COMP9334 Project, Term 1, 2021: Heterogeneous server farms

Due Date: 5:00pm Friday 23 April 2021.

Version 1.02, 12 April 2021

Updates to the project, including any corrections and clarifications, will be posted on the subject website. Make sure that you check the course website regularly for updates.

Change log

Version 1.01. Typo in Section 5.1.1 corrected. The correction is in red.
Version 1.02. Required accuracy of the optimal d is added in Section 5.2. The additional is in red.

1 Introduction and learning objectives

Modern day server farms typically consist of servers with heterogeneous hardware specifications. This is due to incremental deployment where the servers are purchased at different times. A research question is how to distribute jobs to the servers with different processing rates so that the computational resources can be effectively utilised. In this project, you will use simulation to try to answer such a research question.

In this project, you will learn:

1. To use discrete event simulation to simulate a computer system
2. To use simulation to solve a design problem
3. To use statistically sound methods to analyse simulation outputs

2 Support provided and computing resources

If you have problems doing this assignment, you can post your question on the course forum. **We strongly encourage you to do this as asking questions and trying to answer them is a great way to learn. Do not be afraid that your question may appear to be silly, the other students may very well have the same question!**

If you need computing resources to run your simulation program, you can do it on the VLAB remote computing facility provided by the School. Information on VLAB is available here: `https://taggi.cse.unsw.edu.au/Vlab/`

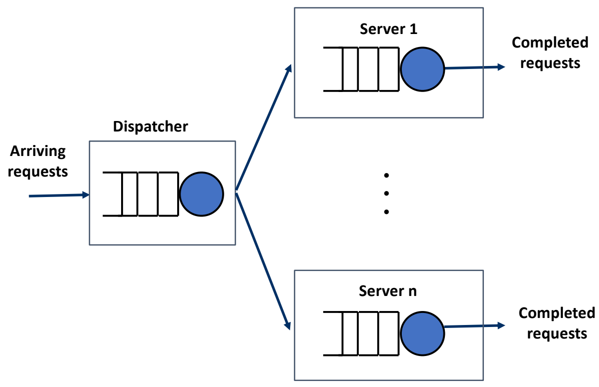

Figure 1: A typical architecture server farm.

3 Load balancing for server farms

3.1 Background

A server farm consists of hundreds (or even thousands) of servers. A typical architecture for server farms is shown in Fig. 1 which has a dispatcher at its front end and servers at its back end. The role of the dispatcher is to choose a server to process an incoming request. It is undesirable to overload one server and leave the other servers idle so a dispatcher tries to spread the requests among the servers evenly. This is known as load balancing [1, Chapter 24].

Many load balancing algorithms have been proposed in the past. A simple algorithm is the round robin (RR) algorithm. Assuming that there are n servers, RR works as follows: send the first request to Server 1, second request to Server 2, ..., n-*th* request to Server n, $(n+1)$-*st* request to Server 1, ..., $2n$-*th* request to Server n and so on. An advantage of RR is that the dispatcher does not need to communicate with the servers to find out what their current load is. However, this advantage is also a disadvantage of RR because the dispatcher does not know how loaded the servers are. So, it is possible that the RR algorithm may send a request to a server which is busy while there are idling servers in the server farm.

A load balancing algorithm that takes the current server load into consideration is the *shortest queue* algorithm. When the dispatcher receives a request, it will query each server to obtain the number of jobs in its queue. After receiving the queue length information from all the servers, the dispatcher chooses the server with the shortest queue length and sends the request to that server.

An assumption behind the shortest queue algorithm is that all servers have the same processing rate. However, typical server farms nowadays consist of servers with heterogeneous processing rates. This is because server farms are incrementally expanded so the hardware are purchased at different times. Therefore, a research problem is to study how load balancing algorithm can be modified to deal with heterogeneous processing rates. This is the subject of investigation of this project.

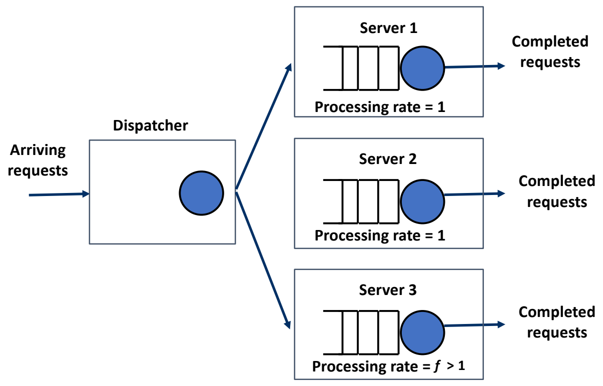

Figure 2: The server farm for this project.

As a remark, we want to point out that another key problem faced by the shortest queue algorithm is the large number of servers in a modern day server farm. The shortest queue algorithm assumes that the dispatcher can obtain the queue length in each server in the farm. However, this is not practical for a farm with a large number of servers. An idea is to query only a fraction of the servers. You can read about this in [2] but we will not be considering this issue in this project.

3.2 Server farm configuration for this project

The configuration of the server farm that you will use in this project is shown in Fig. 2. The farm consists of a dispatcher and 3 servers.

The servers are labelled as Servers 1, 2 and 3 as in Fig. 2. Both Servers 1 and 2 have a processing rate of 1. Server 3 is a faster server which has a processing rate of $f > 1$. The jobs in the servers are processed on a first-come first-serve basis. Each server has a queue and you can assume the number of waiting rooms is unlimited. Note that when we talk about the number of jobs in a server, we are referring to the number of jobs being served plus the number of jobs in the queue.

Upon the arrival of a request, the dispatcher will query the servers for the number of jobs currently in the server. After the dispatcher has received the information from the servers, it will use a load balancing algorithm to choose a server and then send the request to the chosen server. We assume that the time taken to query the servers, receive the replies and execute the load balancing algorithm is negligible. This means that you can assume that the dispatcher has the status of the servers at the arrival time of the requests.

You will use simulation to study the performance of two load balancing algorithms in this project. The rationale of the algorithm is to prioritise the use of an idle server over busy servers and also to prioritise the use of the faster server over the slower servers.

The algorithm assumes that the availability of n_i $(i = 1, 2, 3)$ where n_i is the number of jobs in Server i at the time a request arrives at the dispatcher. The load balancing algorithms make use of a parameter $d(\geq 0)$ which is use to prioritise the use of the faster server over slower servers.

The load balancing algorithm (Version 1) is:

Algorithm 1: The load balancing algorithm (Version 1)

```
   Data: $n_1$,$n_2$,$n_3$,$d$
1  Calculate: $n_s = \min(n_1, n_2)$;
2  if $n_3$ == 0 then
       // Server 3 is idle
3      Route the incoming request to Server 3;
4  else if $n_s == 0$ or $n_s \leq n_3 - d$ then
       // At least one slow server is idle or
       // at least one slow server has far fewer jobs than the fast server
       // Route to the slow server that has fewer jobs
       // If both slow servers have the same number of jobs, use Server 1
5      if $n_1$ == $n_s$ then
6          Route the incoming request to Server 1;
7      else
8          Route the incoming request to Server 2;
9      end
10 else
       // Choose the fast server otherwise
11     Route the incoming request to Server 3;
12 end
```

Let us explain how the load balancing algorithm (Version 1) makes its decision. The key decision making is done in a conditional statement which consists of a *if*-statement in Line 2, a *else*-*if*-statement in Line 4 and an *else*-statement in Line 10.

- In Line 2, the algorithm checks if Server 3 (the faster server) is idle. If this is true, the request will be sent to Server 3.
- In Line 4, the algorithm checks if n_s is 0. If this is true, then this means at least one of the slow servers is idle, the request will be sent to an idling slow server. (We will explain Lines 5-9 later.)
- Line 4 also tests the condition $n_s \leq n_3 - d$ where $d \geq 0$. This condition checks if there is a slow server which has at least d fewer jobs than that in the fast server. If this is true, it means there is a slow server with far fewer jobs and this server will be chosen. (We will explain Lines 5-9 later.)
- The "otherwise" condition in Line 10 is to choose the fast server.

If Line 4 is true, then the algorithm will choose a slow server whose number of jobs is smaller, i.e. equals to n_s. There are three possible cases: (i) Both n_1 and n_2 equal to n_s; (ii) Only n_1 is n_s; (iii) Only n_2 is n_s. For Case (i), either Server 1 or Server 2 can be chosen, and the algorithm chooses Server 1. The explanation for Lines 5-9 is: If Case (i) or Case (ii) is true, then choose Server 1; otherwise, choose Server 2. We would like to remark that most research papers suggest that if Case (i) holds, one can choose a server randomly. We have **not** chosen to do this because it will be difficult for us to verify your simulation.

The load balancing algorithm (Version 1) makes the comparison $n_s \leq n_3 - d$ in Line 4, however, the queue lengths n_s and n_3 may not be directly comparable because the slow and fast servers have different processing rates. The load balancing algorithm (Version 2) will make use of the processing rate f of the faster server. The only difference between Versions 1 and 2 is in Line 4.

Algorithm 2: The load balancing algorithm (Version 2)

```
   Data: $n_1$,$n_2$,$n_3$,$d$,$f$
1  Calculate: $n_s = \min(n_1, n_2)$;
2  if $n_3$ == 0 then
       // Server 3 is idle
3      Route the incoming request to Server 3;
4  else if $n_s == 0$ or $n_s \leq \frac{n_3}{f} - d$ then
       // At least one slow server is idle or
       // at least one slow server has far fewer jobs than the fast server
       // Route to the slow server that has fewer jobs
       // If both slow servers have the same number of jobs, use Server 1
5      if $n_1$ == $n_s$ then
6          Route the incoming request to Server 1;
7      else
8          Route the incoming request to Server 2;
9      end
10 else
       // Choose the fast server for otherwise
11     Route the incoming request to Server 3;
12 end
```

4 Examples

We will now present two examples to illustrate the operation of the server farm that you will simulate in this project. In all these examples, we assume that the system is initially empty.

4.1 Example 1: Load balancing (Version 1)

In this example, we assume the version 1 algorithm will be used. There are 8 requests. The arrival times and service times required at the slow server are given in Table 1. The processing rate f of the faster server is 2. The value of d is 1.

Arrival time	Service time needed at the slow server
1	24
2	7
3	8
5	6
6	8
7	4
8	6
10	2

Table 1: Data for Example 1.

Note that if a job is sent to the faster server, you can assume that the service time needed at the fast server is given by the service time needed at the slow server divided by f.

The events in the server farm are the arrival of a job to the dispatcher and the departure of a completed job from a server. Since we assume that the dispatcher takes negligible time to make a decision, this means the arrival time of a job to the dispatcher is also the arrival time of the job to the server chosen by the dispatcher.

We will illustrate how the simulation of the server farm works using “on-paper simulation”. The quantities that you need to keep track of are:

- **Next arrival time** is the time of the next arrival
- For each server
    - Server status, which can be either busy or idle.
    - **Next departure time** is the time at which the job that is being processed will depart from the server. If the server is idle, the next departure time is ∞. Note that there is a next departure time for each server.
    - The contents of the buffer. Each job in the buffer is characterised by a 2-tuple. The first element of the 2-tuple is the arrival time of the job at the server and the second element is the amount of service needed by the job in that server. Note that each server has its own buffer.

The “on-paper simulation” is shown in Table 2. The notes in the last column explain what updates you need to do for each event.

Master clock	Event type	Next arrival time	Server 1	Server 2	Server 3	Notes
0	–	1	Idle, ∞, –	Idle, ∞, –	Idle, ∞, –	We assume the servers are empty at the start of the simulation. All servers are idle. The next departure times for all servers are ∞. The "–" indicates that the buffer is empty.
1	Arrival	2	Idle, ∞, –	Idle, ∞, –	Busy, 13, –	The event is an arrival. The dispatcher executes the load balancing algorithm (Version 1). The number of jobs in the server at the time that the request arrives is: $n_1 = n_2 = n_3 = 0$. According to the algorithm, the condition in Line 2 is satisfied, so the request will be send to Server 3. Therefore, we need to update the status of Server 3. In particular, note that the number 13 in the Server 3 column is the departure time of the job that is being served. This job requires a processing time of 12 in Server 3, which is the service time needed in the slow server (= 24 for this job) divided by f (which is 2). Since this job arrives at 1, its departure time is 1 + 12 = 13. Lastly, we need to update the next arrival time, which is 2.
2	Arrival	3	Busy, 9, –	Idle, ∞, –	Busy, 13, –	The event is an arrival. Based on $n_1 = n_2 = 0$ and $n_3 = 1$, we have $n_s = 0$. The condition in Line 4 is true. Since $n_1 = 0$ and $n_s = 0$, according to Line 5, the job is sent to Server 1. The status of Server 1 has been updated. The arrival time for the next job has also been updated.
3	Arrival	5	Busy, 9, –	Busy, 11, –	Busy, 13, –	The event is an arrival. Based on $n_1 = 1$, $n_2 = 0$ and $n_3 = 1$, we have $n_s = 0$. The condition in Line 4 is true. Since $n_1 = 1$ and $n_2 = 0$, the job is sent to Server 2. The status of Server 2 has been updated. The arrival time for the next job has also been updated.
5	Arrival	6	Busy, 9, –	Busy, 11, –	Busy, 13, (5,3)	The event is an arrival. Based on $n_1 = 1$, $n_2 = 1$ and $n_3 = 1$, we have $n_s = 1$. The condition in the else statement (Line 10) is true, so the job is sent to Server 3. Since Server 3 is busy, the job is stored in the buffer. The notation (5,3) means the job arrives at the server at 5 and the service time needed for this job is 3. Note that Table 1 says this job requires 6 units of time in the slow server and it has been sent to a fast server, so its service time needs to be adjusted accordingly.
6	Arrival	7	Busy, 9, (6,8)	Busy, 11, –	Busy, 13, (5,3)	The event is an arrival. Based on $n_1 = 1$, $n_2 = 1$ and $n_3 = 2$, we have $n_s = 1$. Since $n_3 - d = 1$, therefore the condition $n_s \leq n_3 - d$ is true. The job is sent to Server 1 because n_1 equals to n_s.

7	Arrival	8	Busy, 9, (6,8)	Busy, 11, (7,4)	Busy, 13, (5,3)	The event is an arrival. Based on $n_1 = 2$, $n_2 = 1$ and $n_3 = 2$, we have $n_s = 1$. Since $n_3 - d = 1$, therefore the condition $n_s \leq n_3 - d$ is true. The job is sent to Server 2.
8	Arrival	10	Busy, 9, (6,8)	Busy, 11, (7,4)	Busy,13, (5,3) (8,3)	The event is an arrival. Based on $n_1 = 2$, $n_2 = 2$ and $n_3 = 2$, we have $n_s = 2$. The condition for the else-statement (Line 7) is true, so the job is sent to Server 3.
9	Depart. Server 1	10	Busy, 17	Busy, 11, (7,4)	Busy,13, (5,3) (8,3)	The event is a departure from Server 1. Since there is a job in the queue in Server 1, that job starts to receive service and its next departure time is $9 + 8 = 17$.
10	Arrival	∞	Busy, 17, (10,2)	Busy, 11, (7,4)	Busy,13, (5,3) (8,3)	The event is an arrival. Based on $n_1 = 1$, $n_2 = 2$ and $n_3 = 3$, we have $n_s = 1$. Since $n_3 - d = 2$, therefore the condition $n_s \leq n_3 - d$ is true. The job is sent to Server 1. Since there are no more arrivals, the next arrival is set to ∞.
11	Depart. Server 2	∞	Busy, 17, (10,2)	Busy, 15, –	Busy,13, (5,3) (8,3)	The event is a departure from Server 2. Since there is a job in the queue in Server 2, that job starts to receive service and its next departure time is $11 + 4 = 15$.
13	Depart. Server 3	∞	Busy, 17, (10,2)	Busy, 15, –	Busy,16, (8,3)	The event is a departure from Server 3. Since there are jobs in the queue in Server 3, that first job starts to receive service and its next departure time is $13 + 3 = 16$.
15	Depart. Server 2	∞	Busy, 17, (10,2)	Idle, ∞, –	Busy,16, (8,3)	The event is a departure from Server 2. Since there are no jobs in the queue in Server 2, it becomes idle and the next departure time is ∞.
16	Depart. Server 3	∞	Busy, 17, (10,2)	Idle, ∞, –	Busy,19, –	The event is a departure from Server 3. The job at the top of the buffer starts to receive service.
17	Depart. Server 1	∞	Busy, 19, –	Idle, ∞, –	Busy,19, –	The event is a departure from Server 1. The job at the top of the buffer starts to receive service.
19	Depart. Server 1	∞	Idle, ∞, –	Idle, ∞, –	Busy,19, –	The event is a departure from Server 1.
19	Depart. Server 3	∞	Idle, ∞, –	Idle, ∞, –	Idle, ∞, –	The event is a departure from Server 3.

Table 2: Table illustrating the updates in a PS server.

Note that the last two jobs in the simulation in Table 2 finish at the same time, see the last two rows in Table 2. In this case, your simulation can handle these two events one after another in any order you like.

The above description has not explained what happens if an arrival and a departure are at the same time. We will leave it unspecified. If we ask you to simulate in trace driven mode, we will ensure that such situation will not occur. If the inter-arrival time and service time are generated randomly, the chance of this situation occurring is practically zero so you do not have to worry about it.

Arrival time	Departure time
1	13
2	9
3	11
5	16
6	17
7	15
8	19
10	19

Table 3: The arrival and departure times for Example 1.

Table 3 lists the arrival times and departure times of all the requests. You can compute the mean response time of the system by using these times. The mean response time is 9.6250 without considering transient removal.

4.2 Example 2: Load balancing (Version 2)

This example illustrates the simulation of the load balancing algorithm (Version 2) assuming $d = 0.5$ and $f = 2$. The arrival and service time needed in the slow server are given in Table 4 .

Arrival time	Service time needed in the slow server
1	12
2	3
3	2
9	10
10	8
11	4
12	6
13	2

Table 4: Arrival times and service times needed in the slow server for Example 2.

In the simulation, there are four possible events

1. Arrival
2. Departure from Server 1
3. Departure from Server 2
4. Departure from Server 3

Table 5 shows the event times in ascending order and the event that happens at that time. Note that if the event is an arrival, we also mention the server that has been chosen.

Table 6 shows the arrival and departure times for this example. The mean response time of the server farm is 4.75 without considering transient removal.

1	Arrival. (Chosen server = 3)
2	Arrival. (Chosen server = 1)
3	Arrival. (Chosen server = 2)
5	Departure from Server 1
5	Departure from Server 2
7	Departure from Server 3
9	Arrival. (Chosen server = 3)
10	Arrival. (Chosen server = 1)
11	Arrival. (Chosen server = 2)
12	Arrival. (Chosen server = 3)
13	Arrival. (Chosen server = 3)
14	Departure from Server 3
15	Departure from Server 2
17	Departure from Server 3
18	Departure from Server 1
18	Departure from Server 3

Table 5: The event times for Example 2.

Arrival time	Departure time
1	7
2	5
3	5
9	14
10	18
11	15
12	17
13	18

Table 6: The arrival and departure times for Example 2.

5 Project description

This project consists of two main parts. The first part is to develop a simulation program for the system which consists of the server farm in Fig. 2 and the load balancing algorithms. The system has already been described in Section 3.2 and illustrated in Section 4. In the second part, you will use the simulation program that you have developed to solve a design problem.

5.1 Simulation program

You must write your simulation program in one (or a combination) of the following languages: Python (either version 2 or 3), C, C++, or Java. All these languages are available on the CSE system.

We will test your program on the CSE system so your submitted program **must** be able to run on a CSE computer. Note that it is possible that due to version and/or operating system differences, code that runs on your own computer may not work on the CSE system. It is your responsibility to ensure that your code works on the CSE system.

Note that our description uses the following variable names:

1. A variable `mode` of string type. This variable is to control whether your program will run simulation using randomly generated arrival times and service times; or in trace driven mode. The value that the parameter `mode` can take is either `random` or `trace`.
2. A variable `time_end` which stops the simulation if the master clock exceeds this value. This variable is only relevant when `mode` is `random`. This variable is a positive floating point number.

Note that your simulation program must be a general program which allows different parameter values to be used. When we test your program, we will vary the parameter values. You can assume that we will only use valid inputs for testing.

For the simulation, you can always assume that the system is empty initially.

5.1.1 The random mode

When your simulation is working in the `random` mode, it will generate the **inter-arrival** times and service times in the slow server time unit in the following manner.

1. We use $\{a_1, a_2, \ldots, a_k, \ldots, ...\}$ to denote the inter-arrival times of the requests arriving at the dispatcher. These inter-arrival times have the following properties:
   (a) Each a_k is the product of two random numbers a_{1k} and a_{2k}, i.e $a_k = a_{1k}a_{2k}$ $\forall k = 1, 2, ...$
   (b) The sequence a_{1k} is exponentially distributed with a mean arrival rate λ requests/s.
   (c) The sequence a_{2k} is uniformly distributed in the interval $\cancel{[a_{2\ell}, a_{a2u}]}$ $[a_{2l}, a_{2u}]$. .
2. The service time in the slow server time unit t is generated by the probability density function $g(t)$ where:

$$g(t) = \begin{cases} 0 & \text{for } 0 \le t \le \alpha \\ \frac{\gamma}{t^\beta} & \text{for } \alpha < t \end{cases} \quad (1)$$

   where

$$\gamma = \frac{\beta - 1}{\alpha^{1-\beta}}$$

   Note that this probability density function has two parameters: α and β. You can assume that $\alpha > 0$ and $\beta > 2$.

5.1.2 The trace mode

When your simulation is working in the `trace` mode, it will read the list of **inter-arrival** times and the list of service times in the slow server time unit from two separate ASCII files.

Let us use Example 1 in Section 4.1 for an illustration. The arrival times are $[1, 2, 3, 5, 6, 7, 8, 10]$ and the service times in the slow server time unit are $[24, 7, 8, 6, 8, 4, 6, 2]$. Your program is required to simulate until all jobs have departed as illustrated in Table 2.

We will supply the inter-arrival times and service times in the slow server time units to you using two ASCII files. The names of these three files have specific format, which we will discuss in Section 6 on testing. For Example 1 in Section 4.1, the service times in the slow server time unit will be specified by a file whose contents are as follows:

```
24.000
7.000
8.000
6.000
8.000
4.000
6.000
2.000
```

where each service time takes up a line of the file. The same format is used for the inter-arrival times. You can assume that the data we provide for `trace` mode are consistent in the following way: The number of inter-arrival times and the number of service times in slow server time unit are equal.

Hint: Do **not** write two separate programs for the `random` and `trace` modes because they share a lot in common. A few `if–else` statements at the right places are what you need to have both modes in one program.

5.2 Determining suitable values of d for load balancing

After writing your simulation program, your next step is to use your simulation program to investigate the following questions:

1. For load balancing algorithm (Version 1), what is the value of d that gives the best mean response time?
2. For load balancing algorithm (Version 2), what is the value of d that gives the best mean response time ? You may determine d to up to an accuracy of 1 decimal place.
3. Statistically speaking, can load balancing algorithm (Version 2) achieve a lower mean response time compared to Version 1?

For this design problem, you will assume the following parameter values: $f = 1.5$, $\lambda = 6.1$, $a_{2\ell} = 0.8$, $a_{2u} = 1.1$, $\alpha = 0.2$ and $\beta = 3.5$.

In solving this design problem, you need to ensure that you use **statistically sound** methods to compare systems. You will need to consider parameters such as length of simulation, number of replications, transient removals and so on. You will need to justify in your report how you answer the above three questions.

6 Testing your simulation program

In order for us to test the correctness of your simulation program, we will run your program using a number of test cases. The aim of this section is to describe the expected input/output file format and how the testing will be performed.

Each test is specified by 4 configurations files. We will index the tests from 1. If 12 tests are used, then the indices for the tests are 1, 2, ..., 12. The names of the configuration files are:

- For Test 1, the configuration files are `mode_1.txt`, `para_1.txt`, `interarrival_1.txt` and `service_1.txt`. The files are similarly named for indices 2, 3, ..,9.
- For Test 10, the configuration files are `mode_10.txt`, `para_10.txt`, `interarrival_10.txt` and `service_10.txt`. The files are similarly named if the test index is a 2-digit number.

We will refer to these files using the generic names `mode_*.txt`, `para_*.txt` etc. We will describe the format of the configuration files in Section 6.1

Each test should produce 4 output files whose format will be described in Section 6.2. We will explain how testing will be conducted in Sections 6.3 and 6.5.

6.1 Configuration file format

6.1.1 `mode_*.txt`

This file is to indicate whether the simulation should run in the `random` or `trace` mode. The file contains one string, which can either be `random` or `trace`.

6.1.2 `para_*.txt`

If the simulation mode is `trace`, then this file has three lines. The first line is the value of f. The second line contains an integer whose value is either 1 or 2; this is used to indicate the version of load balancing algorithm. The third line is the value of d. If the test is Example 1 in Section 4.1, then the contents of this file are:

```
2.000
1
1.0
```

These values are in sample file `para_1.txt`.

If the simulation mode is `random`, then the file has four lines. The meaning of the first three lines are the same as above. The last line contains the value of `time_end`, which is the end time of the simulation. The contents of the sample file `para_4.txt` are shown below where the last line indicates that the simulation should run until `5000.000`.

```
1.350
2
0.5
5000.000
```

You can assume that we will only give you valid values. For f, it is a floating point number strictly greater than 1. For version number, it is an integer of value 1 or 2. The parameter d is non-negative integer or floating point number. For `time_end`, it is a strictly positive integer or floating point number.

6.1.3 `interarrival_*.txt`

The contents of the file `interarrival_*.txt` depend on the mode of the test. If mode is `trace`, then the file `interarrival_*.txt` contains the interarrival times of the requests with one interarrival time occupying one line. You can assume that the list of interarrival times is always positive. For Example 1 in Section 4.1, this file will be

```
1.000
1.000
1.000
2.000
1.000
1.000
1.000
2.000
```

If the mode is `random`, then the file `interarrival_*.txt` contain three lines, corresponding to the values of λ, $a_{2\ell}$ and a_{2u}.

6.1.4 `service_*.txt`

For `trace` mode, the file `service_*.txt` contains one service time in the slow server time unit per line.

For `random` mode, the file `service_*.txt` contains two lines, corresponding to the values of α and β.

6.2 Output file format

In order to test your simulation program, we need four output files **per test**. One file containing the mean response time. The other three files contain the departure times from the three servers.

The mean response time without considering transient removal (a scalar value) should be written to a file whose filename has the form `mrt_*.txt`. For Example 1 in Section 4.1, the contents of this file are:

```
9.6250
```

The other three files contains the departure times of the requests from the three servers. The filename should be of the form `s1_dep_*.txt`, `s2_dep_*.txt` and `s3_dep_*.txt` for respectively, Servers 1, 2 and 3. For Example 1 in Section 4.1, the contents of `s1_dep_*.txt` are:

```
2.0000 9.0000
6.0000 17.0000
10.0000 19.0000
```

The contents of `s2_dep_*.txt` are:

```
3.0000 11.0000
7.0000 15.0000
```

The contents of `s3_dep_*.txt` are:

```
1.0000 13.0000
5.0000 16.0000
8.0000 19.0000
```

Note the following requirements for the files containing the departure times:

1. Each line contains the arrival time and departure time of a request. The arrival time is printed first, followed by blank spaces or tab, then the departure time from that particular component of the system.
2. The requests must be ordered according to the ascending time of the arrival time.
3. If the simulation is in the `trace` mode, we expect the simulation to finish after all requests have left the system. Therefore, each departure file should contain all the requests that have left that particular server.
4. If the simulation is in the `random` mode, each departure file should contain all the requests that have left that particular component by `time_end`.

All numbers in `mrt_*.txt`, `s1_dep_*.txt`, `s2_dep_*.txt` and `s3_dep_*.txt` should be printed as floating point numbers to exactly 4 decimal places.

6.3 The testing framework

When you submit your project, you must include a Linux bash shell script with the name `run_test.sh` so that we can run your program on the CSE system. This shell script is required because you are allowed to use a computer language of your choice.

Let us first recall that each test is specified by a four configuration files and should produce four output files. For example, test number 1 is specified by the configuration files `mode_1.txt`, `interarrival_1.txt`, `service_1.txt` and `para_1.txt`; and test number 1 is expected to produce the output files `mrt_1.txt`, `s1_dep_1.txt`, `s2_dep_1.txt` and `s3_dep_1.txt`.

We will use the following directory structure when we do testing.

```
the directory containing run_test.sh
├── config/
└── output/
```

We will put all the configuration files for all the tests in the sub-directory `config/`. You should write all the output files to the sub-directory `output/`.

To run test number 1, we use the shell command:

```
./run_test.sh 1
```

The expected behaviour is that your simulation program will read in the configuration files for test number 1 from `config/`, carry out the simulation and create the output files in `output/`.

Similarly, to run test number 2, we use the shell command:

```
./run_test.sh 2
```

This means that the shell script `run_test.sh` has one input argument which is the test number to be used.

Let us for the time being assume that you use Python (Version 3) to write your simulation program and you call your simulation program `main.py`. If the file `main.py` is in the same directory as `run_test.sh`, then `run_test.sh` can be the following one-line shell script:

```
python3 main.py $1
```

The shell script will pass the test number (which is in the input argument `$1`) to your simulation program `main.py`. This also implies that your simulation program should accept one input argument which is the test number.

Just in case you are not familiar with shell script, we have provided two sample files: `run_test_ex.sh` and `main_ex.py` to illustrate the interaction between a shell script and a Python (Version 3) file. You need to make sure `run_test_ex.sh` is executable. If you run the command `./run_test_ex.sh 2`, it will read the sample `para_2.txt` in the `conifg/` directory and write a file with the name `dummy_2.txt` to the directory `output/`. You can also try using input arguments 1, 3 or 4 for the sample shell script. You can use these sample files to help you to develop your code.

If you use C, C++ or Java, then your `run_test.sh` should first compile the source code and then run the executable. You should of course pass the test number to the executable as an input.

You can put your code in the same directory that contains `run_test.sh` or in a subdirectory below it. For example, you may have a subdirectory `src/` for your code like the following:

```
the directory containing run_test.sh
├── config/
├── output/
└── src/
```

6.4 Sample files

You should download the file `sample_project_files.zip` from the project page on the course website. The zip archive has the following directory structure:

```
Base directory containing cf_output_with_ref.py, run_test_ex.sh and main_ex.py
├── config/
├── output/
└── ref/
```

Details on the zip-archive are:

- The sub-directory `config/` contains configuration files that you can use for testing.
  - The files `mode_1.txt`, `mode_2.txt`, `mode_3.txt` and `mode_4.txt`. Note that Tests 1, 2 and 3 are for `trace` mode while Test 4 is for `random` mode.
  - The files `para_*.txt`, `interarrival_*.txt` and `service_*.txt` for `*` from 1 to 4, as the input to the simulation.
  - Note that Tests 1 and 2 are the same as, respectively, Example 1 and Example 2 in Section 4.
- The sub-directory `output/` is empty. Your simulation program should place the output files in this sub-dirrectory.
- The sub-directory `ref/` contains the expected simulation results.
  - The files `mrt_*_ref.txt`, `s1_dep_*_ref.txt`, `s2_dep_*_ref.txt` and `s3_dep_*_ref.txt` for `*` from 1 to 4, as the reference files for the output. For Tests 1, 2 and 3, you should be able to reproduce the results in `mrt_*_ref.txt`, `s1_dep_*_ref.txt`, `s2_dep_*_ref.txt` and `s3_dep_*_ref.txt`. However, since Test 4 is in `random` mode, you will not be able to reproduce the results in the output files. They have been provided so that you can check the expected format of the file.
- The Python file `cf_output_with_ref.py` which illustrates how we will compare your output against the reference output. This file takes in one input argument, which is the test number. For example, if you want to check your simulation outputs for test 2, you use:

```
python3 cf_output_with_ref.py 2
```

Note the following:

- The file `cf_output_with_ref.py` expects the directory structure shown earlier.
- For `trace` mode, we will check your mean response time and the departure times from the servers. Note that we are not looking for an exact match but rather whether your results are within a valid tolerance. The tolerance for the `trace` mode is 10^{-3} which is fairly generous for numbers with 4 decimal places.
- For `random` mode, we will only check the mean response time. You can see from the sample file that we check whether the mean response time is within an interval. We obtain this interval using the following method: (i) we first simulate the system many times; (ii) we then use the simulation results to estimate the maximum and minimum mean response times; (iii) we use the estimated maximum and minimum values to form an interval; (iv) in order to provide some tolerance due to randomness, we enlarge this interval further.
- Note that we use a very generous tolerance so if your mean response time does not pass the test, then it is highly likely that your simulation program is not correct.

• The files `run_test_ex.sh` and `main_ex.py` as mentioned in Section 6.3.

6.5 Carrying out your own testing on the CSE system

It is important for you to note the assumption on directory structure mentioned in Section 6.3. You must ensure your shell script and program files are written with this assumption in mind.

Since we will be testing your work on the CSE system, we strongly advise you to carry out the following on the CSE system before submission.

- Create a new folder in your CSE account and `cd` to that folder. We will refer to this directory as the base directory.
  - Copy your shell script `run_test.sh` and program files to the base directory.
  - Copy the `config` and `ref` directories, as well as their contents, to the base directory
  - Create a empty directory `output`
- Make sure your shell script is executable.
- Run your shell script for each test one by one. Make sure that each run produces the appropriate output files for that test in the `output` directory.
- Copy `cf_output_with_ref.py` to the base directory. Run it compare your output against the reference output.

These steps are the same as those that we will use for testing. It is important to know that we will create an empty `output/` directory before we run your code. This means your code does **NOT** have to create the the `output/` directory.

7 Project requirements

This is an individual project. You are expected to complete this project on your own.

7.1 Submission requirements

Your submission should include the following:

1. A written report
   (a) Only soft copy is required.
   (b) It must be in Acrobat pdf format.
   (c) It must be called ”report.pdf”.
2. Program source code:
   (a) For doing simulation
   (b) The shell script `run_test.sh`, see Section 6.3.
3. Any supporting materials, e.g. logs created by your simulation, scripts that you have written to process the data etc.

The assessment will be based on your submission and running your code on the CSE system. It is important that you submit the right version of the code and make sure that it runs on the CSE system.

It is important that you write a clear and to-the-point report. You need to aware that you are writing the report to the marker (the intended audience of the report) not for yourself. Your report will be assessed primarily based on the quality of the work that you have done. You do not have to include any background materials in your report. You only have to talk about how you do the work and we have provided a set of assessment criteria in Section 7.2 to help you to write your report. In order for you to demonstrate these criteria, your report should refer to your programs, scripts, additional materials so that we are aware of them.

7.2 Assessment criteria

We will assess the quality of your project based on the following criteria:

1. The correctness of your simulation code. For this, we will:
   (a) Test your code using test cases
   (b) Look for evidence in your report that you have verified the correctness of the inter-arrival probability distribution and service time distribution. You can include appropriate supporting materials to demonstrate this in your submission.
   (c) Look for evidence in your report that you have verified the correctness of your simulation code. You may derive test cases such as those in Section 4 to test your code. You can include appropriate supporting materials to demonstrate this in your submission.
2. You will need to demonstrate that your results are reproducible. You should provide evidence of this in your report.
3. For the part on determining a suitable value of d for the two versions of load balancing algorithm as well as whether one algorithm is better than the other, we will look for the following in your report:
   (a) Evidence of using statistically sound methods to analyse simulation results
   (b) Explanation on how you choose your simulation and data processing parameters, e.g lengths of your simulation, number of replications, end of transient etc.

7.3 How to submit

You should "zip" your report, shell script, programs and supporting materials into a file called "project.zip". The submission system will only accept this filename. **Please ensure that you run `zip` in the directory containing your `run_test.sh`. If you need to store directories when zipping, you need to use the `-r` switch. The last point is especially important if you put your program code in a sub-directory under the base directory; in this case, the relative path between your `run_test.sh` and your program code need to be preserved.**

You should submit your work via the course website. Note that the maximum size of your submission should be no more than 20MBytes.

You can submit multiple times before the deadline. The latest submission overrides the earlier submissions, so make sure you submit the correct file. Do not leave until the last moment to submit, as there may be technical or communication error and you will not have time to rectify.

8 Plagiarism

You should be aware of the UNSW policy on plagiarism. Please refer to the course web page for details.

References

[1] Mor Harchol-Balter. Performance Modeling and Design of Computer Systems. Cambridge University Press (2013).

[2] Gardner et al., Scalable Load Balancing in the Presence of Heterogeneous Servers. Extended abstract for Performance 2020. . `https://www.performance2020.deib.polimi.it/wp-content/uploads/2020/11/Performance_2020_paper_23.pdf`.