Project report: Stage 2
COMP3100 Distributed Systems, S1, 2022
SID: 46564616, Name: Weibin Chen

[1]

1 Introduction

The overall purpose of this project is to create a client that simulates a scenario based on a set of servers to be scheduled and job setup profiles, following a set of communication protocols to schedule jobs to the various servers in the simulation.
In Stage 1, the scenario was simulated, an experimental environment was built, handshake communication with the servers was implemented, and the most basic allToLargest specification was applied to schedule jobs without considering the optimization of the various resources. To schedule jobs efficiently in a distributed system environment, it must focus on providing good performance metrics, namely turnaround time (TT), resource utilization (RU), and server rental cost (CO).
In Stage 2, we focus our experiments on algorithm optimization. By using a graphical tool to analyze the resource usage of each server at different times and comparing it with various baseline algorithms, we will implement a new scheduling algorithm that provides the minimum turnaround time. Minimizing the average turnaround time will reduce the time between job submission and job completion by as much as possible and balance resource utilization and rental costs.

2 Problem definition

The ATL algorithm in the first phase was inefficient, simply sending all jobs to the largest server without considering objective metrics such as turnaround time, resource utilization, and server cost. In particular, compared to other baseline algorithms, it performs worst when looking at resource utilization and rental costs, as it uses all the most expensive servers, especially when the server capacity is much larger than the job requirements and keeps large servers active.
There are currently four baseline algorithms First Capable (FC), First-Fit (FF), Best-Fit (BF), and worst-fit (WF). Each of them has its focus and has a different effect on optimizing other metrics or different degrees of deterioration.[2]
We need to optimize turnaround times while considering resource utilization and rental costs. We use the alternative baseline algorithms FC, FF, BF, and WF as reference criteria to evaluate the new algorithm's performance we have designed. The acceptance criterion for the project is equal or better performance in terms of resource utilization and rent cost compared to the baseline algorithm. The new algorithm had to outperform all alternative baseline algorithms in terms of turnaround time. In addition to having the lowest average turnaround time, it must also beat other algorithms in most individual comparisons of configurations. In practice, however, this is not so straightforward, as the three metrics are conflicting, and optimizing one of them inevitably leads to a deterioration of the other two. For example, if a job is always submitted to only one server, the user pays for only one server, reducing the total cost of the server lease. Still, if that server cannot accept more requests, the job will be transferred to another server, increasing job turnaround time. The algorithm must not be worse than the other algorithms in all cases. Still, it must be optimal in one of the factors, not just a simple compromise between the various algorithms, so it is difficult to improve the algorithm.

3 Algorithm description

By analyzing all the server configuration XML files, we can confirm that the price of the server type used is proportional to the number of cores on the server. The server's resource utilization is also proportional to the number of cores being used. The primary basis of our new algorithm is to check the usage of cores as the basis for the calculation. The new algorithm compares the number of cores required for a job with the cores available on the server, then selects the server with the slightest difference and schedules the job to that server. Each job is sent to the server that is best suited to each given job. However, depending on the status of the server, other available resources, the number of jobs waiting, and the priority level, there are certain algorithmic adjustments.
In order to minimize job turnaround time, the following guidelines are used to schedule jobs for the server.

1) We will rank the server status from highest to lowest priority: IDLE, ACTIVE, BOOTING, INACTIVE. By prioritizing servers with IDLE, ACTIVE status, we can reduce the turnaround time, ensure higher resource utilization, and minimize leasing costs by not turning on more servers.

2) the server with the lowest number of jobs waiting is ranked first.

3) Servers with sufficient core, memory, and disk resources to meet the current job requests are given priority. To meet resource utilization and rental cost metrics. We will select the server with the slightest difference between the number of cores required for a job and the number of cores available, prioritizing the smaller server to minimize rental costs.

Based on the above criteria, we obtain the resource status of the server one at a time through the command GETS, iterate through all the given servers, and then filter the servers in the following six steps:

1) We will select the servers with the IDLE or ACTIVE status with enough core, memory, and disk resources to satisfy the current job request. The number of jobs currently waiting is required to be 0. This is because if the server has the resources to meet the current demand and has waiting jobs, the server is under-resourced and cannot satisfy the resource requirements of the queued jobs. Suppose the job is directly scheduled to this server. In that case, there is nothing wrong with it, the job will be executed immediately, but the server will not be able to release more resources later to satisfy the queued jobs. It will instead have a longer turnaround time for the queued jobs.

2) If no reasonable server is found in the first point above, servers with a BOOTING status will be selected, provided they have enough resources to satisfy the current job request. However, the number of waiting jobs is not taken into account here. Since the server is booting, all jobs are waiting, and as long as the server has resources left to satisfy the job, it will not affect other queued jobs.

3) If no reasonable server is found in the second point above, servers with the status INACTIVE will be selected, provided that they have enough resources to satisfy the current job request. In the same way, we do not need to consider the number of jobs waiting here.

4) If we do not find a reasonable server at point three above, we will revert to point one and select a server with the status ACTIVE, choosing the server with the least number of waiting jobs and the most available core resources.

5) If no reasonable server is found at point 4 above, all servers do not have enough resources to satisfy the current job request. So we will choose the first server returned by the system.

4 Implementation

On top of the code based on phase 1, we have added the functions to optimize the turnaround time algorithm. These are marked in red in the UML diagram below. The main program of the algorithm is algorithmTT(), which returns the best matching server based on the requested job requirements and the current server status. The main program is divided into four subroutines, algorithmTTActive(), algorithmTTBooting(), algorithmTTInative(), algorithmTTRest(), which correspond to the server states (ACTIVE, IDLE), BOOTING , INATIVE, and insufficient resources.
We first get the job request, job number, job type, submission time, estimated runtime, and the requested core, memory, and disk resources from the class JobSubmission instance. We then obtain the current information about the server with the command GETS and store the data in an instance of class ServerStatus. We can get the current status of the server, available resources, and job queue, such as server type, server ID, server state, server start time, and server waiting.
The JobSubmission and ServerStatus instances are entered into each of the four turnaround time algorithm applets. These four applets will filter and sort the servers in different states according to the section 3 algorithm rules, applying dissimilar conditions to get the most suitable server.
Since the Java language provides powerful filtering and sorting functions, we do not have to struggle with endless for and switch functions like we do in c. We are allowed to use the collection.sort() method for single keyword sorting and the stream().sort() method for multi-keyword sorting. The removeIf() function quickly eliminates any items that match the criteria.[3] [4]

5 Evaluation

To verify the performance of our new algorithm, we need to compare it with the other four baseline algorithms to see the strengths and weaknesses of our unique algorithm. However, since the provided test program,

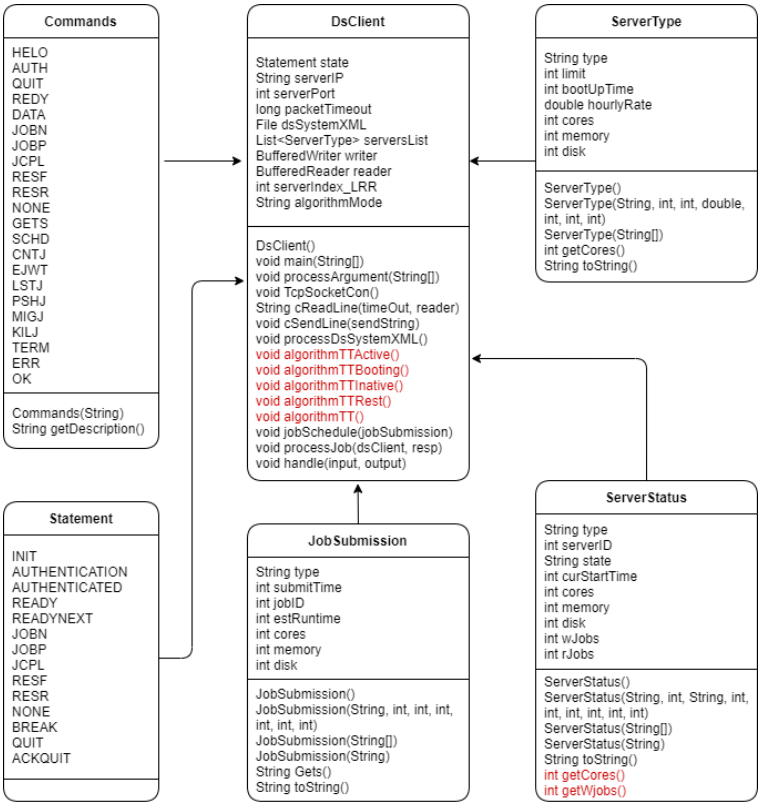

Figure 1: Ds-client java UML

stage2-test-x86, can only summarize and rate for 20 profiles. The simple numerical results do not allow us to analyze the algorithm in detail and identify the root of the problem. So we use ds-viz graphical analysis software when designing our algorithms. This analysis software converts the input server configuration files and run logs into graphs. The vertical axis represents each server that appears in the current simulation configuration, and the horizontal axis represents the time elapsed during the simulation. The graph shows how long the server has been in use, indicated by a green horizontal bar. This allows us to see the operational details of each job, when multiple jobs are allocated, when they are executed and when they are finished on each server, giving us a more direct and objective view of turnaround times, resource utilization, and rental costs.

1) First, we analyse the config20-short-high.xml with the First Capable (FC) algorithm. As you can see, jobs are permanently assigned to the first server of each type, resulting in long queues for most jobs and an average turnaround time of a staggering 124,780s, 41 times that of the FF or BF algorithms. Due to many jobs queued, the running servers are always full, with a resource utilization of 99.74Since the active server is always full, and the price of server type usage is proportional to the number of cores on the server, the rental cost is quite good at $133.57, the least expensive of all the algorithms.

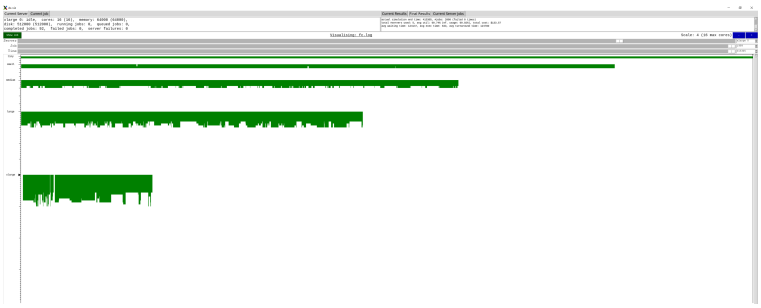

Figure 2: First Capable (FC)

2) The images of First-Fit (FF) and Best-Fit (BF) are very similar. Comparing the data confirms that they have very similar turnaround times, resource utilization, and rental costs. The four baseline algorithms found the best balance of the three evaluation criteria. Our new algorithm aims to outperform FF and BF in terms of turnaround time but not in terms of resource utilization and cost of hire.

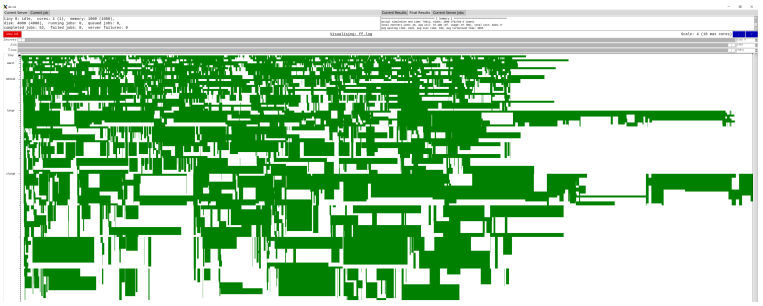

Figure 3: First-Fit (FF)

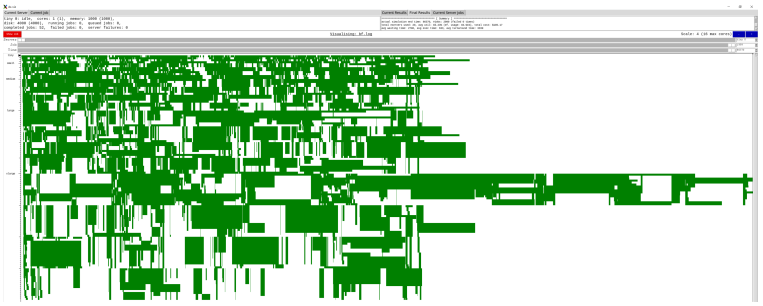

Figure 4: Best-Fit (BF)

3) The worst-fit (WF) algorithm always assigns jobs to the first server with the most resources so that the other servers always have free resources and the servers run under low load for a long time. Hence, the resource utilization and rental cost are the worst of the four baseline algorithms.

   Since the distribution is not evenly distributed, it does not take advantage of the concurrent performance of the Distributed System, which is comparable to that of a single-threaded system. Although the intention was to assign jobs to the server with the most resources in the hope of having a shorter turnaround time, this was not the case, as the average turnaround time was 34,484s, ten times that of the FF or BF algorithms, and only better than FC.

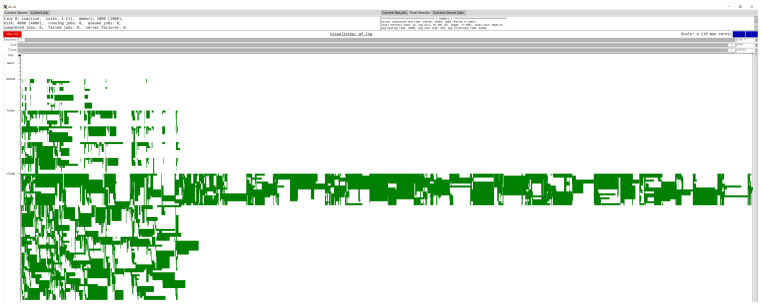

Figure 5: Worst - fit (WF)

4) Our new algorithm considers the strengths and weaknesses of the four baseline algorithms above and tries to allocate jobs to servers with similar resource availability and job requirements to have the fastest turnaround time while still ensuring proper resource utilization and rental costs. Because each server's resources are constantly changing, jobs can be assigned to different servers each time, which allows the Distributed System to take full advantage of the concurrent performance of multiple machines and avoids the problems of the WF algorithm. The green blocks are filled equally on each server. The new algorithm ranks second in resource utilization and leasing cost among the four baseline algorithms because FC's resource utilization has reached a sick 99.74%, which severely affects turnaround time, which is not what we are looking for. The new algorithm ranks first in the turnaround time metric, a quarter of the fastest FF algorithm. This proves the success of our unique algorithm.

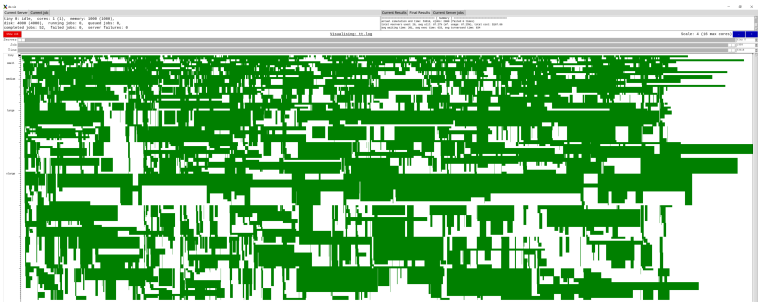

Figure 6: New algorithm (TT)

6 Conclusion

Although our new algorithm achieved its goal, the average turnaround time was reduced to 1475.39s, 21% shorter than the best baseline algorithm, FF. This is without sacrificing much in terms of resource utilization and rental costs.
However, the current algorithm is still relatively crude, simply sorting conditionally on one or two keywords and counting a simple number of queued jobs. There is no in-depth analysis of the running time of each executing or queued job, the proportion of demand for each resource type, accurate prediction of the timing of resource release, and the allocation of jobs in the wrong order to move running or queued jobs to a more suitable server. I hope to study the theory in a future course and improve on the perfect algorithm.

References

[1] C. Weibin, “Ds-sim-client.” [Online]. Available: https://github.com/benbandbetter/DS-SIM-CLIENT

[2] Y. C. Lee, J. King, Y. K. Kim, and S.-H. Hong, “Robust scheduling for large-scale distributed systems,” in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2020, pp. 38–45.

[3] Oracle, “Interface stream,” 2022, accessed May, 2022. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

[4] ——, “Class collections,” 2022, accessed May, 2022. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html