COMP9334 - Capacity Planning of Computer Systems and Networks

T1 2022

Project

Name: Yuhua Zhao – **ZID:** z5404443

1. Simulation Program

```
weill % ./run_test.sh 1
Simulation  1  Done
weill % ./run_test.sh 2
Simulation  2  Done
weill % ./run_test.sh 3
Simulation  3  Done
weill % ./run_test.sh 4
Simulation  4  Done
weill % ./run_test.sh 5
Simulation  5  Done
weill % ./run_test.sh 6
Simulation  6  Done
weill % ./run_test.sh 7
Simulation  7  Done
```

Shell script “run_test.sh” will execute the main.py with the file number and generate corresponding mrt_*.txt and dep_*.txt.

The input files are located at “config” directory and the output file the generate by the main.py are located at “output” directory.

All the logs that created by the simulation and scripts are located at the folder “support_material”.

1.1. Inter-arrival Probability Distribution (Random Mode)

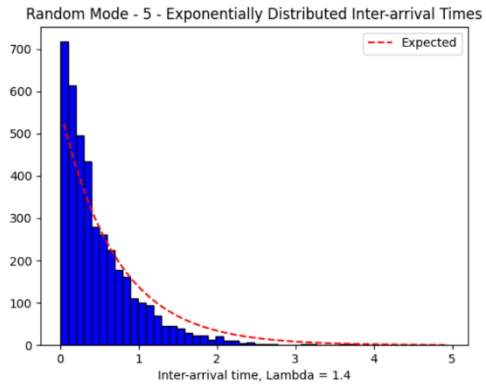
The a1k is exponentially distributed with parameter λ. The a1k is uniformly distributed in the interval of [a2l, a2u] that is provided on the test sample. The inter-arrival time of jobs is the product of a1k and a2k resulting in exponential distribution. In my code (screenshot), I used two modules. One is the random.expovariate and the other one is random.uniform.

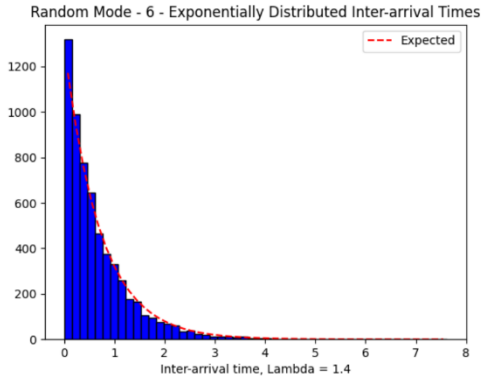
```
#  Calculate the Inter-arrival Time
def generate_inter_arrival_time(a2l, a2u, lamb):
    a1k = random.expovariate(float(lamb))
    a2k = random.uniform(a2l, a2u)
    return a1k * a2k
```

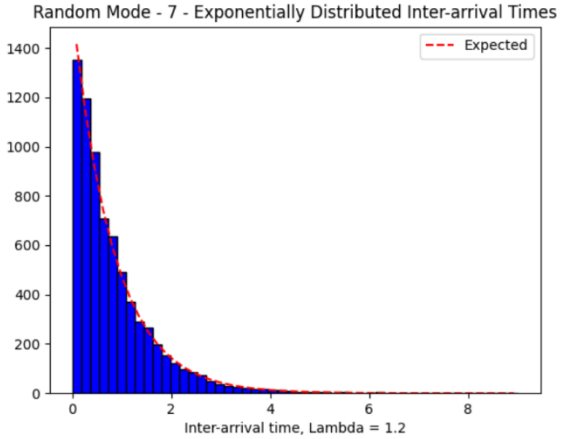
Firstly, I write a code on the main.py to generate an inter-arrival time log when running a random mode sample test (shown below) and the log will save at the support_material.

```
262     ########################################
263     #     Write logs                       #
264     ########################################
265     inter_arr_log_file = os.path.join(logMaterialFolder, "inter_arrival_" + file_number + ".txt")
266     if processing_mode == "random":
267         with open(inter_arr_log_file, "w") as file:
268             value = str(lamb) + "\n" + str(inter_arrival_value_list_store).replace("[", "").replace("]", "")
269             file.write(value)
```

Seondly, to run draw.py in support_material will generate corresponding plot to show the generated inter-arrival time to support my distribution is correct.







	Sample Test 5	Sample Test 6	Sample Test 7
Mode	Random	Random	Random
a2k (interval)	[0.6, 0.8]	[0.8, 1.020]	[0.9, 1.1]
Lambda	1.4	1.4	1.2
Number of Job (Approximately)	4034	6097	7278
End time	2000	4000	6000
Bin	50	50	50
Actual Mean value	0.4958	0.6561	0.8245
Expected Value (1/ λ)	0.7142	0.7143	0.8333

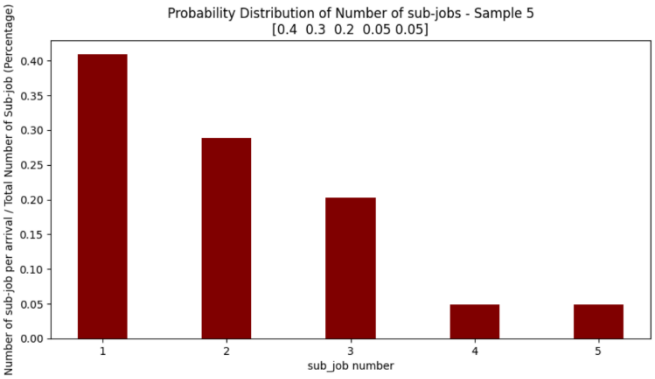
According to the graphs above, all three samples are exponentially distributed. But the end time of Simple 5 is relatively low so the number of jobs is much less than in samples 6 and 7. So the expected value and actual mean value are quite different. But overall, those three random samples are exponentially distributed. The sample 7's expected value and actual Mean Value almost the same. Then we can prove that Inter-arrival time is correct.

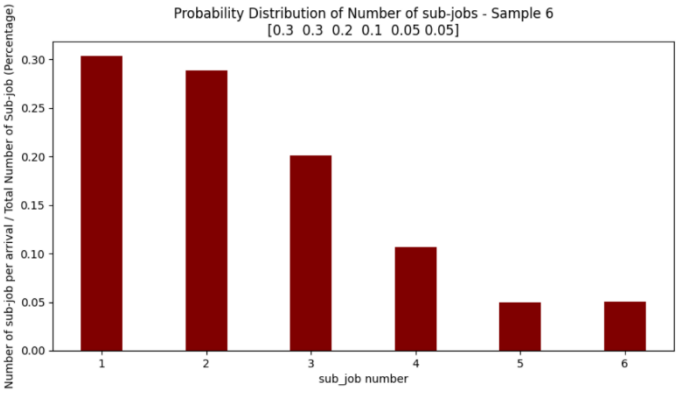
1.2. Probability Distribution of the number of Sub-Job (Random Mode)

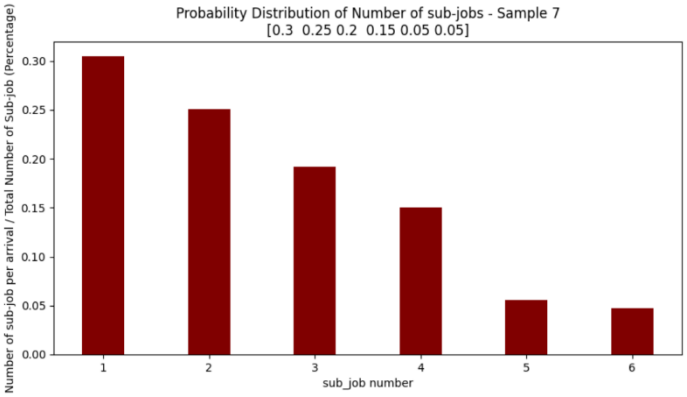
The number of sub-jobs that generate per arrival is basic on the probability sequence on interarrival_*.txt. The module that I use is "random.choice()" which will pick a number of sub-jobs base on the provided weight.

```
# Get Number of Server Base on percentage
def generate_NO_sub_job(prob_array):
    temp_sub_job = []
    temp_P_sequence = []
    for i in range(0, len(prob_array)):
        temp_sub_job.append(int(i + 1))
        temp_P_sequence.append(float(prob_array[i]))

    value = random.choices(temp_sub_job, temp_P_sequence)
    return value[0]
```

After the number of sub-jobs generated per job arrival, the number of service times will be generated according to the sub-job number and stored in "sub_job_service_time_*.txt" and stored in the support_material directory. After running the draw.py, it will run the function of "Generate_subJob_plot" to create diagrams to show the relationship of sub-job number corresponding to sub-job arrival/Total sub-job that was generated.







	Sample 5	**Sample 6**	**Sample 7**
Sub job NO	[1654,1170, 823, 197, 199]	[1868, 1777, 1239, 656, 307, 310]	[2181, 1796, 1373, 1077, 400, 339]
Total Job arrival	4046	6158	7167
Actual Percentage	[0.4088, 0.2892, 0.2034, 0.0487, 0.0492]	[0.3033, 0.2886, 0.2012, 0.1065, 0.0499, 0.0503]	[0.3043, 0.2506, 0.1916, 0.1503, 0.0558, 0.0473]
Excepted Percentage	[0.4, 0.3, 0.2, 0.05, 0.05]	[0.3, 0.3, 0.2, 0.1, 0.05, 0.05]	[0.3, 0.25, 0.2, 0.15, 0.05, 0.05]

According to the Graph and the table above, the actual percentage of sub-job that are created per arrival is very close to the excepted Percentage. By comparing samples 5 and 7. The variable of sample 5 can be up to 0.02 but the variance of Sample 7 is lower than 0.01, this is because the sample of Sample 7 is a lot more than sample 5 (7167 > 4046). The number of sample increase and the variance of the actual percentage and the excepted percentage will be lower.

We can conclude that the number of sub-jobs generated per arrival is correct and match the probability distribution provided by the sample.

1.3. Service time Distribution

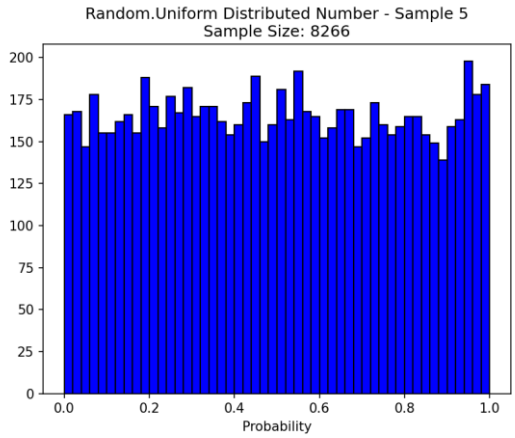
As mentioned above, the txt file that is named “sub_job_service_time_*.txt” stores the service time of each sub-job that generate by using the Cumulation Distribution Function (CDF) that provided on the project requirement.

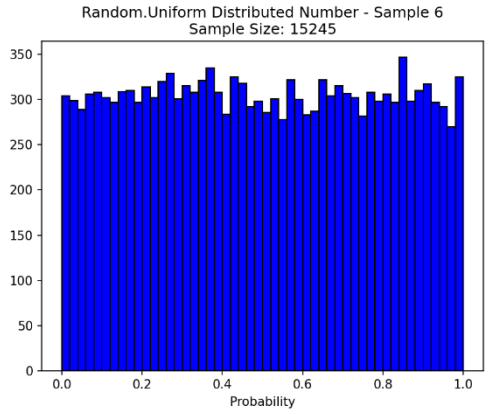
$$S(t) \quad = \quad 1 - \exp(-(\mu\, t)^{\alpha})$$

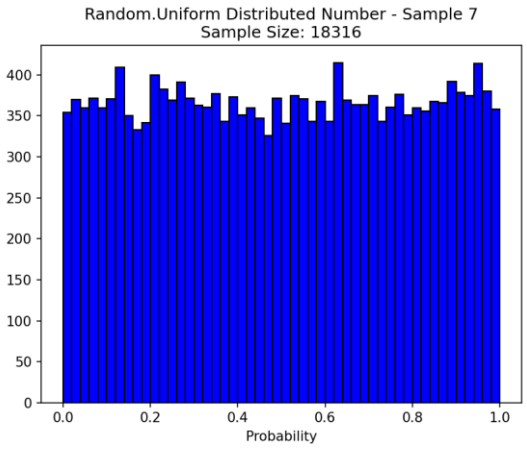
After that we need to change the equation from above to below as “t” is the value we want to get:

$$t = \frac{\sqrt[\alpha]{-\ln(1 - S(t))}}{\mu}$$

Since it’s a CDF which mean that the S(t)’s probability will locate between 0 to 1, so we generate it by using the module called “random.uniform” to uniformly generate number.

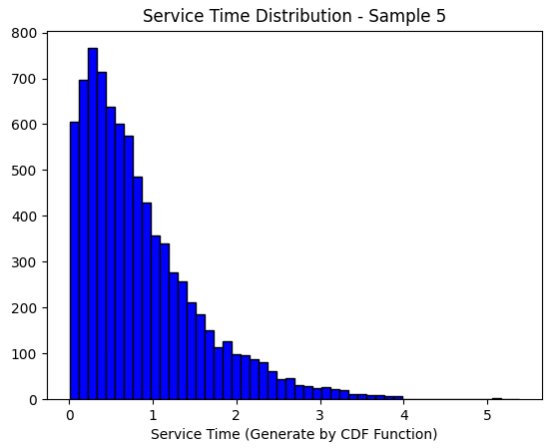


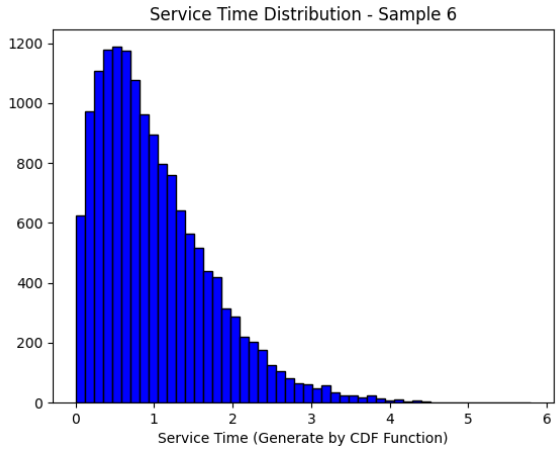


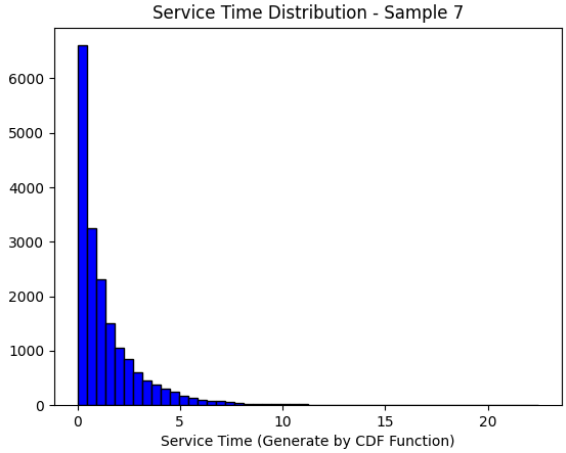


I make the bins equal to 50, so there are around 160 in each set like above for sample 5, and 300 for Sample 6 and 350 for sample 7. The Generate value is uniformly distributed as the Sample size increase.

The service time for all three sample show below:







	Sample 5	**Sample 6**	**Sample 7**
S(t)	Random.uniform(0,1)	Random.uniform(0,1)	Random.uniform(0,1)
μ	1.1	0.9	0.8
α	1.21	1.4	0.8

According to the CDF equation above with given μ, α and the Random generated S(t), we can calculate the service time t. Due to the feature of the function, there is not maximum value t but just extremely low possibility same as the lowest service time is extremely close to 0. And combine with the graph above shows the shape equation. Thus, we can prove the service time distribution correct.

1.4. Simulation Correctness (Section 4)

To verify the correctness of my simulation, I tested the section 4 case.

Sample 1 – n = 4, h = 1**:**

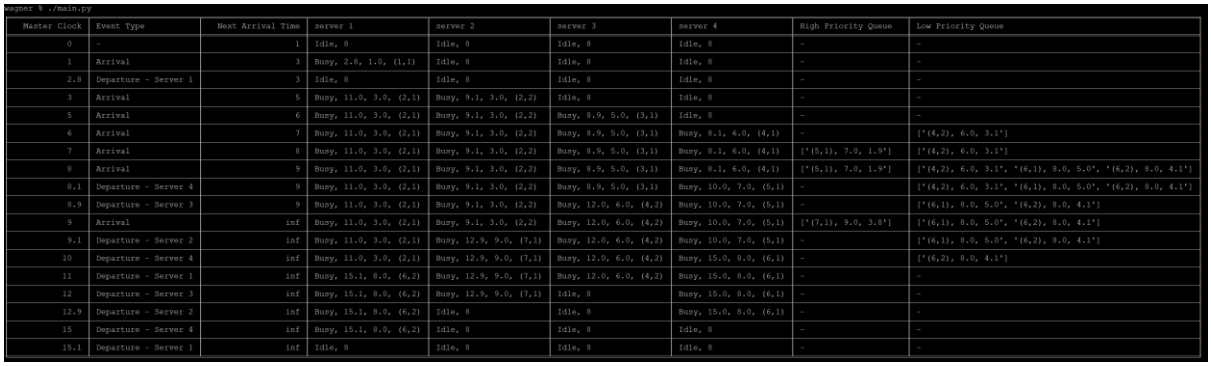
Sub-jobs	Arrival time	Departure time
(1,1)	1.0	2.8
(4,1)	6.0	8.1
(3,1)	5.0	8.9
(2,2)	3.0	9.1
(5,1)	7.0	10.0
(2,1)	3.0	11.0
(4,2)	6.0	12.0
(7,1)	9.0	12.9
(6,1)	8.0	15.0
(6,2)	8.0	15.1

Sample 2 – n = 4, h = 2**:**

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

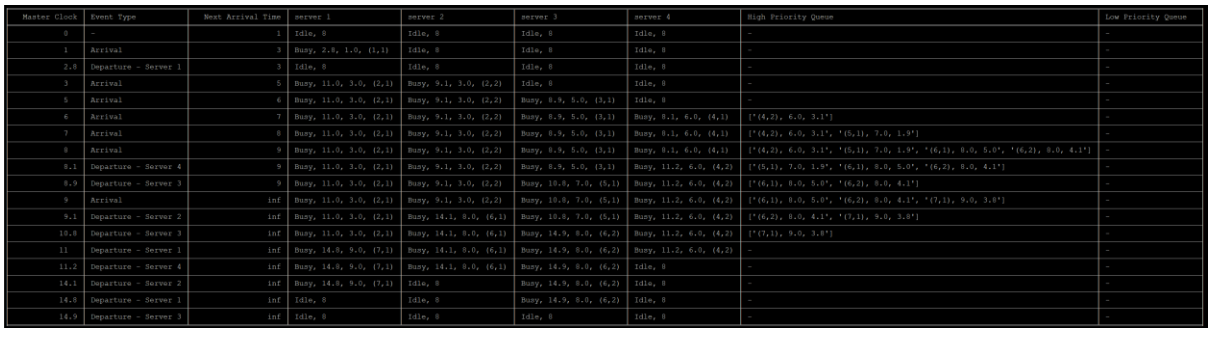
To verify the correctness of my simulation, I used the python library that called “tabulate” to create a table base on the data that I generate (New version of code removed this library). The screenshot below only for demo purpose.

Sample 1 – Server Allocation, Queueing, and departure.



Master Clock	Event Type	Next Arrival Time	server 1	server 2	server 3	server 4	High Priority Queue	Low Priority Queue
0	-	1	Idle, 0	Idle, 0	Idle, 0	Idle, 0	-	-
1	Arrival	3	Busy, 2.8, 1.0, (1,1)	Idle, 0	Idle, 0	Idle, 0	-	-
2.8	Departure - Server 1	3	Idle, 0	Idle, 0	Idle, 0	Idle, 0	-	-
3	Arrival	5	Busy, 11.0, 3.0, (2,1)	Busy, 9.1, 3.0, (2,2)	Idle, 0	Idle, 0	-	-
5	Arrival	6	Busy, 11.0, 3.0, (2,1)	Busy, 9.1, 3.0, (2,2)	Busy, 8.9, 5.0, (3,1)	Idle, 0	-	-
6	Arrival	7	Busy, 11.0, 3.0, (2,1)	Busy, 9.1, 3.0, (2,2)	Busy, 8.9, 5.0, (3,1)	Busy, 8.1, 6.0, (4,1)	-	['(4,2), 6.0, 3.1']
7	Arrival	8	Busy, 11.0, 3.0, (2,1)	Busy, 9.1, 3.0, (2,2)	Busy, 8.9, 5.0, (3,1)	Busy, 8.1, 6.0, (4,1)	['(5,1), 7.0, 1.9']	['(4,2), 6.0, 3.1']
8	Arrival	9	Busy, 11.0, 3.0, (2,1)	Busy, 9.1, 3.0, (2,2)	Busy, 8.9, 5.0, (3,1)	Busy, 8.1, 6.0, (4,1)	['(5,1), 7.0, 1.9']	['(4,2), 6.0, 3.1', '(6,1), 8.0, 5.0', '(6,2), 8.0, 4.1']
8.1	Departure - Server 4	9	Busy, 11.0, 3.0, (2,1)	Busy, 9.1, 3.0, (2,2)	Busy, 8.9, 5.0, (3,1)	Busy, 10.0, 7.0, (5,1)	-	['(4,2), 6.0, 3.1', '(6,1), 8.0, 5.0', '(6,2), 8.0, 4.1']
8.9	Departure - Server 3	9	Busy, 11.0, 3.0, (2,1)	Busy, 9.1, 3.0, (2,2)	Busy, 12.0, 6.0, (4,2)	Busy, 10.0, 7.0, (5,1)	-	['(6,1), 8.0, 5.0', '(6,2), 8.0, 4.1']
9	Arrival	inf	Busy, 11.0, 3.0, (2,1)	Busy, 9.1, 3.0, (2,2)	Busy, 12.0, 6.0, (4,2)	Busy, 10.0, 7.0, (5,1)	['(7,1), 9.0, 3.8']	['(6,1), 8.0, 5.0', '(6,2), 8.0, 4.1']
9.1	Departure - Server 2	inf	Busy, 11.0, 3.0, (2,1)	Busy, 12.9, 9.0, (7,1)	Busy, 12.0, 6.0, (4,2)	Busy, 10.0, 7.0, (5,1)	-	['(6,1), 8.0, 5.0', '(6,2), 8.0, 4.1']
10	Departure - Server 4	inf	Busy, 11.0, 3.0, (2,1)	Busy, 12.9, 9.0, (7,1)	Busy, 12.0, 6.0, (4,2)	Busy, 15.0, 8.0, (6,1)	-	['(6,2), 8.0, 4.1']
11	Departure - Server 1	inf	Busy, 15.1, 8.0, (6,2)	Busy, 12.9, 9.0, (7,1)	Busy, 12.0, 6.0, (4,2)	Busy, 15.0, 8.0, (6,1)	-	-
12	Departure - Server 3	inf	Busy, 15.1, 8.0, (6,2)	Busy, 12.9, 9.0, (7,1)	Idle, 0	Busy, 15.0, 8.0, (6,1)	-	-
12.9	Departure - Server 2	inf	Busy, 15.1, 8.0, (6,2)	Idle, 0	Idle, 0	Busy, 15.0, 8.0, (6,1)	-	-
15	Departure - Server 4	inf	Busy, 15.1, 8.0, (6,2)	Idle, 0	Idle, 0	Idle, 0	-	-
15.1	Departure - Server 1	inf	Idle, 0	Idle, 0	Idle, 0	Idle, 0	-	-

Sample 2 – Server Allocation, Queueing, and departure.



Master Clock	Event Type	Next Arrival Time	server 1	server 2	server 3	server 4	High Priority Queue	Low Priority Queue
0	-	1	Idle, 0	Idle, 0	Idle, 0	Idle, 0	-	-
1	Arrival	3	Busy, 2.8, 1.0, (1,1)	Idle, 0	Idle, 0	Idle, 0	-	-
2.8	Departure - Server 1	3	Idle, 0	Idle, 0	Idle, 0	Idle, 0	-	-
3	Arrival	5	Busy, 11.0, 3.0, (2,1)	Busy, 9.1, 3.0, (2,2)	Idle, 0	Idle, 0	-	-
5	Arrival	6	Busy, 11.0, 3.0, (2,1)	Busy, 9.1, 3.0, (2,2)	Busy, 8.9, 5.0, (3,1)	Idle, 0	-	-
6	Arrival	7	Busy, 11.0, 3.0, (2,1)	Busy, 9.1, 3.0, (2,2)	Busy, 8.9, 5.0, (3,1)	Busy, 8.1, 6.0, (4,1)	['(4,2), 6.0, 3.1']	-
7	Arrival	8	Busy, 11.0, 3.0, (2,1)	Busy, 9.1, 3.0, (2,2)	Busy, 8.9, 5.0, (3,1)	Busy, 8.1, 6.0, (4,1)	['(4,2), 6.0, 3.1', '(5,1), 7.0, 1.9']	-
8	Arrival	9	Busy, 11.0, 3.0, (2,1)	Busy, 9.1, 3.0, (2,2)	Busy, 8.9, 5.0, (3,1)	Busy, 8.1, 6.0, (4,1)	['(4,2), 6.0, 3.1', '(5,1), 7.0, 1.9', '(6,1), 8.0, 5.0', '(6,2), 8.0, 4.1']	-
8.1	Departure - Server 4	9	Busy, 11.0, 3.0, (2,1)	Busy, 9.1, 3.0, (2,2)	Busy, 8.9, 5.0, (3,1)	Busy, 11.2, 6.0, (4,2)	['(5,1), 7.0, 1.9', '(6,1), 8.0, 5.0', '(6,2), 8.0, 4.1']	-
8.9	Departure - Server 3	9	Busy, 11.0, 3.0, (2,1)	Busy, 9.1, 3.0, (2,2)	Busy, 10.8, 7.0, (5,1)	Busy, 11.2, 6.0, (4,2)	['(6,1), 8.0, 5.0', '(6,2), 8.0, 4.1']	-
9	Arrival	inf	Busy, 11.0, 3.0, (2,1)	Busy, 9.1, 3.0, (2,2)	Busy, 10.8, 7.0, (5,1)	Busy, 11.2, 6.0, (4,2)	['(6,1), 8.0, 5.0', '(6,2), 8.0, 4.1', '(7,1), 9.0, 3.8']	-
9.1	Departure - Server 2	inf	Busy, 11.0, 3.0, (2,1)	Busy, 14.1, 8.0, (6,1)	Busy, 10.8, 7.0, (5,1)	Busy, 11.2, 6.0, (4,2)	['(6,2), 8.0, 4.1', '(7,1), 9.0, 3.8']	-
10.8	Departure - Server 3	inf	Busy, 11.0, 3.0, (2,1)	Busy, 14.1, 8.0, (6,1)	Busy, 14.9, 8.0, (6,2)	Busy, 11.2, 6.0, (4,2)	['(7,1), 9.0, 3.8']	-
11	Departure - Server 1	inf	Busy, 14.8, 9.0, (7,1)	Busy, 14.1, 8.0, (6,1)	Busy, 14.9, 8.0, (6,2)	Busy, 11.2, 6.0, (4,2)	-	-
11.2	Departure - Server 4	inf	Busy, 14.8, 9.0, (7,1)	Busy, 14.1, 8.0, (6,1)	Busy, 14.9, 8.0, (6,2)	Idle, 0	-	-
14.1	Departure - Server 2	inf	Busy, 14.8, 9.0, (7,1)	Idle, 0	Busy, 14.9, 8.0, (6,2)	Idle, 0	-	-
14.8	Departure - Server 1	inf	Idle, 0	Idle, 0	Busy, 14.9, 8.0, (6,2)	Idle, 0	-	-
14.9	Departure - Server 3	inf	Idle, 0	Idle, 0	Idle, 0	Idle, 0	-	-

The Generated Tables above show each sub job’s arrival and departure as well as the sub-job ID. And it matches the table provided in Section 4 on the Project requirement.

With in my code, I store the sub-job arrival and departure time and apply an algorithm to find the response time that use the most for each sub job to get respond time of a job.

```
########################################
#     Convert Stored Data to Output    #
########################################
temp_counter = 0
output_sub_job_departure = ""
# Output Departure String
for arr in temp_output_sub_job_departure:
    output_sub_job_departure = output_sub_job_departure + str(arr[0]) + " " * 3 + str(arr[1]) + "\n"
    temp_counter = temp_counter + 1

# Calcualte Response Time
myDic = {}
output_response_time = 0

# Find all the sub-job of a same job but compare the departure value to find the job response time of each job
for i in temp_output_sub_job_departure:
    if i[0] not in myDic:
        myDic[i[0]] = i
    else:
        if float(myDic[i[0]][1]) < float(i[1]):
            myDic[i[0]] = i

# loop the Dic and get the total response time
for i in myDic:
    output_response_time = output_response_time + float(myDic[i][1]) - float(myDic[i][0])
    job_service_time_Store.append(float(myDic[i][1]) - float(myDic[i][0]))

########################################
#     Write Output to file             #
########################################
dep_file = os.path.join(outputFolder, 'dep_' + file_number + '.txt')
mrt_file = os.path.join(outputFolder, 'mrt_' + file_number + '.txt')

# Write Txt to file
with open(dep_file, "w") as file:
    file.write(output_sub_job_departure)

with open(mrt_file, "w") as file:
    file.write(str(round(output_response_time / no_job_arrival, 4)))
```

The sub-job's arrival and departure and the Mean response time of each sample will store in the "output" directory accordingly. And then to run the compare python script that provided name as "cf_output_with_ref.py" to compare the value.

```
wagner % ./cf_output_with_ref.py 1
Test 1: Mean response time matches the reference
Test 1: Departure times match the reference
wagner % ./cf_output_with_ref.py 2
Test 2: Mean response time matches the reference
Test 2: Departure times match the reference
wagner % ./cf_output_with_ref.py 3
Test 3: Mean response time matches the reference
Test 3: Departure times match the reference
wagner % ./cf_output_with_ref.py 4
Test 4: Mean response time matches the reference
Test 4: Departure times match the reference
```

2. Reproducibility

To test the Reproducibility of my simulation program, I will use random mode and the same input value to run the Simulation 100 times (Sample 7 input Value). In theory, I will get a Mean Response time with a very small variance value.

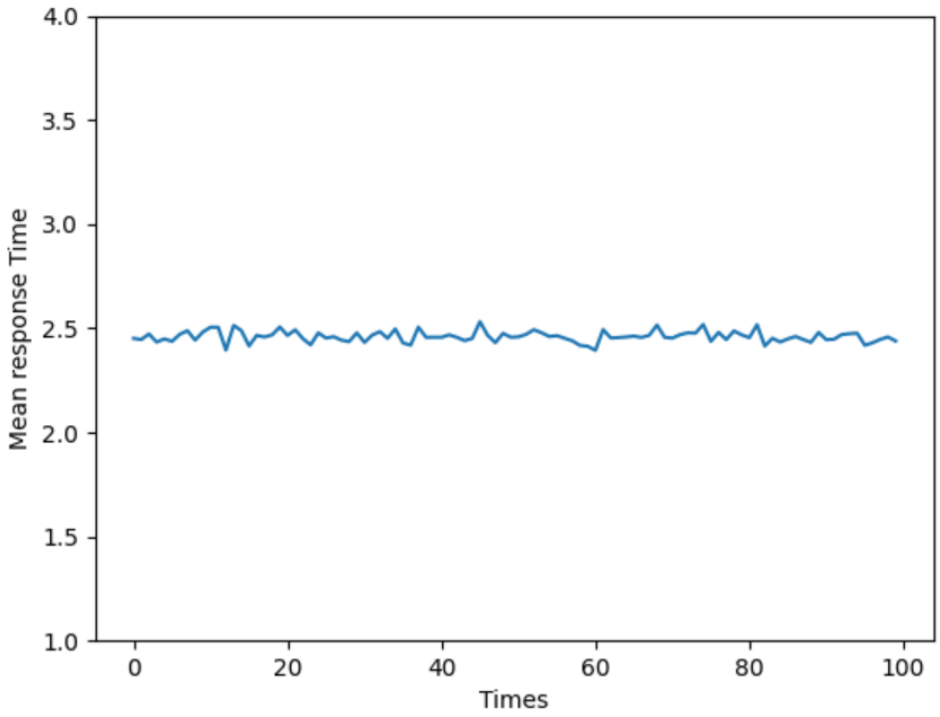
To proof that, I choose the data below:

Parameter	**Value**
Mode	Random
Server Number	10
Threshold	3
End time	6000
Lambda	1.2
A2l	0.9
A2u	1.1
Probability Sequence	[0.300, 0.250, 0.200, 0.150, 0.050, 0.050]
Mu	0.8
Alpha	0.8

Coding Show below, the coding for the Reproducibility will be comment out and will not be execute after. But can be uncomment and the code can be re-execute after.

```
if __name__ == "__main__":
    # main(str(sys.argv[1]))
    #
    # main("100")

    #### Reproducibility Test
    logMaterialFolder = "support_material"
    Reproducibility_log_file = os.path.join(logMaterialFolder, "Reproducibility_random.txt")
    mrt_list = []
    for i in range(0, 100):
        print(i+1)
        output_departure, mrt = main("7")
        mrt_list.append(mrt)

    with open(Reproducibility_log_file, "w") as file:
        value = mrt_list
        file.write(str(value))
```

I use above data to run main.py 100 times. As shown in the blow figure, the mean response time with small variance but almost identical in these 100 times. Thus, given the same parameter, my simulation program is reliable and correct.



The lower bound and the upper bound of this reproducibility is [2.3939, 2.5315] and both extreme value of the 100 times test can all pass the "cf-output_with_ref.py" test.

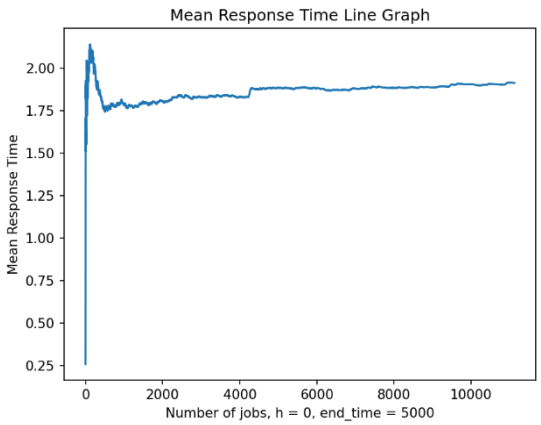
3. Determining a suitable value of the threshold h

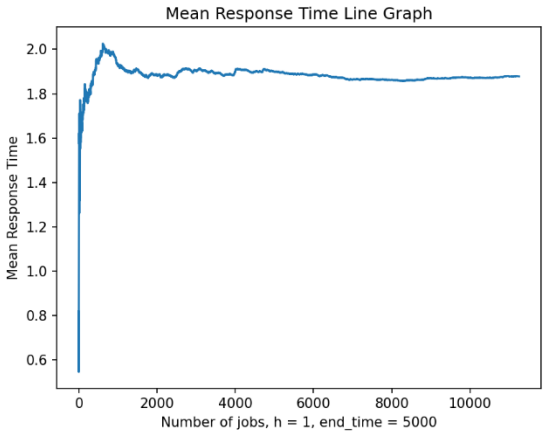
3.1. Transient Behaviour and Removal

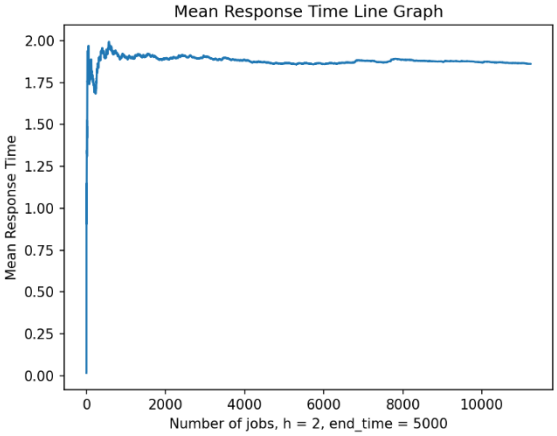
There is a number of methods to analyze the data through a statistic. Since the transient may create variance so we mainly focus on the steady state. And finding the steady state mean response time is more reliable for the result. The parameter that I use shows as follows:

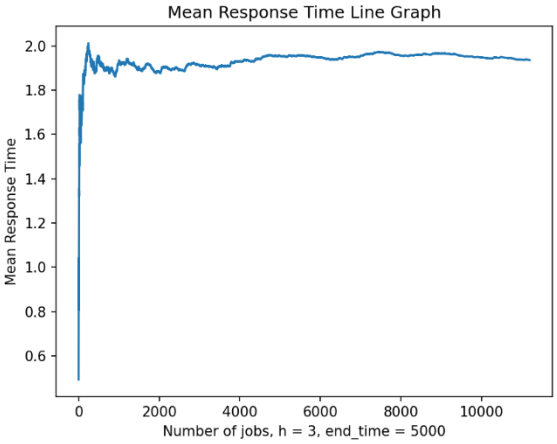
Note: The threshold is used to determine the size of the sub-job per arrival, and the maximum number of sub-jobs per arrival is 5, so the threshold is limited to [0, 5]. And the end time is not provided in the reporting requirement, but the lower end time is the result of a low sample size that is not suitable for finding the Mean response time.

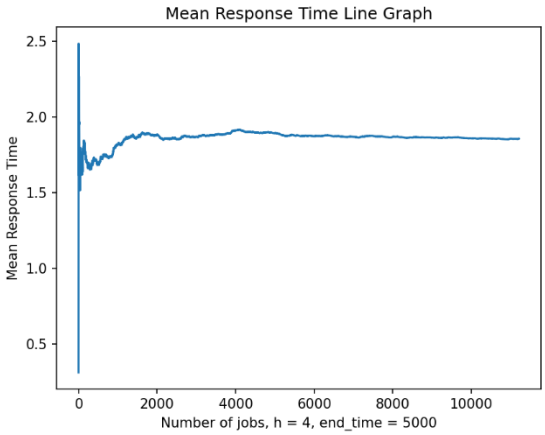
Parameter	**Value**
Number of servers	10
Lambda	1.8
A2l	0.7
A2u	0.9
Sub-jobs per job and sequence	0.4, 0.25, 0.15, 0.11, 0.09
End time	5000
Threshold	0/1/2/3/4/5

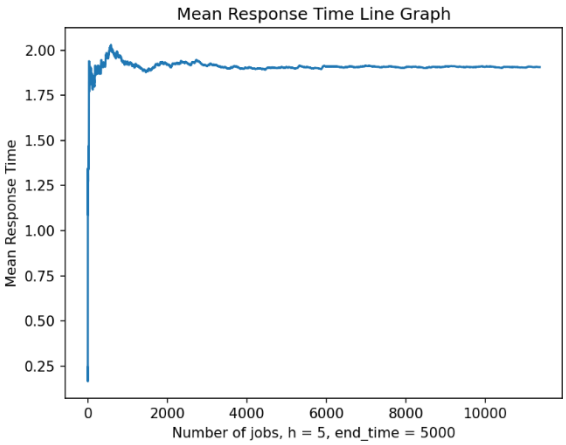

Mean Response Time Line Graph
Mean Response Time
Number of jobs, h = 0, end_time = 5000


Mean Response Time Line Graph
Mean Response Time
Number of jobs, h = 1, end_time = 5000


Mean Response Time Line Graph
Mean Response Time
Number of jobs, h = 2, end_time = 5000


Mean Response Time Line Graph
Mean Response Time
Number of jobs, h = 3, end_time = 5000


Mean Response Time Line Graph
Mean Response Time
Number of jobs, h = 4, end_time = 5000


Mean Response Time Line Graph
Mean Response Time
Number of jobs, h = 5, end_time = 5000

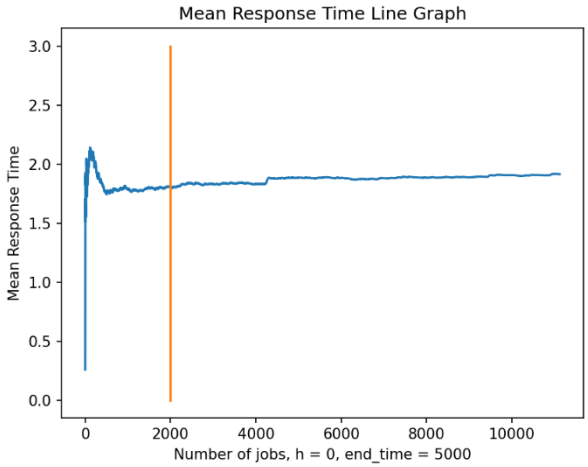
The line graphs above generate by using the following formula:

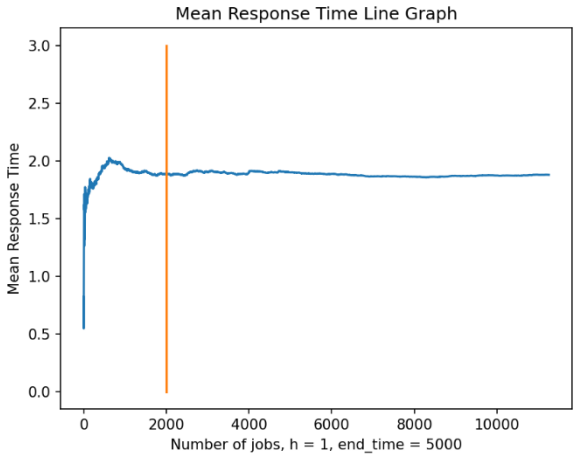
$$M(k) = \frac{X(1)+X(2)+\ldots+X(k)}{k}$$

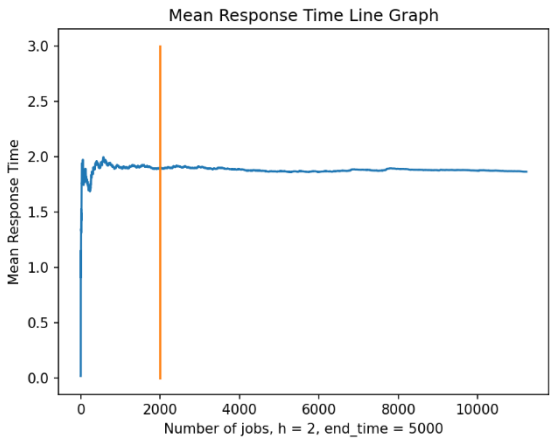
And the "k" represents the k_{th} of jobs.

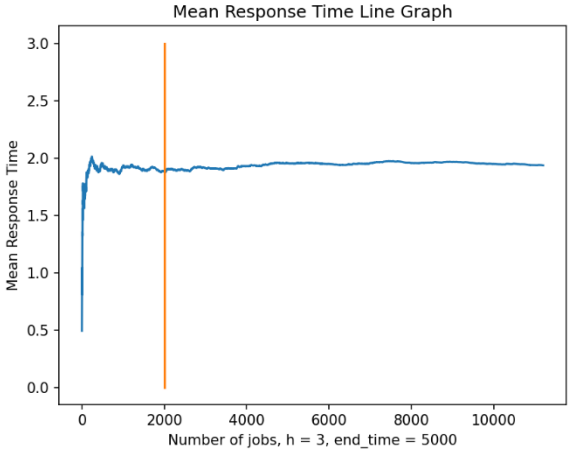
From the 6 graphs above, we can see that the early part of the simulation displays an unusual peak when h = 0, 1, 4, and 5 (without removing the transients). And the later part of the simulation will use the following formula to remove the fluctuates part.

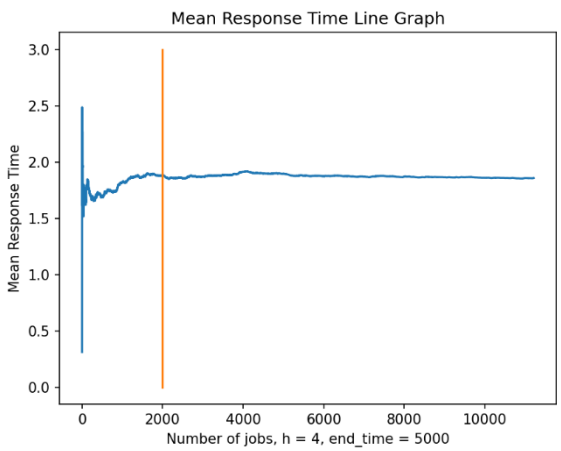
$$\frac{X(m+1)+X(m+2)+\ldots+X(N)}{N-m}$$

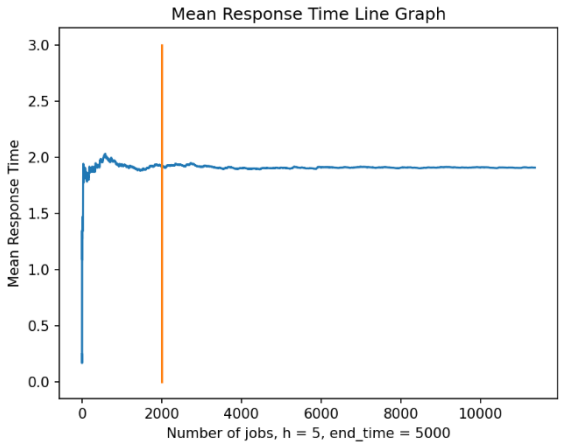








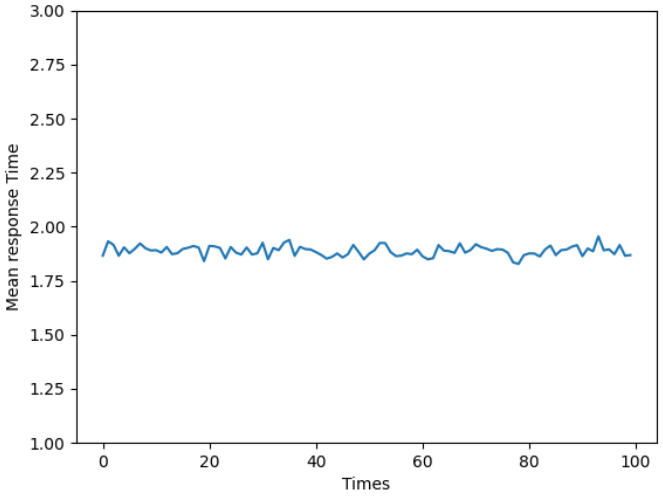




The Orange line refers to the first 2000 number of jobs that need to be removed to get a steady state of mean response time. By comparing all 6 line-graphs, as the number of job increase, the Mean response time is getting more stable. From the graphs above, we can also see that the length of the simulation must be larger than the sample size of jobs equal to 2000. This is because the fluctuation is still relatively big before 2000 and we will not get a relatively stable result if the sample size is small.

3.2. Number of Replications

Use threshold = 3 as an example. I repeat the experiment 100 times using different sets of random numbers. As shown below, there are still some variances, even though the variance is not too big as the upper and lower bound difference is 1.835181289200024 and 1.956473680167565, but potentially will affect the result.



Replicating the simulation multiple times in result a more stable Mean response time leads to a more reliable and stable result.

The provided diagram on the left-hand side repeated the simulation 100 times. And we get a mean value of all 100 mean response time, the value is 1.8889. Similarly, we use 50 replications and the result is 1.8885 which only have 0.0004 variance and in terms of program efficiency and accuracy, choosing a replicant number of 50 is enough as the majority of the extreme situation will appear in the result.

3.3. Computing the confidence Interval

From above 3.2 number of replications, we found out the even though the threshold and other parameter are the same (The number of sub-job and service time of each sub-job are randomly generated), the Mean response time will have some variance. In each replication, we remove the transient part and compute the estimate of the mean steady state response time. Which mean we will use the following function:

$$\hat{T} = \frac{\sum_{i=1}^{n} T(i)}{n}$$

T(i) refers to the estimate from the i^{th} replication.

And we also calculate the sample standard deviations by using the following function:

$$\hat{S} = \sqrt{\frac{\sum_{i=1}^{n} (\hat{T} - T(i))^2}{n-1}}$$

And at the end, there is a probability (1-α) that the mean response time that used to estimate lies in the interval.

$$[\hat{T} - t_{n-1,1-\frac{\alpha}{2}} \frac{\hat{S}}{\sqrt{n}}, \hat{T} + t_{n-1,1-\frac{\alpha}{2}} \frac{\hat{S}}{\sqrt{n}}]$$

where $t_{n-1,1-\frac{\alpha}{2}}$ is the upper $(1 - \frac{\alpha}{2})$

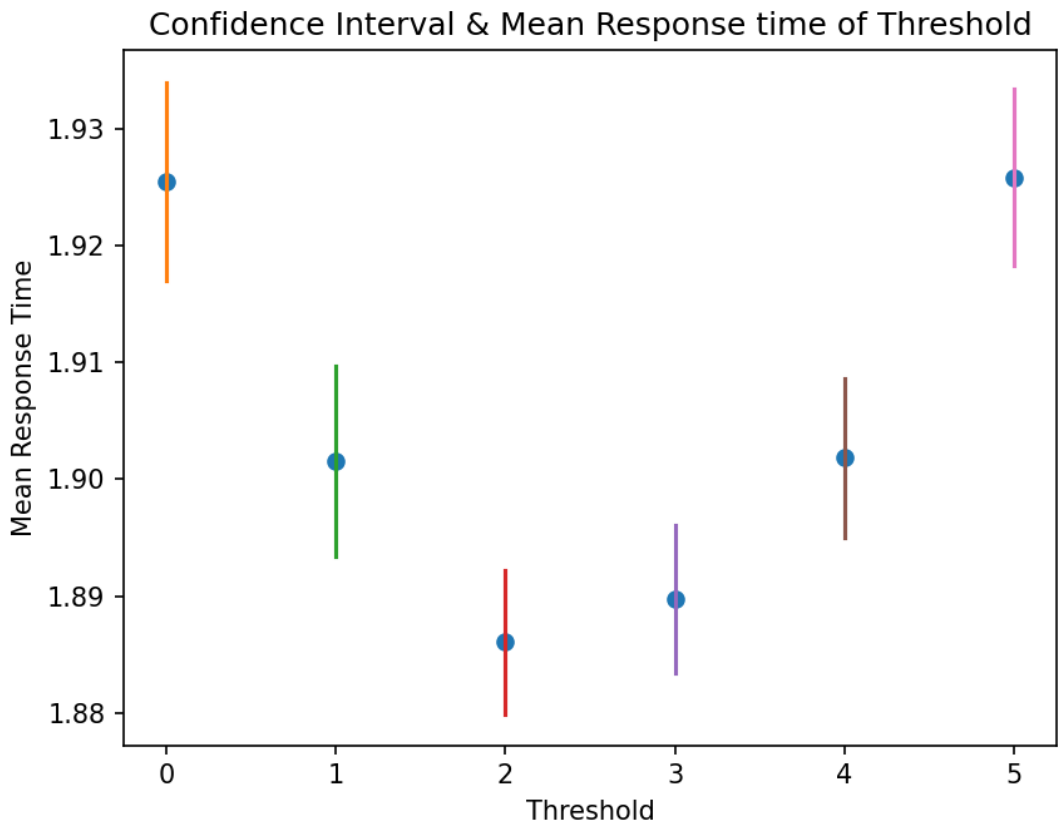
In the project, the T(i) is the mean response time with a different number of replications range of [0, 50]. And we want 95% of confidence interval.

```python
if __name__ == "__main__":
    x = [[0, 0], [1, 1], [2, 2], [3, 3], [4, 4], [5, 5]]
    y_plot = []
    mean_MRT = []
    for threshold in range(0, 6):
        mrt_list = []
        for i in range(0, 50):
            print(i)
            a, b, steady_state_mrt = main(threshold)
            mrt_list.append(steady_state_mrt)

        alp = 0.05
        mean_mrt = np.mean(mrt_list)
        std_mrt = np.std(mrt_list, ddof=1)
        n = len(mrt_list)
        mf = stats.t.ppf(1 - alp / 2, n - 1) / np.sqrt(n)
        confidence_interval = mean_mrt + np.array([-1, 1]) * mf * std_mrt

        thr_interval = [confidence_interval[0], confidence_interval[1]]
        y_plot.append(thr_interval)
        mean_MRT.append(mean_mrt)
        print(mean_mrt)
        print('confidence interval: ', confidence_interval)

    plt.plot(mean_MRT, "o")
    for i in range(0, len(x)):
        plt.plot(x[i], y_plot[i])

    plt.title("Confidence Interval & Mean Response time of Threshold")
    plt.ylabel("Mean Response Time")
    plt.xlabel("Threshold")
    plt.show()
```

Within the support_material directory, there is a python script called “sim_test.py” used to calculate the confidence interarrival value by looping the threshold value from 0 to 5. It will run the random simulation 50 times with pre-set parameters value and get the steady mean response time and store in mrt_list.

Then mean of the mean report time to get an overall mean response time to minimize the variance and use it to calculate the confidence interval.

Threshold	Mean T of 50 replications	Confidence Interval
0	1.925460316793146	[1.91693946 1.93398117]
1	1.9015295186415293	[1.89338533 1.90967371]
2	1.8860906332417642	[1.8799263 1.89225497]
3	1.8897375305235569	[1.8833942 1.89608086]
4	1.9018533701400235	[1.89504868 1.90865806]
5	1.9257986862392387	[1.91823261 1.93336477]



When threshold equal 0 and 5, their mean response time and confidence interval are overlap. This is because when they generate number of sub jobs, they all assign to either high priority or low priority queue.

When Threshold equal 1 and 4, there is only one type of job will assign to either high or low priority list which is also not efficient.

The best Mean response result is when threshold equal 2 and 3.

Conclusion:

The threshold at 2 and 3 can make the system perform better in result of a lowest response time when there are maximum of 5 number of sub jobs. But according to the threshold compare diagram, we can see that when threshold will get the best performance as the mean response time and confidence interval is lower than when threshold equal 3.