COMP9334 Project Report
Server setup in data centres

Cheng Zhang, z5088908

1. Introduction

In this project, I implemented the simulation of server setup in data centres with C++. The program consists of the following files:

`Makefile` : run `make` will generate three executables, `wrapper, design, verify`
`simulation.cpp` : consists of detailed implementation of setup/delayed off system.
`simulation.cpp` : declarations of class and functions.
`wrapper.cpp` : wrapper program which runs all test cases.
`design.cpp` : generate random cases to determine optimum value of delayed-off time.
`verify.cpp` : generate probability distributions to verify the correctness.
`Report.pdf` : The report of the project.

2. Overview

In my implementation, there are mainly three parts: simulation system, wrapper program and design program.

Simulation System:

The simulations system is implemented in `simulation.cpp,simulation.hpp` it mainly consists of three C++ class: dispatcher, server, job; and the function `simulate(...)` which will be invoked by other functions. The system will work under a global master clock in the simulate function, so that the state of system can change.
The three C++ class contains of:

- The dispatcher manages the selection of servers and update of the system states, it consists of a job queue and references to each server, so that it will assign jobs to specific servers.
- The servers processes jobs, it will switch states depending on the current master clock, so that it is able to record the arrival time and departure time of the job.
- The jobs simple records its arrival and departure time, it also store information about mark, and which server it powers on.

The simulation function has two modes depending on the input:

- Trace mode: the program will set the master clock at 0, then initialize the dispatcher which also construct the servers. The update of master clock will depend on the critical time points, which will be updated during execution. It is a global variable implemented as a vector, that can be accessed from all functions in the program. The master clock will be updated at critical time points, such as job arrival, departure, certain behaviour will happen when clock is updated.
- Random mode: Instead of reading arrival time and service time from the input, the random mode will generate the arrival time and service time in random, and feed them to the system. After generation these inputs, the reset parts is the same as trace mode.

After simulation, the reference of all jobs will be returned as a vector, as the jobs have already been processed, they will have information of arrival time and departure time, the response time of each job, and also the mean response time of all jobs can be calculated.

Wrapper program:

The wrapper program simply read data from `mode_*.txt, para_*.txt, arrival_*.txt, service_*.txt` and call the simulate function to process jobs. It will get the processed jobs as a vector after the simulation finishes. Then read arrival time and

departure from jobs to calculate response time and mean response time, the results will be written to `mrt_*.txt` and `departure_*.txt`.

Design program:
The design program is similar to the random mode of the simulations program, it will generate a vector of different delayed-off time, then generate random test cases of jobs with different arrival time and departure time.
Different from the Wrapper program, It is required that the results are reproducible, which I use 1 as seed. In this part, I generated random numbers that is not reproducible, because we want a mean average value of response time, a reproducible sequence will always generate the same sequence under the same condition.
So, I added a bool argument to determine whether we want a reproducible random result. I use mt19937 random number generator in the design program. The pseudo-random number generator will have a period of$2^{19937} - 1$, which will produce better results.
The program will plot the histogram in OpenGL, so that I can compare the performance of different delayed-off value. Each test case will be run 5000 times to ensure a convergence in mean average value.

3. Verification of probability distribution

The verification of probability distribution is in `verify.cpp`, it will use the same function as in simulation to generate the distribution and plot the histogram.

Distribution of arrival time:
As the mean arrival rate of the jobs is exponential distribution with parameter λ, the inter-arrival time can be calculated by $T = \frac{1}{arrivalrate}$.
The histogram of the probability distribution is shown as Figure 1.

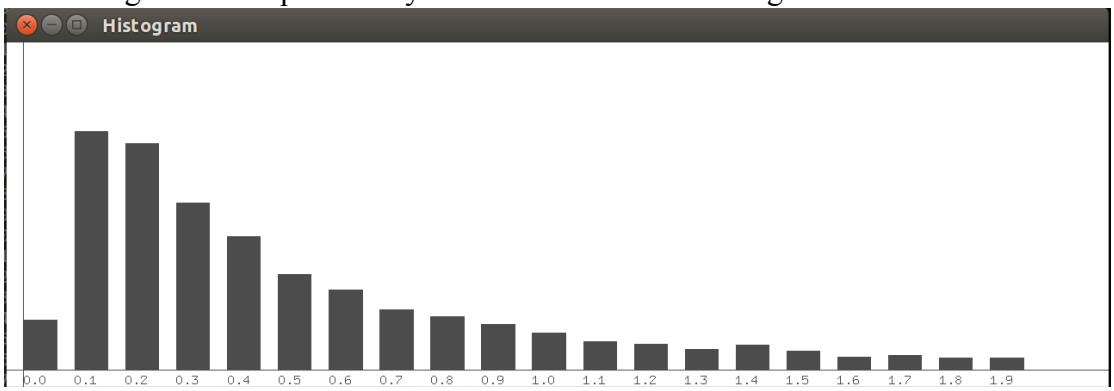

Figure 1. histogram of the probability distribution of arrival time.

Distribution of service time:
The distribution of the service time can be calculated by the following formula:

$$s_k = s_{1k} + s_{2k} + s_{3k}, \forall k = 1,2, \dots$$

Where s_{1k},s_{2k},s_{3k} are exponentially distributed random numbers with parameter μ.

The histogram of the probability distribution is shown as Figure 2.

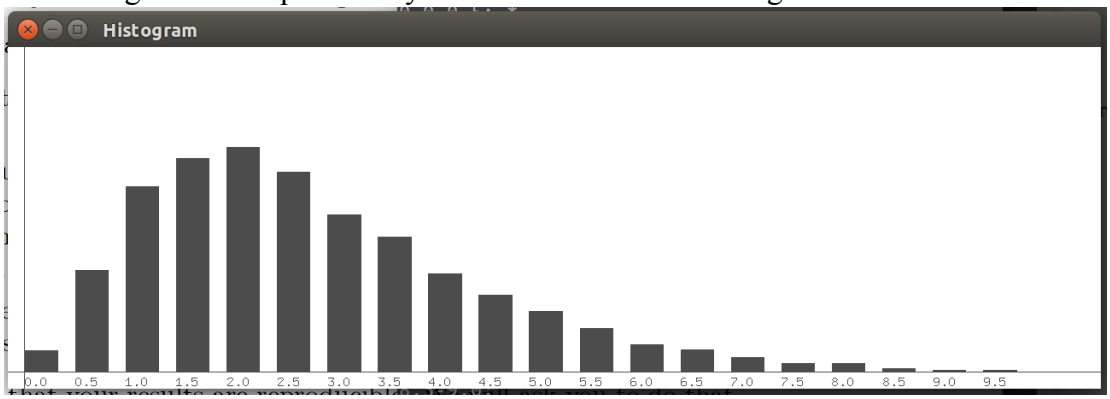

Figure 2. histogram of the probability distribution of arrival time.

Reproducible result:
As the seed of mt19937 random number is 1, it will always produce the same random number sequence, thus the result is reproducible.

4. **Verification of system simulation correctness**
The verification of system simulation correctness is done in two stages:
Comparison with sample test case:
During the early stage of the development of the program, I use simple test cases to verify my correctness of the system.
I print all the information of system state change at critical time point, and verify it with the sample case. It can be done by uncommenting the line `#define DEBUG` in `simulation.hpp`, it will print all the information in regards to job queue, server state and job states.
In this stage, the verification is mainly based on logical correctness, the test is not extensive, and some rare cases that may cause problem was not tested in this part.

Random case test:
After the system is able to correctly run all the basic sample tests, I start to generate several thousands of random test cases to do a auto-test system verification. This part of test ensures the correctness and robustness of the simulation system.

First, I use very strict assert condition in many sections of code based on the logic and requirement of the program. Then I generate random test cases, these cases will trigger error in assertion, then I can analyse this particular case and modify the system so that it can deal with many complex situations that may not appear in the specs. Some of these cases includes:

1. There are two servers both in delayed-off mode and with same delayed-off count down value, it is caused by one server finishes a job, and the other starts at the same time. The dispatcher will choose the first job in the queue.

2. When a server starts to process a marked job, and no other unmarked job in the queue, and at the same time the server correspond to this marked job starts and change its state from SETUP to DELAYEDOFF, which cannot been turned off immediately, the job will go to either server depending on the priority.
These cases are not descripted in the specs, but happened in my random case test. I tried to make the system do reasonable behaviour in these situations.

In the selection of server, I introduced priority to solve this. The pointers of servers are in a vector, they are sorted first by their modes, DELAYEDOFF > OFF > SETUP > BUSY, then if there are two or more servers in DELAYEDOFF mode, they will be sorted by the delayed-off count down values. The dispatcher will choose server at the front of the vector.

5. Design

The design problem is implemented in a separated file: design.cpp, the program will run simulation with different delayed-off time for serval times, then get the mean response time of the system at a particular delayed-off time.

I generated the delayed-off time in logarithmic scale from 0.1, which is: 0.1, 0.2, 0.4, 1.6, 3.2 … , it makes the program to find the optimum value faster. The setting of delayed-off time is related to the value of time_end.
As time_end requires that all job leave the system before this value, It requires us to estimate the departure time during random generation, and stop generation of test cases at a proper time, so that all job will leave the system before or at the time_end.
It means if we set the same time_end for all test cases, once the delayed-off time become larger than time_end, the server will not turn off, so continue increasing delayed-off time will be meaningless. As shown in Figure 1. with actual value in Table 1.

Time_end = 100 :

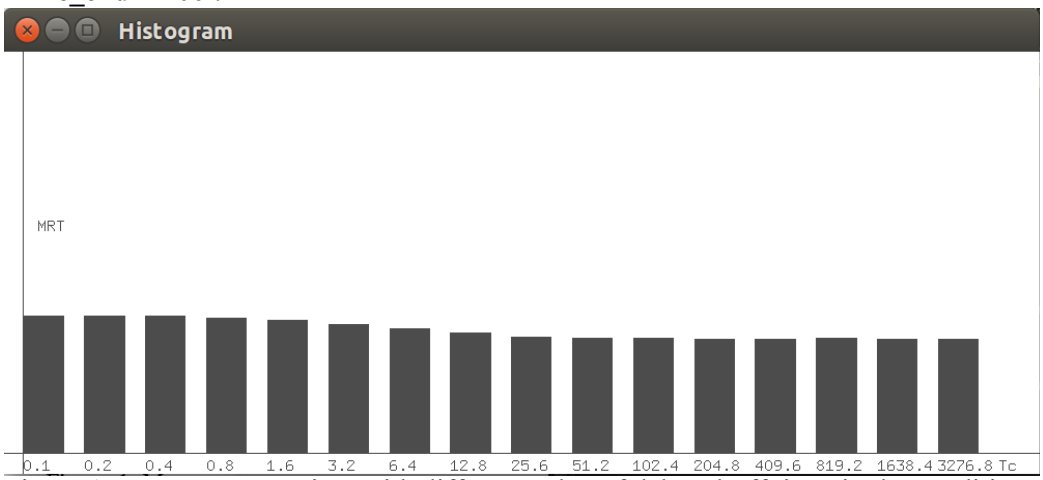

Figure 1. Mean response time with different value of delayed-off time, in the condition $time_{end} = 100$

Tc	0.1	0.2	0.4	0.8	1.6	3.2	6.4	12.8
MRT	6.88	6.89	6.86	6.74	6.67	6.44	6.23	6.02
Tc	25.6	51.2	102.4	204.8	409.6	819.2	1638.4	3276.8
MRT	5.81	5.77	5.74	5.71	5.74	5.75	5.71	5.69

Table 1. value of Mean response time with delayed-off time in Figure 1

As we can see, the mean response time will drop when we increase delayed-off time, but up to a point, the value of mean response time will stop further reduce, even if we have a very large delayed-off time. It meets our previous assumption.
From the histogram in Figure 1, the optimal delayed-off time value in the case of time_end = 100 is around 20 to 50.

By varying the value of time_end, we can get the result under different condition. I tested more cases with time_end = 200, 500, 1000 in the following.

For each combination of time_end and delayedoff time, the simulation will run 5000 times and calculate the mean average of mean response time, to ensure it converges to the expected value.

Time_end = 200 :

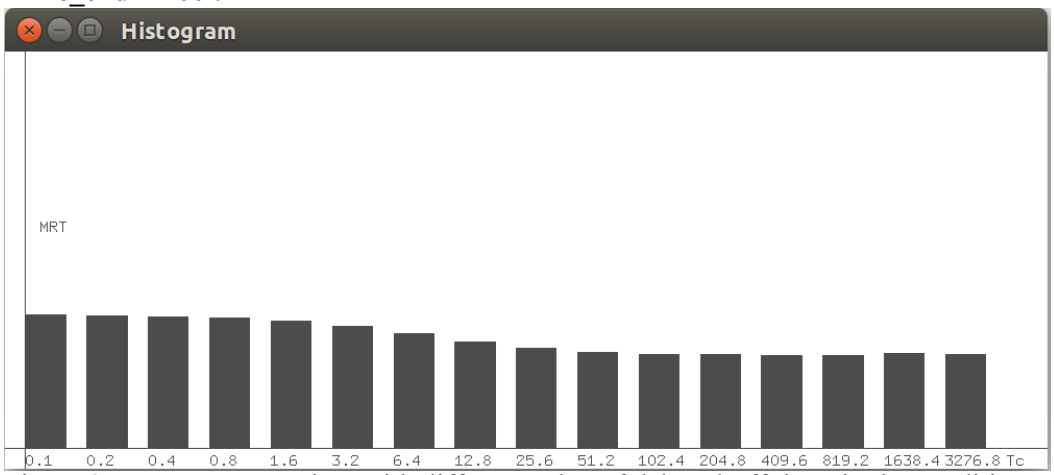

Figure 2. Mean response time with different value of delayed-off time, in the condition $time_{end} = 200$

Tc	0.1	0.2	0.4	0.8	1.6	3.2	6.4	12.8
MRT	6.75	6.71	6.66	6.59	6.46	6.18	5.81	5.43
Tc	25.6	51.2	102.4	204.8	409.6	819.2	1638.4	3276.8
MRT	5.07	4.86	4.79	4.79	4.72	4.72	4.80	4.76

Table 2. value of Mean response time with delayed-off time in Figure 2

From Figure 2, the optimum value of delayed-off time in this case is around 50

Time_end = 500 :

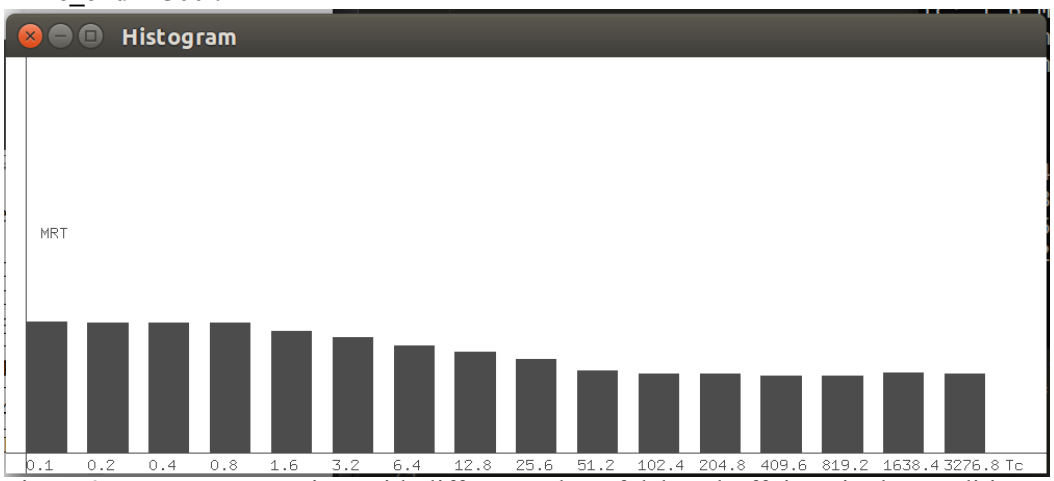

Figure 3. Mean response time with different value of delayed-off time, in the condition $time_{end} = 500$

Tc	0.1	0.2	0.4	0.8	1.6	3.2	6.4	12.8
MRT	6.68	6.61	6.63	6.58	6.19	5.88	5.47	5.13
Tc	25.6	51.2	102.4	204.8	409.6	819.2	1638.4	3276.8
MRT	4.77	4.19	4.02	4.02	3.93	3.93	4.09	4.01

Table 3. value of Mean response time with delayed-off time in Figure 3

From Figure 3, the optimum value of delayed-off time in this case is around 50

Time_end = 1000 :

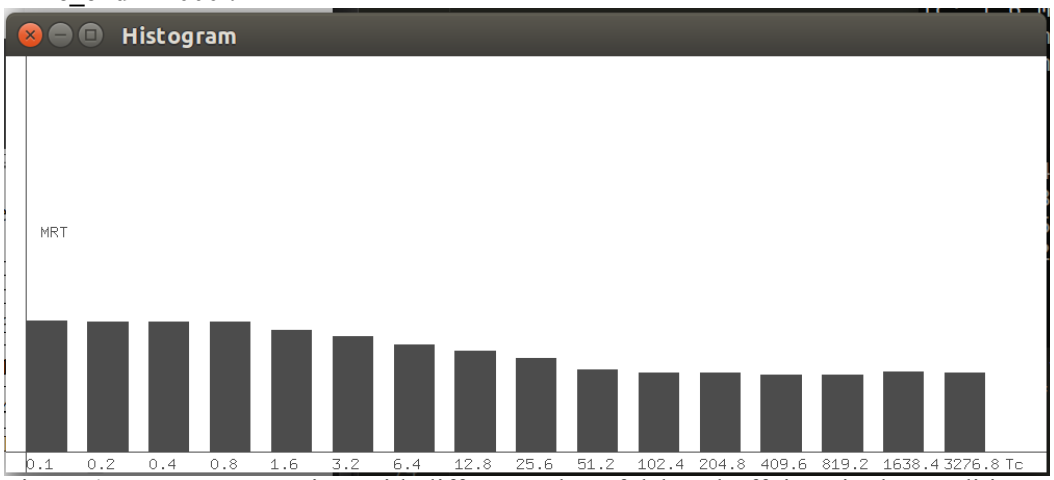

Figure 4. Mean response time with different value of delayed-off time, in the condition $time_{end} = 1000$

Tc	0.1	0.2	0.4	0.8	1.6	3.2	6.4	12.8
MRT	6.56	6.62	6.56	6.46	6.10	5.86	5.41	5.00
Tc	25.6	51.2	102.4	204.8	409.6	819.2	1638.4	3276.8
MRT	4.38	4.05	3.81	3.66	3.70	3.67	3.71	3.85

Table 4. value of Mean response time with delayed-off time in Figure 4

From Figure 4, the optimum value of delayed-off time in this case is around 50

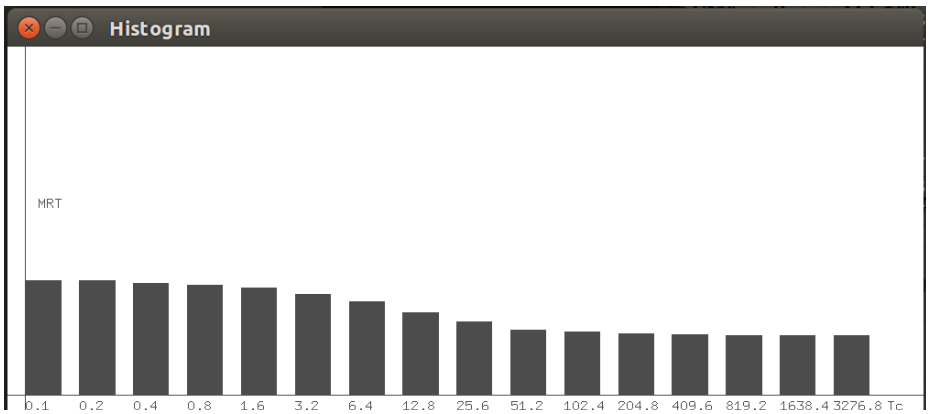

Figure 5. Mean response time with different value of delayed-off time, in the condition $time_{end} = 5000$

Tc	0.1	0.2	0.4	0.8	1.6	3.2	6.4	12.8
MRT	6.60	6.59	6.43	6.36	6.20	5.82	5.38	4.77
Tc	25.6	51.2	102.4	204.8	409.6	819.2	1638.4	3276.8
MRT	4.26	3.78	3.66	3.54	3.52	3.46	3.44	3.43

Table 5. value of Mean response time with delayed-off time in Figure 5

From Figure 5, the optimum value of delayed-off time in this case is around 50

6. Conclusion

In conclusion, the optimum value of delayed off time is related to the time_end when the time_end value is small, but when the test run for long enough, the optimum value of delayed-off time is about 50.