Stage 3 Design Document

Title

Automated Resource Allocation for a Distributed System.

Introduction

During the previous stages, we have learnt how a distributed system is set up and allocated jobs among the different nodes associated. By implementing a number of algorithms as well as the general protocol for the described system, we have been able to develop a good understanding of how we would expect these types of systems to operate.

For this stage, we have been tasked with creating an algorithm that optimises one, or more, of the described metrics used in the system. Namely, these are waiting time, cost, and average utilisation. To improve on these measures, we are required to study how the protocol works and how our previous algorithms distributed and scheduled the jobs sent to it. Furthermore, analysing the data that resulted from each of the algorithms, we can determine what worked well and what did not for each algorithm. Understanding each of these aspects of the system should allow us to implement an algorithm that can result in an improved experience for users of the *ds-sim* system.

Problem Definition

For my algorithm, I decided to focus on optimising and/or maintaining all three metrics at an acceptable level in order to complete with a 'perfect balance' type of optimisation. I concluded that neither of the three metrics are important enough that the others should suffer as a result. I wanted my algorithm to focus on improving and/or maintaining all three metrics at a level that would be expected when compared to other algorithms we have completed. It should be that if one metric were to become slightly worse it was due to the huge advantage of the other metrics. I did not want to see large degradation for the improvement of another aspect.

However, I noted that the most important measure would be the wait time as we would expect customers to want to ensure their users are not waiting large amounts of time on a job to complete so that some other job can occur. This might raise costs but is a necessary aspect of the functionality for such a system. I wanted to ensure that my algorithm would not increase these costs much by utilising active and idle servers, rather than starting new ones. The metric least important to my eventual outcome should be the utilisation as I believed customers, or users, of such an algorithm would be less focused on using the servers completely if it meant it led to reduced waiting times for users, or reduced costs for themselves.

In totality, my objective function is to decrease the wait times for customers while not sacrificing the other measures to the point where there are significant decreases in performance compared to other algorithms we have introduced previously.

Algorithm Description

To be able to optimise all three measures and aim to improve waiting times, I decided I would first need to understand what aspects of the previous algorithms worked as well as the aspects that didn't. As expected, this was mostly the *best-fit* algorithm that resulted in the best overall result. However, there were instances where the other algorithms resulted in a better result. I was able to conclude that this was mostly due to another aspect being sacrificed and so wasn't worth using the algorithm to study.

I deduced that the most important parts of this algorithm were checking for the current state of the servers, and the fitness value. Understanding this, I was able to start my algorithm.

To be able to get the best fit and get a better overall experience for customers, I needed to split the algorithm into parts. The four stages that I concluded with are described as:

1. First, I wanted to find what would be the best fit for the job regardless of whether the server is busy with other jobs, available, or in any other state. This would help us determine which server could be used for it and reduce the overall costs of starting a big server than needed.
2. Once we had this value, iterate through the server information sent from the server client and try to determine if there is an available server with this value. If there is, schedule the job to it.
3. If a job has not got a job at this point, see if any idle servers are available to take the job. By not starting a new server, we should be able to reduce costs by using a currently idle server to just do the job for us instead of running up costs while doing nothing.
4. Finally, if all else fails, we need to find the best server out all servers -- regardless of the state. However, we don't want to run up a large waiting time so I implemented a map that records the estimated run time for a server each time we submitted a job in any of the previous parts. Using this, we can iterate through each one, find the server with the closest estimated finishing time and place the job on that server.

To describe how the expected flow of my algorithm operates, I have created an example below where jobs are in the form `submissionTime id estimatedRuntime cores memory disk` and servers are in the form `type id state availableTime cores memory disk`.

```
SERVERS {
      small  0 0 143 1 4000  16000
      small  1 0 143 1 4000  16000
      medium 0 0 143 2 16000 64000
      medium 1 0 143 2 16000 64000
}

JOBS {
```

```
      83    0 37232 2 500  600
      290   1 40000 1 400  1200
      336   2 129   2 2100 2800
      443   3 40000 2 500  1100
      500   4 40000 1 400  1200
      37233 5 899   1 400  1200
}
```

Figure 1: Server and job descriptions for example.

As described, there are two servers of type `small` and `medium`. The six jobs will execute as following:

- **JOB #1**: Schedule to server ***medium #0***. Requires two cores so we can determine that a medium server is the best fit for it.
- **JOB #2**: Schedule to server ***small #0***. Requires one core and all requirements can run on the first small server in the system
- **JOB #3**: Requirements for a medium server. As one medium server has already been taken, we schedule this on the second one: ***medium #1***.
- **JOB #4**: This job also has requirements for a medium server. Since both medium servers are full and we can't fit it on any larger servers that are idle (there aren't any), we need to determine which of the two active servers should finish quicker. The algorithm would grab the data from when they were scheduled and determine that the ***medium #1*** should finish first so we schedule it for that server.
- **JOB #5**: This job requires a small server. Given that all servers are busy and we have an inactive small server, we can schedule it for this: ***small #1***.
- **JOB #6**: The job requires a small server. However, all our small servers now have an estimated run time of about 40,000 units each. Due to this, our algorithm would look for an idle server of a larger size. It finds that there is a medium server that is now idle and waiting for a job. We can allocate the job to this one to execute to save us waiting the extra time for a small server. We place it on ***medium #0***.

As described in the example, it becomes evident how my algorithm would determine to schedule a given job. It would be ideal to avoid booting up unnecessary servers so we aim to reduce this function by utilising what we have available.

The described example is the flow that I follow in my implemented algorithm. The described flow not only reduces waiting time for jobs but also help reduce the costs by further reducing the number of machines we boot up and use. The algorithm is designed to only start machines when required to ensure that users aren't waiting too long.

Implementation Details

The implementation will be described below using pseudo-code to explain the implementation. Relevant data types will also be highlighted to ensure a complete understanding of how the algorithm operates the type of data it relies on.

```
Server arr: xmlServers     # contains xml data
Server ArrayList: servers  # from resc command
estRunTime = Map(string, integer)

### STAGE ONE ###

integer bestFit = MAX_INT
for Server in xmlServerData
     integer currentFit = server cores - job cores
     if server has resources to run job AND currentFit < bestFit
          bestFit = currentFit

### STAGE TWO ###

quickSort(servers)    # this is done based on CPU core value

int minAvailabletime = MAX_INT
Server bestFitServer = null
for Server in servers
     int currentFit = server cores - job cores
     if (Server can run job AND bestFit >= currentFit AND server
     availableTime < minAvailableTime)
          minAvailableTime = server availableTime
          bestFitServer = Server

if bestFitServer not null
     add (bestFitServer type and id) to estRunTime with value
     (estRuntime + job submitTime)
     return bestFitServer

### STAGE THREE ###

integer bestFitIdle = MAX_INT
Server bestFitIdleServ = null
for Server in servers
```

```
      if Server state = 2 AND Server can run job
            integer possibleFit = job cores - Server cores
            if possiblefit < bestFitIdle
                  bestFitIdle = possibleFit
                  bestFitIdleServ = Server

if bestFitIdleServ not null
      update (bestFitServer type and id) to estRunTime with value
      (estRuntime + job submitTime)
      return bestFitIdleServer
```

STAGE FOUR

```
integer bestEstimate = MAX_INT
Server bestGuess = null
for Server in xmlServers
      i = 0
      while i < server amount available for this type
            integer currentFit = server cores - job cores
            if server can run job and bestFit > currentFit
                  currentEstimate = get value for estRunTime(server
            type
                  and id)
                  bestGuess = Server
                  bestGuess id = i
            i += 1

update (bestGuess type and id) to estRunTime with value (estRuntime +
job submitTime)
return bestGuess
```

QUICK SORT FUNCTION

```
quickSort(arr[], low, high)
    if (low < high)
        pi = partition(arr, low, high)

        quickSort(arr, low, pi - 1)
        quickSort(arr, pi + 1, high)

partition (arr[], low, high)
    i = (low - 1)
```

```
for (j = low; j <= high- 1; j++)
    if (arr[j] <= pivot)
        i++
        swap arr[i] and arr[j]
swap arr[i + 1] and arr[high])
return (i + 1)
```

The described implementation above can be seen written in Java in the source code for my project. The pseudo-code gives a rough outline of how the Java implementation without being too specific for the language. By reading through it, however, one should be able to get a good idea of the process that the algorithm runs through.

It should be noted that my implementation of QuickSort was based on the CPU count for each server in the passed array. The implementation requires an `ArrayList` to operate. I left the pseudo-code version simple in order to improve the readability and not promote the over-complexity of it.

Evaluation

In order to conclude whether my implementation was successful for my desired measures, we need to compare it against the other attributes. Using scores across four different scheduling algorithms, we can determine whether it performs better in the metrics that I set out to improve.

Each of the three metrics will be described and evaluated below in sections in order to highlight the improvements in each core section. Furthermore, I will describe why some results are how they are in some of the configuration files when compared to the other algorithms and if there is a degrade due to the increase in another measure.

Average Waiting Time

Config	First Fit	Best Fit	Worst Fit	My Fit
ds-config-s3-1	5954	5954	8144	520
ds-config-s3-2	320961	366580	598053	148302
ds-config-s3-3	1207216	1252006	1482660	782640
ds-config-s3-4	0	0	0	0
ds-config-s3-5	16	15	7	15
ds-config-s3-6	0	0	0	0
ds-config-s3-7	1	1	0	1165

Figure 2: Table explaining cost across four algorithms.

This was the measure that I set to improve the most, so I expected it to see the largest increase in performance. As seen in the table, almost every instance had an improved wait time – except the last

configuration file. It can be seen that the first three configuration files had the largest improvements with reduced wait times of more than 10x in some cases.

Putting this information into a graph illustrates how big the improvements in wait time are in most cases. This graph can be seen in *figure 3* below.

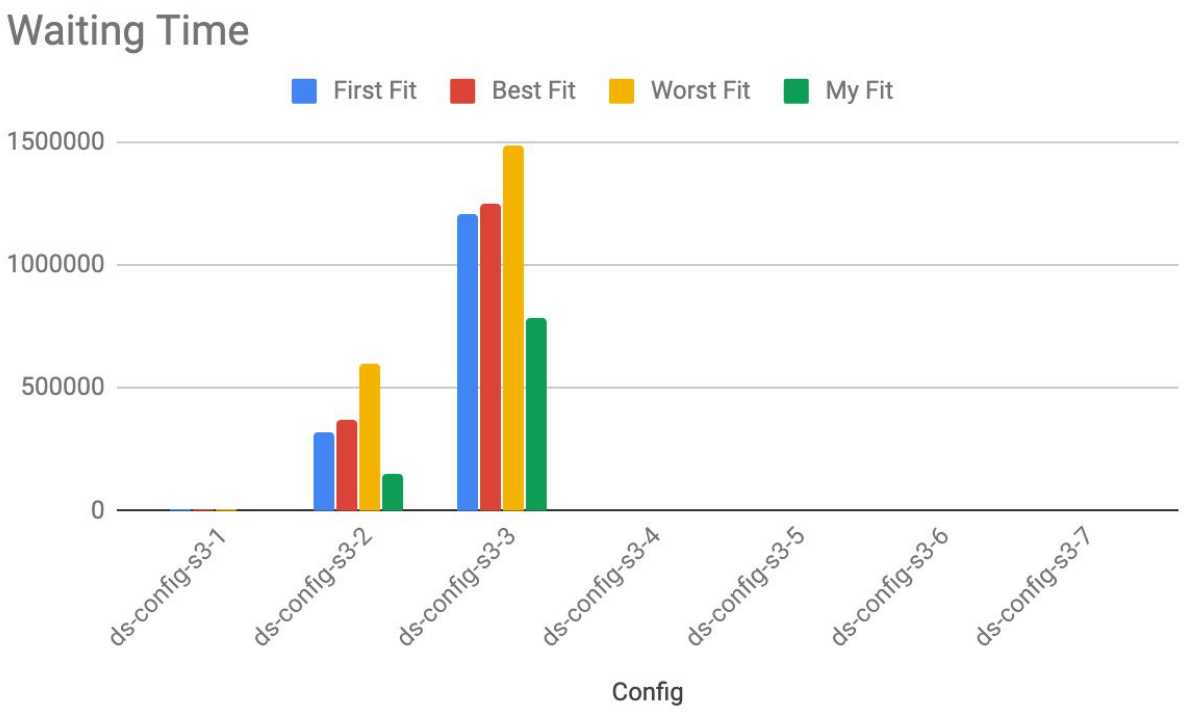

Figure 3: Waiting time graph.

Although it is hard to see all cases but configurations 2 and 3, it is important to note how big the improvements are for these files. Not seeing any degradation in other measures, this algorithm is mostly successful in improving without sacrificing the other aspects. Furthermore *figure 4* highlights differences in *ds-config 1* waiting time.

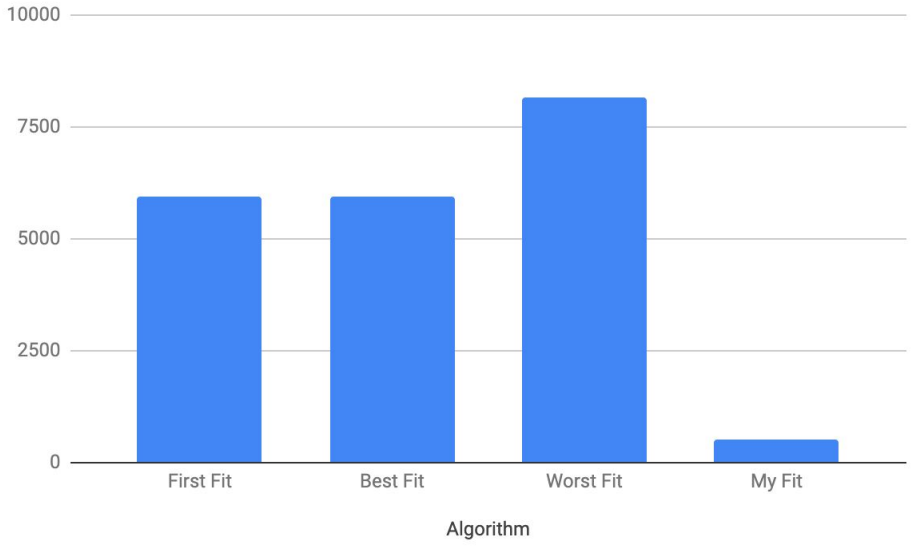

Figure 4: Waiting time graph for *config 1*.

I was unable to fix the bug in configuration file 7 that was causing the increased wait time. I was able to deduce that it was an error to do with the scheduling on medium servers. However, given that the algorithm performs better on the majority of cases, I still believe it is an improvement on previous algorithms.

Cost

Config	First Fit	Best Fit	Worst Fit	My Fit
ds-config-s3-1	$19.85	$19.75	$15.97	$15.86
ds-config-s3-2	$917.62	$898.98	$901.50	$908.65
ds-config-s3-3	$5,060.18	$4,945.91	$5,546.16	$5,077.68
ds-config-s3-4	$86.46	$86.46	$172.92	$86.46
ds-config-s3-5	$56.68	$55.37	$85.04	$55.37
ds-config-s3-6	$74.54	$67.90	$135.81	$67.76
ds-config-s3-7	$614.08	$583.09	$889.60	$577.37

Figure 5: Table explaining cost across four algorithms.

The cost of the servers was able to match the best-fit algorithm fairly consistently. This was important as a didn't want to drive costs up just to decrease the waiting time of users.

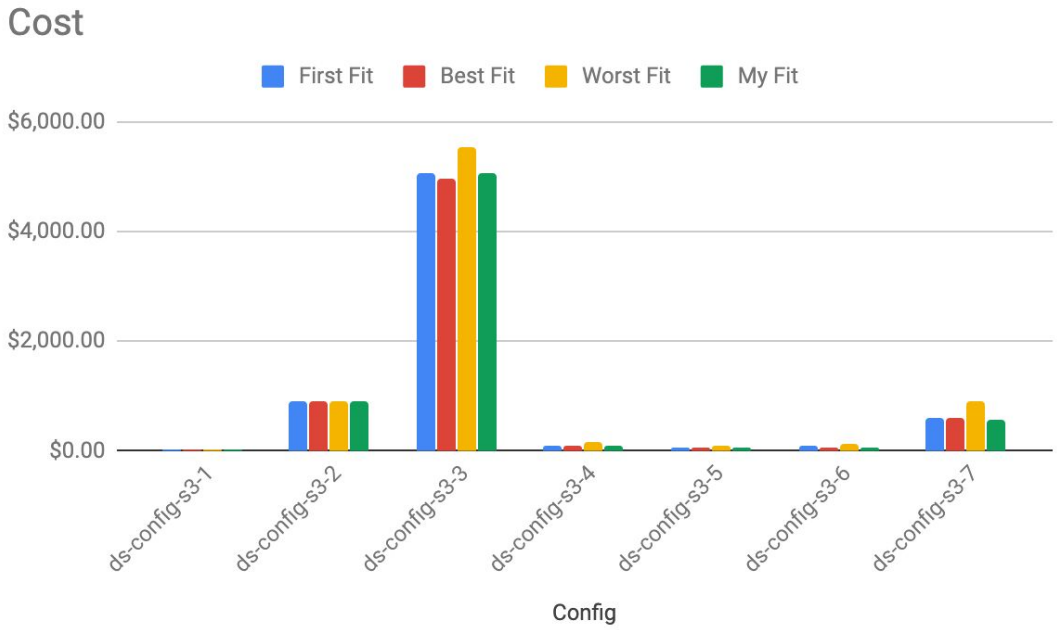

Figure 6: Cost graph.

As seen in figure 5, the cost is mostly cheaper than all the other algorithms, or worst case a little more expensive than our 'best' performing cost algorithm. I was able to maintain a consistent cost that matched that of the rival algorithms to ensure that my main metric was improved.

Utilisation

Config	First Fit	Best Fit	Worst Fit	My Fit
ds-config-s3-1	92.64%	98.04%	93.98%	92.64%
ds-config-s3-2	75.40%	70.90%	59.91%	86.04%
ds-config-s3-3	57.93%	49.89%	57.59%	56.75%
ds-config-s3-4	43.69%	43.69%	43.69%	43.69%
ds-config-s3-5	76.33%	80.68%	91.52%	74.92%
ds-config-s3-6	56.91%	46.33%	95.45%	47.35%
ds-config-s3-7	83.10%	78.10%	82.83%	72.88%

Figure 7: Table explaining utilisation % across four algorithms.

As discussed at the start of the document, utilisation was the least important metric that my algorithm would focus on. As a result, almost all of the configuration files saw a decrease in the utilisation of each server. This is due to the algorithm focusing on placing jobs into their best fit so larger/smaller servers might not be utilised for periods of time due to not being required to be used for a job that suits it. There was little I was able to do to fix this issue without largely impacting the optimisation of the other metrics.

This table can be better visualised in a graph form in *figure 8* below.

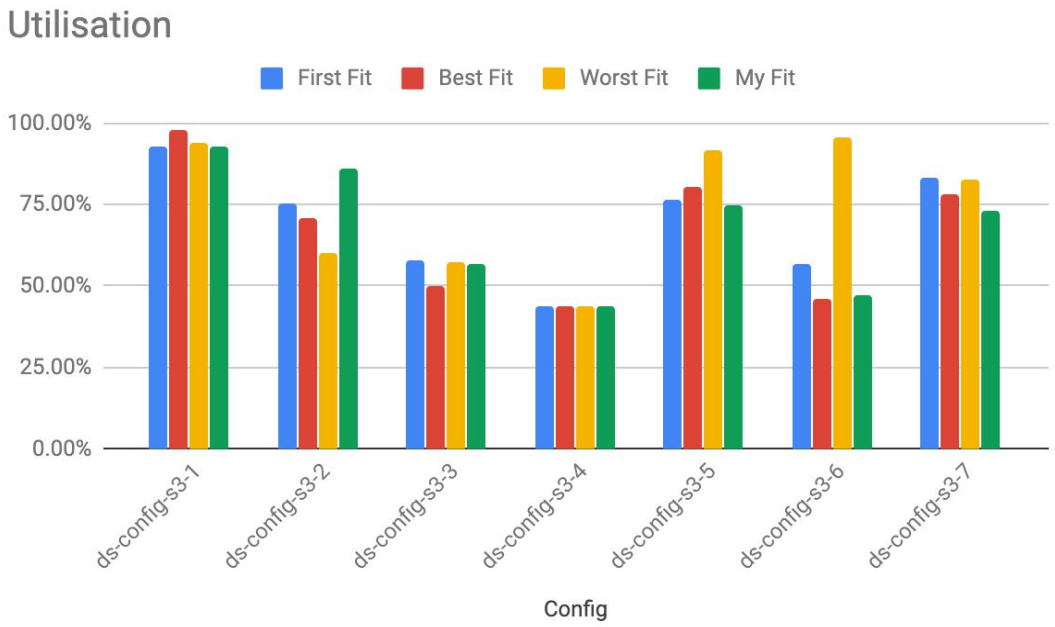

Figure 8: Utilisation graph.

This better depicts that while the utilisation doesn't increase significantly, it mostly stays in line with what you would expect from the other algorithms. Therefore, I was able to conclude that I haven't sacrificed this measure a lot for the other measures to improve.

Conclusion

The algorithm implemented is able to improve, or maintain, acceptable measures when compared to other algorithms we have implemented. Overall, most of the configuration files tested throughout the process have seen a drastic improvement in the wait time while not sacrificing cost and/or seeing a decrease in utilisation of servers.

The stages that I explained in the algorithm and implementation descriptions performed as expected in almost all cases, finding the correct type of server to schedule the job on. As expected, this resulted in improved or maintained measures in the files to ensure that user experience would have been improved in the case of this being used in a real-world distributed system model.

In order to further improve the algorithm, we would need to see better statistics in regards to completion times. Having estimated run times that update as the job goes would result in jobs that can better predict which server to go on. Furthermore, being able to reschedule jobs would allow us to move a job should another server become unexpectedly available.