Queue Simulation

Project - ECE 610

Filza Mazahir

20295951

fmazahir@uwaterloo.ca

Table of Contents

[Note on Code 3](#_Toc509791889)

[Question 1 – Generate Exponential Random Variable 3](#_Toc509791890)

[Question 2 – Simulator for M/M/1, D/M/1 and M/G/1 Queue 4](#_Toc509791891)

[Simulation Initialization 5](#_Toc509791892)

[Event Scheduler 6](#_Toc509791893)

[Run Simulation function (and computing output variables) 9](#_Toc509791894)

[Creating a Queue to run simulation and accessing output variables 10](#_Toc509791895)

[System Stability 10](#_Toc509791896)

[Question 3 – M/M/1, D/M/1, and M/G/1 – E[N] & PIDLE (0.35 ≤ ρ ≤ 0.95) 11](#_Toc509791897)

[E[N] as a function of ρ 11](#_Toc509791898)

[PIDLE as a function of ρ 12](#_Toc509791899)

[Question 4 – M/M/1, D/M/1, and M/G/1 – E[N] & PIDLE (ρ = 1.5) 13](#_Toc509791900)

[Question 5 – M/M/1 and D/M/1 – Comparison of E[N] and Ea[N] 14](#_Toc509791901)

[Question 6 – M/M/1, D/M/1, and M/G/1 – Comparison of E[T] 15](#_Toc509791902)

[Question 7 – Simulator for M/D/1/K Queue 16](#_Toc509791903)

[Question 8 – M/D/1/K– PLOSS (0.4 ≤ ρ ≤ 3) 17](#_Toc509791904)

[Question 9 – M/D/2 and M/D/1 (rate 2C) – E[N] (0.35 ≤ ρ ≤ 0.95) 18](#_Toc509791905)

# Note on Code

The queue simulation is coded in Python 3. The external libraries used for this project are numpy and matplotlib. The makefile submitted makes two assumptions.

1) It assumes that the Python libraries numpy and matplotlib are already installed on the server

2) It assumes that Python 3 is executed with the keyword python3, and executes the command python3 ECE610ProjectFilzaMazahir.py. If that is not the case, and Python 3 on the server has the default keyword python, please use python ECE610ProjectFilzaMazahir.py to run the source code.

The code is commented heavily, and all variables and functions are explained in the source code file. It takes around 18 minutes to run the complete code.

# Question 1 – Generate Exponential Random Variable

Exponential Distribution:

Probability Density Function

Cumulative Density Function

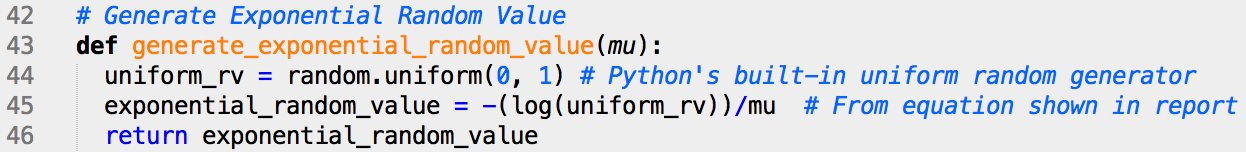
Let U be uniform random variable, and X be exponential random variable

Let be uniformly distributed random variable from 0 to 1, then to generate exponentially distributed random variable from random uniform variables :

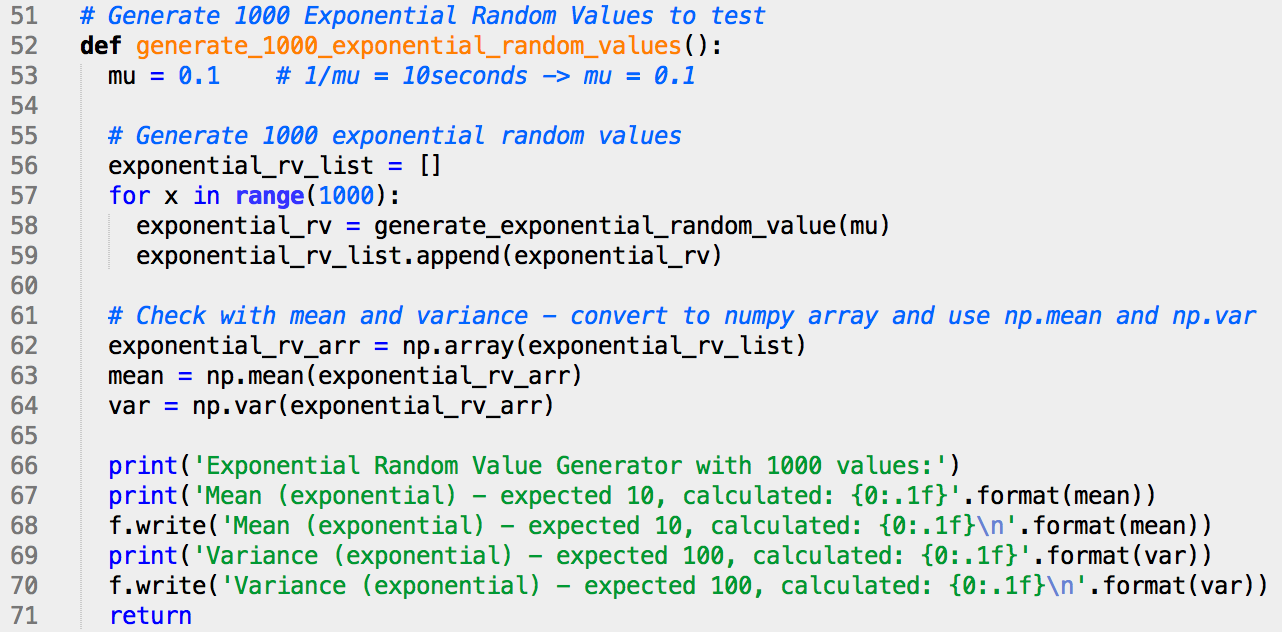
Since and are both uniformly distributed random numbers between 0 and 1:

Therefore:

Using the equation, the following Python code was written to generate an exponential random variable. Note that the variable *mu* in the code is for the parameter μ in the equation.



To generate 1000 exponential random variables:

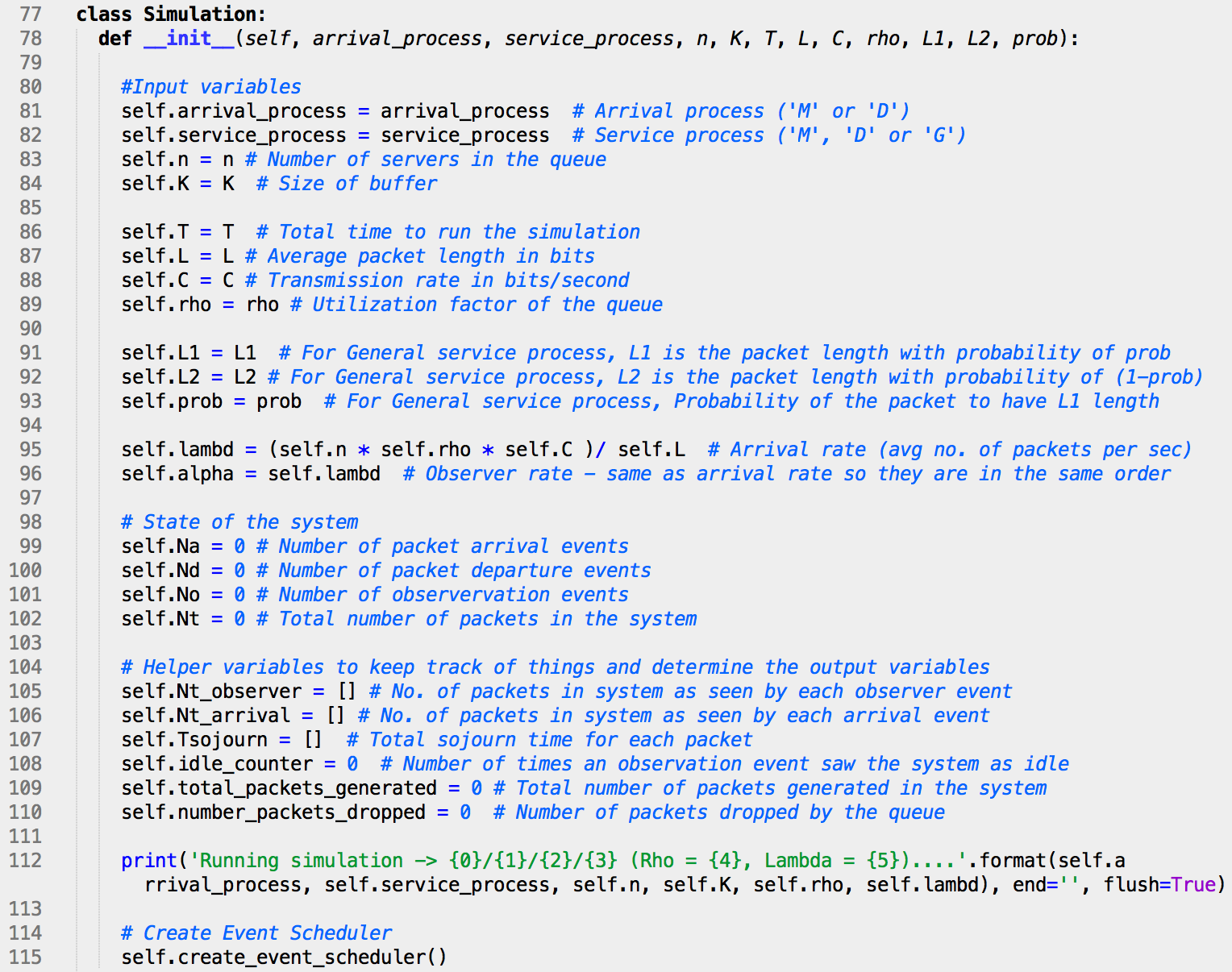


The expected values for the above code are mean of 10 and variance of 100. Running the above code gave mean of 10.4 and variance of 100.7, which is very close to the expected values.

# Question 2 – Simulator for M/M/1, D/M/1 and M/G/1 Queue

The simulator is built using a class Simulation which is initialized in the beginning with all the parameters needed to run the simulation. There are two main functions used for the simulation, first is create\_event\_scheduler(), which creates a double ended queue called Event\_Scheduler consisting of observation, arrival and departure events. Second is run\_simulation()which goes through the list of events in the *Event\_Scheduler*, dequeues events from the beginning, and then updates the system metrics (Nt - number of packets in the system) accordingly.

## Simulation Initialization



The simulator is built using a class Simulation as shown in the code above. When the simulation is initialized in the beginning, it gets all of its parameters such as arrival\_process, service\_process, n, K, T, L, C, rho, L1, L2, prob.

The variables for the type of queue are given as follows: arrival\_process and service\_process are specified by ‘M’, ‘G’, or ‘D’ for Poisson, General or Deterministic distribution respectively. n is for the number of servers in the queue, and K is the size of the buffer given in number of packets.

Other variables provided for the initialization are as follows: T is the total time for the simulation to run, L is the average length of packet in bits, C is the transmission rate of packet in bits/second, rho is the utilization factor of the queue given by are also provided in the initialization. The variables L1, L2 and prob are also provided for the General distribution where the packet length has a bipolar distribution, and is determined as L1 with probability of prob, and L2 with probability of 1 – prob. For queues that do not have a General distribution, a value of 0 can be input for L1, L2 and prob as they do not affect the code otherwise.

Once these variables are initialized, lambd, which is the arrival rate, that is, average number of packets generated per second is calculated using . The variable alpha, which is the observation rate is then set equal to lambd, as both the observation rate and arrival rate are supposed to be of the same order

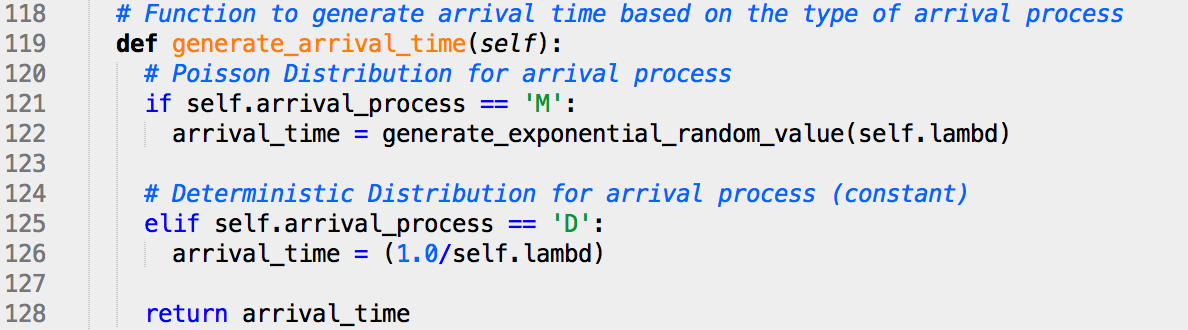
The state of the system is defined as the number of packets in the system, given by the variable Nt. Other variables to keep track of this are Na (number of observation events), Nd (number of departure events), and No (number of observation events). These are all initialized to 0, as only get updated when an event occurs.

Other helper variables are introduced in the Simulation class, which aid in calculating the output variables. Nt\_observer is an array that has the number of packets in the system as seen by all observer events. Similarly, Nt\_arrival is an array that has the number of packets in the system as seen by all arrival packets. The variable Tsojourn is an array that stores the total sojourn time taken by all packets in the system. The variable idle\_counter is the number of times that an observer event sees the system as idle, and total\_packets\_generated keeps track of the total number of packets generated in the system. The variable number\_packets\_dropped keep track of any packets lost because of buffer being too full, and is only updated for the M/D/1/K queue. The variables idle\_counter, total\_packets\_generated, and number\_packets\_dropped are initialized to 0 as that is the case initially.

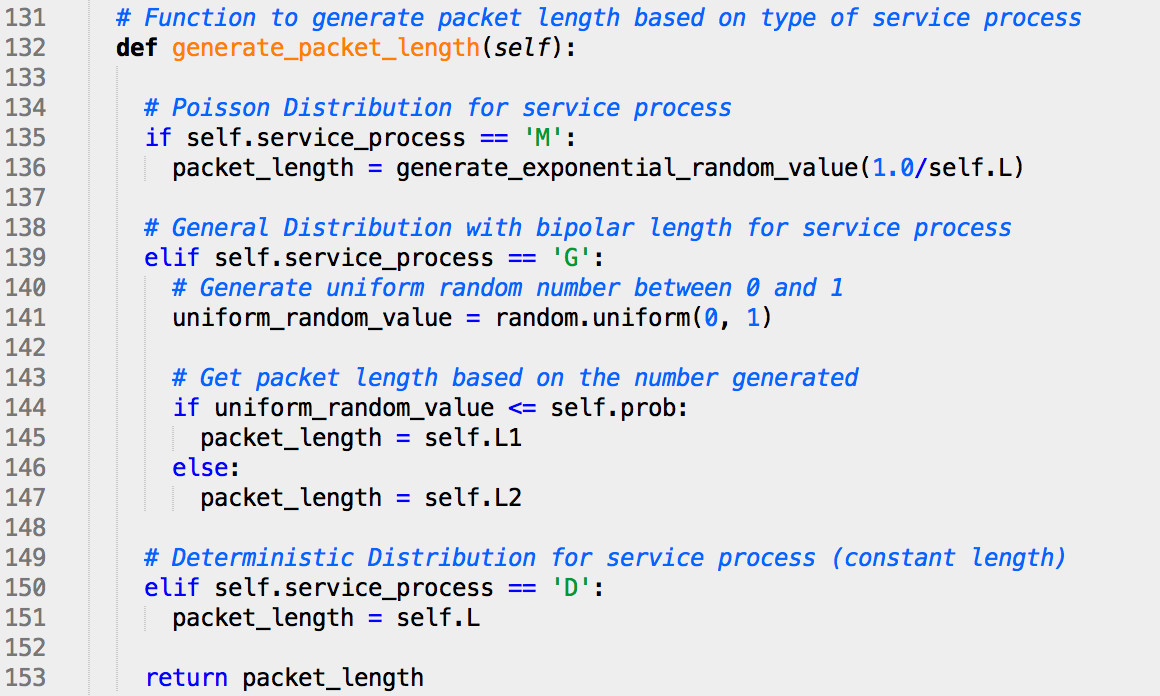
## Event Scheduler

The Event Scheduler is created upon initialization in the create\_event\_scheduler() function. Event\_Scheduler is a double ended queue which consists of events where each event is in the form of a two variable tuple (event\_type, event\_time). The event\_type is given in the form of ‘O’, ‘A’ and ‘D’ for Observer event, Arrival event and Departure event respectively, and event\_time is the time when the particular event happens.

*Helper functions:* Two helper functions are used by the create\_event\_scheduler() function: generate\_arrival\_time() and generate\_packet\_length().



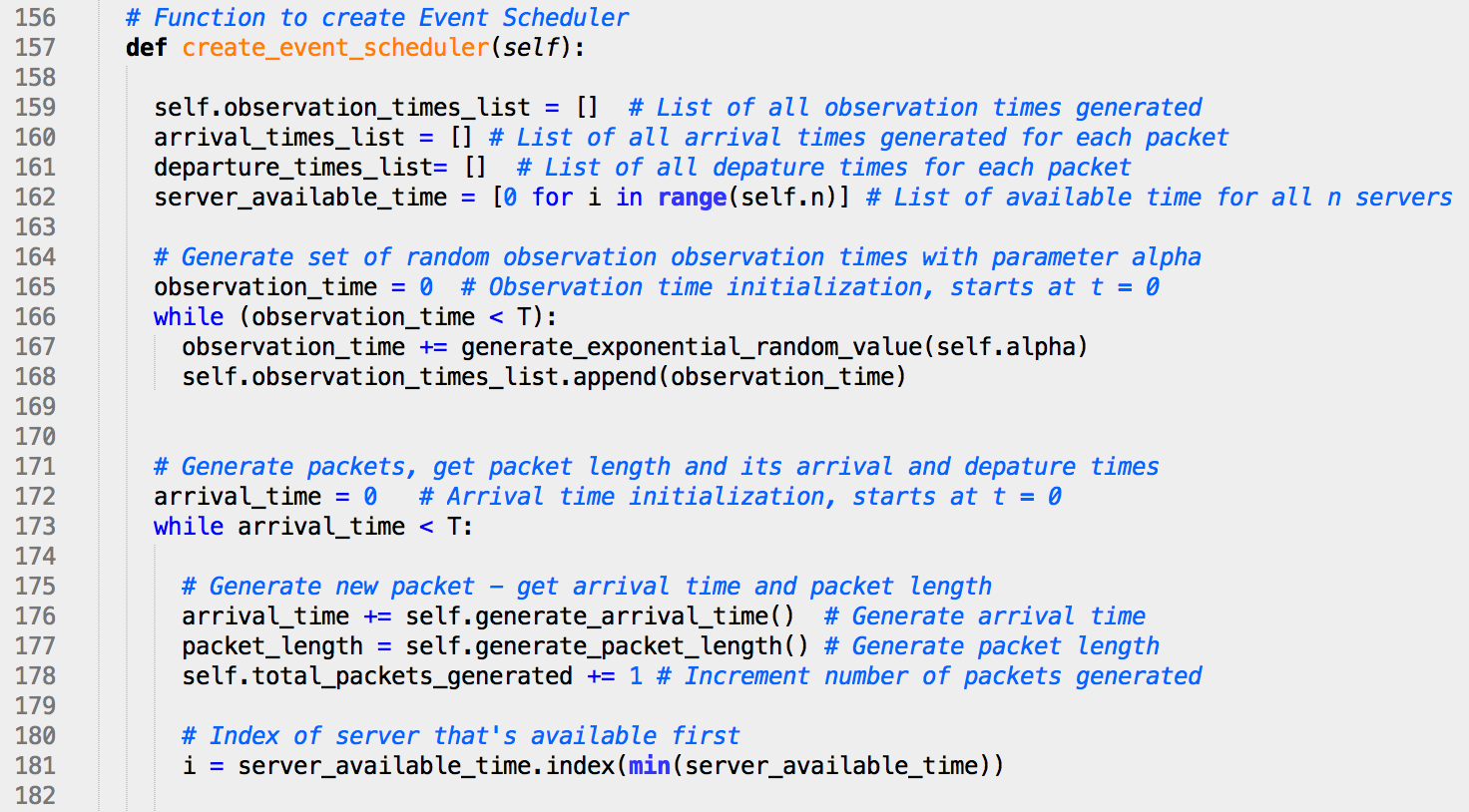
The function generate\_arrival\_time() shown above checks the class variable arrival\_process. If it is ‘M’ for Poisson, then it uses the generate\_exponential\_random\_value() function created in Question 1 to generate arrival time. If the arrival\_process is ‘D’ for Deterministic, then arrival time is calculated as the constant of 1/λ.

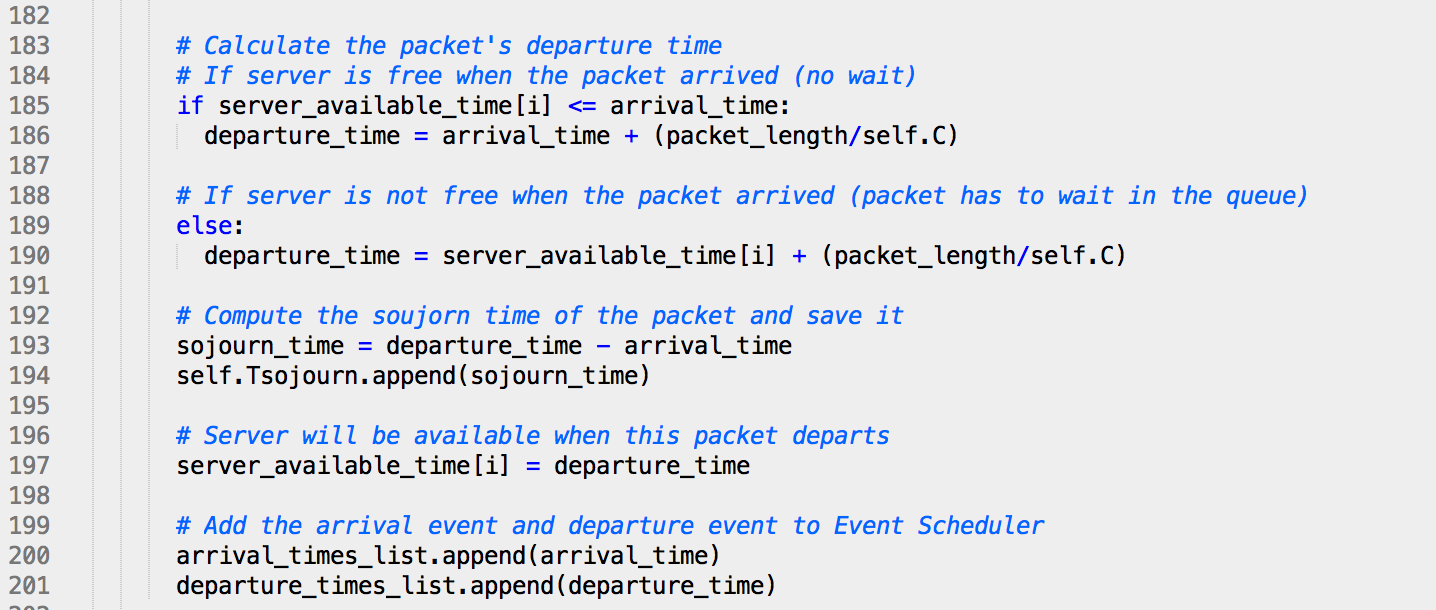


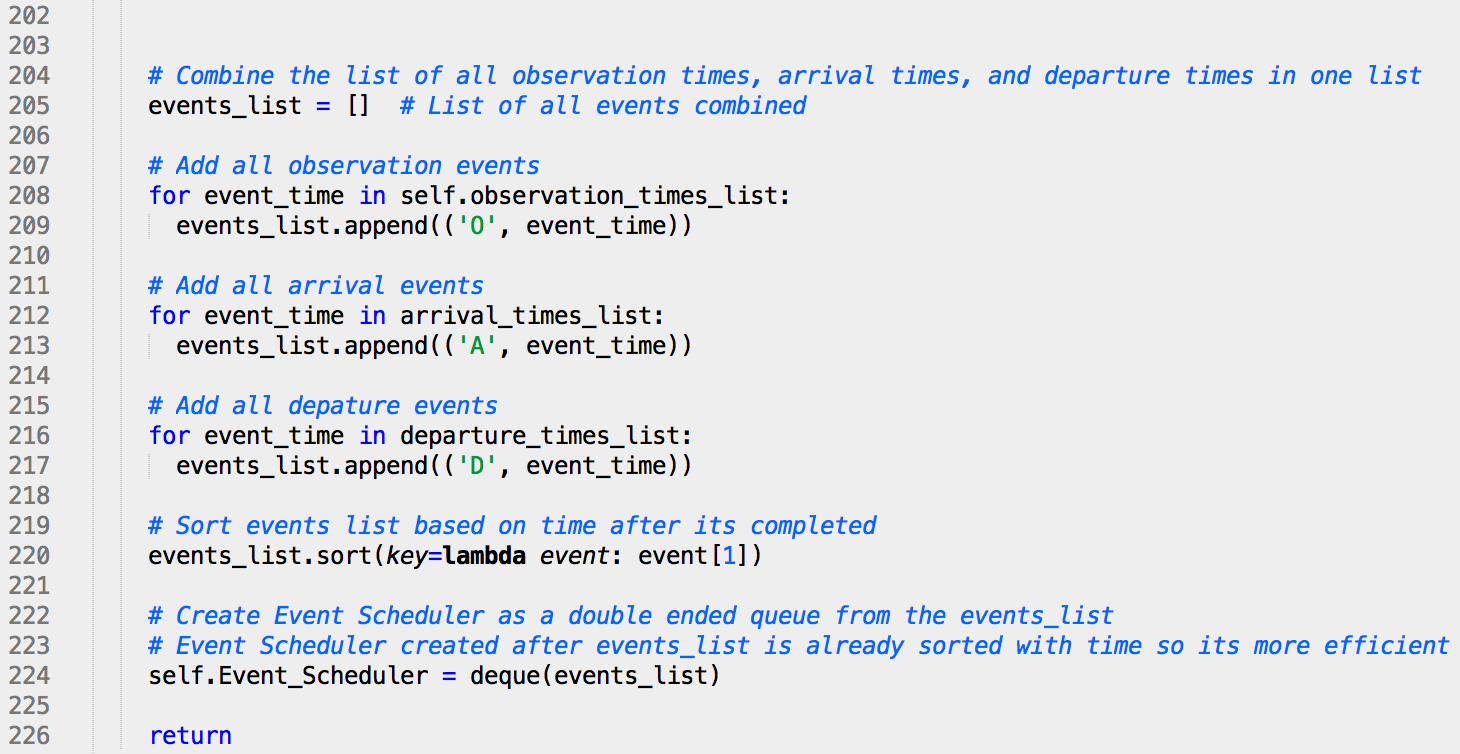
The service rate of a process is given as , and since n and C are constant for each packet, L (packet length) is what changes based on type of service process. Therefore, the function generate\_packet\_length() shown above checks for the class variable service\_process. If it is ‘M’, then it uses the generate\_exponential\_random\_value() function to generate exponential random length with the parameter 1/L as the mean. If the service process is ‘G’, then it uses\_\_\_\_

If the service process is ‘D’ for deterministic, then packet length is calculated as the constant of L.

*Create Event Scheduler Function:*







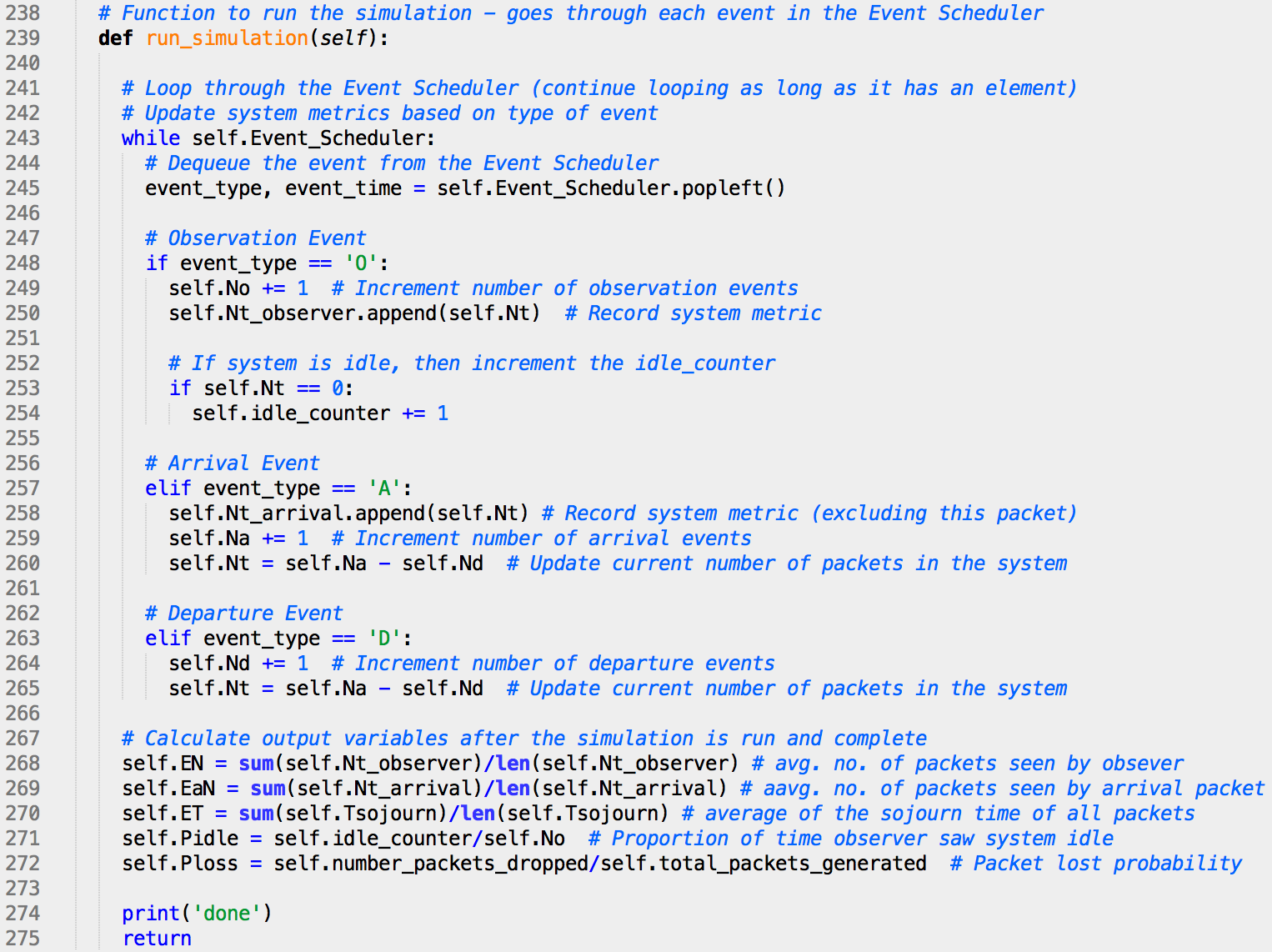
The create\_event\_scheduler() function shown above has three lists observation\_times\_list, arrival\_times\_list, and departure\_times\_list. This is where the time event time for each observation, arrival and departure is added respectively. The server\_available\_time is a list of size n (either 1 or 2), and is initialized as 0. It stores the times that each server will become available.

First, a set of observation times is generated and stored in the observation\_times\_list. Then a new packet’s arrival time and packet length is generated, and it is checked which server will become available first to service this packet. Note that in the case of 1 server, server\_available\_time is of size 1 only, and its minimum is its only element. Then based on the server\_available\_time it is checked if the server is free to service this packet right away or will this packet have to wait. If server is free, then departure time is calculated as arrival\_time of the packet plus transmission time of packet\_length/C . If server is not free, then departure time is calculated as the time server becomes available plus transmission time of packet\_length/C. The sojourn time (total time) of that packet is then calculated and appended in the T\_sojourn list, and the packet’s arrival and departure times are added in the arrival\_times\_list and departure\_times\_list respectively.

Once all the packets are generated with its arrival and departure times computed, the function goes through each of the three lists, observation\_times\_list, departure\_times\_list, and arrival\_times\_list, and adds the events to an events\_list in the form of tuples (event\_type, event\_time), where event\_type is ‘O’, ‘A’, or ‘D’, and event\_time is the corresponding time.

After that, the events\_list is sorted with time using Python’s sort() function so the FIFO method could be applied by transmitting the first packet in the list. A double ended queue called Event\_Scheduler is then created from the events\_list. Note that the reason for a separate events\_list and Event\_Scheduler is that events\_list uses the data structure of a Python list, which is faster to sort, and Event\_Scheduler is a double ended queue which is more efficient when dequeueing the event from the beginning of the list.

## Run Simulation function (and computing output variables)

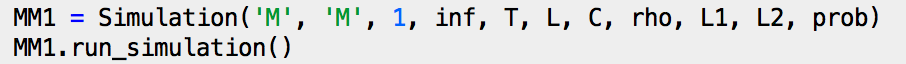


This function run\_simulation() shown on the previous goes through the Event\_Scheduler, and dequeues one event at a time. Based on the type of event (observer, arrival, or departure), it updates the system metrics (Nt - number of packets in the system) accordingly. The observation and arrival events also keep track of the variable Nt in Nt\_observer and Nt\_arrival respectively.

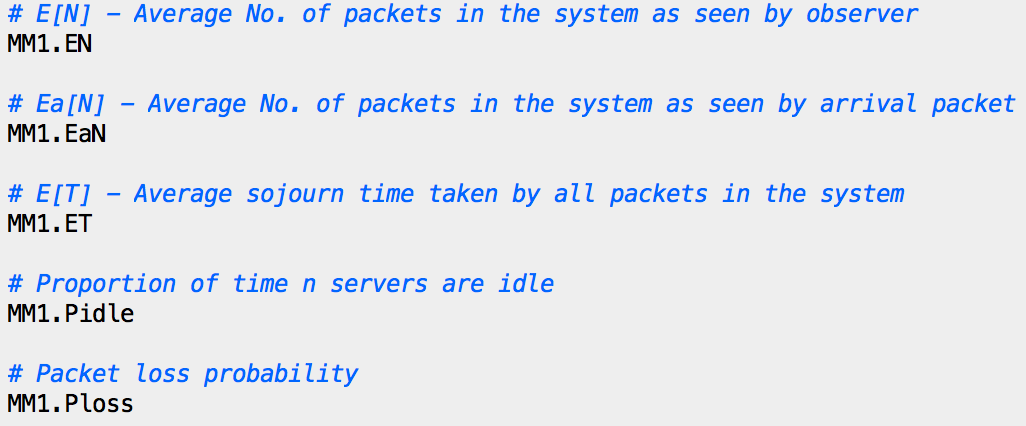
In addition, at each observation event the function also checks whether the system is idle or not, and if it is, it increments the idle\_counter accordingly.

After going through the whole Event\_Scheduler, the output variables are calculated. The variable EN is for E[N] (average number of packets in the system as seen by an observer), and is calculated by taking the average of the Nt\_observer list. The variable EaN is for Ea[N] (average number of packets in the system as seen by an arrival packet), and is calculated by taking the average of the Nt\_arrival list. The variable ET is for E[T] (average sojourn time), and is calculated by taking the average of the T\_sojourn list. PIDLE is denoted by the variable Pidle, and is calculated as the number of times the system was observed to be idle divided by the total number of observation events. PLOSS is denoted by the variable Ploss, and is calculated as the number of packets dropped in the system divided by the total number of packets generated. Note that since the functionality of M/D/1/K queue is not added to the code yet, so Ploss is always 0 since number\_packets\_dropped was initialized to 0 and remains the same.

## Creating a Queue to run simulation and accessing output variables



The above code is used to create a queue for M/M/1 using the class Simulation, and then the simulation is run to compute the output variables. It should be noted that inf passed for the parameter K is for floating point positive infinity in Python. The output variables of the system can then be accessed as follows.



## System Stability

The value of T (total time to run the simulation) was checked after building the simulation to ensure that it gives a stable system. This was done by initially choosing the value of T to be 10,000 seconds, and then simulating three queues, one each for M/M/1, D/M/1, and M/G/1. The same three queues were then simulated again for double the amount of T (20,000 seconds). The output variable of E[N] (average number of packets in system) was compared for the two times with their respective queues, and it was seen that the values were always within 5% of each other.

Difference in values for M/M/1: -0.5%

Difference in values for D/M/1: 0.3%

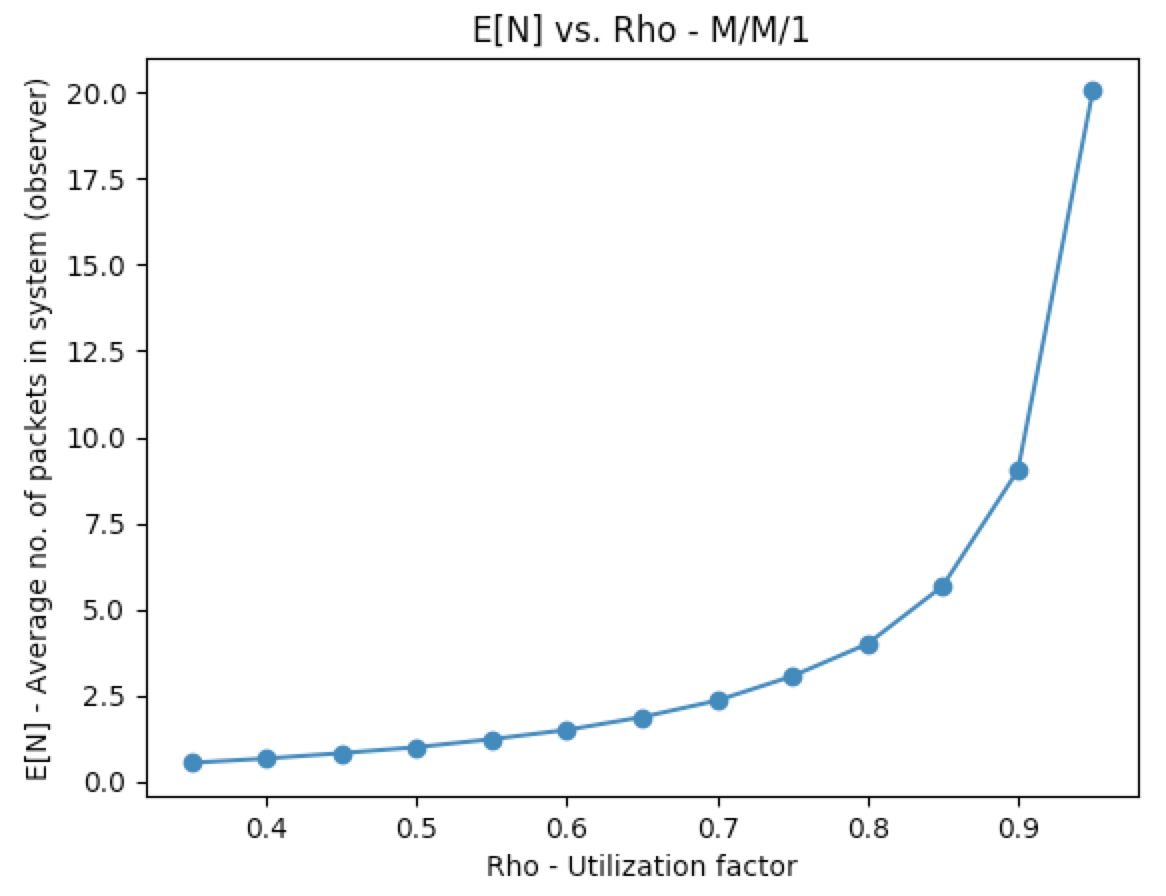
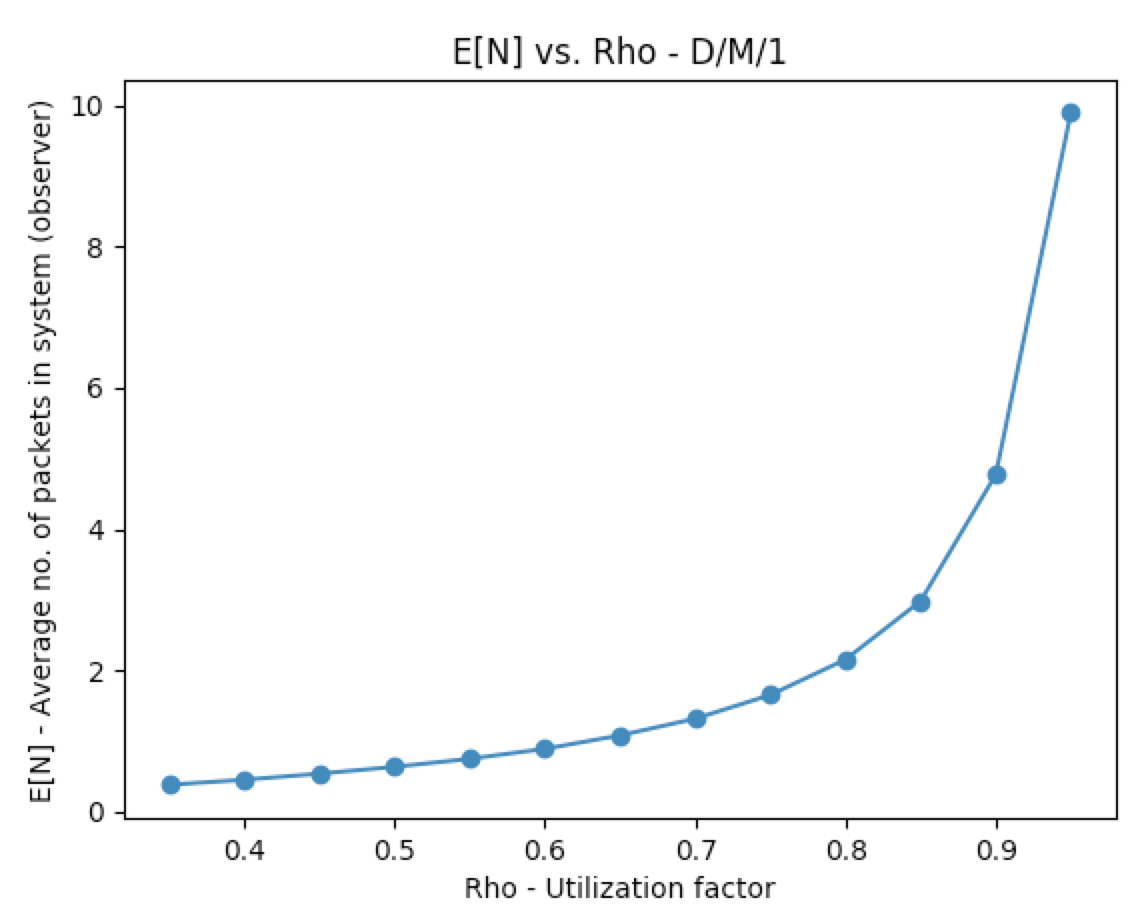
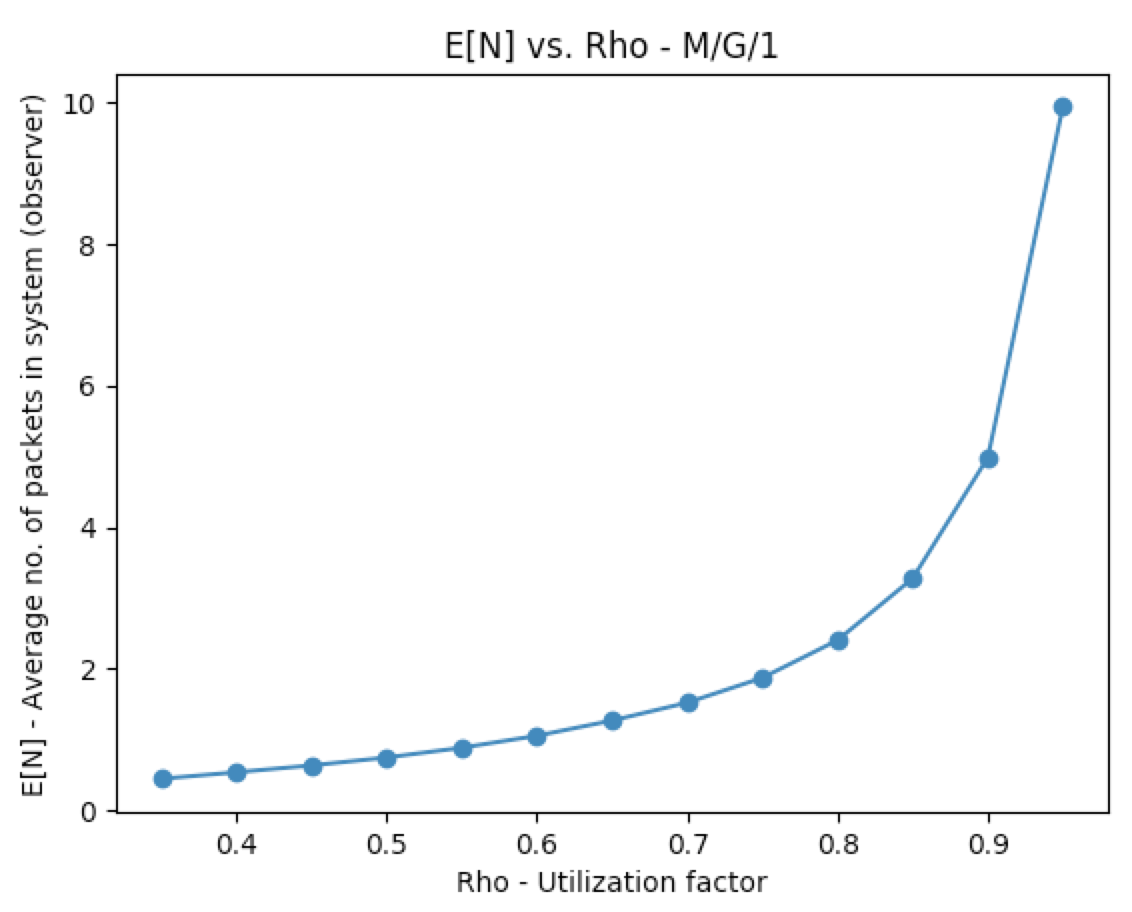
Difference in values for M/G/1: 0.0%

System is stable with T = 10000 (values within 5%)

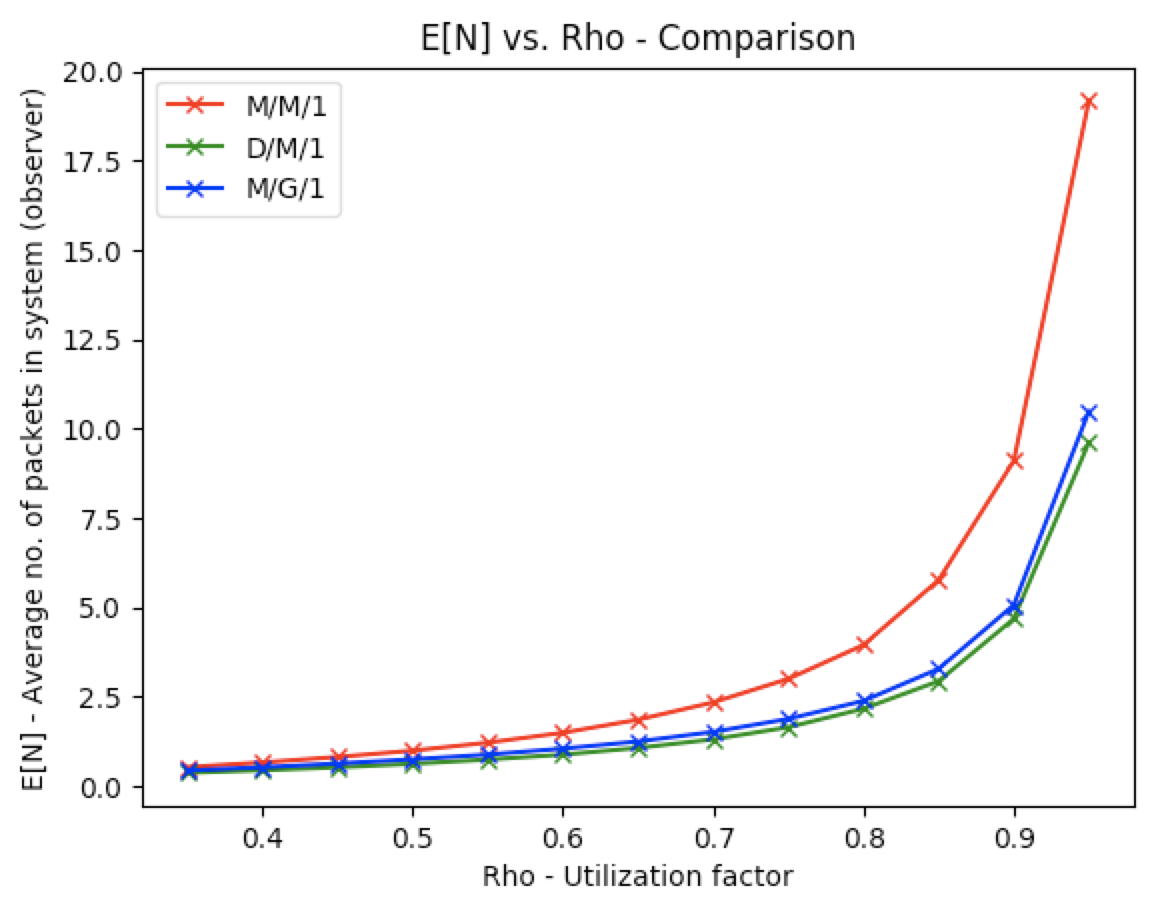
# Question 3 – M/M/1, D/M/1, and M/G/1 – E[N] & PIDLE (0.35 ≤ ρ ≤ 0.95)

For this question, the simulator was run inside a for loop such that the values of ρ went from 0.35 to 0.95 with a step size of 0.05. The values of E[N] (average number of packets in the system as seen by the observer), and PIDLE (proportion of time the system is idle) were returned from the simulator and saved in an array, which was then used to plot the following figures.

## E[N] as a function of ρ



**Fig 1:** E[N] vs. Rho (M/M/1) **Fig 2:** E[N] vs. Rho (D/M/1) **Fig 3:** E[N] vs. Rho (M/G/1)

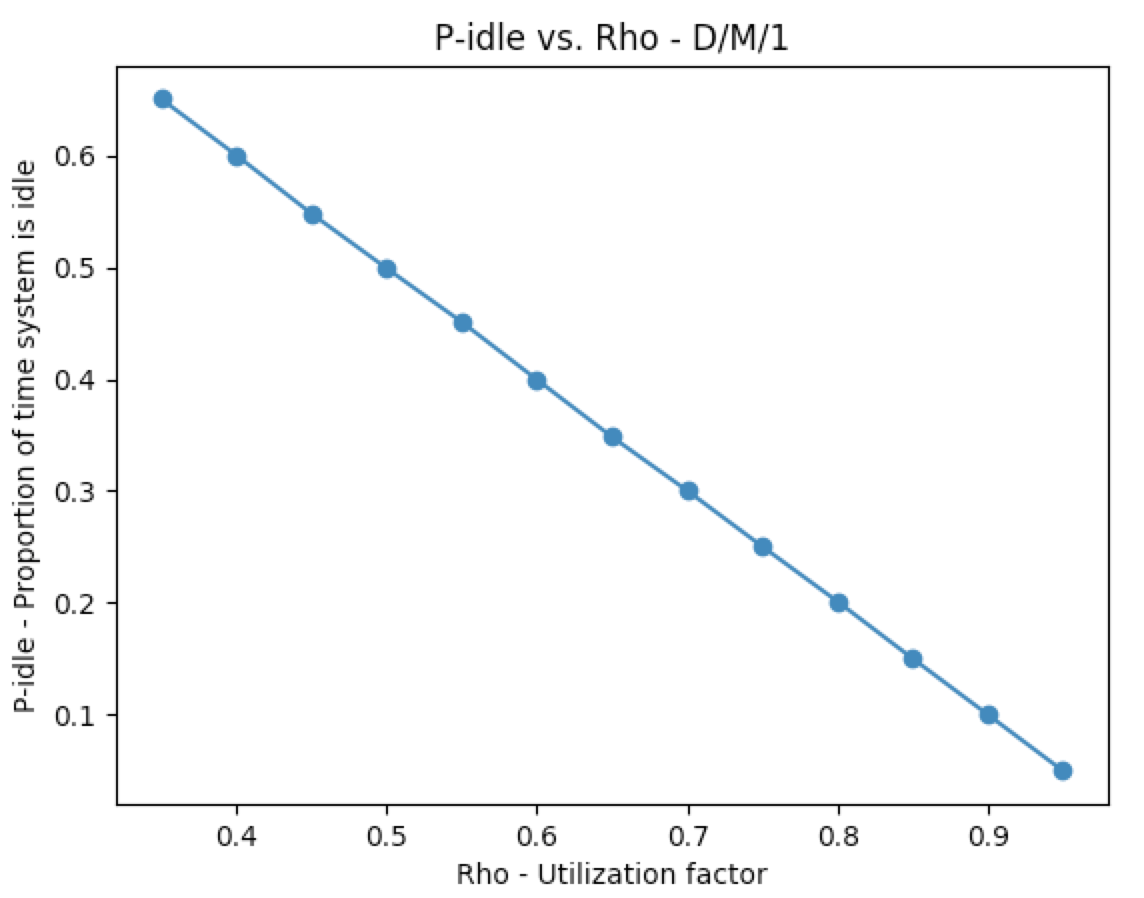
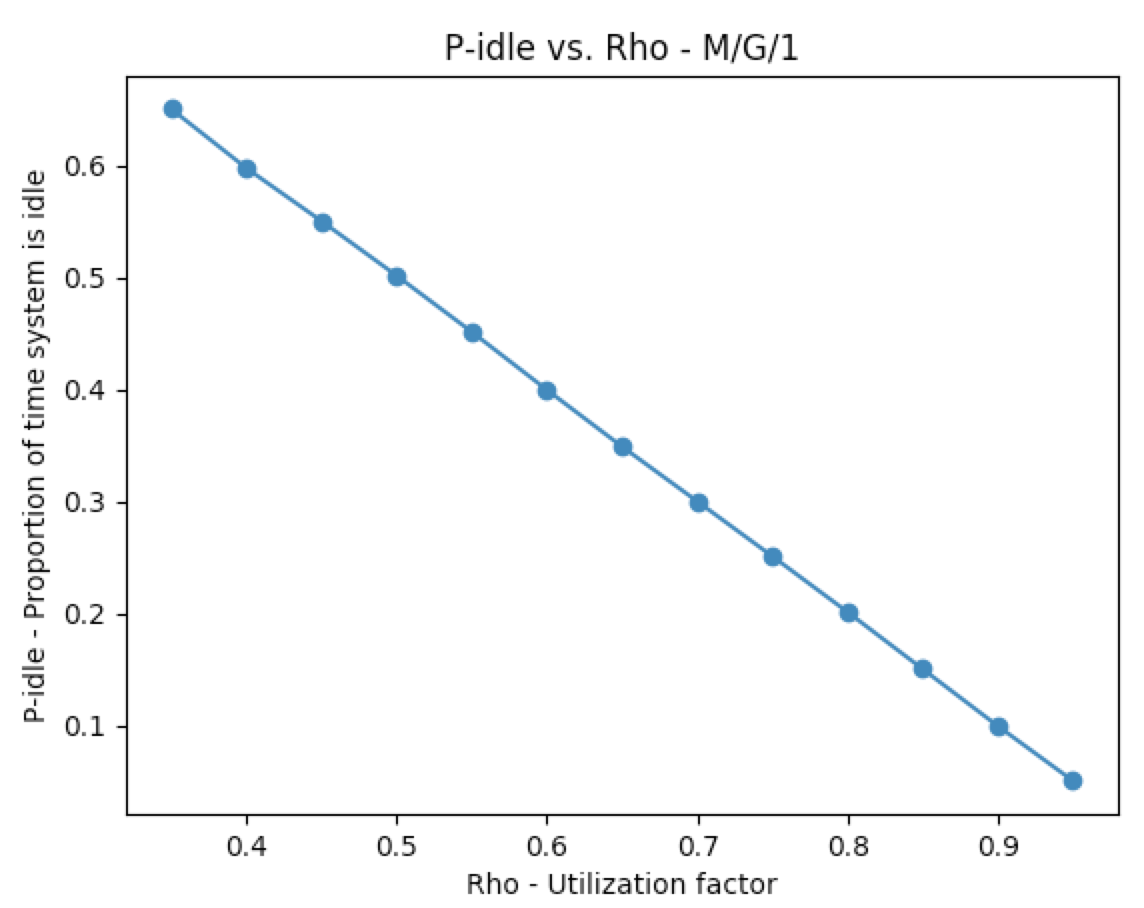


**Fig 4:** E[N] vs. Rho for M/M/1, D/M/1, and M/G/1

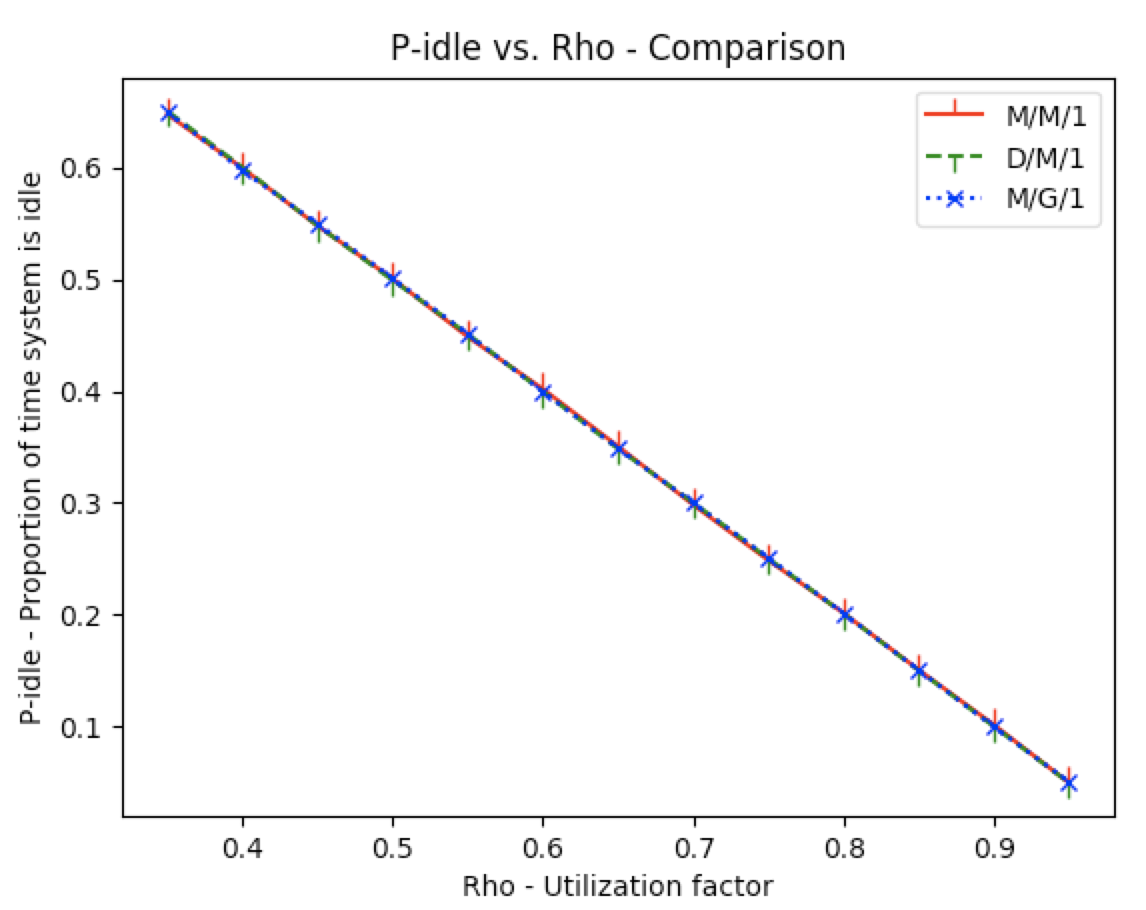
Figures 1, 2, and 3 show E[N] as a function of ρ for M/M/1, D/M/1, and M/G/1 respectively. All three of these queues have an exponential relationship between E[N] and ρ, as ρ increases, the average number of packets in the system increases exponentially.

Comparing the three queues in Figure 4 shows that M/M/1 has the highest average number of packets in the system, as compared to D/M/1 and M/G/1. This is because\_\_\_\_\_

## PIDLE as a function of ρ



**Fig 5:** PIDLE vs. Rho (M/M/1) **Fig 6:** PIDLE vs. Rho (D/M/1) **Fig 7:** PIDLE vs. Rho (M/G/1)



**Fig 8:** PIDLE vs. Rho for M/M/1, D/M/1, and M/G/1

Figures 5, 6, and 7 show PIDLE as a function of ρ for M/M/1, D/M/1, and M/G/1 respectively. Each of these figures show a linear relationship between PIDLE and ρ, as ρ increases, PIDLE decreases linearly. Comparing the three queues in Figure 8 show that the proportion of time system is idle is the exact same for all three of these queues, showing that PIDLE is independent of the type of queue simulated. This is because\_\_\_\_

# Question 4 – M/M/1, D/M/1, and M/G/1 – E[N] & PIDLE (ρ = 1.5)

For this question, the queue M/M/1 was simulated at ρ = 1.5, and E[N] and PIDLE were computes for this simulation.

M/M/1 - E[N] - Average no. of packets in system (rho = 1.5): 249404.413463

M/M/1 - Pidle - Proportion of time system is idle (rho = 1.5): 0.00000

These numbers follow the pattern seen in figure 1 and 5. E[N] grew exponentially from 18.1 at ρ = 0.95 to 29404.4 at ρ = 1.5. PIDLE at ρ = 1.5 is 0, which means the system is never idle, which is expected as ρ = 1.5 means that the utilization factor of the queue is surpassed, so PIDLE will always be 0.

Comparison of M/M/1 at rho=1.5 with other queues at rho=1.5:

D/M/1 - E[N] - Average no. of packets in system (rho = 1.5): 248851.898021

D/M/1 - Pidle - Proportion of time system is idle (rho = 1.5): 0.00000

M/G/1 - E[N] - Average no. of packets in system (rho = 1.5): 249288.776063

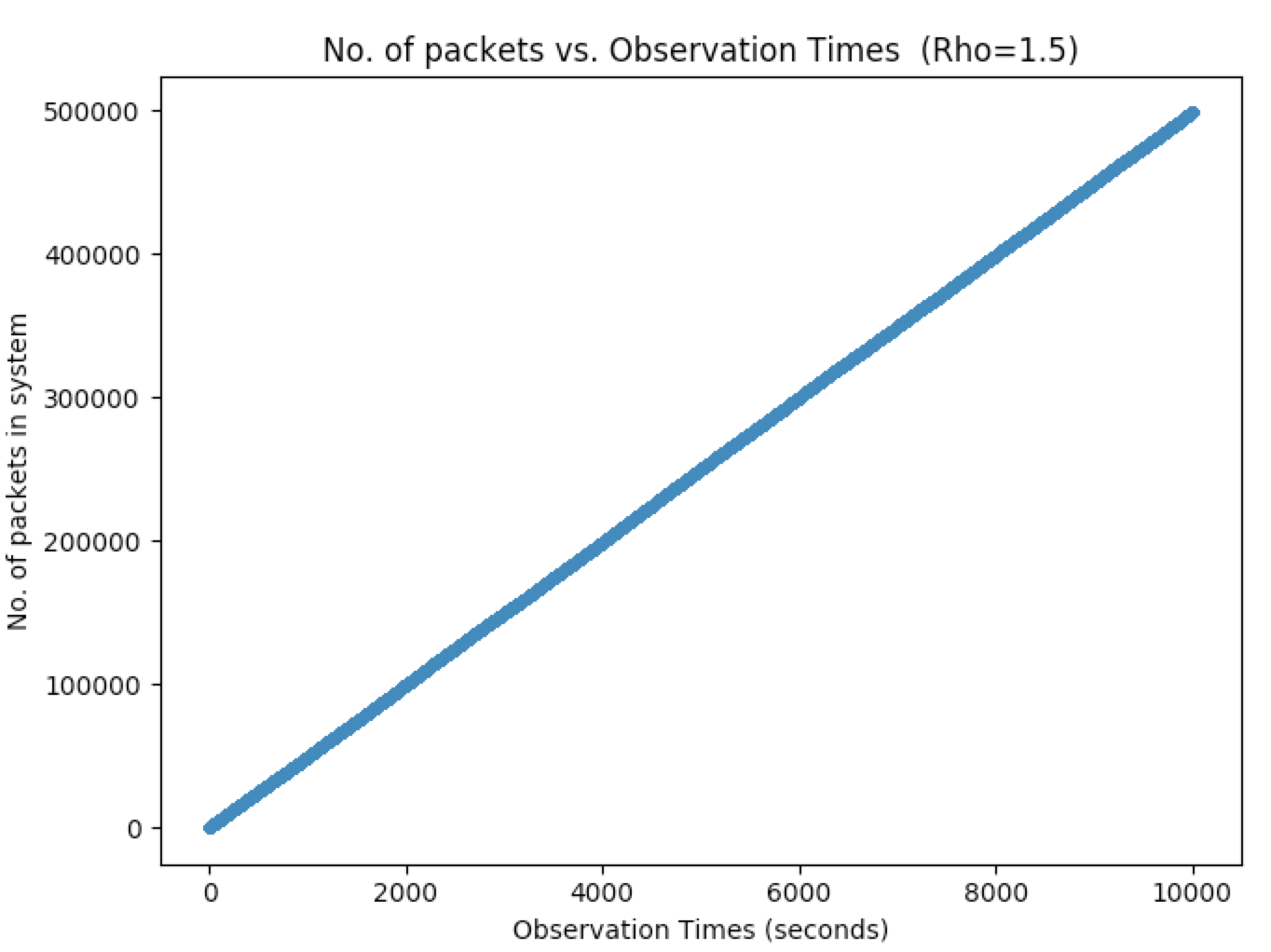
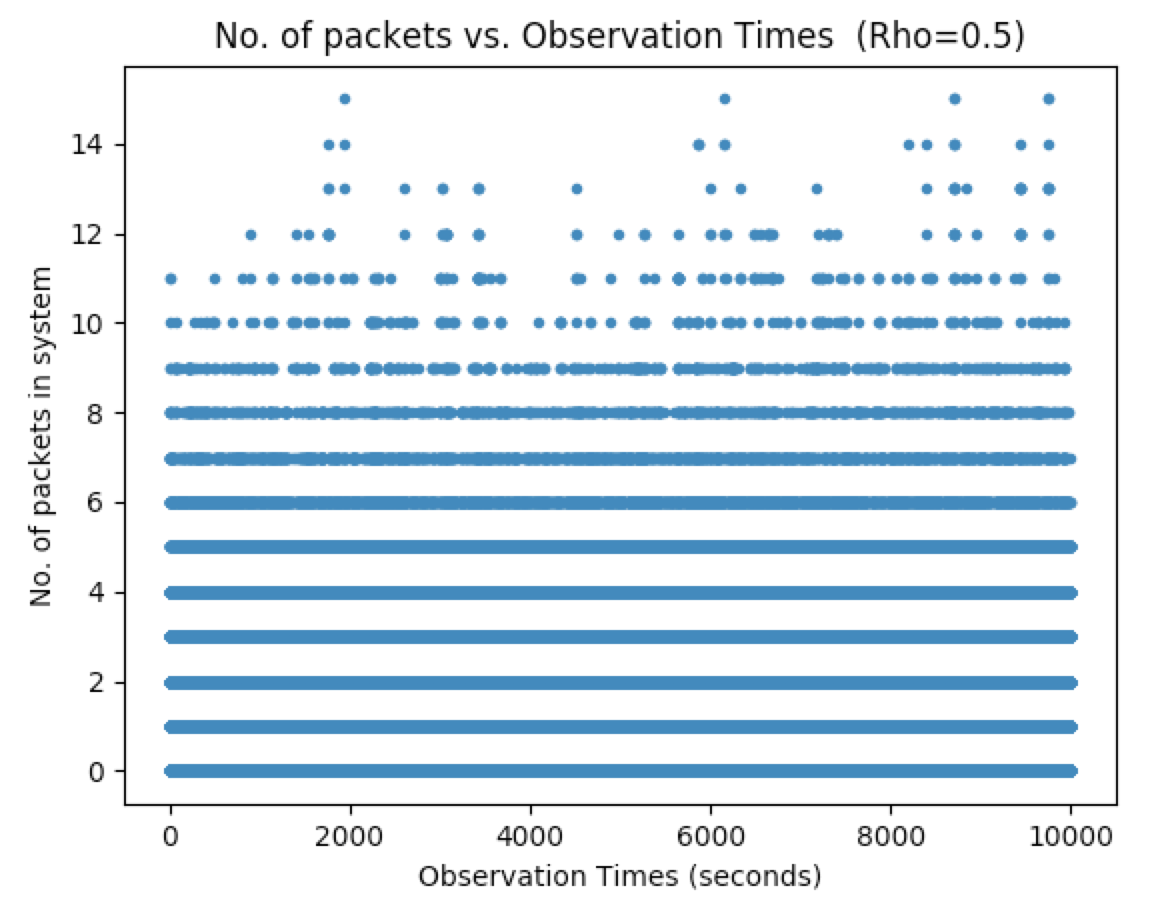
M/G/1 - Pidle - Proportion of time system is idle (rho = 1.5): 0.00000

In order to compare E[N] and PIDLE of M/M/1 with other queues, D/M/1, and M/G/1 were also simulated at ρ = 1.5. It was found that the average number of packets in the system at ρ = 1.5 were very similar for all three queues, and were in the same order. All three of these queues had the expected PIDLE of 0.

Comparison of M/M/1 at rho=1.5 with M/M/1 at rho=0.5

M/M/1 - E[N] - Average no. of packets in system (rho = 0.5): 1.001811

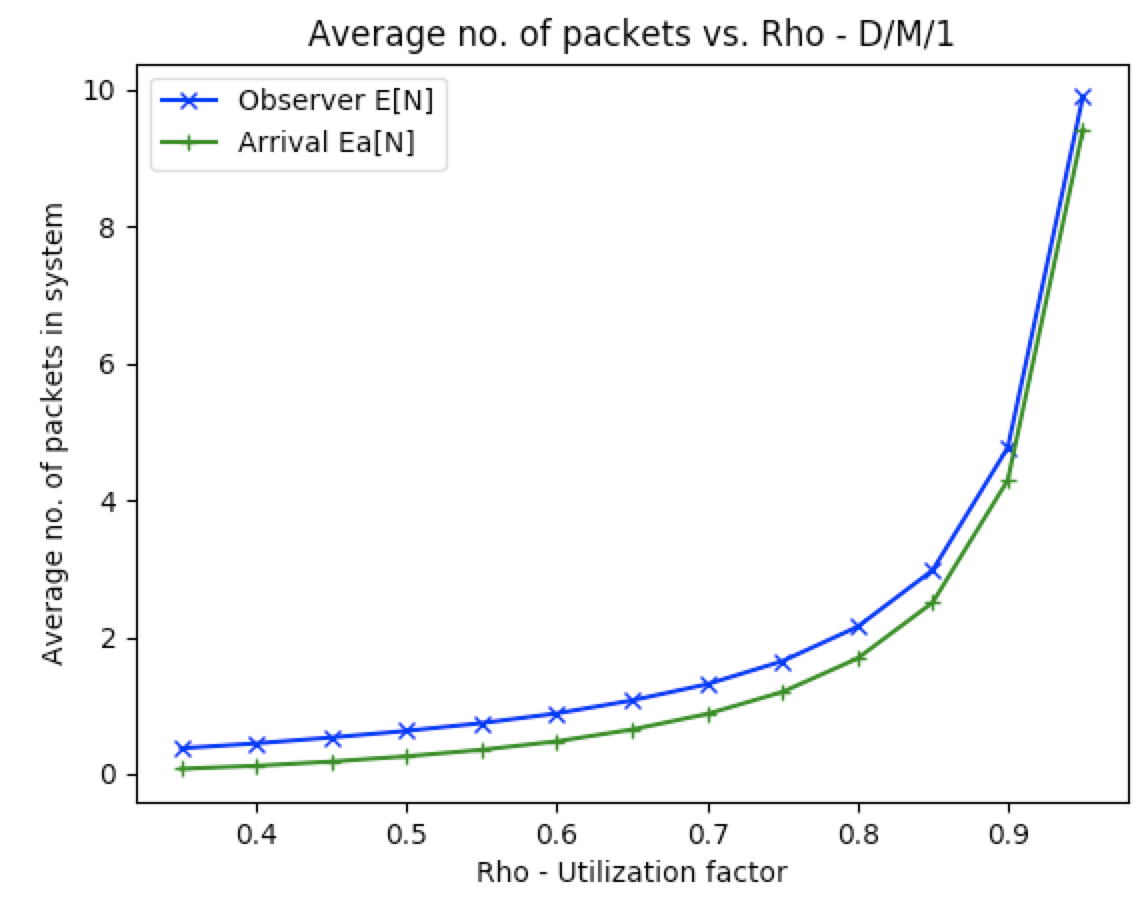
M/M/1 - Pidle - Proportion of time system is idle (rho = 0.5): 0.499541



**Fig 9:** No. of packets vs. Observation times (Rho=1.5) **Fig 10:** No. of packets vs. Observation times (Rho=0.5)

The queue M/M/1 was also simulated at ρ = 0.5, and number of packets in the system as seen by observation events were plotted against observation times to see any trend. Figure 9 show that at ρ = 1.5, number of packets in the system increase linearly with time, whereas at ρ = 0.5, the number of packets in the system fluctuated from 0 to 14, with average being at 1.001. This is because\_\_\_\_

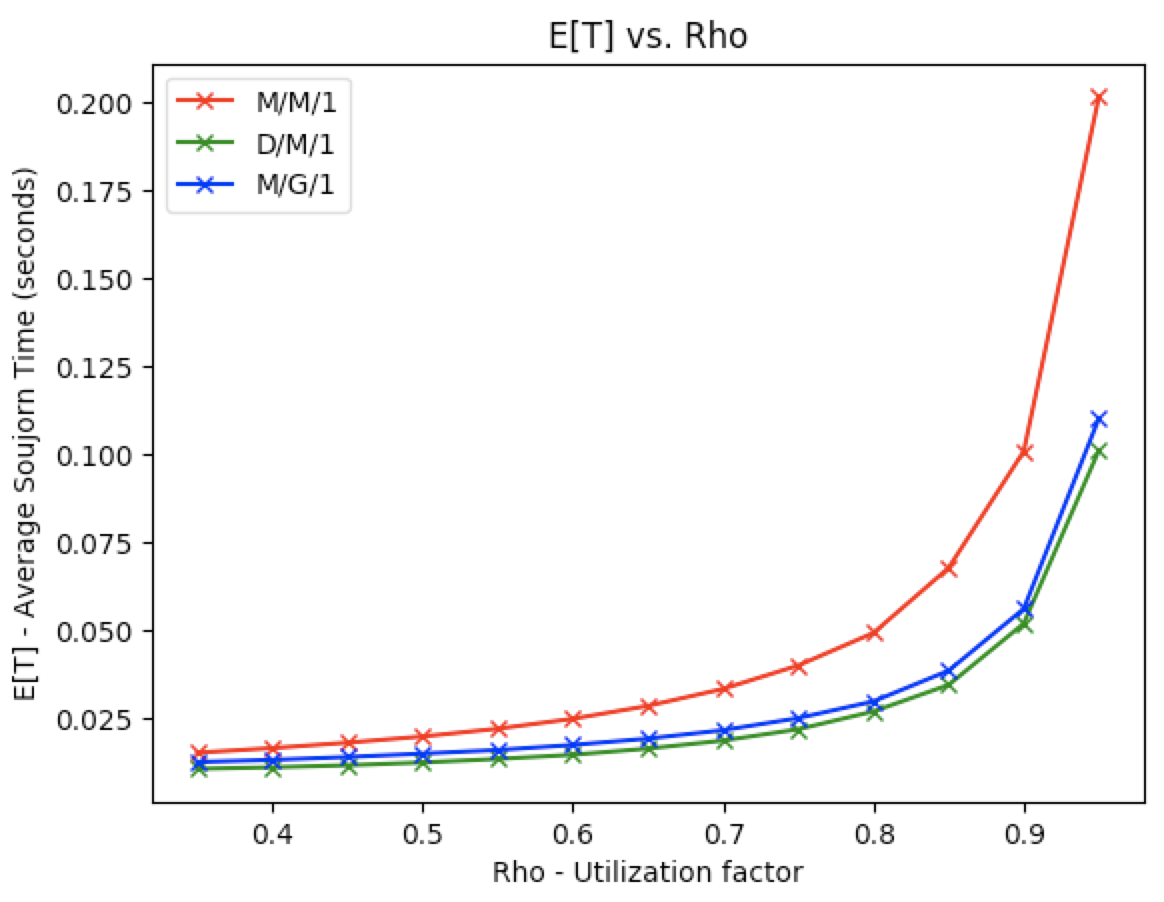
# Question 5 – M/M/1 and D/M/1 – Comparison of E[N] and Ea[N]



**Fig 11:** Comparison of E[N] and Ea[N] (M/M/1) **Fig 12:** Comparison of E[N] and Ea[N] (D/M/1)

Figure 11 and 12 show the comparison of E[N] and Ea[N] for M/M/1 and D/M/1. As seen in figure 11, the average number of packets the system seen by the observation event and the arrival event are the exact same. This is because the arrival process in M/M/1 is Poisson, hence\_\_\_\_\_\_. Figure 12 show that the average number of packets seen by the observation event is a little bit higher than the average number of packets seen by the arrival event. This is because for D/M/1, the PASTA property does not hold true, and\_\_\_\_

# Question 6 – M/M/1, D/M/1, and M/G/1 – Comparison of E[T]



**Fig 13:** E[T] vs. Rho for M/M/1, D/M/1, and M/G/1

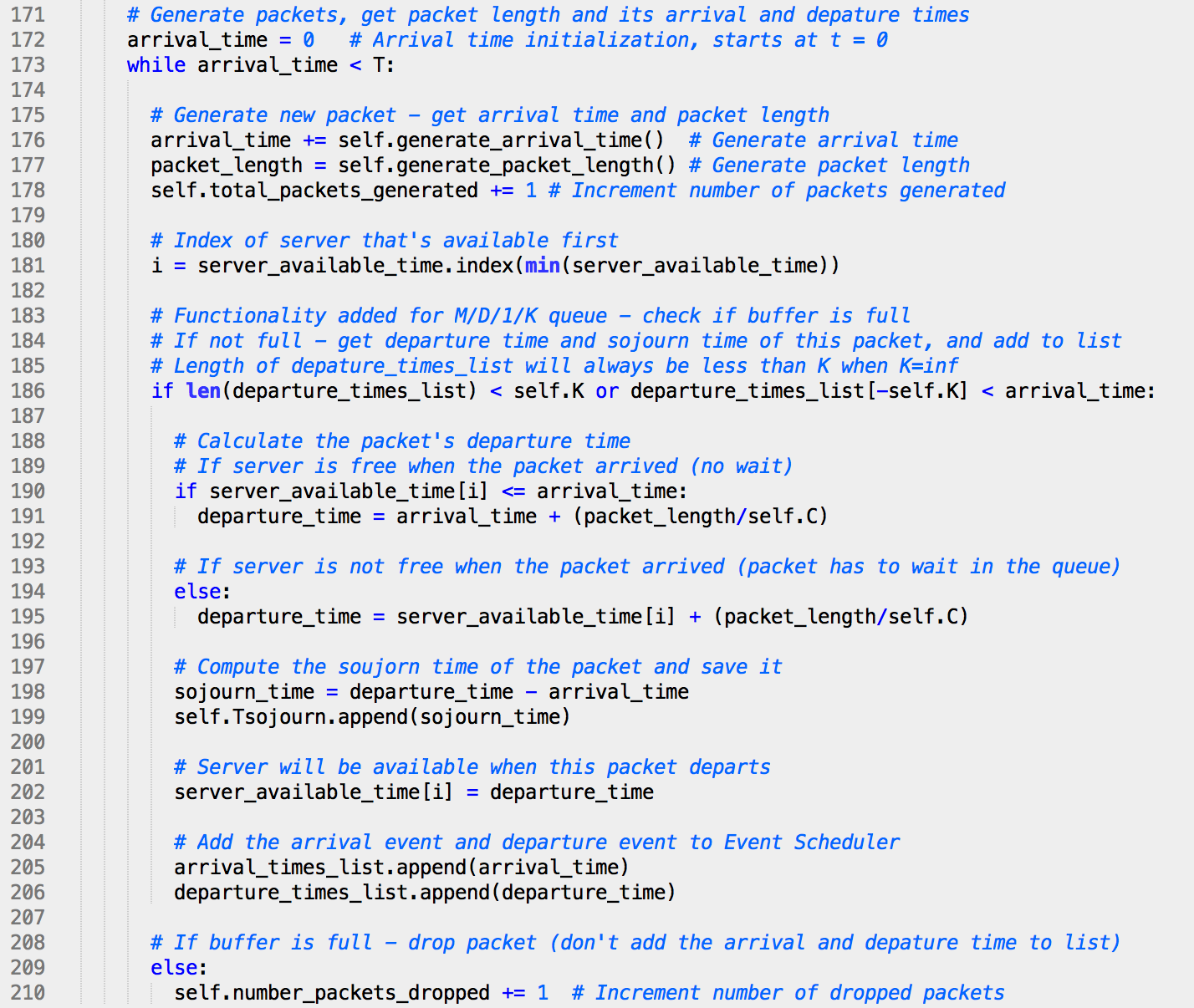
# Question 7 – Simulator for M/D/1/K Queue

To add the functionality of M/D/1/K queue in the class Simulation, the create\_event\_scheduler() function was modified such that it was checked every time a packet was generated whether the buffer was full or not.

*Variables used:*



*Code Changed in create\_event\_scheduler() function:*



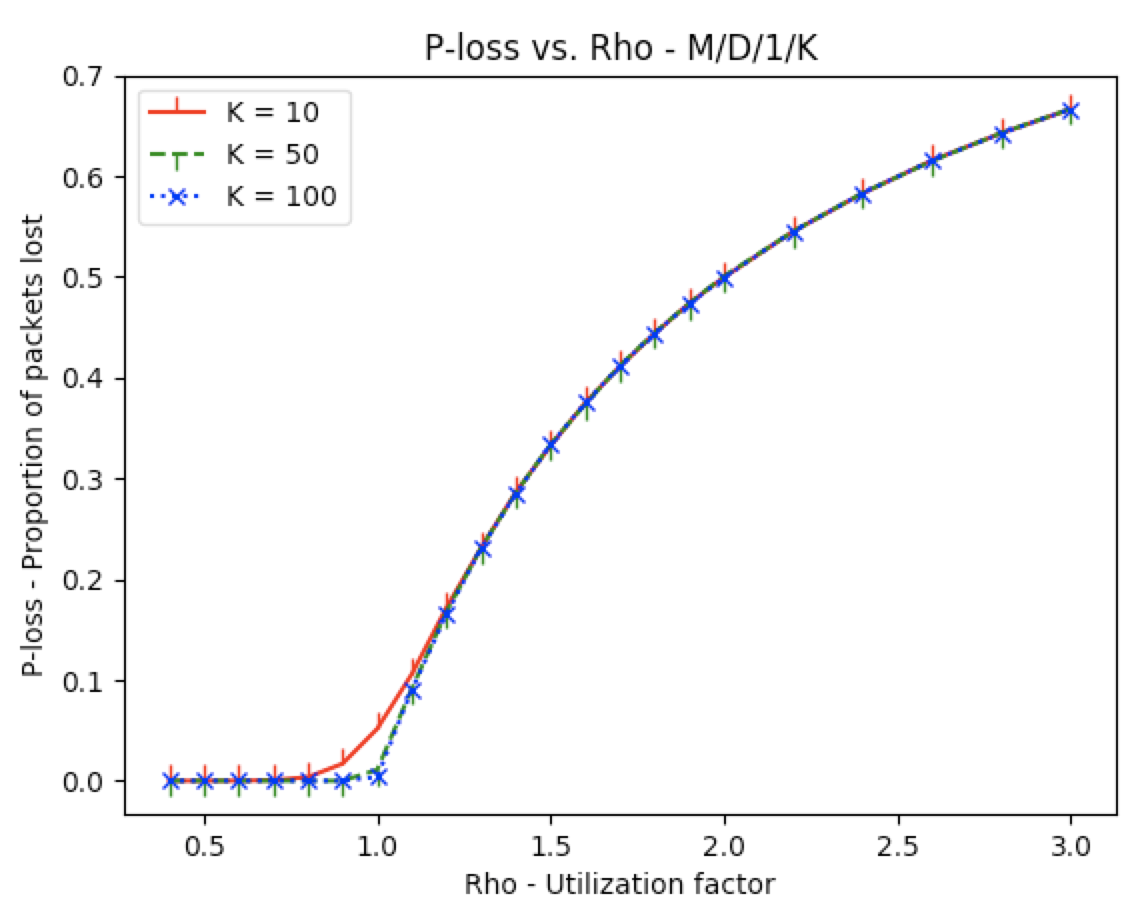
The chunk of code in lines 183 to 201 in Question 1 was put in an if-else condition in the code above in lines 186 to 210. After a packet was generated with its arrival time and packet length, it was checked whether the buffer was full or not. If the buffer was not full, the rest of code was executed where the packet’s departure time and sojourn time were computed, and the events were added to the arrival\_times\_list and departure\_times\_list. However, if the buffer was found to be full, then the packet was dropped, that is, ignored and not added to the arrival\_times\_list and departure\_times\_list, and the number\_packets\_dropped variable was incremented by 1.

The condition to check if the buffer has space is done by checking if the number of departure events (same as the length of the departure\_times\_list) is less than the buffer size. For K = inf (for M/M/1, D/M/1, M/G/1 etc), this condition will always hold true as the number of departure events in the system will always be less than infinity. For the M/D/1/K case, it will still hold true for the first number of K-1 events. After that, it will look for the other if condition, where it looks for the last Kth packet, and checks for its departure time. If that packet would have left before this new packet’s arrival time, it would mean the buffer has space to serve this new packet. If this condition fails, it means the buffer is full.

The rest of the functionality in create\_event\_scheduler of creating observation events, and adding all observation, arrival, and departure events to the Event\_Scheduler remains the same as before.

The variables utilized here are number\_packets\_dropped and total\_packets\_generated, as PLOSS for M/D/1/K is computed as the number of packets dropped by the system divided by the total number of packets generated.

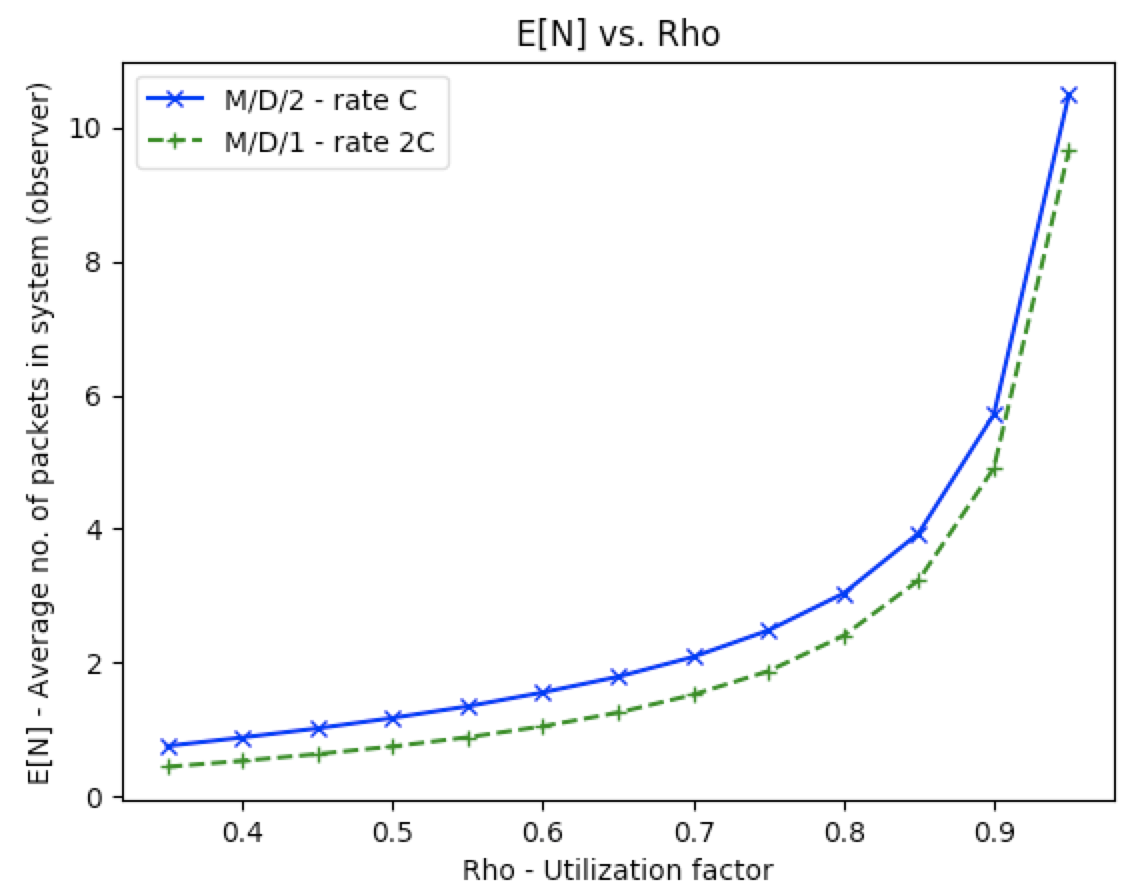
# Question 8 – M/D/1/K– PLOSS (0.4 ≤ ρ ≤ 3)



**Fig 14:** PLOSS vs. Rho for M/D/1/K when K=10, 50, and 100

Figure 14 shows the relationship between PLOSS and ρ for M/D/1/K. As seen, PLOSS is 0 until ρ=0.8 for all three of these queues. From ρ=0.8 to ρ=1.2, the three queues have a slightly different PLOSS. For the queue with K=10, PLOSS increases more rapidly than the queue with K=50 and 100. However, after ρ=1.2, all three of these queues had the same PLOSS, and had a logarithmic relationship between PLOSS and ρ. This is because\_\_\_\_

# Question 9 – M/D/2 and M/D/1 (rate 2C) – E[N] (0.35 ≤ ρ ≤ 0.95)



**Fig 15:** E[N] vs. Rho for M/D/2 and M/D/1

Figure 15 shows a comparison of the average number of packets in the system in M/D/2 with transmission rate of C, and M/D/1 with transmission rate of 2C. As shown, M/D/1 is a better system as it has less average number of packets in the system at each ρ. It should be noted that despite M/D/1 giving a better result, realistically, M/D/2 may be a better choice as it has system reliability. If one server fails by any chance, another can continue transmitting packets in the system.