Amir Lingcoran

45419302

COMP3100 Distributed Systems

Stage 2 Report

OPTIMIZED CLOUD JOB SCHEDULER

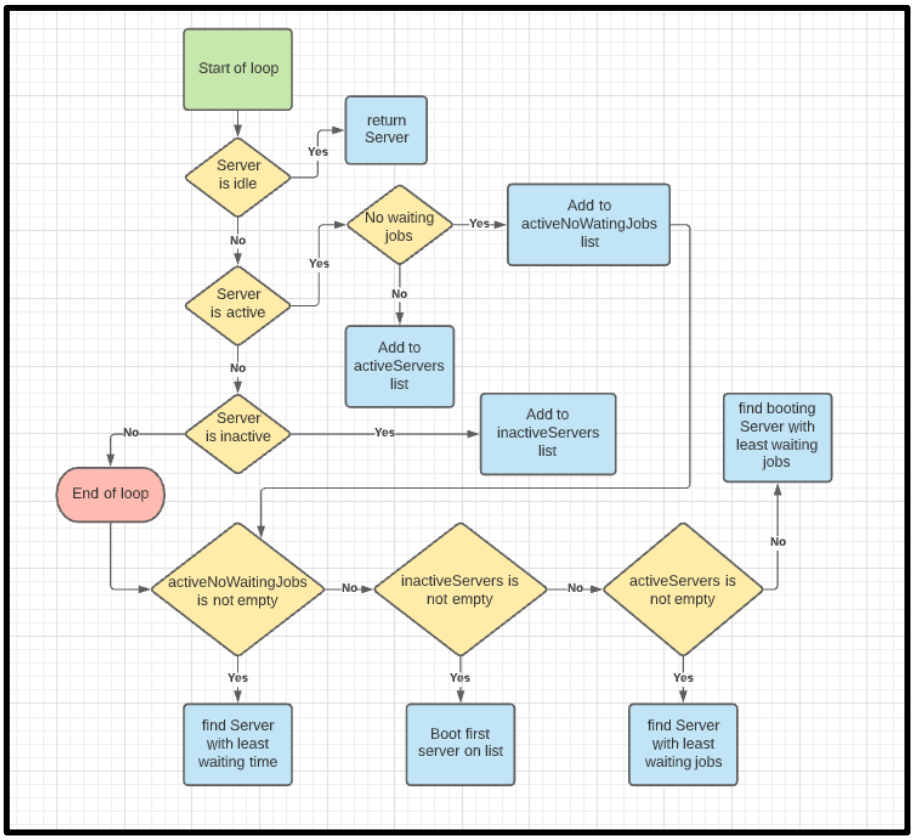
Introduction

Stage 2 of the COMP3100 (Distributed Systems) assignment aims to design and implement a job scheduling algorithm that considers the following factors: turnaround time (from submission of job to completion), resource utilisation (server usage), and overall rental cost. The objective is to have the proposed algorithm perform better than the three baseline algorithms: First-Fit, Best-Fit, and Worst-Fit.

Problem Definition

The proposed algorithm aims to reduce the overall rental cost through maximising the utilisation of all servers used during the job scheduling process. Moreover, this must be achieved without a significant increase in turnaround time.

Algorithm Design



For every job received by the Client from the ds-server, a list of capable servers is generated via GETS Capable. To ensure that booted servers are utilised efficiently, idle servers are given the highest priority. Despite not running jobs, idle servers charge its hourly rate from the time that it starts running until it is terminated or the simulation ends (all jobs have been scheduled). So, it is crucial that all booted servers are always running jobs to keep cost in line with server usage. This is achieved by iterating through the list of capable servers in a loop whilst checking whether an idle server exists. Of course, there are few other server states that the algorithm handles: *inactive* and *active* (divided into *no waiting jobs* and *with waiting jobs* categories). If the algorithm encounters any of the three, it would simply add those servers to separate ArrayLists according to their respective states. On the other hand, if a server is found idle, the method will immediately return that particular Server object and assign it as the optimal server for the current job in queue. This places idle servers at the top of the priority tree.

In the case that no capable servers are found idle, the loop will break upon reaching the last server on the list. At this point, active servers with and without waiting jobs would have been stored in the *activeServers* and *activeNoWaitingJobs* lists respectively, if any. Moreover, inactive servers would have been stored in the list *inactiveServers*, if any.

Placing second in the priority tree are active servers with *no* waiting jobs but are currently running a job(s). The algorithm checks if the list *activeNoWaitingJobs* is not empty and, if so, searches for the server with the least waiting time in the said list. This is achieved by calculating the latest possible completion time within a particular active server and comparing the result with the submit time of the current job to be scheduled.

Est. completion time = startTime + estRunTime

If the current job's submit time is greater than the estimated latest completion of a running job in a server, then the job is scheduled to that server as it is *almost* guaranteed to start upon submission. Otherwise, the difference between the latest completion time and the job's submit time is calculated and compared with other differences. The job is scheduled to the server with the minimum difference.

Next on the priority tree are inactive servers, followed by active servers with 1 or more waiting jobs. This decision revolves around maintaining a good overall turnaround time since queueing more jobs to an already active server with multiple waiting jobs will incrementally increase waiting time for all subsequent jobs. Booting another server can be used for future jobs and give other servers to complete jobs in their queue. However, if all servers have already been booted, received jobs are scheduled to the active server with the least number of waiting jobs.

In the case that none of the conditional branches stated above are satisfied, all servers are presumed booting. The first (smallest) booting server with the least waiting jobs, capable of running the job, is assigned as the optimal server.

Simple Scheduling Scenario

```
CLIENT REDY
SERVER JOBN 101 3 380 2 900 2500
CLIENT GETS Capable 3 700 3800
SERVER  juju 0 booting … 1 0
        Juju 1 booting … 1 0
        Joon 0 active … 0 1 ← 0 waiting jobs & 1 running job
        Joon 1 inactive … 0 0
        Super-silk 0 inactive … 0 0
CLIENT SCHD 3 joon 0 ← selected active server with no waiting jobs
SERVER OK

CLIENT REDY
SERVER JOBN 137 4 111 1 100 2000
CLIENT GETS Capable 1 100 2000
SERVER  juju 0 active … 0 1 ← 0 waiting jobs & 1 running job
        Juju 1 booting … 1 0
        Joon 0 active … 1 1
        Joon 1 inactive … 0 0
        Super-silk 0 inactive … 0 0
CLIENT SCHD 4 juju 0 ← selected active server with no waiting jobs
SERVER OK

CLIENT REDY
SERVER JOBN 156 5 8 3 2700 2600
CLIENT GETS Capable 1 100 2000
SERVER  joon 0 active … 1 1
        joon 1 inactive … 0 0 ← selected first inactive on list
        super-silk 0 inactive … 0 0
CLIENT SCHD 5 joon 1 ← no idle/active (no waiting jobs) servers
SERVER OK
```

Implementation

This implementation uses **GETS Capable** to request a list of all capable servers for a particular job from the ds-server. Messages/replies from the ds-server are read via the BufferedReader, one line at a time. The Client explicitly uses new line characters to send messages to the ds-server and thus -n is required for executing the program on the terminal. Data from String objects containing capable server information are used as fields for the creation of Server instances which are then stored in an ArrayList – *capableServersList*. ArrayLists are used throughout the implementation since they provide ease of adding new elements and are highly dynamic in terms of scaling. Moreover, it is a part of the Java Collections API and thus inherits a multitude of useful functionalities compared to Java arrays.

The algorithm for optimally scheduling jobs to minimise rental cost and maximise resource utilisation occurs via the call to the Client method, *getOptimalServer*().

Client Methods

- ***setup***() – initialise the socket, input and output streams, and the BufferedReader
- ***writeBytes***() – client method for sending messages/commands to the ds-server. The new line character is explicitly added at the end of each String message before writing it into the DataOutputStream.
- ***getCapableServersList***() – returns an ArrayList of Server objects that are capable of scheduling a particular job. This is achieved by sending the GETS Capable command to the ds-server and using the data length sent back by the server as argument for the method to determine the number of loop iterations/line reads (each server state information ends with the new line character).
- ***getOptimalServer***() – returns a Server object that is identified to be the most optimal decision for scheduling the current job based on the algorithm design proposed. The method calls multiple helper functions – *getServerWithLeastWaitingTime*() and *getLeastWaitingJobs*().
- ***getServerWithLeastWaitingTime***() – returns a Server object that is identified to hold running jobs that are expected to finish the earliest compared to all other servers on the list of capable servers. This method employs two helper methods: *getRunningJobsList*() and *getLatestCompletionTime*().
- ***getRunningJobsList***() – returns an ArrayList of String objects that represent all running job on a particular server. Such list is generated by sending the LSTJ command to the ds-server.
- ***getLatestCompletionTime***() – returns an Integer that denotes the completion time of a running job that is estimated to finish last among other running jobs on a server. By taking an ArrayList parameter, *runningJobs*, that holds data of all running jobs from a particular server, the helper method calculates the estimated completion time of all jobs:
- ***getLeastWaitingJobs***() – returns the Server object with the least number of waiting jobs compared to all other capable servers on the list.
- ***close***() – terminates connection between Client and the ds-server

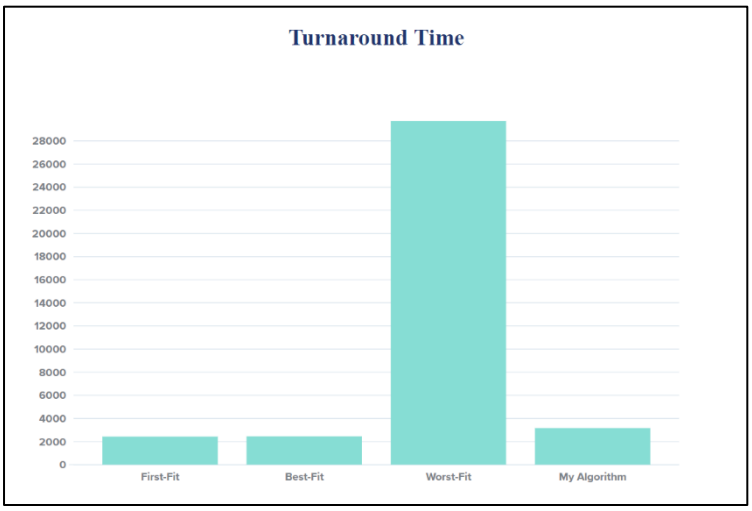
Evaluation

Simulation setup: to evaluate the efficiency of the proposed algorithm, the configuration file 'config100-long-high.xml', located inside the same directory as the ds-server and ds-client, is used. Two terminals were opened to run both ds-server (using the stated configuration file) and the ds-client.

./ds-server -c ds-sample-config01.xml -v brief -n

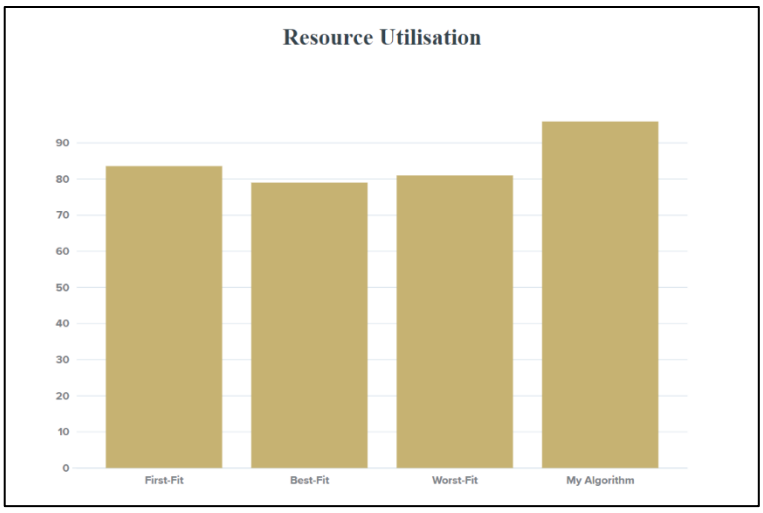
./ds-client -a [ff, bf, wf]

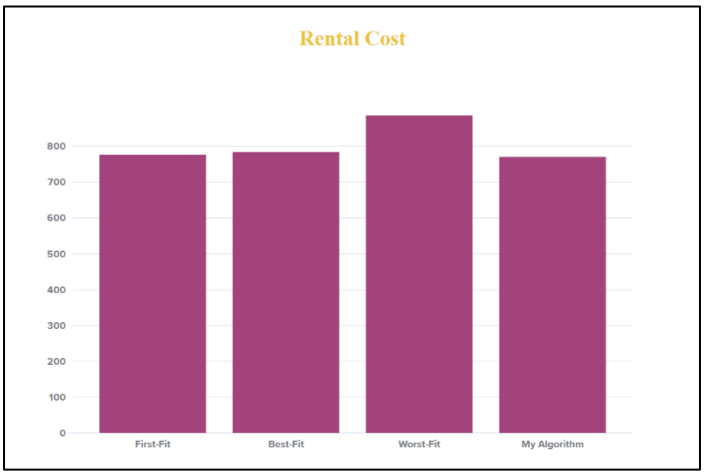
This configuration file is used for all three baseline algorithms (FF, BF and WF) and the results (turnaround time, resource utilisation, and rental cost) were recorded. Afterwards, the same configuration file is used for the new proposed algorithm to compare its results with the others.



The proposed algorithm performs significantly better (~89% improvement) than the Worst-Fit algorithm in terms of turnaround time, though it is slightly behind First-Fit and Best-Fit.

The proposed algorithm maximises resource utilisation better than all 3 baseline algorithms.





The proposed algorithm has lower cost than all 3 baseline algorithms, though only slightly below First-Fit and Best-Fit.

Advantages of proposed algorithm

- Overall rental cost is lower than all three baseline algorithms (First-Fit, Best-Fit, and Worst-Fit).
- Resource utilisation is maximised as a result of the priority tree structure, ensuring that active servers are always running without significant increase in turnaround time.
- Resource utilisation is higher than all three baseline algorithms (FF, BF and WF).
- Turnaround time is minimised and is better than that of Worst-Fit.

Disadvantages of the proposed algorithm

- Complexity of algorithm may affect overall performance, especially when handling large number of servers.
- Code can be further optimised (i.e., minimising use of loops, etc.).
- Turnaround time is worse than First-Fit and Best-Fit.

Conclusion

Ensuring that all booted/active servers are always running jobs maximises resource utilisation and minimises rental cost overall. This is due to the fact that booted servers charge fixed hourly rates despite not running any jobs. Moreover, avoiding extremely long queues on a single server leads to reduced turnaround time.

References

GitHub repository: https://github.com/a-lin420/comp3100-stage2-optimized-job-scheduler