Stage 2 Design Document

Project Title

Automated Resource Allocation for a Distributed System

Group Members

- Bradley Kenny **45209723**
- John Kim **45163782**
- Mark Smith **45176655**

Introduction

Continuing on from the previous checkpoints for this project, we have been tasked with the implementation of improving job to server functionality. Previously, we've used the *allToLargest* function which directs all of the jobs to the largest server type. While this is completely operational, it is highly inefficient. Ideally, using the provided job information such as *job submit time*, *estimated runtime* and required *CPU cores*, we can match the job requirements to server capabilities to best fit the operation.

This stage has been designed to ensure we implement three different algorithms; first-fit, best-fit, worst-fit. These will enable our job-schedule simulator to determine where to schedule jobs, depending on a number of conditions. These three algorithms aim to improve resource automation efficiency from the previous *allToLargest* function while differentiating the criteria which are present in the three algorithms.

Design Consideration

After studying the expectations, we believed it would be best to implement a number of classes. With this, we would be able to create an instance of each class for important information we receive from the server. For example, this information could include details about the job or resources available to us. Furthermore, this enabled us to break up our functions into a more logical way that would make sense when implementing the solution. *Figure 1* displays the relationships between these classes.

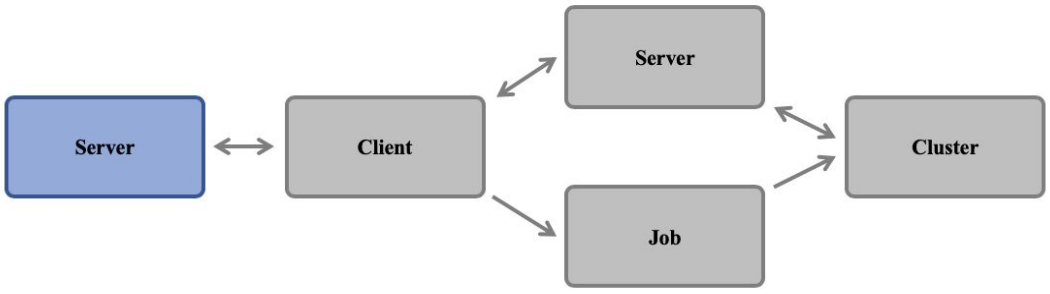

Figure 1: Relationship between classes.

Our first class is the `Client`. Located in here is our `main()` function that runs when you call the program on the command line. Furthermore, this function is the main one used in our implementation. It relies on the other classes to run but contains the core logic for the client-server protocol to operate. In particular, this class is responsible for setting up the send/receive aspect of the project. From here, it relies on other classes to determine how to communicate what information with the server.

We implemented a `Job` and `Server` class. These classes were used when extracting the information given to us from the server. We believed that having these two instances would allow easier manipulation later on when we were going to compare different aspects of each one. Both classes contained properties related to the information given to us in the string from the server. These were then parsed and set to a respective property. They made implementing our algorithms incredibly easier compared to using the strings given to us.

Our last class, `cluster`, was the most significant for this stage. We named it *cluster* due to its responsibility of handling a number of different servers, and determining what server to return to the `client` class for communication with the server provided to us. This class contained our best-fit, first-fit, and worst-fit algorithms. It required that to create an object, we provide it with an `ArrayList` of servers. The idea behind this being that information would be the resource information given by the server. The `cluster` class would then use this data for the methods contained within when called appropriately.

Our implementation checked resource and server state when necessary by sending a `RESC All` message to the server. This would then return all the information about each server to the client, which we parsed into a `Server` object, appended to an `ArrayList` and eventually used to create a `Cluster` class out of. This cluster was then able to do processing on this information.

Algorithm Description

First-Fit

Implemented by John Kim

First-fit is a scheduling algorithm which prioritises on assigning jobs to the first available servers. Availability of the servers is defined by sufficient memory, disk space and the number of CPU cores which are compared against the jobs' own requirements. This algorithm is the simplest implementation out of the three, wherein servers are sorted by *serverType* increasing in size to filter out ill-fits such as assigning small jobs to the largest server, thus producing inefficiency and higher server cost.

This algorithm iterates through the sorted servers (from smallest to largest) where it compares each job to the server with an if condition looking for whether the server in question has the ability to run the job. Sorting the servers were accomplished using a bubble sort [1]. If the said server can house the job, then it assigns the job to that server, otherwise, the algorithm looks for the first active server (defined by server status where it is not unavailable) and assigns the job to that server.

Best-Fit

Implemented by Bradley Kenny

The best-fit scheduling algorithm is responsible for finding the server that best meets the requirements of the job. Within this method, we have had to check a number of properties to determine this 'best' fit requirement including meeting *CPU*, *memory*, and *disk* requirements. Using the provided pseudo-code, we were able to create a solution that met the requirements presented to us. Furthermore, our solution was tested by comparing the given `ds-client`'s output against our own in the same situation.

This method was contained in our `cluster` class. Therefore, it would have access to our `ArrayList` containing all the information received after using the `RESC All` command. Using this `ArrayList`, we are able to iterate through and compare the properties of each `Server`. When the server has sufficient resources to run the given `Job`, we mark it and store a fitness value and available time for later comparisons. The fitness value can be determined by subtracting the required *CPU cores* by the job from the server amount of *CPU cores*. As we iterate through each server, we are able to compare each fitness value with previous scores. If we find a lower score, we know this server is a 'better' fit for the job. It should also be noted that we checked to ensure that the server *state* was able to handle a job at that given point.

Once the algorithm has completed, we can return the server that best fits the algorithm. However, there is the possibility that this 'best' server might be busy and due to our usage of `RESC All`, we have the updated resource information and so might not return any result. In this case, our designed algorithm goes through the initial information, finds the *best fit*. It will then return the best initial fit so that our scheduler can then tell the server to schedule this job to run when the other jobs have completed running on the particular server.

Worst-Fit

Implemented by Mark Smith

The Worst-Fit algorithm is a process that will allocate jobs to the largest available, or soon to be available server. The size of a server is based on its *type*, which is defined by its *coreCount.* The availability of a server is determined by whether it has enough disk, memory, and cores, demanded of the job to be scheduled. The algorithm checks these values, and calculates a *fitness* value, which is the determining variable in which server is the 'worst-fit' for the job to be scheduled to.

In order to check availability of the servers, the algorithm checks a variable in the servers array, which is updated in the Client after each RESC All call. The value of this variable is indicative of the *state* of the server, and if it is available, or likely to be available shortly, the job can be scheduled on that server, and will be assuming it's fitness value is the best of each of the servers assessed.

The calculation for the *fitness* is simply the difference between the *requiredCores* of the *job* to be scheduled, and the *coreCount* of each server. It then tracks the highest of these differences (for servers

available, or soon to be available), after assessing each server, and comparing it to previous servers, will schedule the job to the server with the ‘best’ *worstFit* value.

References

[1] https://www.geeksforgeeks.org/bubble-sort/