time. Although WorstFit looks to be a little cheaper than the Cost Reduction algorithm when it comes to servers that schedule less jobs. However, when it comes to scheduling a higher amount of jobs the cost reduction algorithm outperforms WorstFit. We can as well see that the cost reduction algorithm outperforms WorstFit in time. On some occasions the cost reduction algorithm has a higher turnaround time than BestFit and FirstFit. This is only in servers that deal with larger amount of jobs that need to be schedule but with jobs that require less job scheduling, the cost reduction algorithm performs better in turnaround time. The advantage there is even with slightly higher turnaround times than BestFit and FirstFit, Cost Reduction Algorithm cost less overall and as well maximises more resources than BestFit and FirstFit.

7. Conclusion

In summary the Cost Reduction Algorithm has outperformed the baseline algorithms FirstFit and BestFit regarding the total server cost but only performs better than WorstFit if there are larger amount of jobs than need scheduling. However, even with reducing cost the Cost Reduction Algorithm as well has maintained its turnaround time compared to FirstFit and BestFit while outperforming WorstFit. This shows that the Cost Reduction Algorithm is optimised on par with FirstFit and BestFit in doing so it has also reduce the total server cost.

8. References

[1] A.Ho, “aidanhomq/Comp3100”, Github, 2021. [Online]. Available: https://github.com/aidanhomq/Comp3100. [Accessed: 26/5/2021].

[2] GeeksforGeeks. 2021. *Collection.sort() in Java with Examples* - Geeksforgeeks. [Online]. Available at: https://www.geeksforgeeks.org/collections-sort-java-examples/. [Accessed 20/5/2021].

[3] S. Amjad, “First fit, Best fit, Worst Fit”, Slideshare.net, 2021. [Online]. Available: https://www.slideshare.net/ShahzebAmjad/first-fit-best-fit-worst-fit. [Accessed: 21/5/2021].

[4] Java T Point. 2021. *Java Comparator Interface* – Java T Point [Online]. Available at: https://www.javatpoint.com/Comparator-interface-in-collection-framework#:~:text=Java%20Comparator%20interface%20is%20used,and%20equals(Object%20element). [Accessed: 20/5/2021]