COMP9334 Project Report

Server setup in data centres

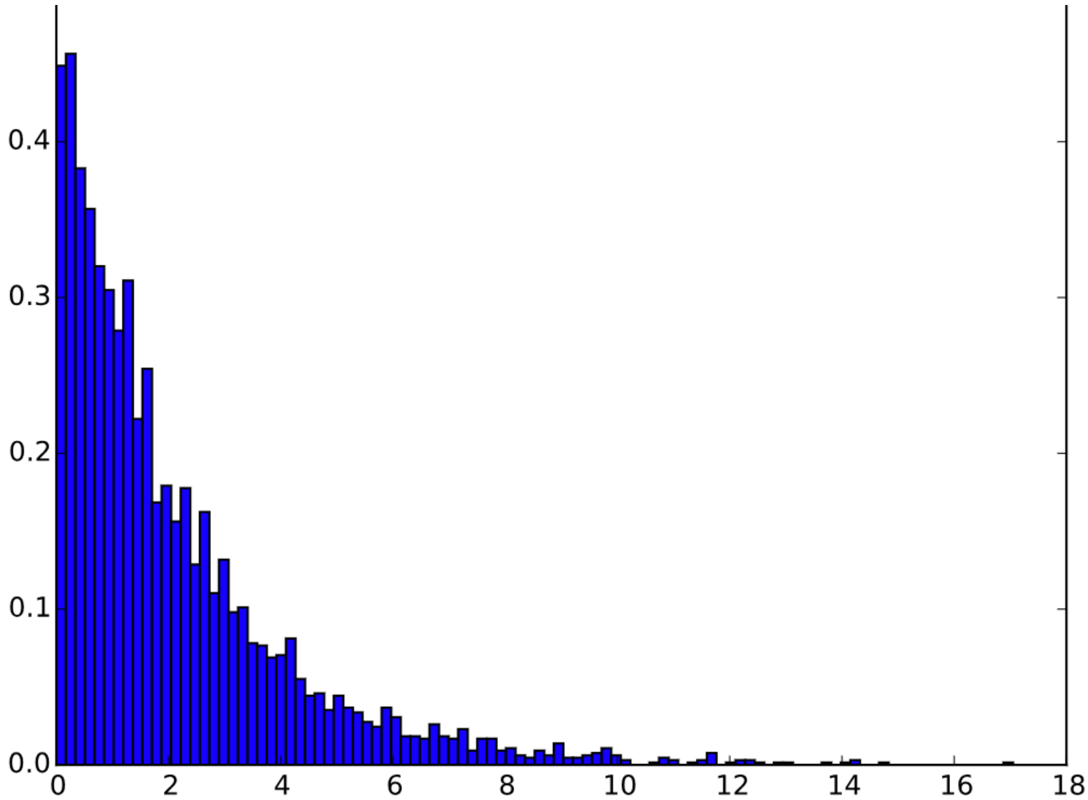
z5102866 Jinzhu WU

This project is designed to use simulation to perform the operation of the computer system. We need to handle arrivals and departures, then get the mean response time and departure time. And we need to choose the value of Tc for the improved system. This report will verify the following criteria.

1. The correctness of the inter- arrival probability distribution and service time distribution.

```
if mode == "random":
    lambda_value = float(read_value(arrival_name))
    mu_value = float(read_value(service_name))
    para = read_data(para_name)
    end_time = para[3]
    arrival = []
    service = []
    curr = 0
    flag = True
    inter_arrival = []
    while flag == True:
        inter = random.expovariate(lambda_value)
        curr += inter
        if curr < end_time:
            arrival.append(curr)
            inter_arrival.append(inter)
        else:
            flag = False
    job_number = len(arrival)
    for i in range(job_number):
        t = 0
        for j in range(3):
            t += random.expovariate(mu_value)
        service.append(t)
```

As the requirements said, the inter-arrival probability distribution is exponentially distributed with lambda. So I use *random.expovariate(lambda)* to generate the inter-arrival times and use *random.expovariate(mu)* to generate the random numbers and sum each three of them as service time. The numbers generated by *random.expovariate()* are obviously exponentially distributed. The plot graph for random.expovariate shows as:



2. The correctness of my simulation code.

In “Trace” mode, I try to derive test cases in Section 3.2 to test my code. The arrival time and service time are in Table1.

Arrival time	Service time
10	1
20	2
30	3
33	4

Table 1: Example 1: Job arrival and service times.

After running my program, we get the departure.txt, which shows as:

departure_1.txt

```
10.000   61.000
20.000   63.000
32.000   66.000
33.000   70.000
```

These results have verified the Table 2 which shows the on-paper simulation with explanatory comments in example.

In “random” mode, I produce the inter-arrival time which then transforms to arrival time, as well as I produce service time, and store them in lists called ‘arrival’ and ‘service’.

```python
    if mode == "random":
        lambda_value = float(read_value(arrival_name))
        mu_value = float(read_value(service_name))
        para = read_data(para_name)
        end_time = para[3]
        arrival = []
        service = []
        curr = 0
        flag = True
        while flag == True:
            inter = random.expovariate(lambda_value)
            curr += inter
            if curr < end_time:
                arrival.append(curr)
            else:
                flag = False
        job_number = len(arrival)
        for i in range(job_number):
            t = 0
            for j in range(3):
                t += random.expovariate(mu_value)
            service.append(t)
        processing(arrival, service,para,number)
```

```python
def processing(arrival,service,para,number):
```

Running the function processing() with these data, the departure.txt and mrt.txt produced. In this simulation, I choose seed = 1 to generate random numbers, so simulation experiments are reproducible.

```python
def main(arrival_name,service_name,para_name,mode_name,number):
    mode = read_value(mode_name)
    random.seed(1)
```

I used a sample para.txt with the following parameter values: the number of servers is 5, setup time is 5, λ = 0.35, μ = 1, T_c = 0.1, and endtime = 500, the partial results are showed below:

departure_3.txt

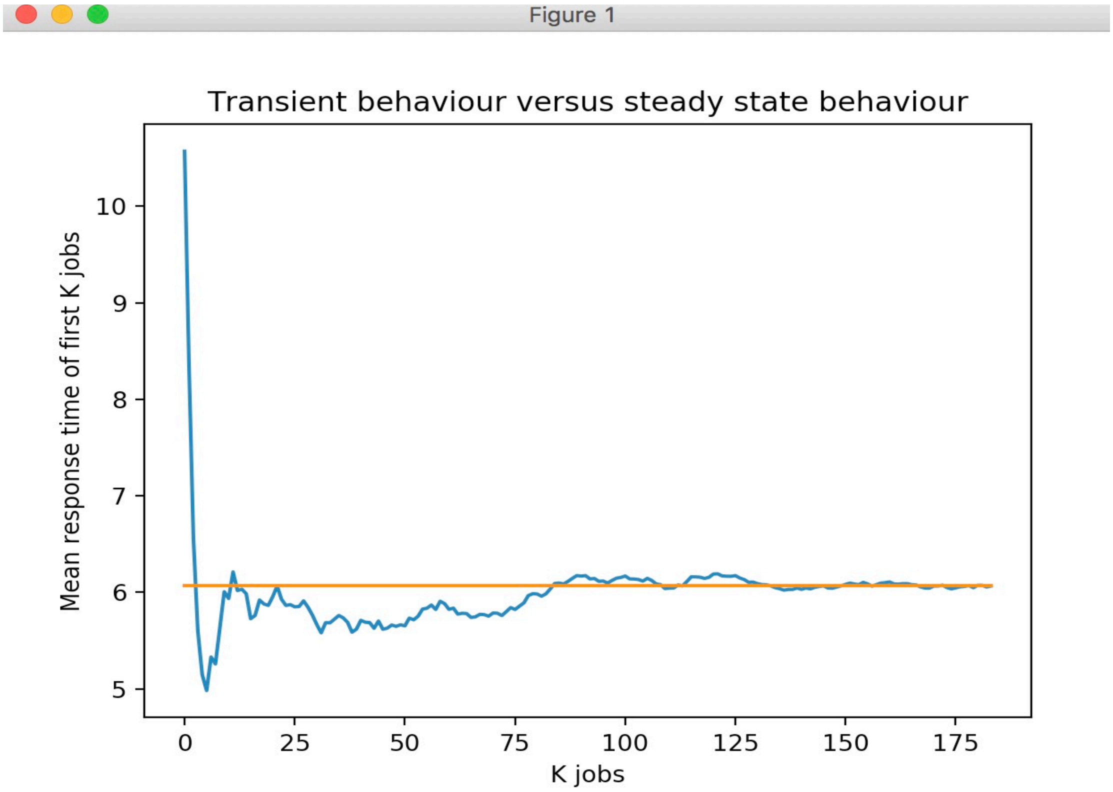
```
0.412   10.982
5.784   11.915
9.907   12.886
10.748  13.473
12.703  16.025
14.408  18.567
17.421  24.837
21.862  26.613
22.226  30.827
22.144  31.544
29.007  34.238
27.387  36.622
33.112  36.817
33.118  39.336
34.802  40.108
39.197  41.059
38.455  44.750
47.498  56.137
54.118  59.272
54.281  59.856
54.207  61.962
56.508  64.909
64.506  67.392
65.878  70.273
66.575  72.655
68.226  73.508
68.942  74.889
68.142  75.629
72.545  76.547
73.303  76.699
```

departure_3.txt

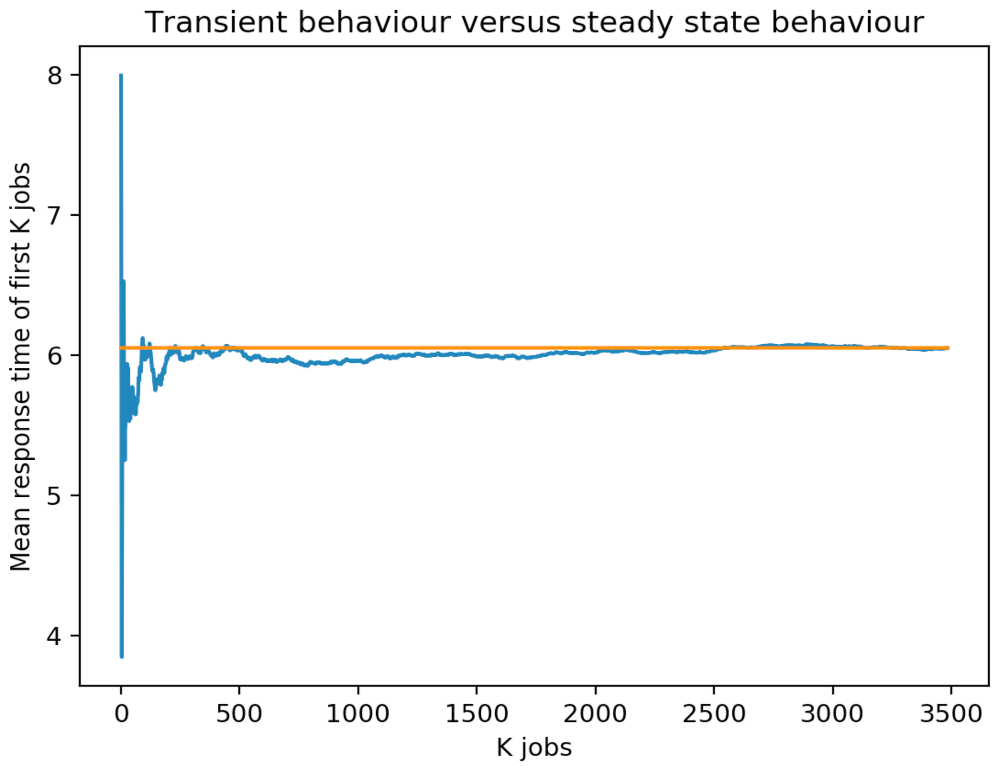
```
397.531 401.251
400.770 403.145
403.018 411.632
403.730 412.324
414.339 421.075
418.906 426.218
424.692 427.380
421.705 427.483
420.983 427.942
426.127 432.199
429.685 434.066
428.578 434.135
432.533 435.204
433.719 437.783
432.706 438.529
443.545 453.343
450.544 456.680
449.499 457.034
456.131 457.917
457.193 460.793
465.198 473.085
469.089 477.117
471.482 478.284
470.626 478.289
477.620 479.674
471.457 481.975
477.509 483.559
482.510 485.429
491.868 499.776
```

3. Transient removal

Response time continuously changes over time, in order to see a steady state, I compute the running mean by python file ‘graph.py’. The graph produced as:

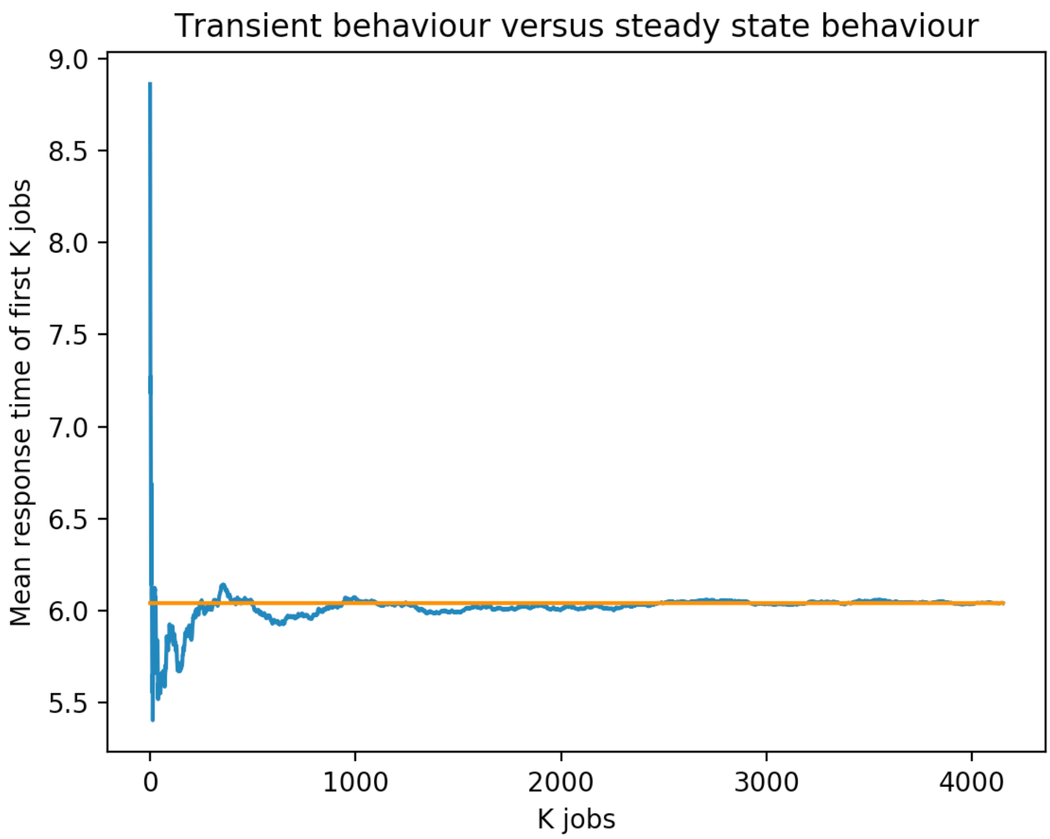


For further test, I try to run the simulation long enough so that I have a good number of jobs in the steady state part. I used a sample para.txt with the following parameter values: the number of servers is 5, setup time is 5, λ = 0.35, μ = 1, T_c = 0.1 and ***endtime = 10000***. The graph shows as:



In this simulation, the length of steady state is not longer than the transient.

So, I chang the endtime into *12000*. The result shows as:



By observation, we can find that after 1600 jobs, the response time seems that it keeps in a steady state value. I get steady mean response time from file graph.py, the value is almost 6.0559541.

4. Determining a suitable value of T_c

In previous simulations, our baseline system uses the following parameter values: the number of servers is 5, setup time is 5, $\lambda = 0.35$, $\mu = 1$, $T_c = 0.1$ and endtime = 12000. This baseline system will give a poor response time because the servers have to be powered up again frequently.

In order to design an improved system, firstly, I have run some simulations with different Tc. And I get following data:

T_c	Steady mean response time
0.1	6.0559541
1	5.6779984
5	4.6295243
7	4.2951698
8	4.1490463
9	4.0593085
9.5	4.0112207

The improved system's response time must be 2 units less than that of the baseline system, so we can consider that the suitable value of T_c is around a value of 9. (The confidence interval produces by cacluate.py.)

Now we start with 5 replications by using **baseline system** (choose seed from 1 to N):

Number of replications	True mean response time	Confidence interval
5	6.0514517681548	(6.007814333636493, 6.095089202673107)
10	6.0478623482776	(5.941889661081383, 6.153835035474007)
20	6.0631314161020	(5.958067528253906, 6.168195303950132)

Then we start with 5 replications by using $\mathbf{T_c = 9}$ (We will call this System1):

Number of replications	True mean response time	Confidence interval
5	4.0649336846382	(3.98223876453532, 4.147628604741029)
10	4.0707949466168	(3.96356652532867, 4.178023367904941)

It obviously shows that, if [p,q] stands for 95% Confidence interval of EMRT System 1 - EMRT System baseline, p and q both less than 0. It means that System 1 is better than baseline system.

For getting better T_c which satisfy the requirement, I need to compare System 1 with others. We start with 5 replications by using $\mathbf{T_c = 9.5}$ (We will call this System2):

Number of replications	True mean response time	Confidence interval
5	4.0291085530387	(3.9515833779643, 4.1066337281131355)
10	4.0335103841728	(3.9354054145918, 4.1316153537538485)

Comparing Systesm2 with System1:

Independent replications	95% Confidence interval of EMRT System 2 - EMRT System1
5	(-0.03065538657101996, -0.04099487662789336)
10	(-0.02816111073677119, -0.0464080141510923)

Hence, System2 is better than System1.

Repeating the above comparison and increasing the number of replications, we can get better system. In this case, T_c = 9.5 meet expectations.