COMP9334 Project, Term 1, 2022: Priority queueing for server farms

Due Date: 5:00pm Friday 22 April 2022

Version 1.00, 18 March 2022

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

Change log

Version 1.00. Issued on 18 March 2022.

1 Introduction and learning objectives

Server farms form the backbone of a lot of corporate information technology infrastructures. You can find them in data centres and cloud platforms. In this project, you will use simulation to study how priority queueing can be used to improve the performance of a server farm.

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/`

3 Server farm configuration for this project

Let us start with a short background. A server farm [1] consists of hundreds (or even thousands) of servers. These server farms are used to process information in data centres [2]. They can also form part of public cloud infrastructure where users can purchase computing time from cloud service providers such as Amazon, Google and Microsoft [2]. When you purchase computing time,

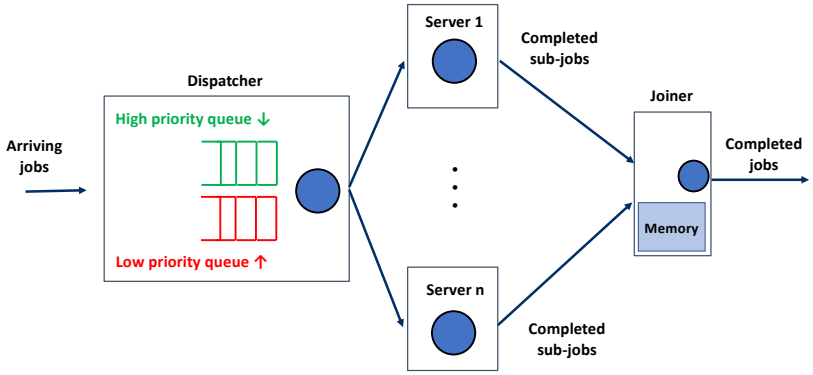

Figure 1: The server farm for this project.

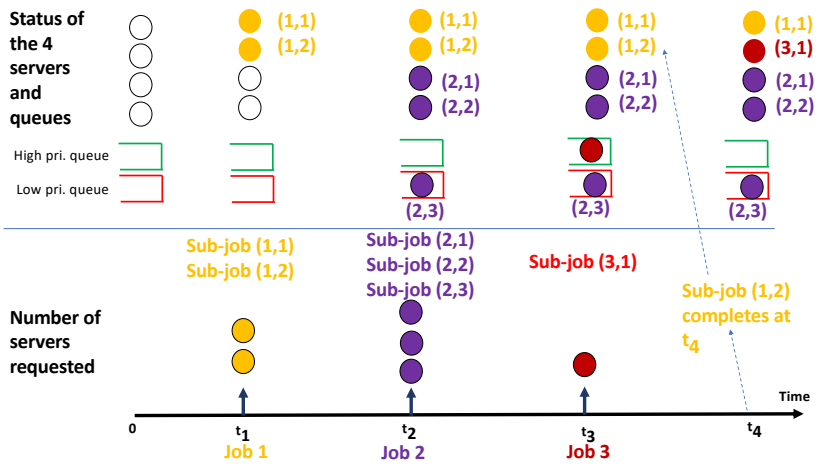

Figure 2: An illustration of the work load and the operation of the server farm. Each job consists of one or more sub-jobs where each sub-job is to be processed by a server.

you may have to specify the type of servers that you want, how many of them and for how long.

The configuration of the server farm that you will use in this project is shown in Fig. 1. The farm consists of a dispatcher, n servers (where $n > 1$) and a joiner. The dispatcher has two queues: a high priority queue and a low priority queue. You can assume that both queues have infinite queueing slots. There are no queues at the server nor joiner.

We will use the word *job* to refer to a request arriving at the server form. Each job will ask to use the service of one or more servers. As an illustration, in Fig. 2, Job 1 asks for 2 servers, Job 2 asks for 3 servers and Job 3 asks for 1 server. If a job asks for k servers for service, then we say that the job consists of k *sub-jobs* where each sub-job is to be processed by a server. For example, in Fig. 2, Job 1 asks for 2 servers, so it consists of 2 sub-jobs which are referred to as using the tuples (1,1) and (1,2). You can interpret the first coordinate of the tuple as the index to a job and the second coordinate of the tuple as the index to a sub-job. You can see other examples in Fig. 2.

We will now use Fig. 2 to illustrate the operation of the server farm assuming there are 4 ($= n$) servers. We assume that the server farm is empty at time 0, i.e. all servers are idle and the queues are empty. Job 1 arrives at time t_1 and asks for 2 servers. Since there are sufficient number of idle servers available, all the sub-jobs will head to the servers to receive their service. Fig. 2 shows the status of the servers and queues just after time t_1. We assume that it takes negligible time for the dispatcher to do its processing and to communicate with the server. This means we can assume that Sub-jobs (1,1) and (1,2) arrive at the servers at time t_1.

In Fig. 2, Job 2 arrives at time t_2 and asks for 3 servers. However, only two servers are idle. So, Sub-jobs (2,1) and (2,2) will head to the two idle servers, while sub-job (2,3) will join the queue. This illustrates that the sub-jobs are ordered and they are processed in an order according to their indices.

We will now explain the operation of the queues. The server farm uses a threshold h to decide whether a sub-job will join the high or low priority queue. If a sub-job comes from a job that asks for h or less servers, then that sub-job will join the high priority queue; otherwise, it will join the low priority queue. For the example in Fig. 2, we assume $h = 2$. Since Sub-job (2,3) comes from Job 2 and Job 2 asks for 3 servers, so the number of servers that Job 2 asks for is higher than the threshold $h = 2$ and this means Sub-job (2,3) will head to the low priority queue. Fig. 2 shows the status of the servers and queues just after time t_2, where you can see that Sub-jobs (2,1) and (2,2) are in the servers but Sub-job (2,3) is in the low priority queue.

Before moving on, we need to further explain how the queues operate at the dispatcher. The queueing discipline in the dispatcher is non-preemptive and we will explain that in Week 7. You can also read about it on p. 500 of [1]. Since we hope that you can get your project started early, we will explain here the rules for the queues at the dispatcher:

1. If a sub-job arrives when all the servers are busy, you will need to determine whether the sub-job is of a high or low priority. After that, a high priority sub-job will go to the back of the high priority queue, and similarly a low priority sub-job will go to the back of the low priority queue.

2. When a server finishes the processing of a sub-job, it will ask the dispatcher for the next sub-job. One of the following three possibilities will happen:

   (a) If the high priority queue is non-empty, the dispatcher will send the first sub-job in the high priority queue to the available server. (Note that if the high priority queue is non-empty, you do not need to check the status of the low priority queue.)

(b) If the high priority queue is empty and the low priority queue is non-empty, the dispatcher will send the first sub-job in the low priority queue to the available server.

(c) If both queues are empty, the server will become idle.

We now continue on the example in Fig. 2 where Job 3 arrives at time t_3 and asks for a server. Since all the servers are busy at the time that Job 3 arrives and Job 3 has a high priority, the only sub-job of Job 3 will join the high priority queue. Fig. 2 shows the status of the server and queues just after the arrival of Job 3.

Next, we assume that Sub-job (1,2) is completed at time t_4, see Fig. 2. A completed sub-job will depart the server and head to the joiner. The available server will contact the dispatcher for a possible next sub-job. Since the high priority queue is non-empty, the sub-job at the head of the queue — which is Sub-job (3,1) — will head to the available server. Fig. 2 shows the status of the server and queues just after Sub-job (3,1) has gone to the server.

We have now explained how the dispatcher and the servers operate, we will now explain the operation of the joiner. The joiner has memory to store the results of the sub-jobs. Once all the results of all the sub-jobs of a job have arrived at the joiner, the jointer will combine all the results and send them to the user who submitted the job. We say that the departure time of a job from the server farm is the time at which the joiner sends all the results of a job to an user. We assume that it takes negligible time for a server to send results to the joiner, it takes negligible time for a joiner to combine all the results from the sub-jobs, and there is sufficient memory at the joiner to store the results from the sub-jobs. Let us consider the example in Fig. 2. Since the server completes the processing of Sub-job (1,2) at time t_4, the result of Sub-job (1,2) will be at the joiner at time t_4. Let us assume that Sub-job (1,1) will be done at the server at time t_5 (and we assume $t_5 > t_4$), so this is also the time that the result of Sub-job (1,1) arrives at the joiner. This means that, at time t_5, all the sub-jobs of Job 1 have been processed. The joiner will combine these results and send them to the user who submitted Job 1. Since it takes negligible time for the joiner to combine the results, we therefore consider that the server farm completes Job 1 at time t_5. In general, consider a job that has k sub-jobs and the times at which these k sub-jobs are completed at the servers are c_1, c_2, ..., and c_k. Let $c_{\max}$ be the maximum of c_1, c_2, ..., and c_k. This means that all the sub-jobs of this job will have been completed at the time $c_{\max}$. This means that $c_{\max}$ is the time that the joiner sends the results of this job to the user and the time $c_{\max}$ is the departure time of this job from the server farm. Consequently, the response time of a job is its departure time minus its arrival time.

We want to make a few remarks concerning this server farm. First, there are no rejections or losses in this server farm since we assume the dispatcher has infinite number of queueing slots and the joiner has sufficient memory. Second, the response time of the server farms depends only on the queues and the servers. This is because it takes negligible processing time at the dispatcher and at the joiner.

We have now completed our description of the operation of the server farm. We will provide a number of numerical examples to further explain their operation in Section 4. Note that in the illustration in this section, we have assumed particular values for the number of servers n and the threshold h for deciding whether a sub-job is considered high or low priority. However, note that n and h are parameters, and we will vary them in the simulation.

You will see that for a given number of servers n, we can use the value of threshold h to influence the mean response time of the server farm. So, a design problem that you will consider in this project is to determine the value of the threshold h to minimise the mean response time of the server farm. You can read in [1] how priority queueing can be used to reduce the mean response time of computer systems.

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: number of servers $n = 4$ and threshold $h = 1$

In this example, we assume the there are $n = 4$ servers in the server farm and the threshold h for determining whether a sub-job is of low or high priority is 1.

In this example, each job may ask for the service of either 1 server or 2 servers. Table 1 shows, for each job, its arrival time and the service times of its sub-jobs. If there is only one service time, then it means the job is only asking for one server. If there are two service times, then the job is asking for two servers.

Job index	Arrival time	Service times of the sub-jobs
1	1	1.8
2	3	8.0, 6.1
3	5	3.9
4	6	2.1, 3.1
5	7	1.9
6	8	5.0, 4.1
7	9	3.8

Table 1: Data for Example 1.

In Table 1, Job 1 has one sub-job, which can be represented as the tuple (1,1), and this sub-job requires a service time of 1.8. Job 2 has two sub-jobs and its two sub-jobs are represented by the tuples (2,1) and (2,2). The service times required by Sub-jobs (2,1) and (2,2) are, respectively, 8.0 and 6.1.

The events in the server farm are the arrival of a job to the dispatcher and the departure of a completed sub-job from a server. 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 job arrival
- For each server, we keep track of the following information:
  - Server status, which can be either busy or idle.
  - **Next departure time** is the time at which the sub-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.
  - For a sub-job in the server, the time that this sub-job arrives at the server farm
  - The tuple to identify the sub-job in the server

  For example, if the server status is "Busy, 7.5, 3.4, (1,2)", this means the server is busy serving the sub-job identified by the tuple (1,2), and this sub-job is scheduled to depart at time 7.5 and has arrived at the server farm at time 3.4. If the server is idle, we will write the status as "Idle, ∞" where the departure time is ∞.
- The contents of the high and low priority queues. Each sub-job in the queue is identified by 3 fields: the tuple identifying the sub-job, the sub-job's arrival time to the server farm, the sub-job's service time. For example, we write a sub-job in a queue as

[(4,2), 6, 3.1]

which means Sub-job (4,2) arrives at the server farm at time 6 and requires a service time of 3.1.

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	Server 4	High priority queue	Low priority queue	Notes
0	–	1	Idle, ∞	Idle, ∞	Idle, ∞	Idle, ∞	–	–	We assume the servers are idle and queues are empty at the start of the simulation. The next departure times for all servers are ∞. The "–" indicates that the queues are empty.
1	Arrival	3	Busy, 2.8, 1, (1,1)	Idle, ∞	Idle, ∞	Idle, ∞	–	–	The event is the arrival of Job 1 with one sub-job (1,1). Since all the servers are idle before this arrival, the sub-job can be sent to any one of the idle servers. We have chosen to send this sub-job to Server 1. Sub-job (1,1) requires a service time of 1.8 and it starts to receive service at time 1, so its departure time is 2.8. Lastly, we need to update the next arrival time, which is 3.
2.8	Departure Server 1	3	Idle, ∞	Idle, ∞	Idle, ∞	Idle, ∞	–	–	The event is a departure from Server 1. Sub-job (1,1) has now been completed with an arrival time of 1 and a departure time of 2.8. Since both queues are empty, Server 1 becomes idle. The server status has been updated accordingly.
3	Arrival	5	Busy, 11.0, 3, (2,1)	Busy, 9.1, 3, (2,2)	Idle, ∞	Idle, ∞	–	–	The event is the arrival of Job 2 which consists of two sub-jobs (2,1) and (2,2). Since there are 4 idle servers before the arrival of these two sub-jobs, they can be sent to any two idle servers. We have chosen to send the two sub-jobs to Servers 1 and 2. Sub-job (2,1) requires a service time of 8.0 and it starts to receive service at time 3, so its departure time is 11.0. Similarly, Sub-job (2,2) will depart at time is 9.1. Lastly, we need to update the next arrival time, which is 5.

5	Arrival	6	Busy, 11.0, 3, (2,1)	Busy, 9.1, 3, (2,2)	Busy, 8.9, 5, (3,1)	Idle, ∞	–	–	The event is the arrival of Job 3 which consists of one sub-job (3,1). Since there are 2 idle servers before the arrival of this sub-job, it can be sent to any of the idle servers. We have chosen to send this sub-job to Server 3. Sub-job (3,1) requires a service time of 3.9 and it starts to receive service at time 5, so its departure time is 8.9. Lastly, we need to update the next arrival time, which is 6.
6	Arrival	7	Busy, 11.0, 3, (2,1)	Busy, 9.1, 3, (2,2)	Busy, 8.9, 5, (3,1)	Busy, 8.1, 6, (4,1)	–	[(4,2), 6, 3.1]	The event is the arrival of Job 4 which consists of two sub-jobs (4,1) and (4,2). Since there is only one idle server before the arrival of these two sub-jobs, the first sub-job (i.e., Sub-job (4,1)) will go to the idle server (i.e., Server 4). The status of Server 4 has been updated accordingly. The second sub-job (i.e., Sub-job (4,2)) will join the queue. Since Job 4 has two sub-jobs and the threshold $h = 1$, therefore Sub-job (4,2) will enter the low priority queue. We enclose the details of each sub-job in the queue within a pair of square brackets. In this case, the sub-job details are [(4,2), 6, 3.1] which refers to Sub-job (4,1) with an arrival time of 6 and a service time of 3.1. Lastly, we need to update the next arrival time, which is 7.
7	Arrival	8	Busy, 11.0, 3, (2,1)	Busy, 9.1, 3, (2,2)	Busy, 8.9, 5, (3,1)	Busy, 8.1, 6, (4,1)	[(5,1), 7, 1.9]	[(4,2), 6, 3.1]	The event is the arrival of Job 5 which consists of one sub-job (5,1). Since all servers are busy, Sub-job (5,1) will be sent to the queue. Since Job 5 has one sub-job and the threshold $h = 1$, therefore Sub-job (5,1) will enter the high priority queue. The sub-job details [(5,1), 7, 1.9] are added to the high priority queue. Lastly, we need to update the next arrival time, which is 8.

8	Arrival	9	Busy, 11.0, 3, (2,1)	Busy, 9.1, 3, (2,2)	Busy, 8.9, 5, (3,1)	Busy, 8.1, 6, (4,1)	[(5,1), 7, 1.9]	[(4,2), 6, 3.1] [(6,1), 8, 5.0] [(6,2), 8, 4.1]	The event is the arrival of Job 6 which consists of two sub-jobs (6,1) and (6,2). Since all servers are busy, both sub-jobs will be sent to the low priority queue. The sub-job details [(6,1), 8, 5.0] and [(6,2), 8, 4.1] have been added to the low-priority queue. Lastly, we need to update the next arrival time, which is 9.
8.1	Departure Server 4	9	Busy, 11.0, 3, (2,1)	Busy, 9.1, 3, (2,2)	Busy, 8.9, 5, (3,1)	Busy, 10.0, 7, (5,1)	–	[(4,2), 6, 3.1] [(6,1), 8, 5.0] [(6,2), 8, 4.1]	The event is a departure from Server 4. Sub-job (4,1) has now been completed with an arrival time of 6 and a departure time of 8.1. Since the high priority queue is non-empty, the first sub-job in the queue will advance to the available server. The sub-job heading to the queue has a service time of 1.9 and the current time (= master clock) is at 8.1, so the departure time of the sub-job will be 8.1 + 1.9 = 10.0. The server status has been updated accordingly.
8.9	Departure Server 3	9	Busy, 11.0, 3, (2,1)	Busy, 9.1, 3, (2,2)	Busy, 12.0, 6, (4,2)	Busy, 10.0, 7, (5,1)	–	[(6,1), 8, 5.0] [(6,2), 8, 4.1]	The event is a departure from Server 3. Sub-job (3,1) has now been completed with an arrival time of 5 and a departure time of 8.9. Since the high priority queue is empty and the low priority queue is non-empty, the first sub-job in the low priority queue will advance to the available server. The server status and queue status have been updated accordingly.
9	Arrival	∞	Busy, 11.0, 3, (2,1)	Busy, 9.1, 3, (2,2)	Busy, 12.0, 6, (4,2)	Busy, 10.0, 7, (5,1)	[(7,1), 9, 3.8]	[(6,1), 8, 5.0] [(6,2), 8, 4.1]	The event is the arrival of Job 7 which consists of one sub-job (7,1). Since all servers are busy, Sub-job (7,1) will be sent to the high-priority queue and the queue status has been updated. Lastly, we need to update the next arrival time to ∞ because there are no more arrivals.

9.1	Departure Server 2	∞	Busy, 11.0, 3, (2,1)	Busy, 12.9, 9, (7,1)	Busy, 12.0, 6, (4,2)	Busy, 10.0, 7, (5,1)	–	[(6,1), 8, 5.0] [(6,2), 8, 4.1]	The event is a departure from Server 2. Sub-job (2,2) has now been completed with an arrival time of 3 and a departure time of 9.1. Since the high priority queue is non-empty, the first sub-job in the high priority queue will advance to the available server. The server status and queue status have been updated accordingly.
10	Departure Server 4	∞	Busy, 11.0, 8, (2,1)	Busy, 12.9, 9, (7,1)	Busy, 12.0, 6, (4,2)	Busy, 15, 8, (6,1)	–	[(6,2), 8, 4.1]	The event is a departure from Server 4. Sub-job (5,1) has now been completed with an arrival time of 7 and a departure time of 10.0. Since the high priority queue is empty and the low priority queue is non-empty. the first sub-job in the low priority queue will advance to the available server. The server status and queue status have been updated accordingly.
11	Departure Server 1	∞	Busy, 15.1, 8, (6,2)	Busy, 12.9, 9, (7,1)	Busy, 12.0, 6, (4,2)	Busy, 15, 8, (6,1)	–	–	The event is a departure from Server 1. Sub-job (2,1) has now been completed with an arrival time of 3 and a departure time of 11.0. Since the high priority queue is empty and the low priority queue is non-empty. the first sub-job in the low priority queue will advance to the available server. The server status and queue status have been updated accordingly.
12	Departure Server 3	∞	Busy, 15.1, 8, (6,2)	Busy, 12.9, 9, (7,1)	Idle, ∞	Busy, 15, 8, (6,1)	–	–	The event is a departure from Server 3. Sub-job (4,2) has now been completed with an arrival time of 6 and a departure time of 12.0. Since both queues are empty, Server 3 becomes idle. The server status has been updated accordingly.
12.9	Departure Server 2	∞	Busy, 15.1, 8, (6,2)	Idle, ∞	Idle, ∞	Busy, 15, 8, (6,1)	–	–	The event is a departure from Server 2. Sub-job (7,1) has now been completed with an arrival time of 9 and a departure time of 12.9. Since both queues are empty, Server 2 becomes idle. The server status has been updated accordingly.

15.0	Departure Server 4	∞	Busy, 15.1, 8, (6,2)	Idle, ∞	Idle, ∞	Idle, ∞	–	–	The event is a departure from Server 4. Sub-job (6,1) has now been completed with an arrival time of 8 and a departure time of 15.0. Since both queues are empty, Server 4 becomes idle. The server status has been updated accordingly.
15.1	Departure Server 1	∞	Idle, ∞	Idle, ∞	Idle, ∞	Idle, ∞	–	–	The event is a departure from Server 1. Sub-job (6,2) has now been completed with an arrival time of 8 and a departure time of 15.1. Since both queues are empty, Server 1 becomes idle. The server status has been updated accordingly.

Table 2: Table illustrating the updates in the server farm.

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.

Table 3 summarises the arrival and departure times of all the sub-jobs. The two sub-jobs of Job 2 complete at times 9.1 and 11.0. Hence, the departure time of Job 2 from the server farm is 11.0, which is the later time of the two departure times. Since Job 2 arrives at time 3, the response time of Job 2 is 11–3 = 8. The departure and response times of other jobs can be similarly computed. The arrival and departure times of all the jobs for Example 1 are given in Table 4. The mean response time of the 7 jobs in this example is $\frac{33.7}{7} = 4.8143$.

Sub-jobs	Arrival time	Departure time
(1,1)	1.0	2.8
(4,1)	6.0	8.1
(3,1)	5.0	8.9
(2,2)	3.0	9.1
(5,1)	7.0	10.0
(2,1)	3.0	11.0
(4,2)	6.0	12.0
(7,1)	9.0	12.9
(6,1)	8.0	15.0
(6,2)	8.0	15.1

Table 3: The arrival and departure times of the sub-jobs for Example 1.

Job	Arrival time	Departure time
1	1.0	2.8
2	3.0	11.0
3	5.0	8.9
4	6.0	12.0
5	7.0	10.0
6	8.0	15.1
7	9.0	12.9

Table 4: The arrival and departure times of the jobs for Example 1.

Note that in this example, we have chosen to obtain the departure times of the sub-jobs and perform post-processing to obtain the departure times of the jobs. Alternatively, the computation of the departure times of the jobs can be incorporated into the simulation exemplified in Table 2. You can use either of the methods.

4.2 Example 2: number of servers $n = 4$, threshold $h = 2$

This example is identical to Example 1 except that the threshold $h = 2$. In other words, for this example, the server farm uses $n = 4$ and $h = 2$, and the arrivals to the server farm are given in Table 1. Note that the setting $h = 2$ essentially means all sub-jobs go into the high priority queue and the low priority queue is not used at all.

Table 5 summarises the arrival and departure times of all the sub-jobs while Table 6 summarises the arrival and departure times of all the jobs. The mean response time of the 7 jobs in this example is $\frac{35.4}{7} = 5.0571$, which is higher than that of Example 1. This demonstrates that priority queueing can be used to influence the mean response time of the server.

Sub-jobs	Arrival time	Departure time
(1,1)	1.0	2.8
(4,1)	6.0	8.1
(3,1)	5.0	8.9
(2,2)	3.0	9.1
(5,1)	7.0	10.8
(2,1)	3.0	11.0
(4,2)	6.0	11.2
(6,1)	8.0	14.1
(7,1)	9.0	14.8
(6,2)	8.0	14.9

Table 5: The arrival and departure times of the sub-jobs for Example 2.

Job	Arrival time	Departure time
1	1.0	2.8
2	3.0	11.0
3	5.0	8.9
4	6.0	11.2
5	7.0	10.8
6	8.0	14.9
7	9.0	14.8

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

5 Project description

This project consists of two main parts. The first part is to develop a simulation program for the server farm in Fig. 1. The system has already been described in Section 3 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 the workload of a job in the following manner.

1. We use $\{a_1, a_2, \ldots, a_k, \ldots, ...\}$ to denote the inter-arrival times of the jobs 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 $[a_{2l}, a_{2u}]$.

   Note: The easiest way to generate the inter-arrival times is to multiply an exponentially distributed random number with the given rate and a uniformly distributed random number in the given range. It would be more difficult to use the inverse transform method in this case, though it is doable.
2. The workload of a job is characterised by the number of sub-jobs and the service times of the sub-jobs. The first step to generate the workload of a job is to generate a random positive integer which is the number of sub-jobs for that job. You will be given a sequence of J non-negative real numbers p_1, p_2, ..., p_k, ... , p_J with the property $\sum_{k=1}^{J} p_k = 1$. Given these numbers, we want the probability that a job consists of k sub-jobs to be equal to p_k,

for $k = 1, ..., J$.

For example, if you are given the sequence 0.5, 0.2, 0.3. This means for the jobs in this workload, you have

(a) Prob[a job consists of exactly 1 sub-job] = 0.5

(b) Prob[a job consists of exactly 2 sub-jobs] = 0.2

(c) Prob[a job consists of exactly 3 sub-jobs] = 0.3

Note that you may interpret J is the maximum number of servers that a job can request.

3. If a job consists of k sub-jobs, then you will need to generate k random service times for the k sub-jobs. These k service times are independent and they all come from the same probability distribution. The cumulative distribution function (CDF) $S(t)$ of the service time t for a *sub-job* is:

$$S(t) \quad = \quad 1 - \exp(-(\mu\, t)^{\alpha}) \tag{1}$$

where the parameters μ and α are positive real numbers.

As an example, if a job consists of 3 sub-jobs, then you will need to generate 3 random numbers which come from a probability distribution whose CDF is given by $S(t)$.

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 of the sub-jobs from two separate ASCII files. We will explain the format of these files in Sections 6.1.3 and 6.1.4 .

An **important requirement** for the `trace` mode is that your program is required to simulate until all jobs have departed. You can refer to Table 2 for an illustration.

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 the threshold h that minimises the mean response time

After writing your simulation program, your next step is to use your simulation program to determine the threshold h that can minimise the mean response time.

For this design problem, you will assume the following parameter values:

- Number of servers: $n = 10$
- For inter-arrival times: $\lambda = 1.8$, $a_{2\ell} = 0.7$, $a_{2u} = 0.9$
- For the number of sub-jobs per job: the sequence p_1, p_2, p_3, p_4, p_5 is $0.4, 0.25, 0.15, 0.11, 0.09$.
- For the service time per job: $\mu = 0.9$, $\alpha = 0.9$.

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 on how you determine the value of the threshold h.

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 2 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

Note that Test 1 is the same as Example 1 discussed in Section 4.1. We will use this test to illustrate the 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 two lines. The first line is the value of n (= number of servers) and the second line has the value of h (= threshold for priority queueing). If the test is Example 1 in Section 4.1, then the contents of this file are:

```
4
1
```

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

If the simulation mode is `random`, then the file has three lines. The meaning of the first two 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_5.txt` are shown below where the last line indicates that the simulation should run until `2000.00`.

```
5
2
2000.00
```

You can assume that we will only give you valid values. You can expect n to be a positive integer greater than 1. You can expect h to be a non-negative integer. 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 jobs 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, the arrival times are $[1, 3, 5, 6, 7, 8, 9]$ which means the inter-arrival times are $[1, 2, 2, 1, 1, 1, 1]$. The inter-arrival times will be specified by a file whose contents are:

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

If the mode is `random`, then the file `interarrival_*.txt` contain 2 lines. The first line contains three values corresponding to the parameters λ, $a_{2\ell}$ and a_{2u}. The second line contains the the values for the sequence $p_1, ..., p_J$. As an example, the contents of `interarrival_5.txt` are:

```
1.400 0.600 0.800
0.400 0.300 0.200 0.050 0.050
```

For this example, the values of λ, $a_{2\ell}$ and a_{2u} are respectively 1.400, 0.600 and 0.800. The values of $p_1, ..., p_5$ are 0.400, 0.300, 0.200, 0.050, 0.050. This means that you can infer the value of J by counting the number of values found in the second line of the file.

6.1.4 `service_*.txt`

For `trace` mode, the file `service_*.txt` contains the service times of the sub-jobs. As an illustration, the service times of the sub-jobs for Example 1 in Section 4.1 will be specified by a file whose contents are:

```
1.800 NaN
8.000  6.100
3.900 NaN
2.100  3.100
1.900 NaN
5.000  4.100
3.800 NaN
```

where you will find the service times of the sub-jobs of each job in a line of the file. Note that the symbol `NaN` is a Python floating point number to denote *not a number* and is often used to indicate an absence of numbers. In this example, if there are two numbers on the line, the job is requesting two servers; if there is a number and an `NaN`, the job is requesting for one server.

In general, if the maximum number of servers that a job can request is J, then every line of `service_*.txt` will have a total of J numbers and `NaN`'s. The following is the first three lines of `service_4.txt` where $J = 5$.

```
1.904 NaN NaN NaN NaN
1.828 0.298 1.303 NaN NaN
2.537 0.016 NaN NaN NaN
```

You can conveniently load the contents of this file by using the function `numpy.loadtxt()` into a `numpy` array. You may also find the function `numpy.isnan()` useful.

For `random` mode, the file `service_*.txt` contains one line, corresponding to the values of μ and α.

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 lines of service times are equal.

6.2 Output file format

In order to test your simulation program, we need two output files **per test**. One file containing the mean response time. The other file contains the departure times of the *sub-jobs* from the *servers*.

We want to start by clarifying what we mean by mean response time. When we talk about mean response time, we are referring to the mean response time of the *jobs* of the server farm. We are *not* referring to the mean response times of the sub-jobs. For `trace` mode, the mean response time will be calculated using all the jobs provided in the `interarrival_*.txt` and `service_*.txt`. For `random` mode, the mean response time should be calculated using all the jobs that have been completed by the end time of simulation as specified by `time_end`. Note that you do not have to consider transient removal for the mean response before you write the result to the output file. However, you should consider transient removal when you do your design.

The mean response time 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:

```
4.8143
```

The other file `dep_*.txt` contains the departure times of of the *sub-jobs* from the *servers*. For Example 1 in Section 4.1, the contents of `dep_*.txt` are:

```
1.0000   2.8000
6.0000   8.1000
5.0000   8.9000
3.0000   9.1000
7.0000   10.0000
3.0000   11.0000
6.0000   12.0000
9.0000   12.9000
8.0000   15.0000
8.0000   15.1000
```

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

1. Each line contains the arrival time and departure time of a sub-job. The arrival time is printed first, followed by blank spaces or tab, then the departure time from the server processing the sub-job.
2. The sub-jobs must be ordered according to *ascending* departure times.
3. If the simulation is in the `trace` mode, we expect the simulation to finish after all sub-jobs have been processed. Therefore, the number of lines in `dep_*.txt` should be equal to the total number of sub-jobs.
4. If the simulation is in the `random` mode, the file should contain all the sub-jobs that have left the server by `time_end`.

All numbers in `mrt_*.txt` and `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 two 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` and `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.sh` and `main.py` to illustrate the interaction between a shell script and a Python (Version 3) file. You need to make sure `run_test.sh` is executable. If you run the command `./run_test.sh 2`, it will read the sample `service_2.txt` in the `config/` 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.sh and main.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_6.txt` and `mode_7.txt`. Note that Tests 1–4 are for `trace` mode while Tests 5–7 are for `random` mode.
  - The files `para_*.txt`, `interarrival_*.txt` and `service_*.txt` for `*` from 1 to 7, 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` and `dep_*_ref.txt` for `*` from 1 to 7, as the reference files for the output. For Tests 1–4, you should be able to reproduce the results in `mrt_*_ref.txt` and `dep_*_ref.txt`. However, since Tests 5–7 are 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 files.
- 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. 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.sh` and `main.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 to 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, probability distribution of the number of sub-jobs, 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 the threshold h that minimises the mean response time, 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. A later 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] Mor Harchol-Balter. Open problems in queueing theory inspired by datacenter computing. *Queueing Systems*, 97:3-27, 2021.