

# Laboratory Report

Nº 2

## Performance evaluation



### *Group 6*

*Félix Saraiva – 98752*

*Miguel Ribeiro – 87553*

*Miguel Dias Gaio – 82608*

## Table of Contents

Table of Figures:.....	3
List of Tables: .....	5
Objectives.....	6
A. Discrete-time Markov chains.....	6
Exercise 1 .....	6
Exercise 2 .....	8
Exercise 3 .....	8
Part a).....	8
Part b).....	10
B. Continuous-time Markov chains.....	11
Exercise 4 .....	11
C. Queuing systems.....	13
Exercise 5 .....	13
Exercise 6 .....	14
Part a).....	14
Part b).....	15
Part c).....	17
D. Workloads .....	19
Exercise 7 .....	19
Part a).....	19
Part b).....	20
Part c) .....	20
Exercise 8 .....	22
E. Server Farms .....	24
Exercise 9 .....	24
F. Packet scheduling .....	25
Exercise 10 .....	25
G. Packet switched networks .....	27
Exercise 11 .....	27
Exercise 12 .....	30

Exercise 13 .....	34
Part a).....	34
Part b).....	36
Part c).....	37
Part d).....	38
H. Circuit switched networks .....	41
Exercise 14 .....	41
.....	41
Part a).....	41
Part b).....	43
Part c).....	44
References .....	45
Annex A .....	46
Annex B .....	47
Annex C .....	48
Annex D.....	50
Annex E .....	53
Annex F.....	56
Annex G.....	59

## Table of Figures:

Figure 1 - 2-DTMC Process.....	6
Figure 2 - 2- DTMC vs Bernoulli Process ( $\alpha = 0.5$ ; $\beta = 0.5$ ).....	6
Figure 3 - 2- DTMC vs Bernoulli Process ( $\alpha = 0.9$ ; $\beta = 0.1$ ).....	7
Figure 4 - Performance of slotted ALOHA graph (N=10) .....	8
Figure 5 - Performance of slotted ALOHA graph (N=25) .....	9
Figure 6 - CTMC.....	11
Figure 7 - Limiting state probabilities equations .....	11
Figure 8 - CTMC theoretical values .....	11
Figure 9 - CTMC View 1 values.....	12
Figure 10 - CTMC View 2 values.....	12
Figure 11 - Parameters of the two flows M/M/1 (Experiment 1) .....	14
Figure 12 - Average delay formula.....	14
Figure 13 - Parameters of the two flows M/D/1 (Experiment 2) .....	15
Figure 14 - Average Delay in Queue M/D/1 formula.....	16
Figure 15 - Parameters of the two flows M/G/1 .....	17
Figure 16 - M/G/1 system with strict priorities Delay in Queue Formula .....	17
Figure 17 - Average Queuing delay over two classes formula .....	20
Figure 18 - The average delay in queue of class k .....	21
Figure 19 - IPP/M/1/K system example .....	22
Figure 20 - Model network (Valadas, 2021) .....	27
Figure 21 - Script Output values .....	28
Figure 22 - Practical values of Network from exercise 13 .....	29
Figure 23 - Average Packet delay in a flow .....	30
Figure 24 - $p=0.05$ Parameters.....	31
Figure 25 - Average Delay of Flow 1 .....	31
Figure 26 - $p=0.5$ Parameters.....	32
Figure 27 - Average Delay of Flow 2 .....	32
Figure 28 - $p=0.95$ parameters.....	32
Figure 29 - Average delay in flow 3.....	32
Figure 30 - $p=0.975$ parameters .....	33
Figure 31 - Average Delay in flow 4 .....	33
Figure 32 - Packet Switched Network.....	34
Figure 33 - Average Packet Delay of a packet for each flow formula.....	34
Figure 34 - Average Delay in Network Formula.....	35
Figure 35 - Pnet simulator input parameters .....	36
Figure 36 - Average Delay results .....	36
Figure 37 - Parameters with new bifurcated flow .....	37
Figure 38 - Results from Exercise 11 Script with new bifurcated flow .....	38
Figure 39 - bg.R Parameters.....	39

Figure 40 - Output from bg.R.....	39
Figure 41 - Parameters from Ex. 11 Script .....	39
Figure 42 - Output from Ex. 11 Script .....	40
Figure 43 - Circuit Switched Network .....	41
Figure 44 - Packet switched network with the identified links .....	42
Figure 45 - Capacity Provided to each Link.....	42
Figure 46 - Definition of each Flow .....	42
Figure 47 - Blocking Probability of each flow .....	42
Figure 48 - Parameters of cnet.R .....	43
Figure 49 - Results of Cnet.R.....	43
Figure 50 - Product bound parameters .....	44
Figure 51 - Blocking probability results product bound .....	44

## List of Tables:

Table 1 - Transition Probabilities table .....	8
Table 2 - ALOHA simulation vs theoretical 1 .....	10
Table 3 - ALOHA simulation vs theoretical 2 .....	10
Table 4 - ALOHA simulation vs theoretical 3 .....	10
Table 5 - Comparison of the Limiting State Probability values.....	13
Table 6 - M/M/1 Average Queuing Delay .....	13
Table 7 - Practical values of point-to-point link with Poisson arrivals and exponentially distributed packet sizes .....	14
Table 8 - Theoretical values of point-to-point link with Poisson arrivals and exponentially distributed packet sizes .....	15
Table 9 - Practical values of point-to-point link with Poisson arrivals and fixed packet sizes .....	16
Table 10 - Theoretical values of point-to-point link with Poisson arrivals and fixed packet sizes .....	16
Table 11 - Practical values of a point-to-point link with strict priority scheduling .....	17
Table 12 - Theoretical values of a point-to-point link with strict priority scheduling .....	18
Table 13 - Average queuing delay of M/M/1 system .....	19
Table 14 - Average queuing delay of M/G/1 system 1 .....	19
Table 15 - Average queuing delay of M/G/1 system 2 .....	20
Table 16 - Sample Variance values of the average queuing delay estimates of exercise 7a) .....	20
Table 17 - With vs Without Strict Priorities .....	21
Table 18 - IPP/M/1/K and the M/M/1/K systems loss.....	22
Table 19 - IPP/M/1/K and the M/M/1/K systems loss ( $K - 1 = 30$ ).....	23
Table 20 - IPP/M/1/K and the M/M/1/K systems loss ( $\rho = 0.5$ ) .....	23
Table 21 - Practical values of JSQ and Random policy average delay 1 .....	24
Table 22 - Practical values of JSQ and Random policy average delay 2 .....	24
Table 23 - DDR Theoretical vs Practical values 1 .....	26
Table 24 - DDR experiment 2 values.....	26
Table 25 - DDR experiment 3 values.....	26
Table 26 - Theoretical vs Simulation Values .....	28
Table 27 - Average Delay of each flow.....	36
Table 28 - Blocking Probability of all methods .....	44

## Objectives

The aim of this laboratory work is to study discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs), queuing systems, and their applications.

### A. Discrete-time Markov chains

Discrete-time Markov chains are a specific type of Stochastic Processes that finds applications in many fields including in computer networks.

#### Exercise 1

In this first exercise, we compare sample paths of Bernoulli and 2-DTMC processes generated by the script present in [Annex A](#). By experimenting with different values of  $\alpha$  and  $\beta$ , we aim to find what type of behaviours can 2-DTMC process model that a Bernoulli process cannot.

##### Experiment 1:

In our first experiment, we have set  $\alpha = 0.5$  and  $\beta = 0.5$  this way, we create a 2-DTMC that represents an equal probability of entering a different state or staying in the same state. These experiments can be more easily observed by analysing *Figure 1* below with the values of each trial.

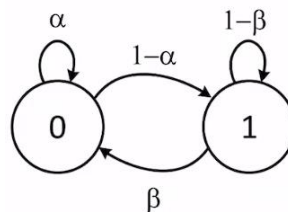


Figure 1 - 2-DTMC Process

And as we can see below, we obtain an equal behaviour in both processes. Both can easily represent jumps from 0 to 1 with a probability of 0.5.

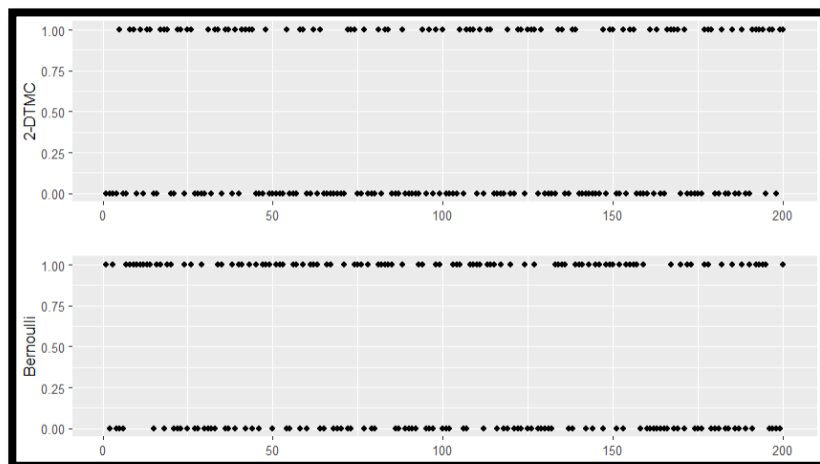


Figure 2 - 2- DTMC vs Bernoulli Process ( $\alpha = 0.5$ ;  $\beta = 0.5$ )

### Experiment 2:

In this second trial, we wish to model a different behaviour where bursts of one state are followed by bursts of another state.

So, we have changed our variables to  $\alpha = 0.9$  and  $\beta = 0.1$ , if we analyse again Figure 1 with these values, we can observe that when a process that enters state 0, it tries to stay in that state for a long time because  $\alpha$  is very close to 1 and the same happens in state 1.

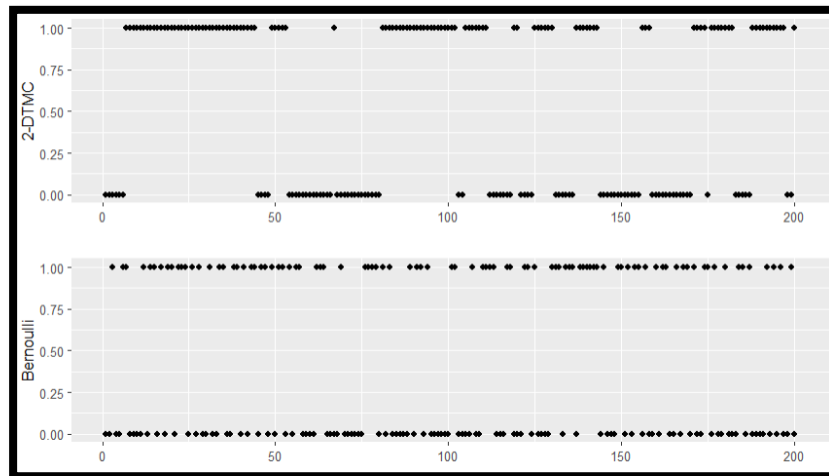


Figure 3 - 2- DTMC vs Bernoulli Process ( $\alpha = 0.9$ ;  $\beta = 0.1$ )

In this case, we can observe that the Bernoulli Process cannot model this kind of behaviour; this happens because, in a Bernoulli Process, the generation of 0's and 1's is independent of the past execution, so the process is just going to jump from one state to another with a probability of 0.5.

We can then conclude that the DTMC is more flexible than the Bernoulli process since this last cannot simulate these kinds of events.



## Exercise 2

In this exercise, we have imported a dataset and estimated the various transition probabilities of a 2-DTMC process.

The relative frequency of transition counts can estimate the transition probabilities. So, our script in [Annex B](#) shows precisely that. We count each time there is a transition from 0 to 0, 0 to 1, 1 to 0 and 1 to 1 and then divide each transition from its respective initial state's total number of transitions.

To conclude, we obtained the results represented in *Table 1* below:

*Table 1 - Transition Probabilities table*

$P_{00}$	0.5191667
$P_{01}$	0.4808333
$P_{10}$	0.7221527
$P_{11}$	0.2778473

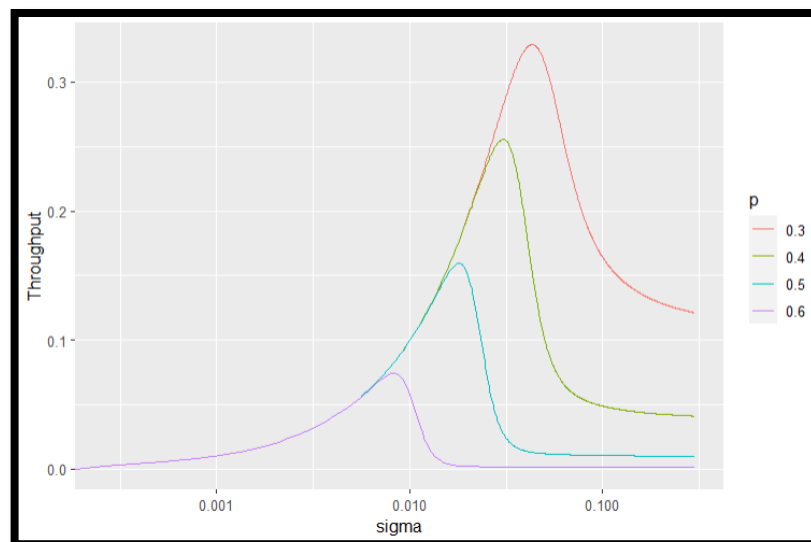
## Exercise 3

This exercise is based on a model, provided in lectures, to compute the throughput of slotted ALOHA with a finite number of users.

### Part a)

In this first part of the exercise The script *aloha\_plot.R* uses the function *aloha\_theo()* to plot the theoretical throughput as a function of  $\sigma$ , for different values of  $N$  and  $p$ . The values used in our experiment were  $N = 10$  and  $p = 0.3, p = 0.4, p = 0.5, p = 0.6$ , for the first graph and  $N = 25$  and  $p = 0.2, p = 0.3$ , for the other.

The goal is to explain the evolution of the throughput with the parameters presented above.



*Figure 4 - Performance of slotted ALOHA graph (N=10)*

Starting by analysing the graph of Figure 4, we can see that the shape of the throughput curves is approximately the same in all cases. The throughput starts by assuming small values for small values of sigma, then increases and reaches its maximum value and finally decreases again for high values of sigma.

The behaviour at the beginning of the graph happens because sigma values are small. This means a more significant percentage of slots with no transmissions; therefore, the low throughput. The final behaviour of the graph is justified because the sigma value is relatively high so, there are a lot of transmissions and many collisions, explaining the decline in the throughput value.

Another conclusion is that the throughput decreases as  $p$  increases. When the value of  $p$  increases, the probability of transmission by backlog users also increases, increasing the likelihood of collisions and therefore decreasing the throughput.

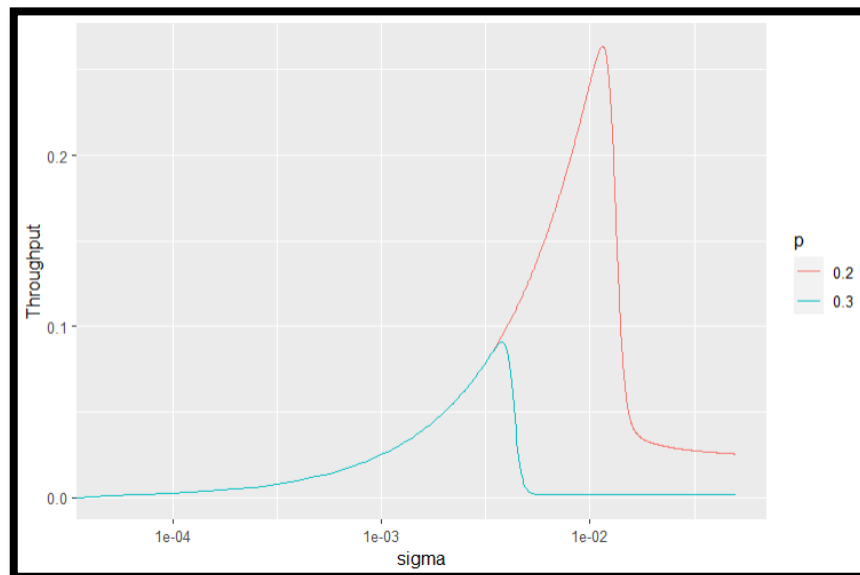


Figure 5 - Performance of slotted ALOHA graph (N=25)

Also, by analysing Figure 5, we can conclude that the throughput decreases as the number of users increases. Considering the experience where we have 10 users and  $p = 0.3$ , represented in the first graph, the maximum throughput is a bit larger than 0.3, comparing this value with 25 users with the same value of  $p = 0.3$ , described in the second graph, the maximum throughput value is less than 0.1. This happens because more users imply more traffic and, therefore, more collisions.

**Part b)**

In this exercise we built a script to estimate the throughput using discrete event simulation in the slotted ALOHA. The code of this exercise can be found in [Annex C](#). The objective is to compare the results obtained in the simulation against the theoretical values.

The theoretical values were obtained through the script *aloha\_theo.R*. In all of the practical experiments a loop with 60000 Time Slots is applied.

**Experiment 1:**

- $N = 10$
- $p = 0.3$
- $\sigma = 0.4$

*Table 2 - ALOHA simulation vs theoretical 1*

Theoretical	Simulation
0.1177026	0.11314

**Experiment 2:**

- $N = 30$
- $p = 0.1$
- $\sigma = 0.9$

*Table 3 - ALOHA simulation vs theoretical 2*

Theoretical	Simulation
0.1293498	0.12495

**Experiment 3:**

- $N = 25$
- $p = 0.2$
- $\sigma = 0.3$

*Table 4 - ALOHA simulation vs theoretical 3*

Theoretical	Simulation
0.02324	0.0234281

By analysing the three experiments we can conclude that in all three the theoretical values are really close to the simulation values, proving the accuracy of our simulator.

## B. Continuous-time Markov chains

Continuous-time Markov Chains are the continuous time analogue of Discrete Time Markov Chains

### Exercise 4

In this exercise, we had the objective of simulating the CTMC in the figure below and estimating its limiting state probabilities, comparing these results with the respective theoretical values. The code of the entire exercise can be found in [Annex D](#).

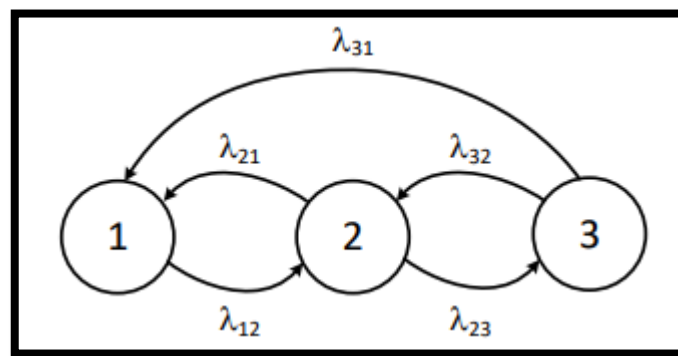


Figure 6 - CTMC

### Theoretical Values:

To calculate the theoretical values, we used the following **balance equations**:

$$\pi_j v_j = \sum_i \pi_i q_{ij}, \quad \sum_i \pi_i = 1$$

Figure 7 - Limiting state probabilities equations

- $v_i$  represents the rate at which the process leaves state  $i$ .
- $q_{ij} = v_i P_{ij}$  is the rate at which the process transitions from state  $i$  to state  $j$ .
- $\pi_i$  is interpreted as the proportion of time the process spends in state  $i$ .

```

> ctmc_theoretical()
==Theoretical Values==
[1] 0.3333333
[1] 0.4444444
[1] 0.2222222
  
```

Figure 8 - CTMC theoretical values

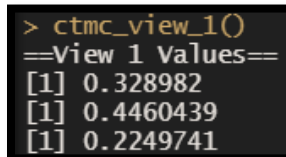
**View 1 Values:**

In the calculation of the limiting state probabilities of View 1, we started by iterating a loop of 10000 processes where we first check the current state of each process.

Then, we determined the amount of time the process spends in that state before making a transition, this time is exponentially distributed with rate  $\nu_i$ .

When a process is leaving a state, it chooses the next state with probability  $P_{ij}$ , and this choice is independent of the amount of time spent on the previous state. In our code, we have the example of state 2 transitions, where we generate a random number between 0 and 1. If that number is larger than the probability of moving from state 2 to state 1 ( $P_{21}$ ), it changes to state 1 otherwise, it changes to state 3.

To estimate the limiting state probabilities below, we divide the time spent in state  $i$  by the total time spent in all states.



```
> ctmc_view_1()
==View 1 Values==
[1] 0.328982
[1] 0.4460439
[1] 0.2249741
```

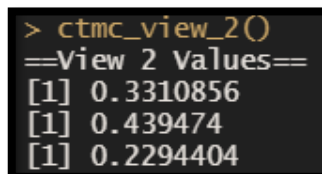
Figure 9 - CTMC View 1 values

**View 2 Values:**

We followed an approach close to the previous view to simulate view 2 of the CTMC and estimate its limiting state probabilities.

We also iterated the CTMC and checked the state of the process, with the difference that a process will stay determined  $T_{ij}$  time in a state.  $T_{ij} = \exp(P_{ij} \nu_i)$  represents an exponentially distributed number with rate  $q_{ij} = \nu_i P_{ij}$ . The CTMC calculates  $T_{ij}$  for all the possible transition states and checks which one is the minimum value; this simulates a situation where the system waits until the first event and jumps to the state of that event.

The limiting state probability is calculated the same way as in view one, by dividing the time spent in state  $i$  by the total time spent in all states.



```
> ctmc_view_2()
==View 2 Values==
[1] 0.3310856
[1] 0.439474
[1] 0.2294404
```

Figure 10 - CTMC View 2 values

### Comparison of Values:

Comparing the values of the three experiments and using the table below, we can conclude that both View 1, and View 2 turn out to be proper ways of simulating a Continuous-Time Markov Chain and, therefore, its limiting state probabilities, since both views come close to the theoretical values.

*Table 5 - Comparison of the Limiting State Probability values*

Theoretical	View 1	View 2
0.333333333	0.328982	0.3310856
0.444444444	0.4460439	0.439474
0.222222222	0.2249741	0.2294404

## C. Queuing systems

### Exercise 5

In this exercise, we used the M/M/1 simulator to study the consequences of the lack of stability in queuing systems.

In our executions we used two values of the system load:  $\rho = 1$  and  $\rho = 2$ , obtaining the following results:

*Table 6 - M/M/1 Average Queuing Delay*

Nr	Value of simulation run	$\rho = 1 (\lambda = 1 \text{ e } \mu = 1)$	$\rho = 2 (\lambda = 2 \text{ e } \mu = 1)$
1	1000	28.40	219.23
2	10000	78.19	2501.65
3	100000	252.91	25049.38

In a stable simulation, we would observe that the longer the simulation, the closer to the theoretical value we would get. On the other hand, and observing the results presented above, we observe the opposite. The longer the simulation, the larger the delay gets. Concluding that this represents an unstable system and confirming that to have a stable system  $\lambda / \mu < 1$ .

## Exercise 6

This exercise required us to use a simulator that estimates the average packet delay and the throughput of each flow, of a packet-switched point-to-point link with two scheduling mechanisms.

### Part a)

In this first part we are to simulate a point-to-point link with FIFO scheduling, which serves two flows, both with **Poisson** arrivals and **exponentially** distributed packet sizes.

```
Flows<-list(list(sourcetype=1,arrivalrate=60,packetsize=1000,priority=1),
            list(sourcetype=1,arrivalrate=30,packetsize=1000,priority=1))
```

Figure 11 - Parameters of the two flows M/M/1 (Experiment 1)

- Source type 1 means that the simulation uses Poisson arrivals and exponentially distributed sizes because of that we are dealing with a M/M/1 system simulation.
- $\lambda_1$  is equal to the arrival rate of the first flow and  $\lambda_2$  is equal to the arrival rate of the second flow.
- To simulate a FIFO scheduling both flows need to have the same priority (1).
- In both experiments we will use a Link Capacity of 100000.
- The simulation end time will be =  $10000 * \frac{1}{\min(\lambda)}$

Experiment 1		Experiment 2	
$\rho = 0.9 (\lambda_1 = 60, \lambda_2 = 30)$		$\rho = 0.5 (\lambda_1 = 30, \lambda_2 = 20)$	
Avg. Delay Flow 1	0.105422	Avg. Delay Flow 1	0.020178
Throughput Flow 1	60703.389658	Throughput Flow 1	29809.455003
Avg. Delay Flow 2	0.107410	Avg. Delay Flow 2	0.020484
Throughput Flow 2	30387.010048	Throughput Flow 2	19846.856570

Table 7 - Practical values of point-to-point link with Poisson arrivals and exponentially distributed packet sizes

### Theoretical Values:

Using the formula bellow we calculated the theoretical average delay in each flow for both experiences.

$$W = \frac{1}{C/\bar{L}_p - \lambda}$$

Figure 12 - Average delay formula

- Experiment 1:

$$W = \frac{1}{\frac{100000}{1000} - (60+30)} = 0.1 \rightarrow \text{Average delay in flow}$$

$$\rho = \frac{60+30}{100} = 0.9 \rightarrow \text{Server utilization}$$

$$\mu = \frac{100000}{1000} = 100 \text{ bits/s} \rightarrow \text{Service rate}$$

- Experiment 2:

$$W = \frac{1}{\frac{100000}{1000} - (30+20)} = 0.02 \rightarrow \text{Average delay in flows}$$

$$\rho = \frac{30+20}{100} = 0.5 \rightarrow \text{Server utilization}$$

$$\mu = \frac{100000}{1000} = 100 \text{ bits/s} \rightarrow \text{Service rate}$$

Experiment 1 ( $\rho = 0.9$ )		Experiment 2 ( $\rho = 0.5$ )	
Avg. Delay Flow 1	0.1	Avg. Delay Flow 1	0.02
Throughput Flow 1	60	Throughput Flow 1	30
Avg. Delay Flow 2	0.1	Avg. Delay Flow 2	0.02
Throughput Flow 2	30	Throughput Flow 2	20

Table 8 - Theoretical values of point-to-point link with Poisson arrivals and exponentially distributed packet sizes

Comparing both experimental and theoretical values, we can conclude that values are close, proving the simulator's precision.

We can also conclude that the bigger the  $\rho$ , the more significant is the delay since the entry rate gets closer to the service rate, this is confirmed by both the simulation and the theoretical values.

### Part b)

In this second part, we simulate a point-to-point link with FIFO scheduling, which serves two flows, both with **Poisson** arrivals and **fixed** packet sizes.

With this, we need to conclude how the performance compares between exponentially distributed packet sizes ([Part a](#)) and fixed packet sizes, and the impact of the service time distribution in the performance of these queuing systems.

```
Flows<-list(list(sourcetype=2,arrivalrate=30,packetsize=1000,priority=1),
            list(sourcetype=2,arrivalrate=20,packetsize=1000,priority=1))
```

Figure 13 - Parameters of the two flows M/D/1 (Experiment 2)

- Source type 2 means that the simulation uses Poisson arrivals and fixed packet sizes because of that we are dealing with a M/D/1 system simulation (deterministic service).
- $\lambda_1$  is equal to the arrival rate of the first flow and  $\lambda_2$  is equal to the arrival rate of the second flow.
- To simulate a FIFO scheduling both flows need to have the same priority (1).
- In both experiments we will use a Link Capacity of 100000.
- The simulation end time will be =  $10000 * \frac{1}{\min(\lambda)}$



Table 9 - Practical values of point-to-point link with Poisson arrivals and fixed packet sizes

Experiment 1		Experiment 2	
$\rho = 0.9 (\lambda_1 = 60, \lambda_2 = 30)$		$\rho = 0.5 (\lambda_1 = 30, \lambda_2 = 20)$	
Avg. Delay Flow 1	0.054980	Avg. Delay Flow 1	0.015072
Throughput Flow 1	60472.188438	Throughput Flow 1	30307.771190
Avg. Delay Flow 2	0.054250	Avg. Delay Flow 2	0.015178
Throughput Flow 2	29672.111114	Throughput Flow 2	20109.848180

**Theoretical Values:**

Using the formula bellow we calculated the theoretical average delay in each flow for both experiences.

$$W_Q = \frac{\lambda}{2\mu(\mu - \lambda)}$$

Figure 14 - Average Delay in Queue M/D/1 formula

- Experiment 1:**

$$W = \frac{60+30}{2*100(100-90)} + \frac{1}{100} = 0.055 \rightarrow \text{Delay in Queue} + \text{Delay in System}$$

$$\rho = \frac{60+30}{100} = 0.9 \rightarrow \text{Server utilization}$$

$$\mu = \frac{100000}{1000} = 100 \text{ bits/s} \rightarrow \text{Service rate}$$

- Experiment 2:**

$$W = \frac{30+20}{2*100(100-50)} + \frac{1}{100} = 0.015 \rightarrow \text{Delay in Queue} + \text{Delay in System}$$

$$\rho = \frac{30+20}{100} = 0.5 \rightarrow \text{Server utilization}$$

$$\mu = \frac{100000}{1000} = 100 \text{ bits/s} \rightarrow \text{Service rate}$$

Table 10 -Theoretical values of point-to-point link with Poisson arrivals and fixed packet sizes

Experiment 1 ( $\rho = 0.9$ )		Experiment 2 ( $\rho = 0.5$ )	
Avg. Delay Flow 1	0.055	Avg. Delay Flow 1	0.015
Throughput Flow 1	60	Throughput Flow 1	30
Avg. Delay Flow 2	0.055	Avg. Delay Flow 2	0.015
Throughput Flow 2	30	Throughput Flow 2	20

In conclusion and comparing the values of this experiment with the ones from [Part a](#) by creating an identical simulation using the same values of  $\lambda$ , we have obtained a smaller Average Delay flow, approximately half of the result that we got from both experiences in Part a, proving that fixed packets sizes have better performance comparing with exponentially distributed packet sizes.

**Part c)**

In the last part of this exercise, we simulate a point-to-point link with strict priority scheduling, which serves two flows with different priorities, both with Poisson arrivals and fixed packet sizes.

With this part we which to compare the results with the theoretical values and draw conclusions.

```
Flows<-list(list(sourcetype=2,arrivalrate=40,packetsize=1000,priority=1),
            list(sourcetype=2,arrivalrate=40,packetsize=1000,priority=2))
```

Figure 15 - Parameters of the two flows M/G/1

- Source type 2 means that the simulation uses Poisson arrivals and fixed packet sizes because of that we are dealing with a M/D/1 system simulation (deterministic service).
- $\lambda_1$  is equal to the arrival rate of the first flow and  $\lambda_2$  is equal to the arrival rate of the second flow.
- To simulate a strict priority scheduling flow, we need to have two different priorities  $\lambda_1 = 1$  and  $\lambda_2 = 2$ , the one with the lowest number has the highest priority.
- In both experiments we will use a Link Capacity of 100000.
- The simulation end time will be  $= 10000 * \frac{1}{\min(\lambda)}$

Table 11 - Practical values of a point-to-point link with strict priority scheduling

Experiment 1	
$\rho = 0.8 (\lambda_1 = 40, \lambda_2 = 40)$	
Avg. Delay Flow 1	0.016506
Throughput Flow 1	39649.930408
Avg. Delay Flow 2	0.043132
Throughput Flow 2	39805.922266

**Theoretical Values:**

Since the service times are fixed, we are dealing with a non-pre-emptive M/D/1 system with priorities, making the following formula the adequate one to use for the calculation of the theoretical values.

$$W_{Qk} = \frac{\rho/2\mu}{(1 - \sum_{i=1}^k \rho_i)(1 - \sum_{i=1}^{k-1} \rho_i)}$$

Figure 16 - M/G/1 system with strict priorities Delay in Queue Formula

Experiment:

$$\mu = \frac{100000}{1000} = 100 \text{ bits/s} \rightarrow \text{Service rate}$$

$$p_1 = \frac{40}{100} = 0.4 ;$$

$$p_2 = \frac{40}{100} = 0.4 ;$$

$$p = 0.4 + 0.4 = 0.8 \rightarrow \text{Server utilization}$$

$$W_1 = \frac{\frac{0.8}{2 \cdot 100}}{1 - 0.4} + \frac{1}{100} = 0.017 \rightarrow \text{Delay in Queue + Delay in System}$$

$$W_2 = \frac{\frac{0.8}{2 \cdot 100}}{(1 - 0.4)(1 - 0.40 - 0.4)} + \frac{1}{100} = 0.043 \rightarrow \text{Delay in Queue + Delay in System}$$

Table 12 - Theoretical values of a point-to-point link with strict priority scheduling

Experiment ( $\rho = 0.8$ )	
Avg. Delay Flow 1	0.017
Throughput Flow 1	40
Avg. Delay Flow 2	0.043
Throughput Flow 2	40

In conclusion we can see that by applying strict priorities we can introduce quality of service in the flows, this is proven by the fact that the flows share the same  $p = 0.4$ , but the flow with the highest priority, Flow 1, has the lowest Average delay.

## D. Workloads

The workload is the load placed into a system; it is the load that the system must work on. The workload has two aspects: the job sizes and the arrival processes of those jobs.

### Exercise 7

In this exercise we studied the impact of the workload variability in the performance of queuing systems using the script *mg1\_bimodal.R*. Considering an M/G/1 system with bimodal (mice-elephant) service distribution we studied two sets of values:

- $S_1 = 1, p_1 = 0.9, S_2 = 11, p_2 = 0.1$
- $S_1 = 1, p_1 = 0.99, S_2 = 101, p_2 = 0.01$

### Part a)

In this first part, we will compare the average queuing delay of M/M/1 and the two M/G/1 systems described above and reflect on the impact of the workload variability in this performance metric.

In these experiences, the job sizes are represented by  $S_1$  and  $S_2$  and occur with probability  $p_1$  and  $p_2$ , respectively. We can see that the  $S_1$  represent the mice due to their small job size, but high probability and the  $S_2$  represent the elephants due to their large job size and low probability. Mice have size 1 in both experiences, and elephants have 11 in experience one and 101 in experience 2.

### Experiment 1:

Table 13 - Average queuing delay of M/M/1 system

M/M/1 ( $\rho = 0.5$ )		$C^2$
Practical Average Queuing Delay M/M/1	1.998123	1.0- Exponential
Theoretical Average Queuing Delay M/M/1	2.0	1.0 - Exponential

Table 14 - Average queuing delay of M/G/1 system 1

Experiment 1 ( $\rho = 0.5$ ) ( $S_1 = 1, p_1 = 0.9, S_2 = 11, p_2 = 0.1$ )		$C^2$
Practical Average Queuing Delay M/G/1	3.107108	2.25 - Bimodal
Theoretical Average Queuing Delay M/G/1	3.25	2.25 - Bimodal

**Experiment 2:**

Table 15 - Average queuing delay of M/G/1 system 2

Experiment 2 ( $\rho = 0.5$ ) ( $S_1 = 1, p_1 = 0.99, S_2 = 101, p_2 = 0.01$ )		$C^2$
Practical Average Queuing Delay M/G/1	25.514384	24.75 - Bimodal
Theoretical Average Queuing Delay M/G/1	25.750000	24.75 - Bimodal

Comparing, for example, M/M/1 system's delay in queue ( $W_Q = 2$ ), with the delay of the second experience M/G/1 system, the delay increases to 25.75, which is a significant amount.

With these, it is possible to conclude that the longer the variability, the worse the performance.

**Part b)**

In this second part, we are to compare the sample variance of the average queuing delay estimates obtained in the three cases of previous exercise. The code of the exercise is present in [Annex E](#).

Experiment	Values	Variance
M/M/1 → Experiment 1	$\rho = 0.5$	0.012769
M/G/1 → Experiment 1	$S_1 = 1, p_1 = 0.9, S_2 = 11, p_2 = 0.1$	0.032054
M/G/1 → Experiment 2	$S_1 = 1, p_1 = 0.99, S_2 = 101, p_2 = 0.01$	21.169962

Table 16 - Sample Variance values of the average queuing delay estimates of exercise 7a)

By analysing the variance values, we can conclude that the variance of the simulations also increases, a large amount, with the variability of the job sizes.

**Part c)**

In this exercise we are to find a way of improving the average queuing delay of the M/G/1 systems.

The solution is in the use of strict priorities. By giving **priority to the smaller jobs** (mice) over the larger ones (elephants) it is possible to minimize the average queuing delay over these two classes.

This can be proven by the formula:

$$W_Q = \frac{\lambda_S W_Q^S + \lambda_L W_Q^L}{\lambda_S + \lambda_L}$$

Figure 17 - Average Queuing delay over two classes formula

Let's pick the following example, similar to the one of the last exercise M/G/1 systems:

$$S_1 = 1, p_1 = 0.8; S_2 = 11, p_2 = 0.1.$$

With these values and with the formula presented below we can calculate the Average delay in Queue for each job, Job 1 with priority 1 and Job 2 with priority 2.

$$W_{Qk} = \frac{\sum_{i=1}^n \lambda_i E(S_i^2)}{2(1 - \sum_{i=1}^k \rho_i)(1 - \sum_{i=1}^{k-1} \rho_i)}$$

Figure 18 - The average delay in queue of class k

Demonstration:

$$\rho_1 = \frac{\lambda_1}{\mu_1}; \rho_2 = \frac{\lambda_2}{\mu_2};$$

$$\mu_1 = \frac{1}{S_1} = 1$$

$$\mu_2 = \frac{1}{S_2} = 0.091$$

$$\lambda_1 = p_1 * \mu_1 = 0.8 * 1 = 0.8$$

$$\lambda_2 = p_2 * \mu_2 = 0.1 * 0.091 = 0.0091$$

$$\rho = 0.8 + 0.1 = 0.9 < 1$$

$$W_{Q1} = \frac{0.8*1}{2(1-0.8)} = 2 \rightarrow \text{Delay of Job 1 (mice)}$$

$$W_{Q2} = \frac{0.0091*121}{2(1-0.8)(1-0.1)} = 3.0586 \rightarrow \text{Delay in Job 2 (Elephants)}$$

$$W_Q = \frac{0.8*2+0.0091*3.0586}{0.8+0.0091} = 2.011 \rightarrow \text{Average Queuing Delay}$$

Table 17 - With vs Without Strict Priorities

Avg Queuing Delay Without strict priorities	Avg Queuing Delay with strict priorities
30.552632	2.011

By comparing the values of the two experiments, with and without the strict priorities we can clearly see a huge improvement in the delay with strict priorities.

## Exercise 8

In this exercise we studied the impact of the traffic burstiness in the performance of queuing systems. Our objective is to study how the loss probabilities evolve as a function of  $K$ ,  $R$ , and  $\rho$  in the IPP/M/1/K system and how the IPP/M/1/K and the M/M/1/K systems compare.

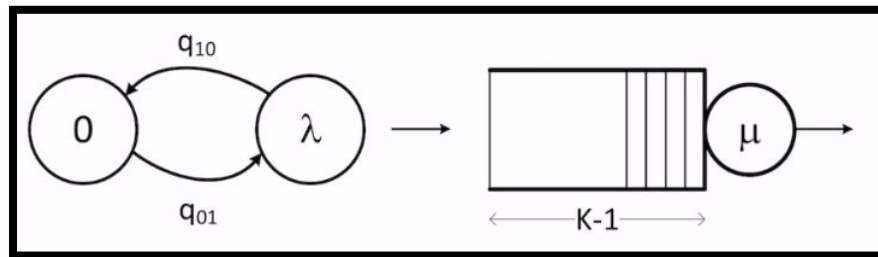


Figure 19 - IPP/M/1/K system example

### Parameters:

- $\lambda_1 \rightarrow$  The peak arrival rate
- $\pi_1 \rightarrow$  The probability of state 1
- $\rho = \lambda/\mu \rightarrow$  The system utilization
- $K - 1 \rightarrow$  The queue size
- $R = \lambda_1/q_{10} \rightarrow$  The ratio between the peak rate and output rate of state 1

### Experiment 1:

- $\rho = 0.9$
- $\lambda = 10$
- $\pi_0 = \pi_1 = 0.5$
- $K - 1 = 10$

R	Loss M/M/1/K	Loss IPP/M/1/K
1	0.043830	0.064650
5	0.040170	0.121090
20	0.045980	0.235020
30	0.044220	0.281830
60	0.042280	0.344880

Table 18 - IPP/M/1/K and the M/M/1/K systems loss

Experiment 2:

- $\rho = 0.9$
- $\lambda = 10$
- $\pi_0 = \pi_1 = 0.5$
- $K - 1 = 30$

R	Loss M/M/1/K	Loss IPP/M/1/K
1	0.005270	0.008960
5	0.003030	0.029540
20	0.004190	0.097760
30	0.002780	0.136020
60	0.002900	0.205430

Table 19 - IPP/M/1/K and the M/M/1/K systems loss ( $K - 1 = 30$ )Experiment 3:

- $\rho = 0.5$
- $\lambda = 10$
- $\pi_0 = \pi_1 = 0.5$
- $K - 1 = 10$

R	Loss M/M/1/K	Loss IPP/M/1/K
1	0.000330	0.001580
5	0.000220	0.010290
20	0.000160	0.034430
30	0.000220	0.046820
60	0.000240	0.058300

Table 20 - IPP/M/1/K and the M/M/1/K systems loss ( $\rho = 0.5$ )

With this study, we can conclude that the larger is the queue size, the fewer packets get dropped and, therefore, the smaller is the loss of packets, this can be observed by comparing the results from experience 1 with experience 2.

We can also conclude that the longer the burst, the higher the loss of packets, therefore, degrading the performance, this can be observed by comparing the several results of the ratio between the peak rate and output rate (R) in all of the experiences.

Furthermore, we can still conclude that the smaller the system utilization, the smaller will be the loss, this can be observed by comparing the results from experience 1 with experience 3.



## E. Server Farms

Server Farms are collections of servers that work together to handle incoming requests, each request may be routed to a different server. Datacentres are made of server farms.

### Exercise 9

In this exercise we are to consider a web server farm, with two servers. Using the script *webserverfarm.R* we are to compare the average delay in system of two different task assignment policies: random and Join Shortest Queue (JSQ).

- **Random Policy** – each job is assigned to one of  $k$  hosts with equal probability
- **JSQ Policy** – each job is assigned to the host with fewer queue jobs

#### Experiment 1:

Experiment 1 ( $\rho = 0.9$ )	
Average Delay of Random Assignment Policy	1.817802
Average Delay of JSQ Assignment Policy	1.316878

Table 21 - Practical values of JSQ and Random policy average delay 1

#### Experiment 2:

Experiment 2 ( $\rho = 0.1$ )	
Average Delay of Random Assignment Policy	1.053307
Average Delay of JSQ Assignment Policy	1.004745

Table 22 - Practical values of JSQ and Random policy average delay 2

By the results of the two experiments, we can conclude that by assigning each job to the server with fewer jobs, regardless of high or low system utilization, the JSQ policy performs better than the random policy approach. However, we do not observe a significant difference between the two average delays, so better approaches should be considered to replace the random policy.

## F. Packet scheduling

### Exercise 10

In this exercise we wrote an R script that simulates two queues served by the Deficit Round Robin scheduling algorithm.

Our implementation was based on the script *rand\_sched.R*, which simulates a random scheduling discipline. The final script can be found in the [Annex F](#).

#### Description of Changes:

Our change to the simulator also reflects the major changes between DDR and the Random scheduling algorithm.

1. We started by creating two new vectors, one that stores the Quantum of each queue and one that keeps the count of the Deficit (Credit) of each queue. We also set the Current Queue variable to 1 at the beginning of the simulator. Our next changes are all located in the Departures section of the code.
2. In the departures section we have three options of departure. The first situation is when the system is empty, when this occurs, we reset the deficit counters of both queues.
3. The second situation is when only one queue is empty. In this one, we start by finding which queue is empty, resetting its deficit counter, and which queue has packets. We then start a while loop that first verifies if the deficit counter of the current queue is higher or equal to the size of the packet that needs to be transmitted. If it is, it transmits the packet, subtracts the size of the packet in the deficit counter, and exits the loop; otherwise, it keeps adding the quantum to the deficit counter until it has enough credit to transmit.
4. The last situation is when no queue is empty. Here we created another while(True) loop that verifies if the deficit counter of the current queue is high enough to transmit the queue. If it is, it transmits the packet, subtracts the size of the packet in the deficit counter, and exits the loop; If it is not, it changes queue and adds the respective quantum value to the new queue deficit counter. This loop goes on until the packet is transmitted. In this situation, the choice of the current queue is no longer random since we have set the Current Queue variable to start at 1 and it changes depending on the deficit of each queue.

Demonstration:

To demonstrate the correctness of our code, we will compare the values of the throughput obtained in each queue with the respective theoretical values obtained using the following formula.

$$r_i = \frac{q_i}{\sum_i q_i} C$$

Experiment 1:

- Theoretical versus Practical values.

$$r_1 = \frac{1000}{1000*2000} * 1000 = 333.33 \rightarrow \text{Queue 1}$$

$$r_2 = \frac{2000}{1000*2000} * 1000 = 666.66 \rightarrow \text{Queue 2}$$

Table 23 - DDR Theoretical vs Practical values 1

Queue	Theoretical Values	Simulation Values
1	333.33	333.47
2	666.66	666.51

Experiment 2:

- With equal amounts of Quantum (both 1000)

Queue	Simulation Values
1	499.87
2	500.09

Table 24 - DDR experiment 2 values

As we can see both queues share the same throughput. So, the quantum has high influence in the throughput values.

Experiment 3:

- With equal amounts of Average Packet size (both 1000)

Queue	Simulation Values
1	333.22
2	666.74

Table 25 - DDR experiment 3 values

As we can see by comparing these values with the ones from the first experiment, we can conclude that the packet size has no influence in the throughput value.

## G. Packet switched networks

### Exercise 11

In this exercise, using the same input data structure as the pnet.R simulator and the network model in the figure bellow (model present in theory slides), we calculated:

1. The average packet delay of each flow
2. The average number of packets in the network
3. The average packet delay in the network using the Kleinrock approximation.

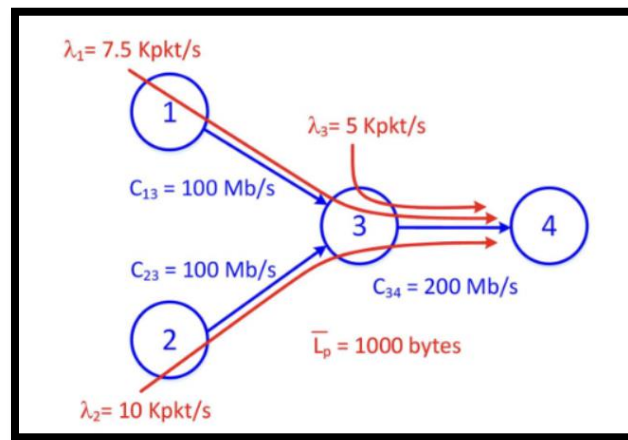


Figure 20 - Model network (Valadas, 2021)

The code execution of this exercise can be found in [Annex G](#).

#### Theoretical Values:

$$L_{ij} = \frac{\lambda_{ij}}{\mu_{ij} - \lambda_{ij}}; \mu_{ij} = \frac{C_{ij}}{L_p}$$

$$\mu_{13} = \mu_{23} = \frac{100 \text{ Mb/s}}{1000 \cdot 8} = 12.5 \text{ Kpkt/s}$$

$$\mu_{34} = \frac{200 \text{ Mb/s}}{1000 \cdot 8} = 25 \text{ Kpkt/s}$$

$$L_{13} = \frac{7.5}{(12.5 - 7.5)} = 1.5 \text{ Kpkt/s}$$

$$L_{23} = \frac{10}{(12.5 - 10)} = 4 \text{ Kpkt/s}$$

$$L_{34} = \frac{7.5 + 10 + 5}{(25 - (7.5 + 10 + 5))} = 9 \text{ Kpkt/s}$$

$$L = L_{13} + L_{23} + L_{34} = 14.5 \text{ pkt} \rightarrow \text{Average Number of Packets in Network}$$

$$W = \frac{L}{(\lambda_1 + \lambda_2 + \lambda_3)} = 644. (4) \text{ us} \rightarrow \text{Average Delay in Network}$$

$$W_{ij} = \frac{1}{(u_{ij} - \lambda_{ij})}; \quad W_1 = W_{13} + W_{34}; \quad W_2 = W_{23} + W_{34}; \quad W_3 = W_{34}$$

$$W_1 = \frac{1}{(u_{13} - \lambda_1)} + \frac{1}{(u_{34} - (\lambda_1 + \lambda_2 + \lambda_3))} = 600 \text{ us} \rightarrow \text{Average Packet Delay Flow 1}$$

$$W_2 = \frac{1}{(u_{23} - \lambda_2)} + \frac{1}{(u_{34} - (\lambda_1 + \lambda_2 + \lambda_3))} = 800 \text{ us} \rightarrow \text{Average Packet Delay Flow 2}$$

$$W_3 = \frac{1}{(u_{34} - (\lambda_1 + \lambda_2 + \lambda_3))} = 400 \text{ us} \rightarrow \text{Average Packet Delay Flow 2}$$

### Experiment:

After calculating the theoretical values, we can verify if they match the practical values we generated in our script. This is the output:

Table 26 - Theoretical vs Simulation Values

Operations	Theoretical Values	Simulation Values
Average Delay of each flow	Flow 1: 600 Flow 2: 800 Flow 3: 400	Flow 1: $6e - 04$ Flow 2: $8e - 04$ Flow 3: $4e - 04$
Average Number of Packets in Network	14.5 pkt	14.5
Average Delay in Network	644. (4) us	0.000644

```
[1] "Average delay in the network: "
> w = sum(L)/sum(lambdasList)
> print(w)
[1] 0.0006444444
>
> print(L)
[1] 1.5 4.0 9.0
>
> print("Average number of packets in the network: ")
[1] "Average number of packets in the network: "
> print(sum(L))
[1] 14.5
>
> print("Average packet delay of each flow: ")
[1] "Average packet delay of each flow: "
> print(W)
[1] 6e-04 8e-04 4e-04
```

Figure 21 - Script Output values

As we can see, our practical values match the theoretical values calculated above.

Here we tested the same script for another network with a different composition, present in [Exercise 13](#), to prove that our script accepts any network topology and any set of flows.

In Exercise 13, [Part a](#)), we calculated the theoretical values of the average packet delay of each flow and the average packet delay of the network. As we can see from the image below, all the values coincide with the theoretical ones.

```
> w = sum(L)/sum(lambdasList)
> print(w)
[1] 0.04435833
>
> print(L)
[1] 5.243902 3.000000 3.000000 1.000000 5.243902 5.243902 1.000000
>
> print("Average number of packets in the network: ")
[1] "Average number of packets in the network: "
> print(sum(L))
[1] 23.73171
>
> print("Average packet delay of each flow: ")
[1] "Average packet delay of each flow: "
> print(W)
[1] 0.07317073 0.03125000 0.03906250 0.00781250 0.00000000 0.00000000 0.00000000
```

Figure 22 - Practical values of Network from exercise 13

We can then conclude that this script works for any network and is generic like requested.

**Exercise 12**

In this exercise we are to compare the average packet delay obtained

1. Through the Kleinrock approximation
2. Through simulation (using the pnet.R simulator), for four distinct values of  $\rho = \lambda/\mu$  namely 0.05, 0.5, 0.95 e 0.975.

**Theoretical Values:**

The average packet delay for a flow can be calculated in the following way:

$$W = \frac{1}{\mu - \lambda}$$

*Figure 23 - Average Packet delay in a flow*

We can do this by considering a network built by two connections with a capacity of 64 Kb/s and a flux where packages are, on average, 1000 bits in size.

By using the following formula:

$$\mu = \frac{C}{L_p}$$

and since we already have the capacity values and the size of the packages, we can calculate the  $\mu$ , which will be 64 in this case.

We also know that

$$\rho = \frac{\lambda}{\mu}$$

so, we can calculate the different values of  $\lambda$  that correspond to their respective  $\rho$  values:

$$\rho = 0.05; \lambda = 3.2$$

$$\rho = 0.5; \lambda = 32$$

$$\rho = 0.95; \lambda = 60.8$$

$$\rho = 0.975; \lambda = 62.4$$

Now we can calculate the theoretical values with the formula in the figure above. We must bear in mind that we must multiply the result by 2, since it represents two fluxes.

$$p = 0.05; W = 0.0329 \text{ s}$$

$$p = 0.5; W = 0.0625 \text{ s}$$

$$p = 0.95; W = 0.625 \text{ s}$$

$$p = 0.975; W = 1.25$$

### Practical Experiment:

To calculate the practical values, we used the pnet simulator where we change the parameters to the different rate values.

- $p = 0.05$

```
parameters <- function() {
  #USER DEFINED PARAMETERS

  #Links must be identified by contiguous integers starting at 1. These numbers
  #must be used as indexes of the LinkCapacities vector and of the vectors
  #defining the routes of each flow. Do not use the node numbers for this
  #purpose!

  #Define here the capacity of each link
  LinkCapacities<-c(64e3,64e3) #In bits/sec

  #Define here the flows. Flows is a list of lists that stores in each list the
  #arrival rate (in packets/second), the mean packet length (in bits) and the
  #route of each flow; the routes must be defined using the link identifiers
  #(and not the node identifiers)
  Flows<-list(list(rate=3.2,packetsize=1000,route=c(1,2)))

  #Define here the simulation end time, function of the minimum rate
  endTime<-10000*(1/3.2)
}
```

Figure 24 -  $p=0.05$  Parameters

**Average delay of flow 1 = 0.033386**

Figure 25 - Average Delay of Flow 1



- $p = 0.5$

```
parameters <- function() {
  #USER DEFINED PARAMETERS

  #Links must be identified by contiguous integers starting at 1. These numbers
  #must be used as indexes of the LinkCapacities vector and of the vectors
  #defining the routes of each flow. Do not use the node numbers for this
  #purpose!

  #Define here the capacity of each link
  LinkCapacities<-c(64e3,64e3) #In bits/sec

  #Define here the flows. Flows is a list of lists that stores in each list the
  #arrival rate (in packets/second), the mean packet length (in bits) and the
  #route of each flow; the routes must be defined using the link identifiers
  #(and not the node identifiers)
  Flows<-list(list(rate=32,packetsize=1000,route=c(1,2)))

  #Define here the simulation end time, function of the minimum rate
  endTime<-10000*(1/32)
}
```

Figure 26 -  $p=0.5$  Parameters

Average delay of flow 1 = 0.064812

Figure 27 - Average Delay of Flow 2

- $p = 0.95$

```
parameters <- function() {
  #USER DEFINED PARAMETERS

  #Links must be identified by contiguous integers starting at 1. These numbers
  #must be used as indexes of the LinkCapacities vector and of the vectors
  #defining the routes of each flow. Do not use the node numbers for this
  #purpose!

  #Define here the capacity of each link
  LinkCapacities<-c(64e3,64e3) #In bits/sec

  #Define here the flows. Flows is a list of lists that stores in each list the
  #arrival rate (in packets/second), the mean packet length (in bits) and the
  #route of each flow; the routes must be defined using the link identifiers
  #(and not the node identifiers)
  Flows<-list(list(rate=60.8,packetsize=1000,route=c(1,2)))

  #Define here the simulation end time, function of the minimum rate
  endTime<-10000*(1/60.8)
}
```

Figure 28 -  $p=0.95$  parameters

Average delay of flow 1 = 0.298356

Figure 29 - Average delay in flow 3

- $p = 0.975$

```
parameters <- function() {
  #USER DEFINED PARAMETERS

  #Links must be identified by contiguous integers starting at 1. These numbers
  #must be used as indexes of the LinkCapacities vector and of the vectors
  #defining the routes of each flow. Do not use the node numbers for this
  #purpose!

  #Define here the capacity of each link
  LinkCapacities<-c(64e3,64e3) #In bits/sec

  #Define here the flows. Flows is a list of lists that stores in each list the
  #arrival rate (in packets/second), the mean packet length (in bits) and the
  #route of each flow; the routes must be defined using the link identifiers
  #(and not the node identifiers)
  Flows<-list(list(rate=62.4,packetsize=1000,route=c(1,2)))

  #Define here the simulation end time, function of the minimum rate
  endTime<-10000*(1/62.4)
}
```

Figure 30 -  $p=0.975$  parameters

Average delay of flow 1 = 0.388066

Figure 31 - Average Delay in flow 4

We can conclude that the Kleinrock approximation is pretty accurate for values near 0, which means for low values of traffic.

When these values get near to 1, the Kleinrock approximation begins to show some flaws, since the theoretical values differ immensely from the practical ones.

### Exercise 13

This exercise will consist of the analysis of the packet switched network represented in the figure below. All links have 256 Kb/s of capacity. The network is initially serving four packet flows using nonbifurcated fixed routing:

1. Flow 1 with route 1-2-4-6 and  $\lambda_1 = 215$  packets/s;
2. Flow 2 with route 1-3-5 and  $\lambda_2 = 64$  packets/s;
3. Flow 3 with route 1-3-5-6 and  $\lambda_3 = 128$  packets/s;
4. Flow 4 with route 2-5 and  $\lambda_4 = 128$  packets/s.

The flows are characterized by Poisson arrivals and exponentially distributed packet sizes with mean 1000 bits.

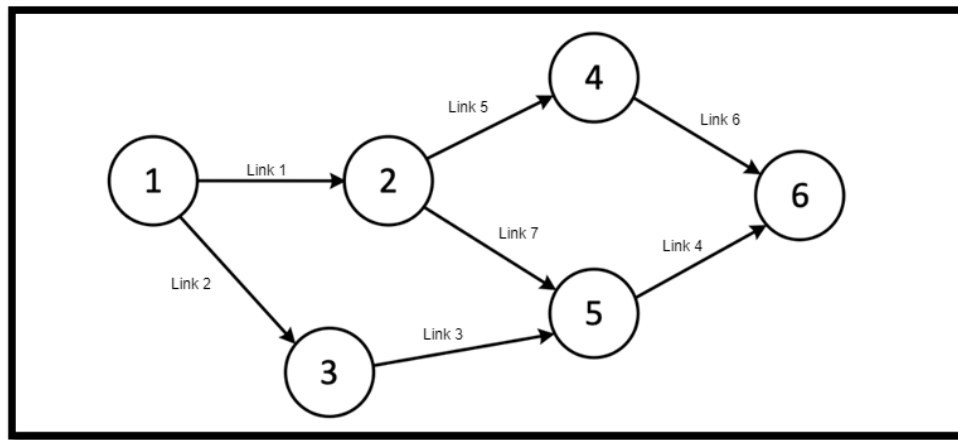


Figure 32 - Packet Switched Network

#### Part a)

To calculate the average packet delay of a packet for each flow we used the formula:

$$W_i = \frac{1}{\mu - \lambda_i}$$

Figure 33 - Average Packet Delay of a packet for each flow formula

We also know the respective rates of each flow:

- $\lambda_1 = 215$  packets/s;
- $\lambda_2 = 64$  packets/s;
- $\lambda_3 = 128$  packets/s;
- $\lambda_4 = 128$  packets/s.

We are only missing  $\mu$ , so we can calculate the average delays.

$$\mu = \frac{C}{L_p} = \frac{256000}{1000} = 256$$

$$W_1 = \frac{1}{(256-215)} + \frac{1}{(256-215)} + \frac{1}{(256-215)} = 0.0731$$

$$W_2 = \frac{1}{(256-(64+128))} + \frac{1}{(256-(64+128))} = 0.03125$$

$$W_3 = \frac{1}{(256-(64+128))} + \frac{1}{(256-(64+128))} + \frac{1}{256-128} = 0.0390625$$

$$W_4 = \frac{1}{256-128} = 0.0078125$$

Finally, to discover the average delay in the network we can use:

$$W = \frac{\lambda_1 * W_1}{\lambda} + \frac{\lambda_2 * W_2}{\lambda} + \frac{\lambda_3 * W_3}{\lambda} + \frac{\lambda_4 * W_4}{\lambda}$$

Figure 34 - Average Delay in Network Formula

Where  $\lambda$  corresponds to the sum of the four given  $\lambda$ .

The final average network packet delay corresponds to:

$$W = 0.04433 \text{ s}$$

**Part b)**

This exercise consists on the confirmation of the results obtained in the previous part a), by using the pnet.R simulator.

The input parameters on the respective file are:

```
parameters <- function() {

  #USER DEFINED PARAMETERS

  #Links must be identified by contiguous integers starting at 1. These numbers
  #must be used as indexes of the LinkCapacities vector and of the vectors
  #defining the routes of each flow. Do not use the node numbers for this
  #purpose!

  #Define here the capacity of each link
  LinkCapacities<-c(256e3,256e3,256e3,256e3,256e3,256e3) #In bits/sec

  #Define here the flows. Flows is a list of lists that stores in each list the
  #arrival rate (in packets/second), the mean packet length (in bits) and the
  #route of each flow; the routes must be defined using the link identifiers
  #(and not the node identifiers)
  Flows<-list(list(rate=215,packetsize=1000,route=c(1,5,6)),
              list(rate=64,packetsize=1000,route=c(2,3)),
              list(rate=128,packetsize=1000,route=c(2,3,4)),
              list(rate=128,packetsize=1000,route=c(7)))

  #Define here the simulation end time, function of the minimum rate
  endTime<-10000*(1/64)

}
```

Figure 35 - Pnet simulator input parameters

And we obtained:

```
Average delay of flow 1 = 0.060411
Average delay of flow 2 = 0.028639
Average delay of flow 3 = 0.036668
Average delay of flow 4 = 0.007844
```

Figure 36 - Average Delay results

Table 27 - Average Delay of each flow

	Theoretical Values	Simulation Values
Average Delay of each flow	<b>Flow 1:</b> 0.0731	<b>Flow 1:</b> 0.060411
	<b>Flow 2:</b> 0.03125	<b>Flow 2:</b> 0.028639
	<b>Flow 3:</b> 0.0390625	<b>Flow 3:</b> 0.036668
	<b>Flow 4:</b> 0.0078125	<b>Flow 4:</b> 0.007844

By the analysing the results obtained above we can conclude that they do not differ heavily on the theoretical values we calculated before.

**Part c)**

In this exercise the flow 1-2-4-6 is bifurcated in the second node and is divided into two different flows:

1.  $X_1 \rightarrow 1-2-4-6$
2.  $X_2 \rightarrow 1-2-5-6$

Flow 1 will go through links 5 and 6 since they are empty. Flow 2 will go through a path that is already with other flow going through it. In this case it is the flow 3 and 4 and there is only a capacity of 128Kb/s available for  $X_2$  to go through.

This leads us into a system of equations where we calculate the new rates of this bifurcated flow.

- $X_1 + X_2 = 215 \rightarrow$  which corresponds to the rate of the original flow 1.
- $X_1 = X_2 + 128 \rightarrow$  which corresponds to the capacity available in the new path.

By solving this system, we will get

- $X_1 = 171,5 \text{ Kb/s}$
- $X_2 = 43,5 \text{ Kb/s}$ .

Now we run our script made in [Exercise 11](#), but with one new flow ( $X_2$ ), and with two different rate values (the ones we have calculated).

The input given looks like this:

```
linkCapacity=c(256e3,256e3,256e3,256e3,256e3,256e3,256e3)
servRate=linkCapacity/1000
lambda1=171.5
lambda2=43.5
lambda3=64
lambda4=128
lambda5=128
flow1=list(lambda1,list(1,5,6))
flow2=list(lambda2,list(1,7,4))
flow3=list(lambda3,list(2,3))
flow4=list(lambda4,list(2,3,4))
flow5=list(lambda5,list(7))
```

Figure 37 - Parameters with new bifurcated flow

The script returns us the following output:

```
> print("Average delay in the network: ")
[1] "Average delay in the network: "
> w = sum(L)/sum(lambdasList)
> print(w)
[1] 0.03619111
>
> print(L)
[1] 5.243902 3.000000 3.000000 2.029586 2.029586 2.029586 2.029586
>
> print("Average number of packets in the network: ")
[1] "Average number of packets in the network: "
> print(sum(L))
[1] 19.36225
>
> print("Average packet delay of each flow: ")
[1] "Average packet delay of each flow: "
> print(W)
[1] 0.04805888 0.04805888 0.03125000 0.04308432 0.01183432 0.00000000 0.00000000
```

Figure 38 - Results from Exercise 11 Script with new bifurcated flow

As we can see, the average network packet delay improved slightly from the one calculated in the Exercise 13, [Part a\)](#).

### **Part d)**

In order to calculate the new average packet delay, we ran the bg.R script given by the teacher, which returned us a set of flows. We had already worked with some of them. According to what was given in the respective parameters, others were completely new and generated by the gb.r script according to what was given in the respective parameters.R.

We then used the script we created on Exercise 11 ([Annex G](#)).

This script gives us a fair amount of information, but we will only focus on the average network packet delay.

Experiment:

We proceed to change our parameters in the script, so they correspond to the flows returned by the bg.R script.

These are the parameters:

```
parameters = function() {
  #Define here the capacity of each link, in bits/sec. The LinkCapacities matrix
  #is a square matrix where the number of rows equals the number of nodes. Rows
  #correspond to origin nodes and columns to destination nodes. Element(i,j)
  #should be equal to the link capacity if there is a link from node i to node j,
  #and should be 0 otherwise
  LinkCapacities<-matrix(c(0, 256e3, 256e3, 0, 0, 0,
                           0, 0, 0, 256e3, 256e3, 0,
                           0, 0, 0, 0, 256e3, 0,
                           0, 0, 0, 0, 0, 256e3,
                           0, 0, 0, 0, 0, 256e3,
                           0, 0, 0, 0, 0, 0),
                        nrow=6,
                        ncol=6,
                        byrow=TRUE)

  #Define here the flows. Flows is a matrix with a user-defined number of rows and
  #3 columns. Each row corresponds to a different flow, and the three columns
  #correspond to (1) the origin node, (2) the destination node, and (3) the
  #offered rate between the two previous nodes, expressed in bits/sec
  Flows<-matrix(c(1, 6, 343,
                  1, 5, 64,
                  2, 5, 128),
                nrow=3,
                ncol=3,
                byrow=TRUE)
}
```

Figure 39 - bg.R Parameters

```
Flow from 1 to 6
Rate of route 1-2-4-6: 171.739221
Rate of route 1-3-5-6: 171.260779
Rate of route 1-2-5-6: 0.000000
Flow from 1 to 5
Rate of route 1-2-5: 43.217805
Rate of route 1-3-5: 20.782195
Flow from 2 to 5
Rate of route 2-5: 128.000000
```

Figure 40 - Output from bg.R

```
linkCapacity=c(256e3,256e3,256e3,256e3,256e3,256e3)
servRate=linkCapacity/1000
lambda1=171.739221
lambda2=20.782195
lambda3=171.260779
lambda4=128
lambda5=43.217805
flow1=list(lambda1,list(1,5,6))
flow2=list(lambda2,list(2,3))
flow3=list(lambda3,list(2,3,4))
flow4=list(lambda4,list(7))
flow5=list(lambda5,list(1,7))
flowsList=list(flow1,flow2,flow3,flow4,flow5)
linkLambda=c()
lambdasList=c(lambda1,lambda2,lambda3,lambda4,lambda5)
```

Figure 41 - Parameters from Ex. 11 Script

We then ran the script and obtained the new and optimized average network packet delay, and as we can see if all flows are bifurcated, the average delay is still clearly improved, when compared to the previous theoretical value we have calculated.



This is the output:

```
> print("Average delay in the network: ")  
[1] "Average delay in the network: "  
> w = sum(L)/sum(lambdasList)  
> print(w)  
[1] 0.03618626
```

Figure 42 - Output from Ex. 11 Script

The average delay improved almost nothing compared to the previous exercise, which was expectable since we are working with a small network. These changes do not have a considerable impact when working with networks this size.

## H. Circuit switched networks

### Exercise 14

This exercise consists of the study of the circuit switched network represented bellow.

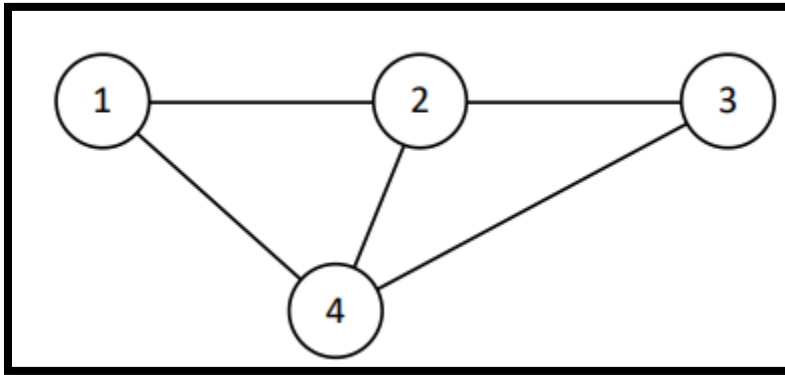


Figure 43 - Circuit Switched Network

The network has 4 nodes, 5 links and 3 call flows:

1. **Flow 1** from node 1 to node 2 with offered load  $\rho = 0.5$ .
2. **Flow 2** from node 2 to node 3 with offered load  $\rho = 0.5$ .
3. **Flow 3** from node 1 to node 3 with offered load  $\rho = 0.5$ .

Each call requires a capacity of  $b = 64$  Kb/s:

Each link has a different operational cost per 64 Kbit/s circuit:

- **Links 1-2 and 2-3:** 500 Euros per 64 Kbit/s circuit.
- **Link 2-4:** 1000 Euros per 64 Kbit/s circuit.
- **Links 1-4 and 4-3:** 100 Euros per 64 Kbit/s circuit.

### Part a)

In this first part we aim to determine:

1. The routes of the 3 flows
2. The capacities of the 5 links that minimize the total cost of the installed solution, such that the blocking probability of each flow is less than 1%.

Experiment:

The execution of the experiment started with the identification of each flow. By trial and error, we attempted each possible path of each flow in order to determine the best route in order to minimize the total cost of the installed solution.

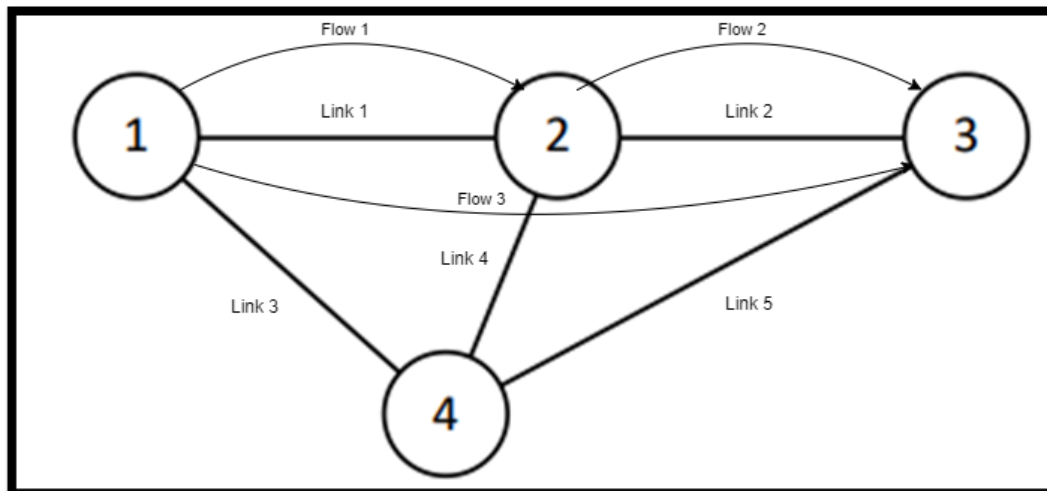


Figure 44 - Packet switched network with the identified links

- **Route of flow 1:** 1-4-3-2 → 700\$
- **Route of flow 2:** 2-3 → 500\$
- **Route of flow 3:** 1-4-3 → 200\$

Then by providing extra capacity to each link we were able to optimize the network by reducing the blocking probability of each flow to less than 1%.

```
LinkCapacities=c(0,5,5,0,5) #In number of circuits
```

Figure 45 - Capacity Provided to each Link

```
Flows=list(list(load=0.5,route=c(3,5,2)),
            list(load=0.5,route=c(2)),
            list(load=0.5,route=c(3,5)))
```

Figure 46 - Definition of each Flow

With the routes and capacities provided above we were able to achieve these Blocking Probability results:

```
Blocking probability of flow 1 = 0.009028
Blocking probability of flow 2 = 0.003031
Blocking probability of flow 3 = 0.006016
```

Figure 47 - Blocking Probability of each flow

**Part b)**

In this part we use the cnet.R simulator to confirm the results of the previous part a).

In order to obtain a correct result in this simulator we must assure that the value of  $p=0.5$ , therefore:

$$p = \text{rate} * \text{call duration} = 0.5$$

**Experiment:**

The values used in the experiment were:

```
LinkCapacities<-c(0,5,5,0,5)

#Define here the flows. Flows is a list of lists that stores in each list the
#call duration, call arrival rate, call bandwidth and the route of each flow;
#the routes must be defined using the link identifiers (and not the node
#identifiers)
Flows<-list(list(duration=2,rate=0.25,bwd=1,route=c(3,5,2)),
            list(duration=2,rate=0.25,bwd=1,route=c(2)),
            list(duration=2,rate=0.25,bwd=1,route=c(3,5)))

#Definition of the simulation end time, function of the minimum arrival rate
endTime<-10000*(1/0.25)
```

Figure 48 - Parameters of cnet.R

And the results were:

```
Blocking probability of flow 1 = 0.005466
Blocking probability of flow 2 = 0.003057
Blocking probability of flow 3 = 0.002078
```

Figure 49 - Results of Cnet.R

These results seem more trustful than the results obtained in the reduced load approximation since this simulation takes into consideration the call duration the rate and the bandwidth of each flow, considerations that are not taken in the previous reduced load simulation.

**Part c)**

Now we use the product bound simulation to compare the results with the previous ones.

**Experiment:**

```
LinkCapacities=c(0,5,5,0,5) #In number of
#Define here the flows. Flows is a list of
#the offered load and (2) the route of each
#using the link identifiers (and not the n
Flows=list(list(load=0.5,route=c(3,5,2)),
           list(load=0.5,route=c(2)),
           list(load=0.5,route=c(3,5)))
```

Figure 50 - Product bound parameters

And the output results are:

```
Blocking probability of flow 1 = 0.009174
Blocking probability of flow 2 = 0.003067
Blocking probability of flow 3 = 0.006126
```

Figure 51 - Blocking probability results product bound

Table 28 - Blocking Probability of all methods

	Reduced Load	cnet.R Simulator	Product Bound
Blocking Probability of Flow 1	0.009028	0.005466	0.009174
Blocking Probability of Flow 2	0.003031	0.003057	0.003067
Blocking Probability of Flow 3	0.06016	0.002078	0.006126

Comparing these three experiments, we can see that this last one, Product Bound, has closer values to the reduced load than the cnet.R Simulator this can be explained by both sharing the same parameters, the load and the route of each flow where the cnet simulator takes a closer to reality approach.

## References

Valadas, R. (2021). *Laboratory guide nº 2 – Performance evaluation*. Lisboa: Instituto Superior Tecnico.

## Annex A

- *dtmc\_bernoulli\_plot.R* script provided in class.

```
1. library(ggplot2)
2. library(gridExtra)
3.
4. #Input parameters
5. alpha=0.9
6. beta=0.1
7.
8. #2-DTMC transition matrix
9. P=matrix(c(alpha,1-alpha,beta,1-beta),
10.         nrow=2,
11.         ncol=2,
12.         byrow=TRUE)
13. #DTMC mean
14. Avg=P[1,2]/(1+P[2,1]-P[1,1])
15.
16. Points1=c()
17. Points2=c()
18. NumPoints=200
19.
20. #2-DTMC simulation
21. CurrentState=0
22. for (i in 1:NumPoints) {
23.   p=P[CurrentState+1,CurrentState+1] #Extracts probability of transition to
   same state
24.   if (rbinom(1,1,p)==0) { #Changes state if no success
25.     CurrentState = (CurrentState+1)%2
26.   }
27.   Points1=c(Points1,CurrentState)
28. }
29.
30. #Bernoulli process simulation
31. for (i in 1:NumPoints) {
32.   Points2=c(Points2,rbinom(1,1,Avg))
33. }
34.
35. p1=qplot(y=Points1,ylab="2-DTMC")
36. p2=qplot(y=Points2,ylab="Bernoulli")
37. p3=grid.arrange(p1, p2, ncol = 1)
38. p3
39.
40. ggsave(p3,file="p3.jpeg",device="jpeg")
41.
```

## Annex B

```

1. mydata = read.table("2dtmcddata.txt", sep = "\t")
2.
3. mydatavector <- c(mydata$x)
4. mydatalength = length(mydatavector)
5.
6.
7. From0 = 0
8. From1 = 0
9.
10. From0to1 = 0
11. From0to0 = 0
12. From1to0 = 0
13. From1to1 = 0
14.
15.
16.
17. #Counting Transitions
18. for (i in 1:mydatalength) {
19.   #From 0 to 0
20.   if (setequal(mydatavector[i],0) && setequal(mydatavector[i+1],0)) {
21.     From0to0 = From0to0 + 1
22.     From0 = From0 + 1
23.   }
24.   #From 0 to 1
25.   else if (setequal(mydatavector[i],0) && setequal(mydatavector[i+1],1)) {
26.     From0to1 = From0to1 + 1
27.     From0 = From0 + 1
28.   }
29.   #From 1 to 0
30.   else if (setequal(mydatavector[i],1) && setequal(mydatavector[i+1],0)) {
31.     From1to0 = From1to0 + 1
32.     From1 = From1 + 1
33.   }
34.   #From 1 to 1
35.   else if (setequal(mydatavector[i],1) && setequal(mydatavector[i+1],1)) {
36.     From1to1 = From1to1 + 1
37.     From1 = From1 + 1
38.   }
39. }
40.
41. #Calculating Transition Probabilities
42. P00 = From0to0 / From0
43. P01 = From0to1 / From0
44. P10 = From1to0 / From1
45. P11 = From1to1 / From1
46.
47. cat("Transition Probability of P00 =", P00, "\n")
48. cat("Transition Probability of P01 =", P01, "\n")
49. cat("Transition Probability of P10 =", P10, "\n")
50. cat("Transition Probability of P11 =", P11, "\n")
51.

```



## Annex C

```

1. TimeSlots = 0
2. Success = 0
3. Thinking = 1
4. Backlogged = 2
5.
6. #Parameters
7. N = 30
8. p = 0.1
9. o = 0.9
10.
11. Clients <- c(N, 0)
12.
13. while(TimeSlots<60000) {
14.   ThinkingClients = 0
15.   BackloggedClients = 0
16.   #if number of clients thinking is greater than 0
17.   if (Clients[Thinking] > 0) {
18.     #for each client thinking see if it can transmit
19.     for (i in 1:Clients[Thinking]) {
20.       #Thinking Client transmits
21.       if (o > runif(1)) {
22.         ThinkingClients = ThinkingClients + 1
23.       }
24.     }
25.   }
26.   #if number of clients Backlogged is greater than 0
27.   if (Clients[Backlogged] > 0) {
28.     #for each client backlogged see if it can transmit
29.     for (i in 1:Clients[Backlogged]) {
30.       #If Backlogged Clients transmits
31.       if (p > runif(1)) {
32.         BackloggedClients = BackloggedClients + 1
33.       }
34.     }
35.   }
36.
37.   # If a Collision takes place
38.   if (ThinkingClients + BackloggedClients > 1) {
39.     #All thinking clients become backlogged clients
40.     Clients[Thinking] = Clients[Thinking] - ThinkingClients
41.     Clients[Backlogged] = Clients[Backlogged] + ThinkingClients
42.   }
43.
44.   else{
45.     #A backlogged Client transmits with success
46.     if (Clients[Backlogged] > 1 && BackloggedClients == 1) {
47.       Success = Success + 1
48.       #Backlogged Client goes to Thinking state
49.       Clients[Backlogged] = Clients[Backlogged] - 1
50.       Clients[Thinking] = Clients[Thinking] + 1
51.     }
52.     #A Thinking Client transmits with success
53.     else if (Clients[Thinking] > 1 && ThinkingClients == 1) {
54.       Success = Success + 1
55.     }
56.   }
57.   TimeSlots = TimeSlots +1
58. }
59. SystemThroughput = Success / TimeSlots
60. cat('System Throughput = ', SystemThroughput,'\n')
61. cat('N = ',N,'\n')

```

```
62. cat('Sigma = ', o, '\n')
63. cat('p = ', p, '\n')
64.
```

## Annex D

```

1. #Transition Values
2. lambda2to1 = 1
3. lambda2to3 = 1
4. lambda3to1 = 1
5. lambda3to2 = 1
6. lambda1to2 = 2
7.
8. #Transition Probability from i to j
9. P1to2 = lambda1to2 / lambda1to2
10. P2to1 = lambda2to1 / (lambda2to1 + lambda2to3)
11. P2to3 = lambda2to3 / (lambda2to1 + lambda2to3)
12. P3to1 = lambda3to1 / (lambda3to1 + lambda3to2)
13. P3to2 = lambda3to2 / (lambda3to1 + lambda3to2)
14.
15. #Transition rate out of state Vi
16. V1 = lambda1to2
17. V2 = lambda2to3 + lambda2to1
18. V3 = lambda3to2 + lambda3to1
19.
20. #Limiting State Probabilities - Theoretical Values
21. ctmc_theoretical <- function() {
22.   a <- matrix(c(-2, 2, 1,
23.                 1, -2, 1,
24.                 1, 1, 1), nrow = 3, ncol = 3)
25.   b <- matrix(c(0, 0, 1), nrow = 3, ncol = 1)
26.   result = solve(a, b)
27.   Pi1 = result[1]
28.   Pi2 = result[2]
29.   Pi3 = result[3]
30.
31.   cat("==Theoretical Values==\n")
32.   print(Pi1)
33.   print(Pi2)
34.   print(Pi3)
35. }
36.
37. #Limiting State Probabilities - View 1
38. ctmc_view_1 <- function() {
39.   CurrentState = 1
40.   StateTime1 = 0
41.   StateTime2 = 0
42.   StateTime3 = 0
43.
44.   for (i in 1:10000) {
45.     if (CurrentState == 1) {
46.       T1 = rexp(1, V1)
47.       StateTime1 = StateTime1 + T1
48.       CurrentState = 2
49.     }
50.     else if (CurrentState == 2) {
51.       T2 = rexp(1, V2)
52.       StateTime2 = StateTime2 + T2
53.
54.       #It creates a 0.5 chance of jumping from state 2 to state 1 or 3
55.       JumpProb = runif(1)
56.       if (JumpProb > P2to1) {
57.         CurrentState = 1
58.       } else{

```

```

59.     CurrentState = 3
60.   }
61. }
62.
63. else if (CurrentState == 3) {
64.   T3 = rexp(1, V3)
65.   StateTime3 = StateTime3 + T3
66.
67.   #It creates a 0.5 chance of jumping from state 3 to state 1 or 2
68.   JumpProb = runif(1)
69.   if (JumpProb > P3to2) {
70.     CurrentState = 1
71.   } else{
72.     CurrentState = 2
73.   }
74. }
75. }
76. #Calculation of the proportion of time that the process spends in state i
77. Pi1 = StateTime1 / (StateTime1 + StateTime2 + StateTime3)
78. Pi2 = StateTime2 / (StateTime1 + StateTime2 + StateTime3)
79. Pi3 = StateTime3 / (StateTime1 + StateTime2 + StateTime3)
80. cat("==View 1 Values==\n")
81. print(Pi1)
82. print(Pi2)
83. print(Pi3)
84. }
85.
86. #Limiting State Probabilities - View 2
87. ctmc_view_2 <- function() {
88.   CurrentState = 1
89.   StateTime1 = 0
90.   StateTime2 = 0
91.   StateTime3 = 0
92.   #Instantaneous transition rate
93.   q1to2 = P1to2 * V1
94.   q2to1 = P2to1 * V2
95.   q2to3 = P2to3 * V2
96.   q3to1 = P3to1 * V3
97.   q3to2 = P3to2 * V3
98.
99.   for (i in 1:10000) {
100.    if (CurrentState == 1) {
101.      T1to2 = rexp(1, q1to2)
102.      StateTime1 = StateTime1 + T1to2
103.
104.      CurrentState = 2
105.    }
106.    else if (CurrentState == 2) {
107.      T2to1 = rexp(1, q2to1)
108.      T2to3 = rexp(1, q2to3)
109.      StateTime2 = StateTime2 + min(T2to1, T2to3)
110.
111.      if (T2to1 < T2to3) {
112.        CurrentState = 1
113.      } else{
114.        CurrentState = 3
115.      }
116.    }
117.    else if (CurrentState == 3) {
118.      T3to1 = rexp(1, q3to1)
119.      T3to2 = rexp(1, q3to2)

```

```
120.         StateTime3 = StateTime3 + min(T3to1, T3to2)
121.
122.         if (T3to1 < T3to2) {
123.             CurrentState = 1
124.         } else{
125.             CurrentState = 2
126.         }
127.     }
128. }
129. Pi1 = StateTime1 / (StateTime1 + StateTime2 + StateTime3)
130. Pi2 = StateTime2 / (StateTime1 + StateTime2 + StateTime3)
131. Pi3 = StateTime3 / (StateTime1 + StateTime2 + StateTime3)
132.
133. cat("==View 2 Values==\n")
134. print(Pi1)
135. print(Pi2)
136. print(Pi3)
137. }
138.
139. ctmc_theoretical()
140. ctmc_view_1()
141. ctmc_view_2()
142.
```

## Annex E

```

1. #Compare the sample variance of the average queuing delay estimates
2.
3. #input parameters
4. Ro = 0.5 #system utilization
5. s1 = 1
6. p1 = 0.99 #mice (size and probability)
7. s2 = 101
8. p2 = 0.01 #elephants (size and probability)
9. StoppingCondition = 10000
10.
11. #parameters calculated from the input parameters
12. ServiceRate = 1 / (s1 * p1 + s2 * p2)
13. ArrivalRate = ServiceRate * Ro
14.
15. n = 0
16. AvgDelayMM1Total = 0
17. AvgDelayBimodalTotal = 0
18. CollectionAvgDelayMM1 <- c()
19. CollectionAvgDelayBimodal <- c()
20.
21. while (n < 100) {
22.   #M/M/1
23.   Time = 0
24.   NumQueueCompleted = 0
25.   ServerStatus = 0
26.   NumInQueue = 0
27.   AcumDelay = 0
28.   QueueArrivalTime = c()
29.   EventList = c(rexp(1, ArrivalRate), Inf)
30.
31.   #simulation cycle
32.   while (NumQueueCompleted < StoppingCondition) {
33.     NextEventType = which.min(EventList)
34.     Time = EventList[NextEventType]
35.     if (NextEventType == 1) {
36.       EventList[1] = Time + rexp(1, ArrivalRate)
37.       if (ServerStatus == 1) {
38.         QueueArrivalTime = c(QueueArrivalTime, Time)
39.         NumInQueue = NumInQueue + 1
40.       }
41.     } else {
42.       NumQueueCompleted = NumQueueCompleted + 1
43.       ServerStatus = 1
44.       EventList[2] = Time + rexp(1, ServiceRate)
45.     }
46.   }
47.   else {
48.     if (NumInQueue == 0) {
49.       ServerStatus = 0
50.       EventList[2] = Inf
51.     }
52.     else {
53.       AcumDelay = AcumDelay + Time - QueueArrivalTime[1]
54.       QueueArrivalTime = QueueArrivalTime[-1]
55.       NumInQueue = NumInQueue - 1
56.       NumQueueCompleted = NumQueueCompleted + 1
57.       EventList[2] = Time + rexp(1, ServiceRate)
58.     }
59.   }
60. }
61.

```

```

62. AvgDelayMM1 = AcumDelay / NumQueueCompleted
63. CollectionAvgDelayMM1 <- append(CollectionAvgDelayMM1, AvgDelayMM1)
64.
65.
66. #M/G/1 Bimodal
67. Time = 0
68. NumQueueCompleted = 0
69. ServerStatus = 0
70. NumInQueue = 0
71. AcumDelay = 0
72. QueueArrivalTime = c()
73. QueueJobSize = c()
74. EventList = c(rexp(1, ArrivalRate), Inf)
75. #simulation cycle
76. while (NumQueueCompleted < StoppingCondition) {
77.   NextEventType = which.min(EventList)
78.   Time = EventList[NextEventType]
79.   if (NextEventType == 1) {
80.     EventList[1] = Time + rexp(1, ArrivalRate)
81.     JobSize = ifelse(runif(1) < p1, s1, s2)
82.     if (ServerStatus == 1) {
83.       QueueArrivalTime = c(QueueArrivalTime, Time)
84.       QueueJobSize = c(QueueJobSize, JobSize)
85.       NumInQueue = NumInQueue + 1
86.     } else{
87.       NumQueueCompleted = NumQueueCompleted + 1
88.       ServerStatus = 1
89.       EventList[2] = Time + JobSize
90.     }
91.   }
92.   else {
93.     if (NumInQueue == 0) {
94.       ServerStatus = 0
95.       EventList[2] = Inf
96.     } else {
97.       AcumDelay = AcumDelay + Time - QueueArrivalTime[1]
98.       JobSize = QueueJobSize[1]
99.       QueueArrivalTime = QueueArrivalTime[-1]
100.      QueueJobSize = QueueJobSize[-1]
101.      NumInQueue = NumInQueue - 1
102.      NumQueueCompleted = NumQueueCompleted + 1
103.      EventList[2] = Time + JobSize
104.    }
105.  }
106. }
107. AvgDelayBimodal = AcumDelay / NumQueueCompleted
108. CollectionAvgDelayBimodal <-
109.   append(CollectionAvgDelayBimodal, AvgDelayBimodal)
110.   n = n + 1
111. }
112. #Sample variance of the 100 values of Avg Delay
113. AvgDelayMM1Var = var(CollectionAvgDelayMM1)
114. AvgDelayBimodalVar = var(CollectionAvgDelayBimodal)
115.
116. #M/G/1 Pollaczek-Khinchine
117. es = s1 * p1 + s2 * p2
118. es2 = s1 ^ 2 * p1 + s2 ^ 2 * p2
119. wqPK = ArrivalRate * es2 / (2 * (1 - ArrivalRate * es))
120. C2PK = es2 / es ^ 2 - 1
121.
122. cat(sprintf("Arrival rate = %f", ArrivalRate), "\n")
123. cat(sprintf("Service rate = %f", ServiceRate), "\n")
124. cat("\n")
125.
126.

```

```
127. cat(  
128.     sprintf(  
129.         "Estimated Variance of the average queuing delay M/M/1 = %f",  
130.         AvgDelayMM1Var  
131.     ),  
132.     "\n"  
133. )  
134. cat("\n")  
135. cat(  
136.     sprintf(  
137.         "Estimated Variance of the average queuing delay M/Bimodal/1 = %f",  
138.         AvgDelayBimodalVar  
139.     ),  
140.     "\n"  
141. )  
142. )  
143.
```



## Annex F

```

1. # simulate two queues served by the DRR (Deficit Round
2. # Robin) scheduling algorithm. The queues have
3. # independent Poisson arrivals and uniformly
4. #distributed packet sizes
5.
6.
7.
8. #Input parameters
9. ArrivalRate = c(1, 2) #Arrival rate at each queue
10. AvgPacketSize = c(1000, 3000) #Average packet size of packets arriving at each queue
11. LinkCapacity = 1000 #Link capacity
12. Quantum = c(1000, 2000)
13. DeficitCounter = c(0,0)
14.
15.
16.
17. #Initialization
18. Time = 0 #Simulation clock
19. NumSysCompleted = 0 #Number of packets that crossed the system
20. LinkStatus = 0 #Link status (idle or busy)
21. NumInQueue = c(0, 0) #Number of packets in each queue
22. QueuePacketSize = list(c(), c()) #Initializes list storing packet sizes at each queue
23. AccumBits = c(0, 0) #Accumulator of bits transmitted from each queue
24. EventList = c(rexp(1, ArrivalRate[1]), rexp(1, ArrivalRate[2]), Inf) #Event list
    initialization
25. CurrentQueue = 1
26.
27.
28.
29. #Simulation loop
30. while (NumSysCompleted < 10000) {
31.   NextEventType = which.min(EventList) #Determines next event
32.   Time = EventList[NextEventType] #Jumps clock to time of next event
33.   if (NextEventType == 1 || NextEventType == 2) {
34.     #Arrival events at queue 1 or queue 2
35.     EventList[NextEventType] = Time + rexp(1, ArrivalRate[NextEventType]) #Next arrival to
    queue
36.     PacketSize = sample(seq(AvgPacketSize[NextEventType] - 500, AvgPacketSize[NextEventType]
    +
37.                             500, 100), 1) #Define packet size of arriving packet
38.     if (LinkStatus == 1) {
39.       #Link is busy
40.       QueuePacketSize[[NextEventType]] = c(QueuePacketSize[[NextEventType]], PacketSize)
    #Queues arriving packet
41.       NumInQueue[NextEventType] = NumInQueue[NextEventType] + 1 #Increments number of
    packets in this queue
42.     }
43.     else {
44.       #Link not busy
45.       LinkStatus = 1 #Link becomes busy
46.       ServedPacketSize = PacketSize #Places packet in link (its packet size)
47.       ServedQueue=NextEventType #Stores queue of packet being served
48.       EventList[3] = Time + ServedPacketSize / LinkCapacity #Schedules departure of this
    packet
49.     }
50.
51.   } else {
52.     #Departure event
53.     NumSysCompleted = NumSysCompleted + 1 #Increments number of packets that crossed the
    system

```

```

54.     AcumBits[ServedQueue] = AcumBits[ServedQueue] + ServedPacketSize #Accumulates bits of
served packet
55.
56.     if (all(NumInQueue == 0)) {
57.         #System is empty
58.         EventList[3] = Inf #No departure from empty system
59.         LinkStatus = 0 #Link becomes idle
60.         #Resets the deficit counters of both queues
61.         DeficitCounter[1] = 0
62.         DeficitCounter[2] = 0
63.         CurrentQueue=1
64.
65.     } else if (any(NumInQueue == 0)) {
66.         #Only one queue empty
67.         CurrentQueue = which(NumInQueue != 0)#Discovers non-empty queue
68.
69.         EmptyQueue = which(NumInQueue == 0)#Discovers empty queue
70.         DeficitCounter[EmptyQueue] = 0 #Resets the deficit counters of empty queue
71.
72.         PacketSize = QueuePacketSize[[CurrentQueue]][1] #Reads packet size of non-empty queue
73.         ServedPacketSize = PacketSize #Places packet in link (its packet size)
74.         ServedQueue=CurrentQueue #Place packet in link (its queue)
75.
76.         while (TRUE) {
77.             if (DeficitCounter[CurrentQueue] >= PacketSize) {
78.                 QueuePacketSize[[CurrentQueue]] = QueuePacketSize[[CurrentQueue]][-1] #Removes
packet size from non-empty queue
79.                 NumInQueue[CurrentQueue] = NumInQueue[CurrentQueue] - 1 #Updates number of clients
in non-empty queue
80.                 EventList[3] = Time + ServedPacketSize / LinkCapacity #Schedules departure of this
packet
81.                 DeficitCounter[CurrentQueue] = DeficitCounter[CurrentQueue] - PacketSize
82.                 break
83.             } else{
84.                 DeficitCounter[CurrentQueue] = DeficitCounter[CurrentQueue] +
Quantum[CurrentQueue]
85.             }
86.         }
87.
88.
89.     } else {
90.         #No queue empty
91.
92.         while (TRUE) {
93.
94.             PacketSize = QueuePacketSize[[CurrentQueue]][1] #Places packet in link (its packet
size)
95.             if (DeficitCounter[CurrentQueue] >= PacketSize) {
96.                 ServedPacketSize=PacketSize
97.                 ServedQueue=CurrentQueue #Place packet in link (its queue)
98.                 QueuePacketSize[[CurrentQueue]] = QueuePacketSize[[CurrentQueue]][-1] #Removes
packet size from current queue
99.                 NumInQueue[CurrentQueue] = NumInQueue[CurrentQueue] - 1 #Updates number of clients
in current queue
100.                 EventList[3] = Time + ServedPacketSize / LinkCapacity #Schedules departure of
this packet
101.                 DeficitCounter[CurrentQueue] = DeficitCounter[CurrentQueue] - PacketSize
#Removes packet size from DeficitCounter
102.                 break
103.             }
104.             else {
105.                 #Change Queue
106.                 if (CurrentQueue == 1) {
107.                     CurrentQueue = 2
108.                 } else {

```

```
109.         CurrentQueue = 1
110.     }
111.     DeficitCounter[CurrentQueue] = DeficitCounter[CurrentQueue] +
Quantum[CurrentQueue]#Adds Quantum to the Deficit Counter
112. }
113. }
114. }
115. }
116. }
117.
118.
119.
120. #Print results
121. sprintf("The throughput of queue 1 is %f", AcumBits[1] / Time)
122. sprintf("The throughput of queue 2 is %f", AcumBits[2] / Time)
123.
```

## Annex G

```

1. linkCapacity=c(100e6,100e6,200e6)
2. servRate=linkCapacity/8000
3.
4. #rate of the flows
5.
6. lambda1=7500
7. lambda2=10000
8. lambda3=5000
9.
10. #list of flows
11.
12. flow1=list(lambda1,list(1,3))
13. flow2=list(lambda2,list(2,3))
14. flow3=list(lambda3,list(3))
15. flowsList=list(flow1,flow2,flow3)
16. linkLambda=c()
17.
18. #list of rates
19. lambdasList=c(lambda1,lambda2,lambda3)
20.
21. for (i in 1:length(linkCapacity)){
22.   linkLambda[i]=0
23. }
24.
25. for ( j in 1:length(flowsList)){
26.   for (i in 1:length(linkCapacity)){
27.     if ( i %in% flowsList[[j]][[2]]){
28.       linkLambda[i]=linkLambda[i]+flowsList[[j]][[1]]
29.     }
30.   }
31. }
32.
33. L=c()
34. W=c()
35.
36.
37. W=rep(0,length(linkCapacity))
38.
39. #We calculate the packet size
40.
41. for(i in 1:length(linkCapacity)){
42.   L[i]=linkLambda[i]/(servRate[i]-linkLambda[i])
43. }
44.
45.
46. #We calculate the average delay of each flow
47.
48. for (i in 1:length(flowsList)){
49.   for (j in 1:length(servRate)){
50.     if (j %in% flowsList[[i]][[2]]){
51.       W[i]=W[i]+(1/(servRate[j]-linkLambda[j]))
52.     }
53.   }
54. }
55.
56. print("Average delay in the network: ")
57.
58. # We calculate the average delay in the network
59. w = sum(L)/sum(lambdasList)
60. print(w)
61.

```

```
62. print(L)
63.
64. print("Average number of packets in the network: ")
65. print(sum(L))
66.
67. print("Average packet delay of each flow: ")
68. print(W)
69.
70.
71.
72.
73.
74.
```