Resource Utilization and Cost Reduction in Distributed Systems

Andrew Campbell – 45638292

Introduction

The aim of this project is to create a client for the ds-sim server that takes a job and schedules it to a server based on different factors that will allow it to reduce the overall cost and improve the resource utilization of the servers being used.

This stage of the project focuses on taking one of the pre-defined algorithms (first-fit, best-fit or worst-fit) and improving on it to achieve at least one of the following:

* Minimisation of average turnaround time
* Maximisation of average resource utilisation
* Minimisation of total server rental cost

In this instance, the best-fit algorithm is being improved upon to minimize turnaround time and minimize server rental cost.

Problem Definition

The best-fit algorithm makes use of the fitness value calculated using the CPU cores required vs the CPU cores available from a server. Although this method is effective in finding a server that would leave the least amount of CPU cores unused, when choosing a server, it does not take into account the ability to run multiple jobs simultaneously. To improve on the best-fit algorithm, my objective function will look at not only the fitness value of the available servers, but will also look to see if any current jobs are being completed by those servers, giving higher priority to servers that already have jobs running that are capable of running the current job simultaneously.

Algorithm Description

The algorithm takes into account the available servers for the job using the “GETS Avail” command and calculates and compares the fitness values of each of the servers. Once the lowest fitness value is found, all the servers that returned the lowest fitness value are compared to see how many jobs they are running (using the “CNTJ” command). The server with the most running jobs is selected and if there are multiple with the same amount of running jobs, the first selected server is chosen. Below are two examples of how the algorithm would function in two different scenarios.

Example 1:

For this example, imagine you have 3 servers as follows:

Server-A: A server with 2 CPU Core

Server-B: A server with 2 CPU Cores

Server-C: A server with 3 CPU Cores

Now imagine you have the following jobs:

Job-A: Requires 1 CPU Core

Job-B: Requires 2 CPU Cores

Job-C: Requires 2 CPU Cores

Job-D: Requires 1 CPU Cores

Job-A is sent to the client and the client calculates and compares the fitness scores of the servers (finds the server(s) with the least remaining CPU cores if the job was to be run on them) and schedules the job for Server-A. The client is then sent Job-B and calculates that it is to be sent to Server-B as Server-A does not currently have enough CPU Cores available and Server-B’s fitness score is better than Server-C’s. The client then schedules Job-B to Server-B. Job-C is then sent to the client which only receives Server-C as available (has enough CPU Cores available) and hence schedules Job-C to Server-C. Job-D is then sent to the client which then compares the fitness values of Server-A and Server-C (as they are the only 2 servers with 1 CPU Core available) and finds that they are both the same. The client then schedules the job to Server-A as it is found before Server-C.

Example 2:

For this example, imagine you have 3 servers as follows:

Server-A: A server with 1 CPU Core

Server-B: A server with 2 CPU Cores

Now imagine you have the following jobs:

Job-A: Requires 1 CPU Core

Job-B: Requires 1 CPU Core

Job-C: Requires 1 CPU Core

Job-A is sent to the client which then requests the available servers and calculates and compares the fitness values of the servers and schedules Job-A to Server-A. Job-B is then sent to the client and is scheduled to Server-B (as Server-A has no available CPU Cores). Job-A completes and the CPU Core on Server-A becomes available again. Job-C is sent to the client and the client calculates and compares the fitness values of the servers. It finds that they have the same fitness value and requests and compares their running jobs and finds that Server-B has a running job while Sever-A does not and schedules Job-C to Server-B.

Implementation details

The program has been split into five main classes that hold their own functions. The classes are split up as per the following; client.java, bestFitFinder.java, handshaker.java, msgReader.java, msgWriter.java. The main function is implemented in the “client.java” file and is used to call functions from other classes to allow interaction and scheduling with the server. The main function makes use of the handshake function found in the handshaker class to make the handshake with the server at the start of the interaction between the client and the server. The main method then uses the msgRead and msgSend functions found in the msgReader and msgWriter classes respectively to talk to the server, depending on what the server has sent through. The main function requests a job then requests the available servers and chooses the best server out of the available servers by passing the CPU Cores required and the available servers to the bestFit function in the bestFitFinder class. The bestFit function compares the available servers requests to see their waiting jobs from the server by using the msgRead and msgSend functions. The bestFit function then returns the best server to be scheduled by the main function.

Evaluation

Conclusion

In conclusion, whilst working towards a solution to maximize the resource utilization of my client when compared to the Best Fit algorithm, I was also able to minimize the running cost of the servers. This was able to be done by the implementation of my client of which its main priority was to make as much use of all CPU Cores available in any given server at any given time.

References