**DISTRIBUTED SYSTEMS – JOB SCHEDULER**

**Name:** Samuel Ryu (45377553)

**Introduction**

The distributed systems job scheduler is a client implementation of a client-server-based interaction where the client receives jobs from the host server and allocates these jobs on different servers.

In the first stage, the simulation entailed a socket connection, a simple handshake and an All-To-Largest (ATL) job allocation. The primary focus of the first stage was to establish a socket connection, use the commands available on the host server, and provide a simple scheduling algorithm that allocates every job to the largest server. However, ATL is a naïve algorithm and does not account for metrics such as turnaround time (TT), resource utilisation (RU), and costs.

The introduction these new measurements provide a need for a more sophisticated algorithm that can utilise the information concerning each job and server. Where ATL suffers from extremely large TT, it provides superficially high RU and low costs. Thus, the goal of stage 2 is to provide an algorithm that significantly surpasses the TT of ATL but also provides a reasonable RU and costs. The algorithm will also be compared to Best Fit (BF), Worst Fit (WF), and First Fit (FF) baseline algorithms to ensure that reasonable comparisons can be made to existing algorithms.

**Problem Definition**

The scheduling problem involves finding the best server to allocate a job while also considering future jobs. However, there is little information on what the oncoming jobs are, or information on how many jobs are left. Thus, calculations can only be performed on the current job. The best solution is to allocate the most suitable job to the server at its current time such that future jobs have as many choices as possible to also provide also suitable allocation. This becomes difficult because the different metrics are closely related as demonstrated by ATL where the algorithm’s TT is severely high whereas its RU is efficient, and costs are extremely low. Improving turnaround time would require using multiple servers to reduce the wait time on each server, but consequentially would increase costs and RU.

The objective of stage 2 implementation is to create an algorithm that significantly surpasses the TT of ATL however improve the RU while also maintaining TT and costs in relation to baseline algorithms. The reason for choosing RU is that BF provides good TT however has very poor RU and suffers from large costs. WF significantly provides RU however suffers very large TT. RU is tied linearly with costs and better RU would result in smaller costs. Thus, the stage 2 algorithm will provide generally better TU compared to WF while also providing significantly better RU.

**Algorithm Description**

Two algorithms have been implemented however, the main algorithm is Half Fit, whereas the other algorithm is named Prioritise Idle.

Half Fit was based on multiple principles inspired from the WF and FF baseline algorithm. The approach is to try to allocate jobs on servers with the higher core count and eventually work down to the servers with lower cores if required. In theory, if smaller jobs are started on big servers, it reduces the number of servers required to complete the smaller jobs but additionally reduces the bootup time and the number of servers that have been booted. The largest servers are reserved for the largest jobs since only the largest servers can process these jobs. The selection of servers is prioritised by idle, inactive, then active. When choosing active servers, it selects based on which server has fewer waiting jobs. Both largest jobs and other jobs are selected with the same prioritisation. When the algorithm starts, it obtains a list of available servers and searches for the largest and second largest cores. It creates a list of the largest servers and stores the number of the second largest cores. The algorithm then requests for jobs and sees whether the current job has larger than the second largest server cores, which means whether this job can only be performed by the largest servers. It requests a GETS Capable query which returns the updated list of capable servers of running it. If it is true then, the prioritisation phase is initiated. It traverses through each server in the list and looks for any idle servers that can be reused, if not then find inactive servers that are not being used. If the server none of these two conditions can be met, then the server with least waiting jobs is chosen via a linear search. If the job is not a largest core server required job, then the job will choose a server based on Worst Fit initially, and then fills that server up until it chooses the next large server. The other jobs do not however remove all the largest servers. It keeps the servers that are idle, in case a small job and be fitted alongside the larger jobs. With less servers being activated, this will increase the RU and decrease costs, but will also result in jobs having to wait longer as small and medium jobs share servers.

Prioritise Idle is the secondary algorithm that was used to improve TT but was unsuccessful. It was based off BF however attempted to prioritise idle servers over normal best fitness value calculations. The algorithm starts by requesting for jobs to be scheduled, and iterates through each job. Each job is selected based on the lowest fitness value. In theory, it matches the job with the smallest server possible capable of processing the job, which would be efficient and result in higher utilisation and costs. An after check is performed to see whether an idle servers can be used instead and schedules to the first available idle server.

**Implementation**

The socket and communication methods were refactored to follow a singleton instance. Due to the introduction of multiple classes for different algorithms. It was decided necessary to implement a singleton pattern such that, only a single instance of the socket, data input stream, and data output stream was created. By forming a private constructor which is called within the object itself, it ensures that no other classes can call the constructor. However, all the classes can call its methods. Communication builds these methods upon the socket.

The algorithm was designed as a while loop as the number of iterations were unknown. The while loop conditions were based on the response from the server. The message read from the server would be stored and continuously updated.

The server class object was a class object file which stored all the details necessary regarding a server. A server had a type, id, current start time, cores, memory, disk, waiting jobs, and running jobs. All these information about the server needs to be stored, and multiple of these server objects can exist at once. Information is also prone to change, for example, if a server is allocated with a job, the core count will reduce to reflect the remaining available cores. The primary reason for the object however was that the data structure contained different primitive types of fields, both String and integers. The need a data structure for multiple servers with changing information means an ArrayList should be used. An ArrayList of servers allows only servers to be used but also has useful methods such as the “removeIf(Predicate p)” method which allows the algorithm to filter the list based on a specifiable condition.

The deep copied ArrayList was required because the original ArrayList needed to remain unchanged in case the deep copy ArrayList returns no results. The importance of the deep copy was emphasised because soft copied array list, which are created by creating a new ArrayList then referencing the old ArrayList would result in modifications to both ArrayLists if one was modified.

Fitness Values was calculated based on server cores minus the job cores.

**Evaluation**

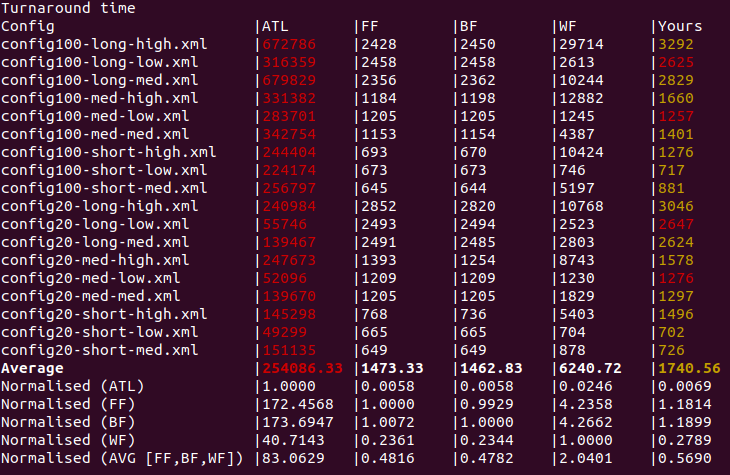
* Simulation Setups

Setup:

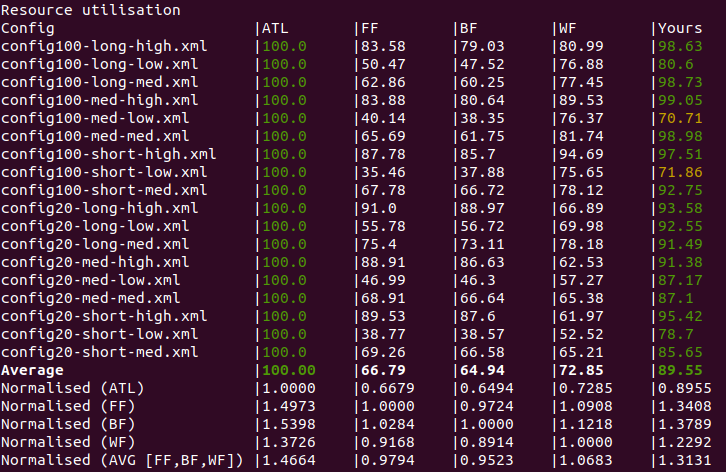
* Download the COMP3100\_DS\_Scheduler
* Extract the .zip
* Inside the folder, open the Linux terminal
* Compile the Java files using the command:
  + “javac Client.java”
* Then run the test binary file using the command:
  + “./test\_results “java Client -hf” -o ru -n -c other/

The stage 2 DS Scheduler shown considerable improvements on TT compared to ATL. ATL still maintains a higher RU and lower costs however these results are disregarded as they are superficial and completely ruined costs.

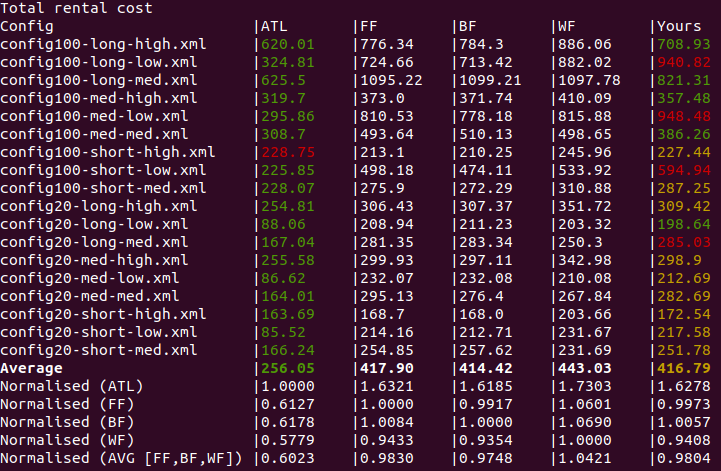
BF and FF had lower TT on every config compared to HF, however HF had shown near same TT on some configuration files. On closer examinations, the configurations that were low.



The RU on Half Fit significantly was improved on average compared to BF and FF on all configurations. HF was on average better than WF however some configurations such as low configurations were worse than WF. HF has achieved its goal of increasing the RU.



Costs had some mixed results and was sometimes able to perform higher. But interestingly enough it was unable to.



**Conclusion**

Overall, the DS Scheduler stage 2 implementation was able to create an algorithm that had improved the metrics on the baseline algorithms and was a successful project.

**References**

GitHub Repository: <https://github.com/SamuelRyu/COMP3100_ds_scheduler>