DS-SIM EFFICIENT RESOURCE ALLOCATOR

Tanay Bhaskar Gandhi(45533296)

Introduction

In real life scheduling, that takes place in data centres throughout the world, there are lot of factors that are involved while deciding the most appropriate server to select for running the jobs. We are trying to replicate the situation using ds-sim program, in stage 2, where our aim is to implement the scheduling decision while keeping in mind the three main factors involved, turnaround time, rental cost and resource utilization. The aim is to design a new algorithm that optimises one or more of the above-mentioned factors. There are three main baseline algorithms that have been given, and we need to try to design an algorithm whose overall performance is better than the three baseline algorithms(First-Fit, Best-Fit and Worst-Fit).

Problem Definition

The problem we have been given is to create an algorithm whose performance is superior to three baseline algorithms and is significantly different from AllToTheLargest algorithm technique that we used in stage1. We have also been advised to use GETS Capable command instead of reading server information from the XML file that ds-server sends us when we complete handshake and authentication. The function that I have decided to implement is an optimised Best-Fit algorithm. The objective is to create an algorithm that takes a balanced approach for two-performance metrics – Resource Utilisation and Turnaround time (at the expense of rental cost) and provide a better scheduling than all three baseline scheduling algorithms. In the algorithm I have implemented, I have given priority to Resource utilisation and turnaround time as service providers all over the world value performance of their servers the most. We can see that by the model followed by major cloud service providers such as AWS and Azure, where they sign Service level Agreements for guaranteed uptime and certain performance assurance and are willing to pay their clients money if any conditions in the SLA is not met.

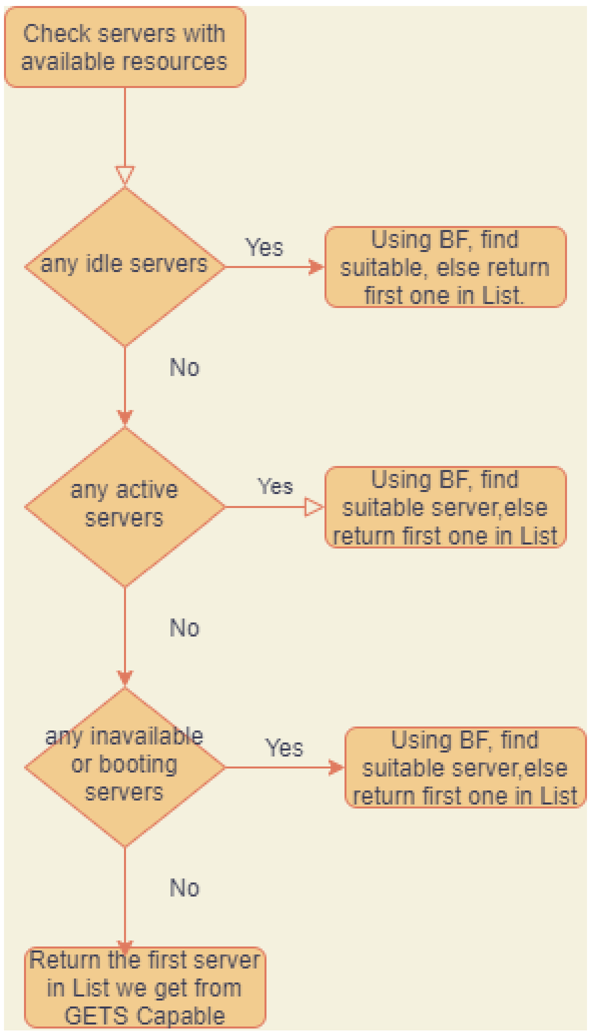
Algorithm Description

The algorithm that we have decide to use is an optimised Best-Fit baseline algorithm. In BF, we chose the smallest possible server that has the resources available, with no waiting jobs and least fitness value. The area that I found in BF to improve is to utilise the remaining resources that are available to us. A simple example is to schedule a new job that requires 2 cores to an active server with 7 cores, who is running a job on 4 cores, but has 3 cores that are not used rather than booting a small new server with 2 cores. This implementation has removed booting time from equation and hence the waiting time is reduced. Since we are utilising more resources, the resource utilisation of the server goes up, which also helps us to reduce rental costs. But we see an increase in rental cost in the algorithm as in the case when we do not find any active idle server with required resources, we use best fit to find the smallest server and boot it up, but this also has a downside. Since we are starting small servers, we will use a greater number of servers, and will face an increase in the rental cost for the use. Priority is given to idle servers, then active servers and then rest of servers with available resources.

Simple Scenario:

- **Server:** SENDS JOBN 37 0 106 4 300 1200 ← First Job requiring 4 cores
- **Client:** GETS Capable
- **Server:** DATA 3 124
- **Client:** OK
- **Sever:** juju 0 inactive -1 2 16000 64000 0 0
  joon 0 inactive -1 7 24000 108000 0 0 ← **Server selected using optimised BF**
  super-silk 0 inactive -1 16 48000 1230000 0 0
- **Client:** OK
- **Server:** .
- **Client:** SCHD 0 joon 0
- **Server:** OK
- **Client:** REDY
- **Server:** JOBN 12 1 345 2 1200 36000 ←*second job requires 2 cores*
- **Client:** GETS Capable 2 1200 36000
- **Server:** Data 3 111
- **Client:** OK *// Assuming First job is still running*
- **Server: :** juju 0 inactive -1 **2** 16000 64000 0 0 ← *Selection according to BF, Value =0*
  joon 0 active 1 **3** 24000 108000 0 0 ← **Selection made based on state & BF-value**
  super-silk 0 inactive -1 **16** 48000 1230000 0 0
- **Client:** OK
- **Server:** .
- **Client:** SCHD 1 joon 0
- **Server:** OK
- **Client:** REDY
- **Server:** JOBN 36 2 111 **2** 1000 38000
- **Client :** GETS Capable
- **Server:** Data 3 111
- **Client:** OK *//Assuming first 2 jobs are completed*
- **Server:** juju 0 inactive -1 2 16000 64000 0 0
  joon idle 1 7 24000 108000 0 0 ← **Server selected due to state == idle**
  super-silk 0 inactive -1 16 48000 1230000 0 0
- **Client:** OK
- **Server: .**
- **Client:** SCHD 2 joon 0

Algorithm Logic Flowchart


Check servers with available resources
any idle servers
Yes
Using BF, find suitable, else return first one in List.
No
any active servers
Yes
Using BF, find suitable server,else return first one in List
No
any inavailable or booting servers
Yes
Using BF, find suitable server,else return first one in List
No
Return the first server in List we get from GETS Capable

Implementation

In optimised Best-Fit algorithm, we use Array List as data structure. The main reasons of using Array List over other data structures such as arrays, linked list or Hash Map are due to three main reasons. Their size can be dynamically increased. We can create Array list with of a type of object, such as Servers in our case. This helps us to implement the OOP techniques and utilise the methods and variables in different class. We can also easily iterate over Array List and there are lot of helpful methods provided in the java.util package such as size(), or clear(), which we use frequently in our program. For implementation, we get the data from server using GETS Capable command, read one line at a time, and iterate over the list of servers sent to us. We than parse the fields in the line, using .split(" "), and create a server object. We than add this server object to a serverList and use it to find the most optimal server. For JOBN, we use the same techniques to split the fields and create a Job object and use it to find the most efficient server. Then in our scheduling algorithm, we check if there are adequate resources as data we get from GETS Capable may also have servers with 0 cores, as they are capable to run the job, but just don't have available resources. So, we create three Array List, idleList, activeList and tempList. We iterate over the data from GETS, find server with resources, and based on their states add them to one of the three Lists. We calculate fitness value as server.core – job.core and the server with least fitness value is chosen. Then we first check the idleList, find a best Fit server in the List or else just return the first server in the list. Then we give priority to active servers. If no idle server s found, we iterate over activeList and return the Best Fit server. And lastly, we iterate over tempList to find the best Fit server and schedule job on them. If we cannot find a suitable server in any of the three lists mentioned above, we schedule job to first server we get from GETS Capable.

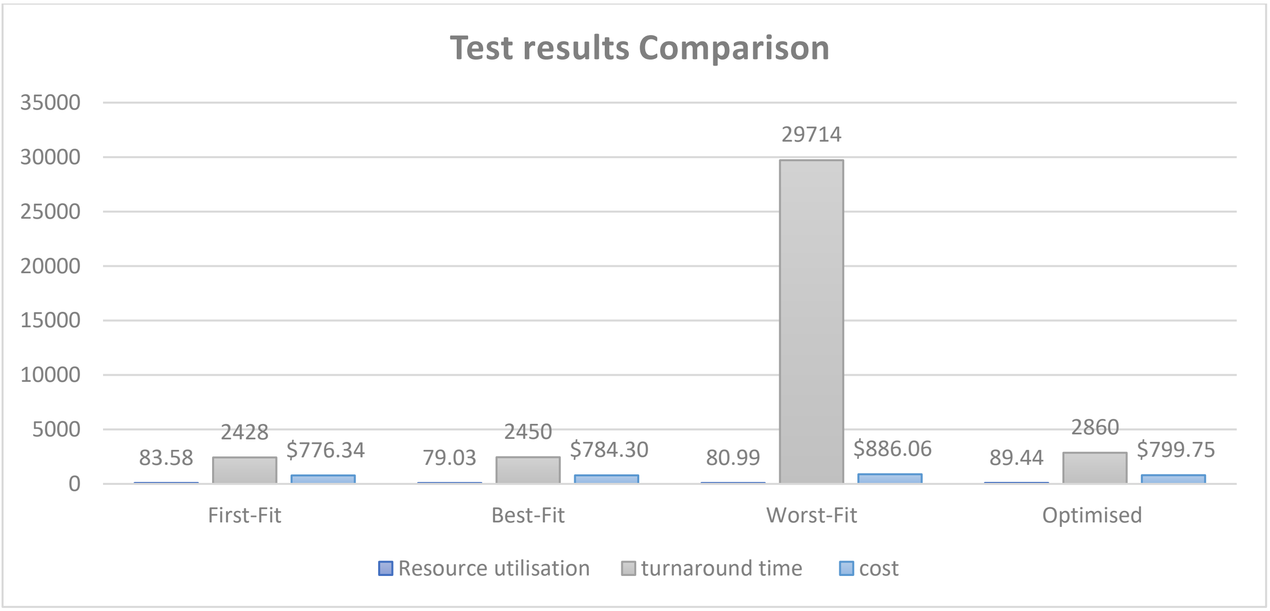
Another optimisation we made was to only select the server while iterating over activeList or tempList if they have less or no waiting jobs w.r.t other servers.

Evaluation

We will evaluate our code, with comparison to three baseline algorithms. For evaluation we use the -a mode in the ds-client and run it on the config file called config100-long-high.xml which can be found in ds-server/sample-configs/other.

Test Setup: open two terminal windows in Ubuntu. After cloning ds-sim, in one terminal run *./ds-server -c config100-long-high.xml -v all -n* and run *./ds-client -a ff*. Note down the turnaround time, Resource utilisation percentage and rental cost.Run the second test with same ds-server command and on client side, we run *./ds-client -a bf* and note down the results.We run the third test with same ds-server command and on client side, we run *./ds-client -a wf* and note down the results.

And lastly, we run the same ds-server command, but compile the supplied Client code and run java Client. (Please note that Server.java and Job.java are in the same directory). We note down the results of difference performance variables. We can clearly see the results in the chart below:



Minimum Turnaround Time: FF > BF > Optimised > WF

Maximum Resource Utilisation: Optimised > FF > WF > BF

Minimum Cost: FF > BF > Optimised > WF

PROS:-

- Better utilisation of resources
- Turnaround Time is close to that of BF and FF.
- Significantly better than WF.

CONS:-

- Since we are using more servers than in AllToTheLargest, the rental cost is also high.
- Additional time may be consumed in choosing which server to schedule the job to.
- We do not kill any idle servers, which may lead to higher cost.
- When we try to utilise idle or active servers by giving them priority, there may not be enough resources for next job which required exact resources of the idle server.

Conclusion

We conclude that our algorithm is highly optimised for the maximum resource utilisation, which may eventually lead to lower costs and less waiting time, but in our case, we take a hit in turnaround time and rental costs. On average, we can conclude that we cannot optimise on all three-performance metrics and will always have to choose the one that matters the most.

References:

GitHub Link: https://github.com/tgandhi-20/COMP3100-JobSchedular-