Cloud Job Scheduler for Optimised Performance

Huu Cuong Nguyen (44491158)

Introduction

This project is to introduce the basic concept of a distributed system, also known as distributed computing. This system has multiple components placed in several machines such as computers or any devices that have local memory and are able to process multitasking. Those components communicate and interact with each other via a network to achieve designed tasks. A Distributed System is also able to scale both horizontally and vertically to meet a huge number of arising demands on the explosion of technology nowadays such as Google Data Centres, AWS clouds, etc.

In the approach to the Distributed System, the main focus of this Stage 2 is on designing and implementing a new scheduling algorithm to optimise performance in terms of minimised average turnaround time, maximised resource utilisation, and minimised total rental cost. More specifically, we are about to devise a simple system that is able to dispatch jobs to servers which, in this case, are the ones that can achieve one or more of the following: maximised resource utilisation, minimised average turnaround time, and minimised total rental cost. This report will take into account Problem Definition, Algorithm Description, Implementation, Evaluation and Conclusion to accomplish the objective.

Problem Definition

We already worked out a system that can schedule jobs to the largest server, which is the one with the biggest number of cores, in Stage 1. However, that does not guarantee we will have a system that is capable of attaining either minimised average turnaround time, maximised resource utilisation, or minimised total rental cost, because a largest server could have a huge total rental cost and high turnaround time, and also using just 1 type of server does not lead to resource being utilised wisely and properly (not 100% number of servers are used, which means the resource utilisation will not be maximised). Thus, performance might not be optimised in such a system.

In this stage, however, we can hardly design a system that is able to achieve all 3 of these goals: minimised average turnaround time, maximised resource utilisation, and minimised total rental cost, since by minimising waiting time (turnaround time), we may end up using more resources leading to high costs; or by maximising resource utilisation, we may have a system delivering high turnaround time. Therefore, we will need to decide on possibly one or two of these three objectives: minimised average turnaround time, maximised resource utilisation, minimised total rental cost. This report will focus on getting the resource utilisation as high as possible (maximised resource utilisation) as wasting resources is said to have a substantially negative impact on the operation of a business, entity or system. In the meantime, when we maximise the resource utilisation, we may have a high chance of reducing server rental costs, leading to potentially minimised total rental costs. That means we can achieve 2 objectives, which are maximised resource utilisation and minimised total rental cost, with the algorithm stated below in this report.

Algorithm Description

This kind of algorithm takes into account a list of servers capable of handling the current passed job from the server side and chooses the smallest one, which is the one with the least number of cores, and then schedule the specific job to this server. After that, it will again choose the smallest server (possibly the previous one or the new one with the least number of cores) from the lists of capable servers obtained for the next job just for the purpose of utilising all the resources that a type of server can provide. Then, we just continue the above steps until all jobs have been scheduled to the appropriate servers. The above description is a general picture of the functionalities of each simulator, this will be concentrated more in the Implementation part.

To have a more detailed picture of what looks like when we run the ds-server with given configurations, we take a closer look at the example results when running ds-server (‘./ds-server -c [configuration file] -v all’) and our client (comprising new scheduling algorithm) with the sample configuration file 1 as below:

# 0 juju servers used with a utilisation of 0.00 at the cost of $0.00

# 1 joon servers used with a utilisation of 100.00 at the cost of $0.58

# 0 super-silk servers used with a utilisation of 0.00 at the cost of $0.00

# actual simulation end time: 5295, #jobs: 10 (failed 0 times)

# total servers used: 1, avg util: 100.00% (ef. usage: 100.00%), total cost: $0.58

# avg waiting time: 2204, avg exec time: 728, avg turnaround time: 2932

We can see that for this configuration file, we have used just 1 server, which is a server of type ‘joon’ (since the smallest one among the lists of capable servers is of this type), and scheduled all 10 jobs to this server in order to use up all resources, which this smallest server can provide, as best as possible. In this way, we can maximise resource utilisation at 100% with a relatively low and seemingly affordable total cost of $0.58. However, as a trade-off with these 2 benefits, this algorithm results in a high turnaround time at 2932.

With this new scheduling algorithm, we can be confident in achieving maximised resource utilisation as it would use up all the resources that a type of server could provide, in terms of core count. If we use various servers for handling jobs (possibly a different one for each job), then we cannot utilise all the resources that a server can provide as with the resource capacity of a particular server, it can handle a large number of jobs, not just one. Therefore, if we just schedule one job to each server, then the left amount of resources of that server could not be used wisely.

Implementation

For the implementation, first of all, the network, input and output data streams libraries need to be imported for the connection between client and server, and for reading outputs and writing inputs from and to the server respectively. Socket, data input and output streams declarations and initializations are a must in this type of communication as the socket is one endpoint of a two-way communication link between two programs, which, in this case, are client and server, running on the network. The socket mechanism provides a means of inter-process communication by setting up named contact points between which the communication occurs. Basically, the server creates a socket, attaches to it a network port and IP address, then waits for the client to contact it. The client then creates a socket and attempts to connect to the server socket at the created port and IP address. When the connection is established, transfer of data begins, where data input and output streams are required to process the data exchange between them.

What’s next, the client says “HELO” to the server to confirm if the server is actually listening to it to start the data exchange. If what the server received from the client is indeed a valid message, the server will confirm “OK” for the client to be sure that it can start on the authorization and transfer of data. Otherwise, the server will send an “ERR” message to the client to indicate that the message is invalid or in the wrong format. After the authorization is done with the reply as “OK” from the server, now it is ready to schedule jobs (if any) by processing through the loops until there are no ones left.

The loop now starts: If what the client received from the server is indeed “NONE” or “QUIT”, indicating that there are no jobs left or the server wants to terminate the connection (communication) respectively, the client simply sends “QUIT” to it to stop the communication and close all connections and socket.

Otherwise, if it is not actually a real job (starting with “JOBN”) that needs to be scheduled, it simply sends “REDY” to the server to just say that “it is not actually a job that needs to be scheduled, just give me the next one, and I am ready for it”. Then, the server will send the next piece of information (probably a job or not) and the client will continue processing this piece of information from the beginning of the loop.

If it is now indeed a job with information like submit time, job id, estimated run time, core, memory, disk, then the client knows that it will have to schedule the job with its job id (each job has a specific and different job id). To start with, the client “GETS Capable” from the server to make requests for server state information based on initial resource capacity. That is, the client requests a list of servers that are capable of handling the job. What the client receives in response to “GETS Capable” is the data with a number of records (servers) and the maximum length of one record in bytes including spaces between fields. To really get a full list of records and their information, the client will need to read in the outputs from the server in (number of records x maximum length of one record) bytes. Now, a full list of records (servers) with their information is obtained. This list of servers is an array of strings separated by new line characters. In this list, each server is another array of strings separated by spaces with information like server name, limit, bootup time, hourly rate, core count, memory, disk. We will process through each server to get the name of the smallest server, which is the one with the lowest core count located at the string index 4 of each record. After attaining the smallest server name, the client will need to dispatch job to this server with its id by calling the command SCHD along with the job id and the server with its id. After the job has been scheduled, the server will return “OK” as a confirmation of the successful scheduling. Then, the client is ready for the next job by telling the server that it is “REDY”. Now, it will return to the beginning of the loop and process another piece of information.

After all the jobs have been scheduled, the client and server end the communication by sending each other “QUIT” messages and then close all the connections and socket. The communication process ends here.

Evaluation

Below is a table that compares this algorithm with the other three baseline algorithms: First-fit (FF), Best-fit (BF), Worst-fit (WF) (run with ‘./ds-server -c [configuration file] -v all’):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Configuration files | Features | This algorithm | FF | BF | WF |
| Configuration file 1 | - Average Resource Utilisation  - Total Server Rental Cost  - Average Turnaround Time | - 100%  - 0.58  - 2932 | - 100%  - 0.75  - 873 | - 100%  - 0.75  - 873 | - 100%  - 0.60  - 754 |
| Configuration file 2 | - Average Resource Utilisation  - Total Server Rental Cost  - Average Turnaround Time | - 100%  - 437.40  - 377673 | - 76.24%  - 601.60  - 6328 | - 75.67%  - 559.33  - 6327 | - 91.38%  - 690.71  - 10255 |
| Configuration file 3 | - Average Resource Utilisation  - Total Server Rental Cost  - Average Turnaround Time | - 100%  - 1054.96  - 661335 | - 83.51%  - 1148.18  - 6185 | - 80.29%  - 1159.27  - 6185 | - 81.93%  - 1117.35  - 17515 |
| Configuration file 4 | - Average Resource Utilisation  - Total Server Rental Cost  - Average Turnaround Time | - 100%  - 691250.69  - 28863782 | - 92.88%  - 680058.12  - 11409075 | - 91.95%  - 680509.38  - 11578125 | - 99.43%  - 715831.38  - 12434272 |
| Configuration file 5 | - Average Resource Utilisation  - Total Server Rental Cost  - Average Turnaround Time | - 100%  - 263.13  - 218529 | - 63.26%  - 489.38  - 4784 | - 63.31%  - 447.60  - 4783 | - 94.67%  - 888.62  - 4780 |

REFERENCES

https://www.javatpoint.com/socket-programming

<https://www.geeksforgeeks.org/socket-in-computer-network/>

Project Git Repository

<https://github.com/DacNguyen234/Comp3100_project>