# COMP3100 Stage 2 Report

## Cloud Job Scheduler for Distributed Systems

Student Name: Quoc Trung Nguyen Student ID: 45470383

## 1. Introduction

The purpose of this project is to develop a job scheduler written in Java that will act as a client to a server, or more specifically, a cloud data center simulated by ds-server in the ds-sim protocol. The client will communicate with the server using the TCP protocol in order to receive jobs and schedule them to the appropriate servers for execution based on some particular performance metrics and constraints.

For stage 2 of the project in particular, a new scheduling algorithm for the Java job scheduler needs to be designed and implemented so that the resulting job scheduling must satisfy one or more of the following (possibly clashing) goals:

- Minimizing the average turnaround time
- Maximizing the average resource utilization
- Minimizing the total server rental cost

## 2. Problem Definition

The problem with job scheduling based on the 3 performance objectives mentioned in the introduction (the minimization of the average turnaround time, the maximization of the average resource utilization and the minimization of the total server rental cost), is that they are potentially conflicting, i.e., the satisfaction of one goal may be the sacrifice of another. Therefore, only a maximum of 2 out of the 3 goals may be chosen to facilitate the creation of a new scheduling algorithm.

My performance objectives are the maximization of the average resource utilization and the minimization of the total server rental cost due to a few reasons. The first reason is that when we pay for something, we always want to get the most out of it and the servers in this case are not an exception. More specifically, we want all the servers rented by us to execute jobs to their full potential with as little idle time as possible so that the total rental cost will be worthwhile, which means the average resource utilization will need to be maximized. In addition, we do not want to pay a lot for renting the servers even though our money spent is worth it as in real life this rental cost can become excruciatingly high and sometimes can even go way over our budget without us realizing before it is too late. Consequently, the total server rental cost will also need to be minimized.

## 3. Algorithm Description

My scheduling algorithm is called `allToCheapest` algorithm. It will schedule every job j to the first server s that satisfies both conditions:

- Having enough resource capacity to execute j, i.e., capable of executing j
- Having the cheapest hourly rate per core, i.e., the hourly rate divided by the core count yields the smallest result out of all server types

A scheduling scenario:

- Sample config (ds-config01--wk9.xml):

```
<config randomSeed="65535">
    <servers>
        <server type="tiny" limit="1" bootupTime="40" hourlyRate="0.4" coreCount="1"
memory="4000" disk="32000"/>
        <server type="small" limit="1" bootupTime="40" hourlyRate="0.4" coreCount="2"
memory="8000" disk="64000"/>
        <server type="medium" limit="1" bootupTime="60" hourlyRate="0.8" coreCount="4"
memory="16000" disk="128000"/>
    </servers>
    <jobs>
        <job type="short" minRunTime="1" maxRunTime="60" populationRate="30"/>
        <job type="medium" minRunTime="61" maxRunTime="600" populationRate="40"/>
        <job type="long" minRunTime="601" maxRunTime="3600" populationRate="30"/>
    </jobs>
    <workload type="unknown" minLoad="20" maxLoad="60"/>
    <termination>
        <condition type="endtime" value="86400"/>
        <condition type="jobcount" value="5"/>
    </termination>
</config>
```

- Schedule (my-wk9-all.txt / my-wk9-brief.txt):
  + Job 0 => Server small 0 because job 0 requires 1 core, 700 MB of memory and 600 MB of disk space and small 0 is the first server that is capable of executing job 0 with the cheapest hourly rate per core (0.4 / 2 = 0.2)
  + Job 1 => Server small 0 because job 1 requires 1 core, 400 MB of memory and 800 MB of disk space and small 0 is the first server that is capable of executing job 1 with the cheapest hourly rate per core (0.4 / 2 = 0.2)
  + Job 2 => Server small 0 because job 2 requires 2 cores, 900 MB of memory and 1600 MB of disk space and small 0 is the first server that is capable of executing job 2 with the cheapest hourly rate per core (0.4 / 2 = 0.2)
  + Job 3 => Server small 0 because job 3 requires 1 core, 500 MB of memory and 3300 MB of disk space and small 0 is the first server that is capable of executing job 3 with the cheapest hourly rate per core (0.4 / 2 = 0.2)
  + Job 4 => Server medium 0 because job 4 requires 4 cores, 1600 MB of memory and 4600 MB of disk space and medium 0 is the first server that is capable of executing job 4 with the cheapest hourly rate per core (0.8 / 4 = 0.2)

## 4. Implementation Details

The algorithm is implemented as the only method `allToCheapest` inside the Scheduler class. The method accepts 2 parameters:

- `servers`: an `ArrayList` of `Server` objects (with attributes `type, id, state, curStartTime, core, memory, disk, waitingJobs, runningJobs`) capable of executing a particular job in the ascending order of core count created by parsing the response of ds-server to a `GETS Capable` request from the client
- xmlServers: an `ArrayList` of `Server` objects (with attributes `type, limit, bootupTime, hourlyRate, core, memory, disk`) in the ascending order of core count created by parsing ds-system.xml written by ds-server after receiving an `AUTH` request from the client

First of all, the method creates a new `ArrayList` of `Server` objects (`tempServers`) with only `Server` objects from `xmlServers` whose `type` attributes are also present in `servers`. After that, the method finds the first `Server` object (`xmlCheapest`) with the cheapest hourly rate per core in `tempServers`. Last but not least, the method finds the first `Server` object (`cheapest`) with the same `type` attribute as `xmlCheapest` in `servers` and returns that `Server` object.

The method functions as expected because it makes use of:

- The `ArrayList` class from the `java.util` library
- The `Server` class with private attributes `id, type, limit, bootupTime, hourlyRate, core, memory, disk, state, curStartTime, core, memory, disk`, their corresponding getters and 2 separate constructors for the 2 different use cases as described above.

## 5. Evaluation

Sample config (ds-S1-config02--demo.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- generated by: Y. C. Lee -->
<config randomSeed="2048">
  <servers>
       <server type="m1.tiny" limit="10" bootupTime="60" hourlyRate="0.1"
coreCount="1" memory="2000" disk="16000" />
       <server type="m1.small" limit="10" bootupTime="60" hourlyRate="0.2"
coreCount="2" memory="4000" disk="32000" />
       <server type="m1.medium" limit="10" bootupTime="60" hourlyRate="0.4"
coreCount="4" memory="8000" disk="64000" />
       <server type="m1.large" limit="10" bootupTime="60" hourlyRate="0.8"
coreCount="8" memory="16000" disk="128000" />
       <server type="m1.xlarge" limit="10" bootupTime="60" hourlyRate="1.6"
coreCount="16" memory="32000" disk="256000" />
```

```
  </servers>
  <jobs>
      <job type="instant" minRunTime="1" maxRunTime="30" populationRate="10" />
      <job type="short" minRunTime="31" maxRunTime="180" populationRate="40" />
      <job type="medium" minRunTime="181" maxRunTime="600" populationRate="30" />
      <job type="long" minRunTime="601" maxRunTime="1800" populationRate="20" />
  </jobs>
  <workload type="medium" minLoad="40" maxLoad="70" />
  <termination>
      <condition type="endtime" value="86400" />
      <condition type="jobcount" value="1000" />
  </termination>
</config>
```

Results:

- My algorithm (my-all.txt / my-brief.txt):

```
# ------------------------------------------------------------------------
---------
# 1 m1.tiny servers used with a utilisation of 100.00 at the cost of $3.76
# 1 m1.small servers used with a utilisation of 100.00 at the cost of $6.52
# 1 m1.medium servers used with a utilisation of 100.00 at the cost of $7.83
# 1 m1.large servers used with a utilisation of 100.00 at the cost of $11.02
# 1 m1.xlarge servers used with a utilisation of 100.00 at the cost of $10.68
# ================================== [ Summary ]
==================================
# actual simulation end time: 135332, #jobs: 1000 (failed 0 times)
# total #servers used: 5, avg util: 100.00% (ef. usage: 100.00%), total cost:
$39.81
# avg waiting time: 45383, avg exec time: 396, avg turnaround time: 45779
```

- FF algorithm (ff-all.txt / ff-brief.txt):

```
# ------------------------------------------------------------------------
---------
# 10 m1.tiny servers used with a utilisation of 72.62 at the cost of $2.71
# 10 m1.small servers used with a utilisation of 75.03 at the cost of $5.44
# 10 m1.medium servers used with a utilisation of 79.74 at the cost of $11.09
# 10 m1.large servers used with a utilisation of 71.16 at the cost of $22.25
# 10 m1.xlarge servers used with a utilisation of 62.47 at the cost of $29.11
# ================================== [ Summary ]
==================================
# actual simulation end time: 11175, #jobs: 1000 (failed 0 times)
# total #servers used: 50, avg util: 72.21% (ef. usage: 71.12%), total cost:
$70.60
# avg waiting time: 7, avg exec time: 396, avg turnaround time: 403
```

- BF algorithm (bf-all.txt / bf-brief.txt):

```
# ------------------------------------------------------------------------
---------
# 10 m1.tiny servers used with a utilisation of 72.62 at the cost of $2.71
# 10 m1.small servers used with a utilisation of 74.13 at the cost of $5.49
# 10 m1.medium servers used with a utilisation of 72.79 at the cost of $11.06
# 10 m1.large servers used with a utilisation of 54.20 at the cost of $22.17
# 10 m1.xlarge servers used with a utilisation of 63.25 at the cost of $30.99
```

```
# ================================== [ Summary ]
==================================
# actual simulation end time: 11175, #jobs: 1000 (failed 0 times)
# total #servers used: 50, avg util: 67.40% (ef. usage: 67.15%), total cost:
$72.42
# avg waiting time: 6, avg exec time: 396, avg turnaround time: 402
```

- WF algorithm (wf-all.txt / wf-brief.txt):

```
# ------------------------------------------------------------------------------
---------
# 0 m1.tiny servers used with a utilisation of 0.00 at the cost of $0.00
# 0 m1.small servers used with a utilisation of 0.00 at the cost of $0.00
# 2 m1.medium servers used with a utilisation of 100.00 at the cost of $0.04
# 10 m1.large servers used with a utilisation of 64.10 at the cost of $21.93
# 10 m1.xlarge servers used with a utilisation of 99.62 at the cost of $51.91
# ================================== [ Summary ]
==================================
# actual simulation end time: 22391, #jobs: 1000 (failed 0 times)
# total #servers used: 22, avg util: 83.51% (ef. usage: 83.39%), total cost:
$73.88
# avg waiting time: 438, avg exec time: 396, avg turnaround time: 834
```

Comparisons:

- The average resource utilization of my algorithm is the highest out of the 4: a perfect 100% compared to 72.21% of FF algorithm, 67.40% of BF algorithm and 83.51% of WF algorithm (pro)
- The total server rental cost of my algorithm is the lowest out of the 4: just $39.81 compared to $70.60 of FF algorithm, $72.42 of BF algorithm and $73.88 of WF algorithm (pro)
- The average turnaround time of my algorithm is the highest out of the 4: 45779 compared to just 403 of FF algorithm, 402 of BF algorithm and 834 of WF algorithm (con)

⇨ The pros outweigh the cons

⇨ My algorithm is resource-efficient and cost-efficient but not time-efficient

## 6. Conclusion

My algorithm achieves 2 out of the 3 stated performance objectives (maximization of the average resource utilization and minimization of the total server rental cost). More specifically, my algorithm will be best suited for scenarios where a resource-efficient and / or cost-efficient approach is needed but will not be suitable for situations where a time-efficient solution is required instead.

## 7. References

Project git repository URL: https://github.com/frankie2305/2021COMP3100_STAGE2