Stage 2: Cost-efficient Resource Allocation for Distributed Systems

Christopher Nance - 43254748

Introduction

Distributed systems are composed of multiple, independent and various systems that are, on a varying scale, geographically separate from each other, yet linked in a way that allows two or more systems in such a system to perform in concert with one another to achieve a common goal. Usually, in such a system, all the components must be subject to a universally agreed-upon set of rules and protocols that allow efficient communication with one another.

In this assignment, we will aim to develop and implement a job scheduler algorithm that fulfils the above-stated purpose, specifically; scheduling jobs in an *efficient* manner such that the entire simulated distributed system provided to us benefits from it overall relative to the scheduler submitted in stage 1. This algorithm aims to outperform the three baseline algorithms; First Fit (FF), Best Fit (BF), and Worst Fit (WF) in at least one key area out of the three defined in the assignment specifications.

Problem Definition

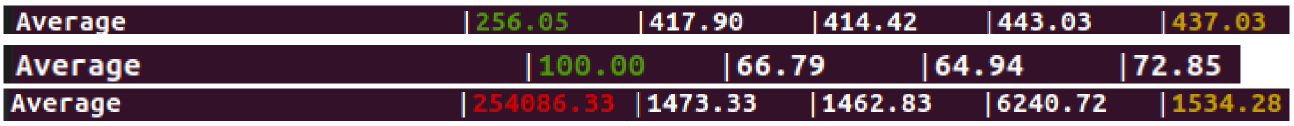
Our goal is to improve at least one of the following metrics relative to both our stage 1 job scheduler, as well as the three baseline algorithms:

- Minimisation of average turnaround time
- Maximisation of average resource utilisation
- Minimisation of total server rental cost

As these objectives are somewhat in conflict with one another, our scheduling algorithm aims to significantly reduce average turnaround time with respect to that of the allToLargest algorithm, and to obtain a comparable average turnaround time with respect to the three baseline algorithms.

This increase in average turnaround time comes with a cost to the second two objectives, thus, its overall usefulness is highly dependent on how critical time is in relation to the purpose of the distributed system, and specifically, real-time systems in which the end-user experience and the overall performance of the system is very much tied to how quickly operations can be carried out. This can include distributed systems involving Massively Multiplayer Online Games, virtual reality communities and air-traffic control environments. Systems that focus on distributed database management, for example, are not suited for this kind of algorithm, as the end-user experience is relatively immaterial in the context of moving/managing files and does not justify the increased cost associated with poorer resource utilisation.

The following are the turnaround time, the resource utilisation, and the total rental cost of the allToLargest algorithm after running the provided test file over the various config files (which will be the standard format for all tests going forward).



```
Average                     |256.05     |417.90     |414.42     |443.03     |437.03
Average                        |100.00     |66.79      |64.94      |72.85
Average                     |254086.33  |1473.33    |1462.83    |6240.72    |1534.28
```

Algorithm Description

Our solution makes use of multiple algorithms working in tandem with one another, and can overall be described as a form of a server-side non preemptive priority scheduling algorithm. To analogise the bin-packing problem, we will place each of our jobs inside a distinct “bin” that allows it to be completed in the most efficient manner possible.

We should first note a few details about ds-sim that our algorithm(s) take advantage of. Firstly, a server being hosted by ds-sim can be in one of a variety of states during the process of job scheduling, as can a job that has either been submitted, or is waiting to be submitted.

We will only refer to the states immediately relevant to our algorithm. Given the fact that we are allocating jobs to multiple servers that are capable of operating simultaneously, we can visualise our problem as a particular form of the bin-packing problem, therefore; we need to take into account both the jobs being allocated and the servers they are being allocated to; it will provide various optimisations both to the server side and the job side.

Firstly, we store all of the available servers in a global arrayList ***allServers***, which will then be referenced continuously throughout the operation of our algorithm. We ensure that allServers is always arranged in ascending order according to its cores, which increases that chances that we will obtain a “perfect fit,” or a close fit, meaning a job will more likely be scheduled on a server that has the exact number of cores the job requires to run. This ensures a greater degree of efficiency overall.

We then assign a priority to each available server according to their states, where by our client will sequentially assess each server grouped by their state. The order of priority is as follows:

1. Idle servers
2. Inactive servers
3. Active and booting servers with no waiting jobs
4. Active and booting servers with one or more waiting jobs

Via a loop, (2), (3) and (4) are stored inside their own respective arrayLists (we will call them our reference arrayLists), in the form of integers that point to their location within *allServers*. If a server doesn’t have a sufficient number of cores that can fit our job then it is filtered out. (1) is the exception, if we come across an idle server during our iteration we immediately schedule a job and move on to the next job, as an idle server has no running or waiting jobs and is immediately open to scheduling. Furthermore, it incurs a cost to keep an idle server running despite it not running jobs.

Once we have filled out our three reference arrayLists, we will loop through each of them in the order of (3), (2), and finally, (4).

To schedule our job in a way that leads to the least amount of waiting time being added, we will find, out of (3), the server which has the closest finish time based on the jobs it is currently running to the submit time of the job that we want to schedule. This ensures that we add the smallest amount of waiting time possible when we decide to schedule our current job.

For (2), since we have arranged allServers in ascending order via the core count, we can safely schedule our job to the first server point to inside this arrayList as it makes most efficient use of the potential core difference between job and server.

For (4), we find the server that has the smallest number of waiting jobs and schedule to that one. This helps ensure that the total waiting time exhibited by the servers at any given time is extended in a way that is evenly distributed amongst all the servers, helping to reduce our total waiting time overall.

In the event that none of the cases above apply to a particular job, it will just be scheduled to the first server that it fits in. This is not ideal in theory, however this should only account for a very small number of cases.

Scheduling Scenario

Here we will make use of the configuration file ds-config01-wk9. The server types and jobs to be scheduled:

type	limit	bootupTime	hourlyRate	coreCount	memory	disk
juju	2	60	0.2	2	4000	16000
joon	2	60	0.4	4	16000	64000
super-silk	1	80	0.8	16	64000	512000

J(1):	**60 1 2025 2 1500 1900**
Servers:	**juju 0 inactive -1 2 4000 16000 0 0**
	juju 1 inactive -1 2 4000 16000 0 0
	joon 0 booting 97 1 15300 60200 1 0
	joon 1 inactive -1 4 16000 64000 0 0
	super-silk 0 inactive -1 16 64000 512000 0 0
Scheduled:	**1 juju 1**

Here, we can see this particular job being scheduled to an inactive server, as our booting server has one waiting job and that puts it at a lower priority.

J(7):	**225 7 442 2 500 2100**
Servers:	**juju 0 active 156 0 2500 13900 0 1**
	juju 1 active 120 0 2500 13100 0 1
	joon 0 active 97 0 15200 58200 0 2
	joon 1 active 161 2 15100 61500 0 1
	super-silk 0 booting 236 9 57300 501800 2 0
Scheduled:	**7 joon 1**

Here, J(7) is scheduled to Joon 1. Juju 0 and 1 both have insufficient cores, and joon 1 has a fewer number of running jobs than joon 0 has. Thus, it takes priority. Here we utilised the *canFit()* and the *fewestWaitingJobs()* methods.

Implementation

This whole program is an extension of that which was submitted in stage 1, however much of the core was streamlined. For instance, readXML() was removed entirely as it was redundant, and we also reduced the total number of data structures we were using, and abstracted them out of the main SocketClient class into their own separate classes.

The most relevant software libraries that we used are as follows:

- *import java.io.DataInputStream;*

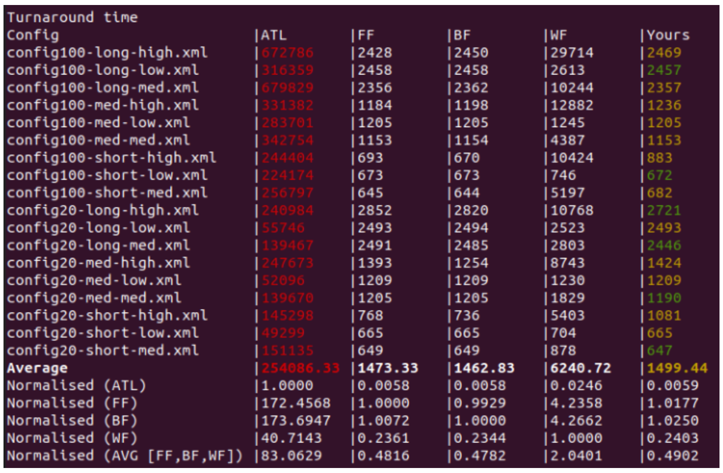
- *import java.io.DataOutputStream;*
- *import java.io.IOException;*
- *import java.net.Socket;*
- *import java.util.ArrayList;*

Components/Functions (most relevant ones to assignment 2):

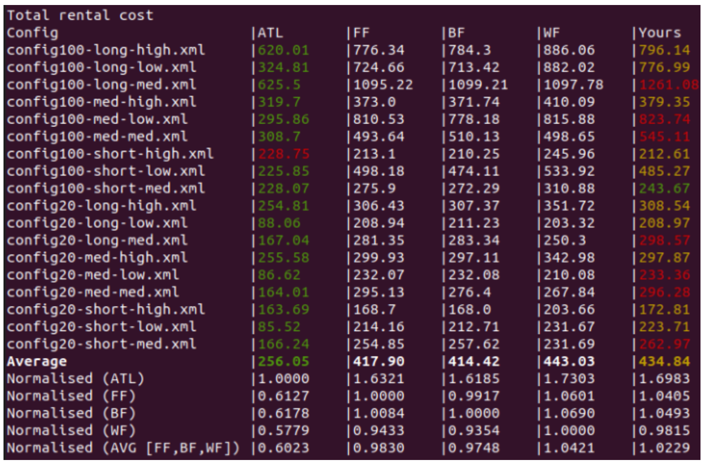
- *public void sortServers(ArrayList<SocketServer> servers)*
    - Sorts our arrayList containing all of the servers in ascending order by core count.
- *public boolean canFit(SocketServer server, SocketJob currentJob)*
    - Returns a boolean based on whether a job can “fit” within a server, by comparing the number of cores the job needs to run to the number of available cores the server has.
- *public String serverState(SocketServer server)*
    - Returns a string containing the state of a server passed in the parameter (“idle”, “booting”, “active”, or “inactive”).
- *public int priorityScheduler(SocketJob currentJob)*
    - Returns the index of the server in *allServers* that is the optimal fit based on our algorithm described above.
- *public int closestToFinish(ArrayList<Integer> freeServers, int currentJobSubmitTime)*
    - Returns the index of the server in *allServers* that contains the smallest difference between the submit time of our current job, represented by currentJobSubmitTime, and the latest finish time out of all the potential jobs running in each server.
- *public int fewestWaitingJobs(ArrayList<Integer> activeServers)*
    - Returns the index of the server in *allServers* that contains the smallest number of waiting jobs.

Evaluation

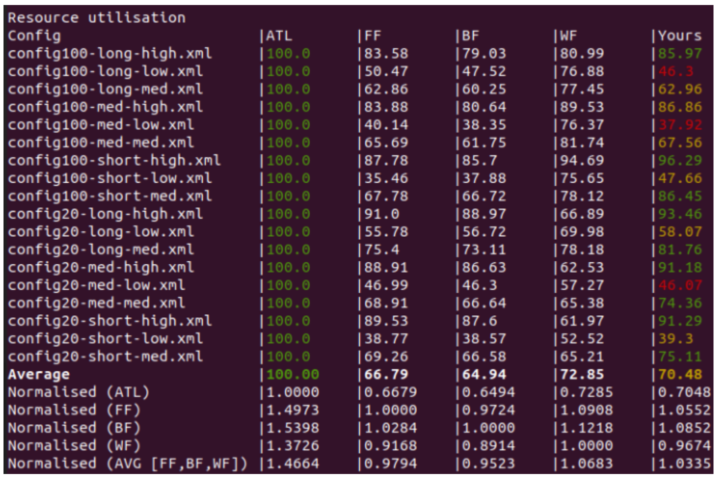
These are based on the provided test file (test_results):



```
Turnaround time
Config                      |ATL        |FF       |BF       |WF       |Yours
config100-long-high.xml     |672786     |2428     |2450     |29714    |2469
config100-long-low.xml      |316359     |2458     |2458     |2613     |2457
config100-long-med.xml      |679829     |2356     |2362     |10244    |2357
config100-med-high.xml      |331382     |1184     |1198     |12882    |1236
config100-med-low.xml       |283701     |1205     |1205     |1245     |1205
config100-med-med.xml       |342754     |1153     |1154     |4387     |1153
config100-short-high.xml    |244404     |693      |670      |10424    |883
config100-short-low.xml     |224174     |673      |673      |746      |672
config100-short-med.xml     |256797     |645      |644      |5197     |682
config20-long-high.xml      |240984     |2852     |2820     |10768    |2721
config20-long-low.xml       |55746      |2493     |2494     |2523     |2493
config20-long-med.xml       |139467     |2491     |2485     |2803     |2446
config20-med-high.xml       |247673     |1393     |1254     |8743     |1424
config20-med-low.xml        |52096      |1209     |1209     |1230     |1209
config20-med-med.xml        |139670     |1205     |1205     |1829     |1190
config20-short-high.xml     |145298     |768      |736      |5403     |1081
config20-short-low.xml      |49299      |665      |665      |704      |665
config20-short-med.xml      |151135     |649      |649      |878      |647
Average                     |254086.33  |1473.33  |1462.83  |6240.72  |1499.44
Normalised (ATL)            |1.0000     |0.0058   |0.0058   |0.0246   |0.0059
Normalised (FF)             |172.4568   |1.0000   |0.9929   |4.2358   |1.0177
Normalised (BF)             |173.6947   |1.0072   |1.0000   |4.2662   |1.0250
Normalised (WF)             |40.7143    |0.2361   |0.2344   |1.0000   |0.2403
Normalised (AVG [FF,BF,WF]) |83.0629    |0.4816   |0.4782   |2.0401   |0.4902
```



```
Total rental cost
Config                      |ATL        |FF       |BF       |WF       |Yours
config100-long-high.xml     |620.01     |776.34   |784.3    |886.06   |796.14
config100-long-low.xml      |324.81     |724.66   |713.42   |882.02   |776.99
config100-long-med.xml      |625.5      |1095.22  |1099.21  |1097.78  |1261.08
config100-med-high.xml      |319.7      |373.0    |371.74   |410.09   |379.35
config100-med-low.xml       |295.86     |810.53   |778.18   |815.88   |823.74
config100-med-med.xml       |308.7      |493.64   |510.13   |498.65   |545.11
config100-short-high.xml    |228.75     |213.1    |210.25   |245.96   |212.61
config100-short-low.xml     |225.85     |498.18   |474.11   |533.92   |485.27
config100-short-med.xml     |228.07     |275.9    |272.29   |310.88   |243.67
config20-long-high.xml      |254.81     |306.43   |307.37   |351.72   |308.54
config20-long-low.xml       |88.06      |208.94   |211.23   |203.32   |208.97
config20-long-med.xml       |167.04     |281.35   |283.34   |250.3    |298.57
config20-med-high.xml       |255.58     |299.93   |297.11   |342.98   |297.87
config20-med-low.xml        |86.62      |232.07   |232.08   |210.08   |233.36
config20-med-med.xml        |164.01     |295.13   |276.4    |267.84   |296.28
config20-short-high.xml     |163.69     |168.7    |168.0    |203.66   |172.81
config20-short-low.xml      |85.52      |214.16   |212.71   |231.67   |223.71
config20-short-med.xml      |166.24     |254.85   |257.62   |231.69   |262.97
Average                     |256.05     |417.90   |414.42   |443.03   |434.84
Normalised (ATL)            |1.0000     |1.6321   |1.6185   |1.7303   |1.6983
Normalised (FF)             |0.6127     |1.0000   |0.9917   |1.0601   |1.0405
Normalised (BF)             |0.6178     |1.0084   |1.0000   |1.0690   |1.0493
Normalised (WF)             |0.5779     |0.9433   |0.9354   |1.0000   |0.9815
Normalised (AVG [FF,BF,WF]) |0.6023     |0.9830   |0.9748   |1.0421   |1.0229
```



```
Resource utilisation
Config                      |ATL       |FF        |BF        |WF        |Yours
config100-long-high.xml     |100.0     |83.58     |79.03     |80.99     |85.97
config100-long-low.xml      |100.0     |50.47     |47.52     |76.88     |46.3
config100-long-med.xml      |100.0     |62.86     |60.25     |77.45     |62.96
config100-med-high.xml      |100.0     |83.88     |80.64     |89.53     |86.86
config100-med-low.xml       |100.0     |40.14     |38.35     |76.37     |37.92
config100-med-med.xml       |100.0     |65.69     |61.75     |81.74     |67.56
config100-short-high.xml    |100.0     |87.78     |85.7      |94.69     |96.29
config100-short-low.xml     |100.0     |35.46     |37.88     |75.65     |47.66
config100-short-med.xml     |100.0     |67.78     |66.72     |78.12     |86.45
config20-long-high.xml      |100.0     |91.0      |88.97     |66.89     |93.46
config20-long-low.xml       |100.0     |55.78     |56.72     |69.98     |58.07
config20-long-med.xml       |100.0     |75.4      |73.11     |78.18     |81.76
config20-med-high.xml       |100.0     |88.91     |86.63     |62.53     |91.18
config20-med-low.xml        |100.0     |46.99     |46.3      |57.27     |46.07
config20-med-med.xml        |100.0     |68.91     |66.64     |65.38     |74.36
config20-short-high.xml     |100.0     |89.53     |87.6      |61.97     |91.29
config20-short-low.xml      |100.0     |38.77     |38.57     |52.52     |39.3
config20-short-med.xml      |100.0     |69.26     |66.58     |65.21     |75.11
Average                     |100.00    |66.79     |64.94     |72.85     |70.48
Normalised (ATL)            |1.0000    |0.6679    |0.6494    |0.7285    |0.7048
Normalised (FF)             |1.4973    |1.0000    |0.9724    |1.0908    |1.0552
Normalised (BF)             |1.5398    |1.0284    |1.0000    |1.1218    |1.0852
Normalised (WF)             |1.3726    |0.9168    |0.8914    |1.0000    |0.9674
Normalised (AVG [FF,BF,WF]) |1.4664    |0.9794    |0.9523    |1.0683    |1.0335
```

As we can see, we have a vastly improved turnaround time overall, but at the cost of poorer resource utilisation and a higher total rental cost. As mentioned in the introduction section, this makes this algorithm useful if implemented for real-time systems. Economically, the cost is more contingent on a smoother end-user experience than it is on raw rental costs.

Conclusion

Overall, this algorithm is very useful when implemented within distributed systems that are heavily focused on facilitating communication within a real-time system, where the turnaround time is of primary importance. Based on that metric, this algorithm was quite successful.

References/Github Repository

- https://github.com/Abstractvice/COMP3100_Project