COMP3100 Stage 2

Louie Coghill

May 29, 2022

46399089 - Session 1, 2022
Client GitHub repo
ds-sim Github repo

Contents

1 Introduction **2**

2 Problem definition **2**
2.1 The baseline algorithms 2

3 Scheduling Algorithm **3**
3.1 Overview 3
3.1.1 New job scheduling 3
3.1.2 Job migration 4
3.2 Considerations and constraints 4

4 Implementation **4**
4.1 Client structure 4
4.2 Java library imports 4
4.3 Classes and class methods 5
4.3.1 State interface 5
4.3.2 Job class 5
4.3.3 Server class 5

5 Evaluation **5**
5.1 Algorithm Comparison 5

6 Conclusion **6**

1 Introduction

In this report, I will going over the stage two assignment where I am tasked with creating a new job scheduling algorithm for the MQ COMP3100 ds-sim. The ds-sim is a distributed system simulation that simulates time, processing servers, and creates jobs to be scheduled by a client via socket communication [1]. This report will go over, the set out requirements, the details about my algorithm, and finally end with an evaluation of its performance.

2 Problem definition

The primary goal of stage two is to write a new job scheduling algorithm. This algorithm must outperform all baseline algorithms present in the pre-compiled client for the ds-sim. The main performance metric we are focused on is the average job turnaround time for a given ds-sim configuration, however we must also consider other metrics such as average resource utilisation and rentals costs as they are also important to know about a scheduling algorithm.

2.1 The baseline algorithms

Provided in ds-sims pre-compiled client are four baseline algorithms. These algorithms are called First Capable (FC), First Fit (FF), Best Fit (BF), and Worst Fit (WF) and can be described as follows.

Algorithm	Description
First Capable (FC)	Schedules each job to the first eventually capable server. Does not consider state of servers. Has terrible average turnaround time as it is incredibly naive, however is an incredibly low cost algorithm.
First Fit (FF)	Schedules each job to the first server that has required resources immediately available. Avoids already running servers if possible. One of the better two baseline algorithms, providing fairly quick turnaround time at not too great of a cost.
Best Fit (BF)	Schedules each job to the server that has immediately available resources that closest fit the job requirements. Avoids already running servers if possible. The other better baseline algorithm, providing a very similar turnaround time and cost as seen in First Fit.
Worst Fit (WF)	Schedules each job to the server with the largest immediately available resources. Avoids already running servers if possible. Better than FC, but not a good choice when FF and BF are present. Cost is similar to FF and BF, although average turnaround time is more resembles FC.

3 Scheduling Algorithm

This section will provide an overview of the design of my scheduling algorithm including the considerations, justifications for decisions made, and the constraints posed by the ds-sim.

3.1 Overview

Upon assessment of the baseline algorithms, I had initially set out to use the ideas of Worst Fit as a basis for my work. However after some experimentation with adding features I was left disappointed with its performance. Instead I changed the basis for my algorithm to use the ideas of First Fit, and added 2 key features to improve its performance. This includes changing how the jobs are initially scheduled to a ds-sim server if no server can immediately begin the job, and the addition of job migration.

3.1.1 New job scheduling

Upon receiving a job to be scheduled, all servers with the required resources immediately available are queried for and the job is scheduled to the first result if any are returned. If no servers are returned, the client queries for all servers eventually having the required resources. For every server returned, the client checks their currently scheduled jobs disregarding all servers with waiting jobs. The server then records the soonest estimated time that the required resources will be available, and repeats the process for all other returned servers. The job is then scheduled to the server estimated to have the resources for the job available soonest. In the event that all servers have waiting jobs, the job is scheduled to the largest server relying on assumption one in the user guide [1].

Figure 1: Scheduling decision example

Suppose we had three (four core) servers and one new job to be scheduled. A query for servers able to immediately start the job will return nothing, so a query for all servers eventually capable of the job is made returning all three servers. The client then sees that Server 0 has a waiting job and disregards it. Server 1 has a job larger or equal to the new job estimated to completed at t 300. Server 3 also has a job larger or equal to the new job, but is estimated to be completed later than Server 1s job. The job is then scheduled to Server 1 as it is expected to provide the necessary resources at the earliest point in time.

3.1.2 Job migration

Upon receiving a job completion update the client finds the server the job had completed on, and queries for its scheduled jobs. If there are no waiting jobs, the client queries all other servers for their waiting jobs and migrates jobs to the server which had completed a job until its resources are all taken up by the migrated jobs. Only waiting jobs are migrated to reduce each jobs waiting time, while running jobs are ignored as migration resets their progress which ultimately extends the jobs turnaround time.

3.2 Considerations and constraints

With the two main additions mentioned above, we must justify and consider the effect it will have on the performance metrics. The use of time aware scheduling is done in an effort to reduce waiting times for jobs that must be scheduled to a waiting state. However this time awareness relies on estimations, so we cannot be certain that each decision is perfect as jobs may complete sooner or later than expected. The use of job migrations helps correct these inaccuracies by allowing the waiting jobs to start sooner by moving to any server completing jobs sooner than expected.

4 Implementation

4.1 Client structure

In preparation for stage two, I had significantly refactored the client to implement the state pattern. This separates the tasks performed by the scheduler into distinct states, and follows the principle of least knowledge by providing the client with only an interface for its state. Additionally, the management of the socket is handled by the Main class and not the client, where the client is provided with only the necessary input and output streams. The outcome of which drastically improves the programs code maintainability, making future additions to the client much easier to implement.

4.2 Java library imports

- java.net.Socket [2]

  To allow for communication between the client and the ds-sim server.
- java.io.InputStreamReader [3] and java.io.BufferedReader [4]

  Used to read messages from the server to the client via a socket.
- java.io.DataOutputStream [5]

  Used to send messages from the client to the server via a socket.
- java.util.ArrayList [6]

  Used to store a dynamic quantity of data, e.g. a GETS response.

4.3 Classes and class methods

4.3.1 State interface

The interface in which all client states implement. It is made up of a single method signature which the client can continuously call. During runtime, each state will appropraitely change the Clients state, and therefore the action being performed by the client.

4.3.2 Job class

The job class is used for storing the necessary job data received from the ds-sim. This includes the jobs ID, and the CPU, memory and disk space requirements. This class features a getQueryString() method which is used to help populate a GETS message.

4.3.3 Server class

Much like the job class, the server class is used for storing the necessary data about servers. Server data is queried for in the GetServersState and stored in a list by the Client. It features a getName() method to make building messages such as SCHD, MIGJ, and LSTJ easier.

5 Evaluation

The following evaluation will make use of the provided test suite for stage two. This test suite comes with many configurations with varying loads, job lengths and quantities of servers and server type. It runs all of the baseline algorithms in the pre-compiled ds-client in addition to my client. After each run the average turnaround time, resource utilisation and rental costs metrics are collected. With these metrics we can more easily assess where my algorithm falls short, or excels in comparison to the baseline algorithms.

In this evaluation I will not be going over the run/configuration specific metric, but instead use the aggregate values such as averages and normalised values for comparison.

5.1 Algorithm Comparison

	First Capable	First Fit	Best Fit	Worst Fit	Mine
Average turnaround time (seconds)	246382.78	1803.5	1866.72	19609.94	1434.72
Average resource utilisation (%)	94.37	63.3	60.72	68.79	63.67
Average rental cost ($)	379.2	638.18	636.89	665.09	638.66

The above provides figures to have real numbers associated with the different algorithms. However for a true comparison I will also include the normalised values for comparison where my algorithm is represented as 1.0.

	First Capable	First Fit	Best Fit	Worst Fit	Mine
Average turnaround time	0.0058	0.7955	0.7686	0.0732	1.0
Average resource utilisation	0.6747	1.0058	1.0486	0.9256	1.0
Average rental cost	1.6842	1.0007	1.0028	0.9603	1.0

Assessing the relative comparison table above, we can make a few observations. We first see that no baseline algorithm outperformed my own which has 80% of the turnaround time of First Fit. Continuing the comparison with First Fit, we have a near identical resource utilisation and rental cost. This means that for the same resources and cost, more work can be performed through scheduling efficiency gains alone. By contrast, the worst algorithm is the First Capable algorithm, achieving a relative performance of 0.0058, a very undesirable outcome.

6 Conclusion

This report has covered the problem posed by stage two, and how my solutions performance is to be measured. It has also gone over the refactoring as preparation for stage two, and discussed the ideas behind my algorithm. Additionally, a simple example was given to demonstrate how it works. Then from the evaluation using the test suite we can see that my solution has outperformed the baseline algorithms, while maintaining the same cost and resource requirements/utilisation as First Fit. This means that any First Fit scheduling done can be instead use my scheduling algorithm to improve the amount of work done without increasing the cost and resource utilisation.

My client Github link
ds-sim Github link

References

[1] "ds-sim: an open-source and language-independent distributed systems simulator User Guide"
https://github.com/distsys-MQ/ds-sim/blob/master/docs/ds-sim_user-guide.pdf

[2] "Socket (Java Platform SE 8)"
https://docs.oracle.com/javase/8/docs/api/java/net/Socket.html

[3] "InputStreamReader (Java Platform SE 8)"
https://docs.oracle.com/javase/8/docs/api/java/io/InputStreamReader.html

[4] "BufferedReader (Java Platform SE 8)"
https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html

[5] "DataOutputStream (Java Platform SE 8)"
https://docs.oracle.com/javase/8/docs/api/java/io/DataOutputStream.html

[6] "ArrayList (Java Platform SE 8)"
https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html