University of Waterloo
Department of Electrical and Computer Engineering

Lab 1: M/M/1 and M/M/1/K Queue Simulation

ECE 358: Computer Networks
October 8, 2021

Adrian Wong (20720975)
Enoch Tang (20720705)

Question 1

Question 1: Write a short piece of code to generate 1000 exponential random variable with λ=75. What is the mean and variance of the 1000 random variables you generated? Do they agree with the expected value and the variance of an exponential random variable with λ=75? (if not, check your code, since this would really impact the remainder of your experiment).

Function name in lab1.py: *exponential_random_variable_question1()*

After modeling the random variable x with exponential distribution in Python3, we executed the program for 1000 different x's with λ=75 which gave the following output.

```
Expected value for Exponential Random Variable: 0.013333333333333334
Expected value for generated array: 0.013925809128649502
Variance for Exponential Random Variable: 0.00017777777777777779
Variance for generated array: 0.00017623798420647455
```

Figure 1: Expected Value and Variance output for 1000 exponential random variables with λ=75.

From Figure 1, we can see that the expected value and variance is very similar. The difference in percentage between the initial expected value (1/λ) and the expected value of the array is 4.44%. Additionally, the difference in percentage between the initial variance ($1/\lambda^2$) and the variance of the array is 0.87%. From this observation, we say that the two value pairs agree with each other.

Question 2

Question 2: Build your simulator for this queue and explain in words what you have done. Show your code in the report. In particular, define your variables. Should there be a need, draw diagrams to show your program structure. Explain how you compute the performance metrics.

Class and function name in lab1.py: `InfiniteEventQueue.create_queue()`

For the M/M/1 queue, we started by creating a class to encapsulate all the members and attributes that the queue might contain. As shown in Figure 2, the attribute variables include the following: the average length of a packet in bits, transmission rate, simulation time, arrival/departure/observer counters, and the performance metrics. When the object is created, the variables `rho`, `L`, `C`, and `simulation_time` are passed in depending on the specifications of the buffer. The remaining variables are updated during runtime when the queue is generated.

```
class InfiniteEventQueue:
    def __init__(self, rho, L, C, simulation_time) -> None:
        self.rho = rho
        self.L = L
        self.C = C
        self.simulation_time = simulation_time
        self.Na = 0
        self.Nd = 0
        self.No = 0

        self.queue = deque()
        self.num_packets_in_buffer = []
        self.idle_time = 0
        self.Pidle = 0
        self.Ploss = 0
        self.average_num_of_packets = 0
```

Figure 2: Attributes of InfiniteEvent Queue class

The `create_queue()` function is responsible for the creation of Arrival, Departure, and Observer events with their respective timestamps. Firstly, the arrival, departure, and observer arrays are created which will hold an tuple event (eg. (“arrival”, 0.01562)). Additionally, the `current_time` of the simulation is set to 0. The first while loop is responsible for creating the arrival and departure events. While the `current_time` is less than the `simulation_time`, we calculate the `arrival_rate` with a helper function shown in Figure 4. Using this, we can use the `exponential_random_variable` function and `current_time` to generate an arrival time. Secondly, we can calculate the packet’s `service_time` by using a λ=1/L which will give us an exponential random variable with an expected value of L. Thirdly, if there are no departures or if the most recent departure is earlier than the next arrival time, then the `departure_time` of the current arrival is the sum of the `current_time` and `service_time`. However, if this is not the case, then the current departure time is the sum of the most recent `departure_time` and `service_time`. Lastly, this is added to the array and the `current_time` is set to the `arrival_time`. After generating these arrival and departure events, we generate the observer events. This is simply done by using a λ value which is 5 times the `arrival_rate`. Finally, the arrays are appended to each other, sorted according to their timestamp, and added to the infinite queue.

```python
def create_queue(self):
    arrivals = []
    departures = []
    observers = []

    current_time = 0
    while current_time < self.simulation_time:
        arrival_rate = compute_arrival_rate(self.rho, self.L, self.C)
        arrival_time = exponential_random_variable(arrival_rate) + current_time
        arrivals.append(('arrival', arrival_time))

        packet_length = exponential_random_variable(1/self.L)
        service_time = compute_service_time(packet_length, self.C)

        if len(departures) == 0 or departures[-1][1] < arrival_time:
            departure_time = arrival_time + service_time
        else:
            departure_time = departures[-1][1] + service_time

        departures.append(('departure', departure_time))

        current_time = arrival_time

    current_time = 0
    while current_time < self.simulation_time:
        observer_time = exponential_random_variable(5*arrival_rate) + current_time
        observers.append(('observer', observer_time))

        current_time = observer_time

    self.Na = len(arrivals)
    self.Nd = len(departures)
    self.No = len(observers)
    all_events = arrivals + departures + observers
    all_events.sort(key=lambda x: x[1])
    self.queue = deque(all_events)
```

Figure 3: The create_queue function

```python
def exponential_random_variable(lambda_value):
    random.seed(datetime.datetime.now())
    U = random.random()
    x = -(1/lambda_value)*math.log(1-U)

    return x


def compute_arrival_rate(rho, L, C):
    return rho * compute_service_rate(L, C)


def compute_service_rate(L, C):
    return C/L


def compute_service_time(L, C):
    return L/C
```

Figure 4: Helper Functions

The performance metrics, E[N], Pidle, and Ploss are computed in the `run_simulation()` function and will be discussed in Question 3 and Question 6.

Question 3

Question 3: The packet length will follow an exponential distribution with an average of L = 2000 bits. Assume that C = 1Mbps. Use your simulator to obtain the following graphs. *Provide comments on all your figures*.

1. E[N], the average number of packets in the queue as a function of ρ (for $0.25 < \rho < 0.95$, step size 0.1). Explain how you do that.
2. P_{IDLE}, the proportion of time the system is idle as a function of ρ, (for $0.25 < \rho < 0.95$, step size 0.1). Explain how you do that.

To calculate both E[N] and Pidle, we need to first run the simulation. Before discussing how E[N] and Pidle were obtained, we will first go over how we chose an appropriate simulation time by computing error differences between results obtained from two simulation times.

T and 2T 5% Error Calculation

When running our simulation, we chose T = 1000 and T = 2000 with rho = [0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85, 0.95] in order to verify our network queue. The following results were obtained when running the simulation for the two time durations:

T = 1000

```
E[N] = [0.3326103079864197, 0.5378453300127373, 0.8219747887602625,
1.2068248498486487, 1.8971789788243707, 3.062956668047436,
5.712468612318742, 20.010510744316413]

Pidle = [0.6247775678860241, 0.5414000080528097, 0.45747387401612716,
0.37650456304776114, 0.2898310741704061, 0.20543322394748825,
0.1232706721464893, 0.04272618890653632]
```

T = 2000

```
E[N] = [0.3339067964987273, 0.5380708775420803, 0.8174753872480388,
1.215125221921476, 1.839884740951903, 2.991235163396533,
5.788277884364904, 18.522936378342965]

Pidle = [0.6246695704230016, 0.541751048449427, 0.45800854557259063,
0.37481889210254427, 0.2922284453745265, 0.20857260597359173,
0.12371730587568902, 0.04159037144072416]
```

E[N] Error Formula

The following formula was used to compute the error for E[N]:

$$E[N]\ error\ = \frac{\sum_{i=0}^{i=N-1} \frac{abs(E[i]_{1000} - E[i]_{2000})}{E[i]_{1000}}}{N} * 100$$

Equation 1: E[N] Error formula

Where N is the size of $E[N]_{1000}$ or $E[N]_{2000}$ arrays.

Pidle Error Formula

The following formula was used to compute the error for Pidle:

$$Pidle\ error\ = \frac{\sum_{i=0}^{i=N-1} \frac{abs(Pidle[i]_{1000} - Pidle[i]_{2000})}{Pidle[i]_{1000}}}{N} * 100$$

Equation 2: Pidle Error formula

Where N is the size of $Pidle[N]_{1000}$ or $Pidle[N]_{2000}$ arrays.

Error Results

From Equation 1, the error for E[N] between the two simulation times, T = 1000 and T = 2000, was calculated to be **1.97%**, which is within the 5% threshold required for this lab.

From Equation 2, the error for Pidle between the two simulation times, T = 1000 and T = 2000, was calculated to be **0.75%**, which is within the 5% threshold required for this lab.

As a result, simulation data recorded for the simulation T = 1000 will be used during analysis for the remainder of question 3.

E[N] Calculation

To calculate E[N], we leverage the Observer events to take snapshots of the number of packets in the network buffer. The number of packets in the buffer is calculated by taking the difference in the number of Departures (Nd) in the event queue with the number of Arrivals (Na) in the event queue.

When an Observer event is pulled from the event queue, the number of packets in the buffer is appended to the array `num_of_packets_in_buffer`. Once the simulation has finished running, the average number of events in the buffer is computed by taking the sum of the values in `num_of_packets_in_buffer` and dividing by the length of `num_of_packets_in_buffer`. This average value is appended to E[N], where each entry in E[N] represents the average number of packets in the buffer corresponding to a given rho value. Figure 5 and 6 show the code containing the logic described above.

```
def run_simulation(self):
    last_departure_time = 0
    last_oberserver_time = 0
    idle_in_progress = False
    while self.queue:
        event, current_time = self.queue.popleft()
        if event != "observer" and idle_in_progress:
            self.idle_time += (last_oberserver_time-last_departure_time)
            idle_in_progress = False
        if event == "arrival":
            self.Na -= 1
        elif event == "departure":
            self.Nd -= 1
            last_departure_time = current_time
        else:
            self.No -= 1
            num_of_packets_in_buffer = self.Nd - self.Na
            self.num_packets_in_buffer.append(num_of_packets_in_buffer)
            if num_of_packets_in_buffer == 0:
                last_oberserver_time = current_time
                idle_in_progress = True

    self.average_num_of_packets = sum(self.num_packets_in_buffer) / len(self.num_packets_in_buffer)
    self.Pidle = self.idle_time / self.simulation_time
```

Figure 5: The run_simulation Method for Processing M/M/1 queue.

```
# Question 3
print("Simulating Question 3...")
question3_En = []
question3_Pidles = []
question3_rhos = [.25, .35, .45, .55, .65, .75, .85, .95]
for rho in question3_rhos:
    Queue = InfiniteEventQueue(rho, 2000, 1000000, 1000)
    Queue.create_queue()
    Queue.run_simulation()
    question3_En.append(Queue.average_num_of_packets)
    question3_Pidles.append(Queue.Pidle)
    print("Infinite Queue - Finished simulating rho={}".format(rho))

question3_plot_graphs(question3_rhos, question3_En, question3_Pidles)
```

Figure 6: Executing the simulation

Pidle Calculation

When the buffer receives an incoming observer event and there are no packets in the buffer (Nd-Na=0), we save the current_time into the `last_observer_time` and set the `idle_in_progress` flag to true. This means that the buffer is currently in an idling state. Notice that we do not immediately increment the idle_time attribute since there could be consecutive observer events incoming. If this was the case, it would add overlapping idle times which we do not want as shown in the diagram below (Figure 7). Here we want to capture the total idle time which is L2 rather than summing each time we see an observer (L1 + L2). Lastly, this flag prevents any edge cases when the first event in the queue is an observer which can cause negative idle times.

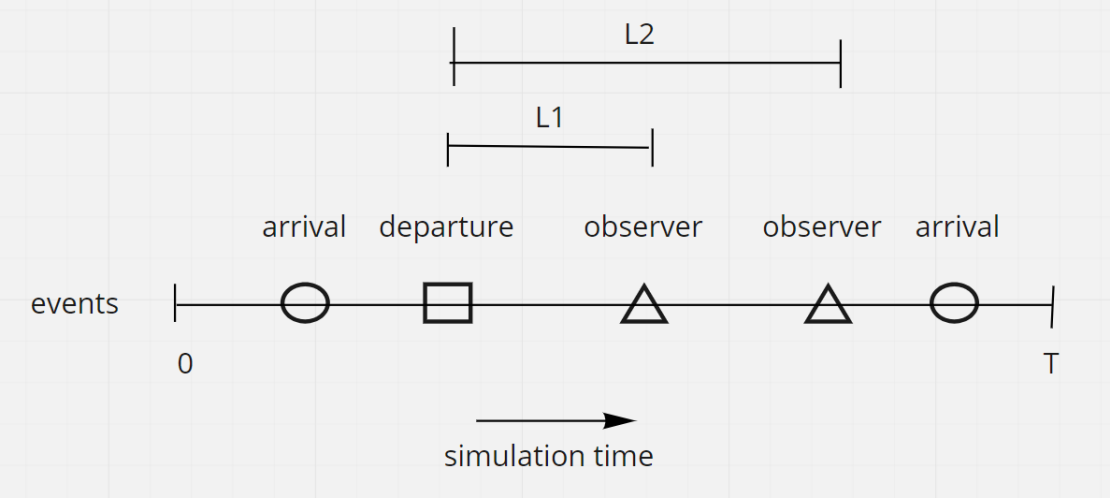

Figure 7: Diagram of example event queue and idle time calculate

Results

Recall that for a simulation time of T = 1000 and rho input [0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85. 0.95], the following results were obtained:

```
E[N] = [0.3326103079864197, 0.5378453300127373, 0.8219747887602625,
1.2068248498486487, 1.8971789788243707, 3.062956668047436,
5.712468612318742, 20.010510744316413]

Pidle = [0.6247775678860241, 0.5414000080528097, 0.45747387401612716,
0.37650456304776114, 0.2898310741704061, 0.20543322394748825,
0.1232706721464893, 0.04272618890653632]
```

Plotting E[N] and Pidle against rho, we obtain the following two graphs in Figure 8 and Figure 9 respectively.

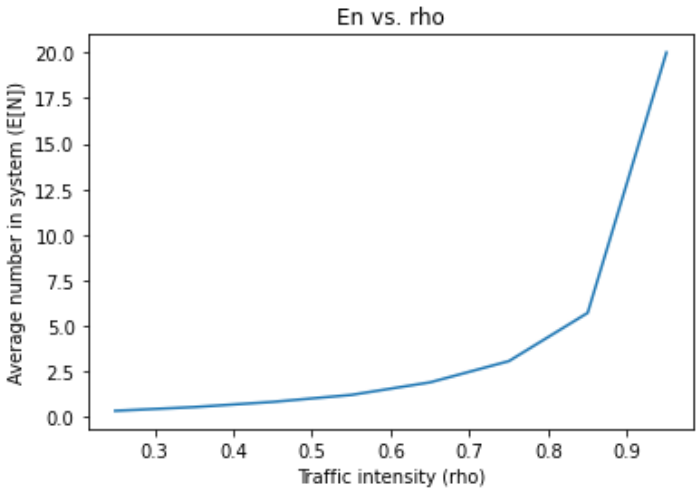

Figure 8: E[N] vs rho graph for M/M/1 queue

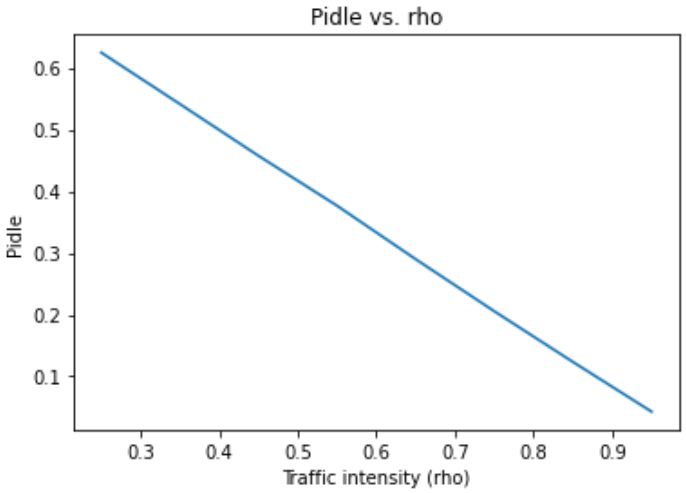

Figure 9: PIdle vs rho graph for M/M/1 queue

From Figure 8, we can see that the average number of packets in the buffer grows exponentially as the traffic intensity (rho) increases. Because the buffer is unbounded, the graph will continue to grow exponentially until infinity, as values of rho greater than 1 mean that the arrival rate of the packets is faster than the service rate.

Conversely, from Figure 9, we can see that the proportion of idle time in the buffer decreases at a linear rate as traffic intensity (rho) increases. The idle time decreases as traffic intensity increases since this means there are more events being processed in the infinite buffer. As a result, these events become tightly packed together and there is less time between events for idling. The graph will decrease at a linear rate for values of rho less than 1 for this region as the amount of packets departing is linearly proportional to the amount of packets arriving. Because idle time is calculated based on the empty window between the last departure time and the next

arrival time, this explains why Pidle is a linear function up to rho values of 1. Values of rho greater than 1 will result in Pidle approaching a horizontal asymptote of 0. This is because in this region, the departures are not linearly proportional with arrivals because packets are arriving at a rate faster than the network can service them. This behaviour is evident in question 4 where a rho value of 1.2 results in a near 0 Pidle time in the network buffer.

Question 4

Question 4: For the same parameters, simulate for $\rho=1.2$. What do you observe? Explain.

Using a simulation time of T = 1000 with rho = 1.2, the following values for E[N] and Pidle are obtained:

```
E[N] = 50078.43128034603 for rho=1.2
Pidle = 5.884713981479394e-06 for rho=1.2
```

For a rho value of 1.2, we observe that the number of packets in the infinite buffer is slightly over 50 thousand, while the proportion of idle time is nearly zero. This makes intuitive sense as a rho value greater than 1 means that the arrival rate of the packets is greater than the service rate of the packets. Thus, with a rho value of 1.2, the number of packets in the buffer will grow exponentially with time, which is evident with the large E[N] value obtained. Similarly, because the buffer is receiving packets at a rate greater than they can be serviced, it is extremely unlikely for there to be any idle time in the buffer, which is evident from the near zero Pidle time obtained from the simulation.

Question 5

Question 5: Build a simulator for an M/M/1/K queue, and briefly explain your design.

Class and function name in lab1.py: FiniteEventQueue.create_queue() and FiniteEventQueue.run_simulation()

For the M/M/1/K queue, we created a new class to encapsulate this data structure called FiniteEventQueue which contains the attributes specified in Figure 10. These attributes are similar to that of the M/M/1 queue with the addition of the K value and a buffer queue. The buffer queue will be the data structure that encapsulates our network buffer. For our simulation, this buffer queue will store our departure events that have not yet been departed. Essentially, this means the buffer will store the packets that need to be processed.

```python
class FiniteEventQueue:
    def __init__(self, rho, L, C, simulation_time, K) -> None:
        self.rho = rho
        self.L = L
        self.C = C
        self.simulation_time = simulation_time
        self.K = K
        self.Na = 0
        self.No = 0
        self.num_of_packet_loss = 0

        self.queue = deque()
        self.buffer = deque()
        self.num_packets_in_buffer = []
        self.idle_time = 0
        self.Pidle = 0
        self.Ploss = 0
        self.average_num_of_packets = 0
```

Figure 10: FiniteEventQueue class for M/M/1/K

The event queue creation process is also similar to the finite queue; however, it does not generate departure events as that is generated during the simulation. Again, we create the arrival and observer events for the entire simulation time by leveraging the helper functions shown in Figure 4. The λ for the exponential random variable function is still set to be 5 times the arrival rate for observer events. Lastly, we sort these events and enqueue them into our event queue. Figure 11 shows the process described above.

```python
def create_queue(self):
    arrivals = []
    observers = []

    current_time = 0
    while current_time < self.simulation_time:
        arrival_rate = compute_arrival_rate(self.rho, self.L, self.C)
        arrival_time = exponential_random_variable(arrival_rate) + current_time
        arrivals.append(('arrival', arrival_time))

        current_time = arrival_time

    current_time = 0
    while current_time < self.simulation_time:
        observer_time = exponential_random_variable(5*arrival_rate) + current_time
        observers.append(('observer', observer_time))

        current_time = observer_time

    self.Na = len(arrivals)
    self.No = len(observers)
    all_events = arrivals + observers
    all_events.sort(key=lambda x: x[1])
    self.queue = deque(all_events)
```

Figure 11: Creation of Event Queue for Finite Buffer

The simulation process is captured within the run_simlation method of the `FiniteEventQueue` class shown in Figure 12. The simulation for the `FiniteEventQueue` is similar to the `InfiniteEventQueue`.

We begin by pulling events from the head of the event queue. If we receive an arrival event from the event queue, we create a corresponding departure event with a timestamp depending on whether the departure buffer is empty or not. We then append this departure event to the departure buffer. Note that because we are simulating a Finite network queue, if the departure buffer is equal to the max size of the buffer (K), we do not append a departure event to the buffer and instead, increment a variable `num_of_packet_loss` by 1. At the end of the simulation, we divide the total number of packets lost by the number of arrival events to obtain the proportion of packets lost during the simulation.

Because departure events are generated on demand during simulation, we cannot simply merge the departures into the event queue without incurring a large performance overhead. As such, when we pull an event from the event queue, we compare the timestamp of that event with the timestamp of the head departure event in the departure buffer. If the timestamp of the polled event from the event queue is greater than the timestamp of the departure event, we pop the departure event off the buffer and continue this process until the timestamp of the event from the event queue is <= the timestamp of the head departure event. These departure events can be popped as they technically have happened in the past, and thus do not need any action on them. This process essentially keeps the event queue and the departure buffer in sync and sorted by timestamp.

Finally, when we receive an Observer event, the Observer will simply record the number of events in the departure buffer and append that to an array containing snapshots of the number of events in the buffer. This snapshot array is then used later on to calculate the average number of packets, by summing the snapshots and dividing by the length of the snapshot array.

```
def run_simulation(self):
    last_departure_time = 0
    last_oberserver_time = 0
    idle_in_progress = False
    while self.queue:
        event, current_time = self.queue.popleft()
        while self.buffer and current_time > self.buffer[0]:
            last_departure_time = self.buffer.popleft()
        if event == "arrival":
            if idle_in_progress:
                self.idle_time += (last_oberserver_time-last_departure_time)
                idle_in_progress = False
            if len(self.buffer) >= self.K:
                self.num_of_packet_loss += 1
            else:
                # Generate departure event
                packet_length = exponential_random_variable(1/self.L)
                service_time = compute_service_time(packet_length, self.C)
                if len(self.buffer) == 0:
                    corresponding_departure_time = current_time + service_time
                else:
                    corresponding_departure_time = self.buffer[-1] + service_time
                self.buffer.append(corresponding_departure_time)
        else:
            self.num_packets_in_buffer.append(len(self.buffer))
            if not self.buffer:
                last_oberserver_time = current_time
                idle_in_progress = True

    self.average_num_of_packets = sum(self.num_packets_in_buffer) / len(self.num_packets_in_buffer)
    self.Pidle = self.idle_time / self.simulation_time
    self.Ploss = self.num_of_packet_loss / self.Na
```

Figure 12: The run_simulation Method for Processing M/M/1/K queue.

Question 6

Question 6: Let L=2000 bits and C=1 Mbps. Use your simulator to obtain the following graphs:

- E[N] as a function of ρ (for $0.5 < \rho < 1.5$, step size 0.1), for K = 10, 25, 50 packets. Show one curve for each value of K on the same graph.
- P_{Loss} as a function of ρ (for $0.5 < \rho < 1.5$) for K = 10, 25, 50 packets. Show one curve for each value of K on the same graph. Explain how you have obtained P_{Loss}.

T and 2T 5% Error Calculation

Similar to question 3, we chose T = 1000 and T = 2000 to verify our network queue. The following results were obtained when running the simulation for the two time durations for each K value:

T = 1000

K=10

E[N] for T = 1000: [0.9871017414569846, 1.4517618520867046, 2.1393373734557444, 2.978615574289294, 3.9693160180854727, 4.994170081043447, 5.9384354257338385, 6.7026854691191895, 7.3190105906130585, 7.796143942275946, 8.127904551588761]

Ploss for T = 1000: [**0.0005208938538532121**, 0.0023801439404531263, 0.008868314222633716, 0.023730917774034407, 0.05022822924359579, 0.09079356566218526, 0.1387123283686293, 0.19259084977790236, 0.24440093728803108, 0.29349540013213826, 0.3379489485122546]

K=25

E[N] for T = 1000: [1.0029482167078911, 1.4975159196224703, 2.3293056952189413, 3.9802973754815114, 7.32703462459604, 12.691534168558546, 17.287511143941515, 20.158724149858674, 21.690314418083098, 22.485915877684473, 23.024298823144104]

Ploss for T = 1000: [0.0, 0.0, **2.863475228075802e-05**, **0.0008886508027896125**, 0.007797253663556045, 0.03966093428952902, 0.09749221141357167, 0.16695486166640014, 0.2300385049918148, 0.2861624124892687, 0.3355607842719973]

K=50

E[N] for T = 1000: [0.9925851708221907, 1.4961986031549661, 2.315250200611687, 4.050277546163189, 8.747778930093427, 25.338878501832884, 40.85078721456534, 45.054905561341386, 46.6694131606579, 47.49495197305739, 48.00785656932651]

Ploss for T = 1000: [0.0, 0.0, 0.0, 0.0, **0.0003169473756535654**, 0.020251002851577965, 0.09278672565728774, 0.1679378505194392, 0.2303085028190695, 0.2867857046011478, 0.3349141433652604]

T = 2000

K=10

E[N] for T = 2000: [0.9971746800105009, 1.4630418453183538, 2.118764013793707, 2.9633056365483514, 3.9732769589827748,

5.019318876206567, 5.930203077955896, 6.721754716013294,
7.321858004513623, 7.775806793362058, 8.136750169032634]

Ploss for T = 2000: [**0.0005217506032116648**, 0.0023161117336321434,
0.008787065074558648, 0.02349811045349116, 0.05112598836880157,
0.09181458561726583, 0.14039292241613568, 0.19301975212046046,
0.24552614832190212, 0.292509830316847, 0.3379009323985999]

K=25

E[N] for T = 2000: [0.9913893556392462, 1.4987760171397622,
2.310878437601543, 3.9486097642823172, 7.156647597368459,
12.51185314275637, 17.398601043834844, 20.213569969980583,
21.67920426736887, 22.49797544524413, 22.986685633910874]

Ploss for T = 2000: [0.0, 0.0, **5.2989465121474766e-05**,
0.0008577240804160087, 0.007407423860122961, 0.038696050890697206,
0.09855589562295074, 0.16893679813920917, 0.22977033359482235,
0.2856800915138875, 0.33346590428377376]

K=50

E[N] for T = 2000: [1.0021829396226747, 1.4888974410235907,
2.3585385281761413, 4.032169283590525, 8.832305118337674,
24.77159512178152, 40.313815178412575, 44.971873835599716,
46.59370374906075, 47.510404195753665, 48.002805538605685]

Ploss for T = 2000: [0.0, 0.0, 0.0, **1.2486576929800464e-06**,
0.0006434512994048909, 0.018012463391539866, 0.0918852648833714,
0.16576184036994374, 0.22822604360564924, 0.28675495413099295,
0.3334230957916516]

The bolded numbers within these data results are the En and Ploss values that are less than 0.001. These values are ignored and set to 0 because they cause a very high error difference (due to division by such a small number) and can be considered indistinguishable from one another.

Ploss Error Formula

The following formula was used to compute the error for Ploss:

$$Ploss\ error\ = \frac{\sum_{i=0}^{i=N-1} \frac{abs(Ploss[i]_{1000} - Ploss[i]_{2000})}{Ploss[i]_{1000}}}{N} * 100$$

Equation 3: Ploss Error formula

Error Results

From Equation 1 in Question 3, the error for E[N] between the two simulation times, T = 1000 and T = 2000, was calculated to be the following:

- For K=10, error for E[N] is 0.42%
- For K=25, error for E[N] is 0.70%
- For K=50, error for E[N] is 0.79%

From Equation 3, the error for Ploss between the two simulation times, T = 1000 and T = 2000, was calculated to be the following:

- For K=10, error for Ploss is 0.89%
- For K=25, error for Ploss is 0.96%
- For K=50, error for Ploss is 1.33%

By comparing the errors for E[N] and Ploss for each K value, we can see they are all within the 5% threshold. Therefore, the system is considered stable and a value of T=1000 will be used for the remainder of Question 6.

Ploss Calculation

To calculate Ploss, we simply check if the length of the buffer is greater or equal to K for every arrival event that is trying to enter the buffer. If this is the case, the algorithm does not create a corresponding departure event and increments the `num_of_packet_loss` attribute shown in Figure 12. After all events are processed, the algorithm calculates Ploss by dividing the `num_of_packet_loss` by the total number of Arrival events (`Na`). Ultimately, this gives us a ratio of the total number of packets dropped due to buffer condition and the total number of generated packets.

Results

Recall the simulation was conducted with T=1000 and rho=[0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5]. From plotting E[n] vs rho and Ploss vs. rho for each K value, we obtain the following two graphs Figure 13 and Figure 14 shown below.

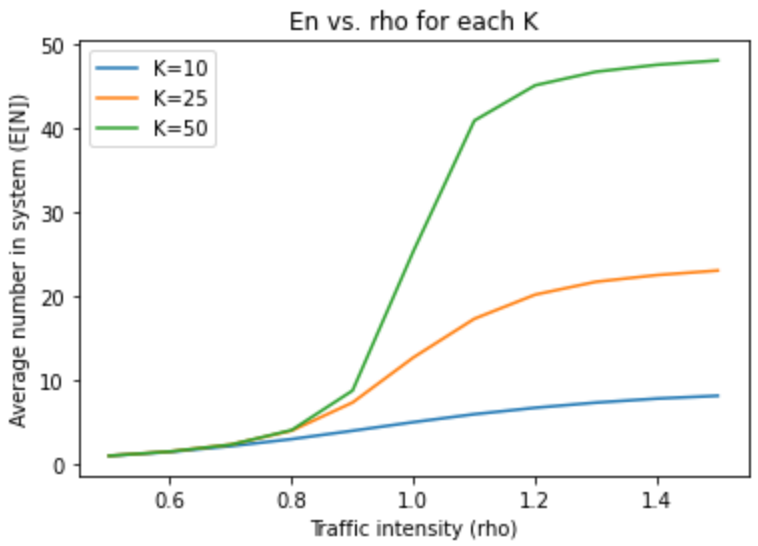

Figure 13: E[n] vs rho for M/M/1/K

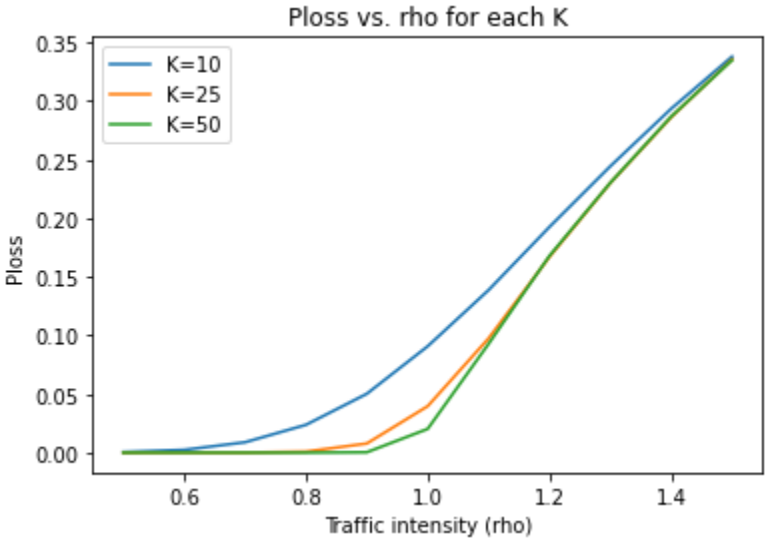

Figure 14: Ploss vs rho for M/M/1/K

For Figure 13, we notice that the average number of packets in the buffer is well below the max capacity of the buffer (K) for low values of rho. However, as rho increases, the average number of packets in the buffer increases exponentially, similar to the exponential increase seen previously in the infinite queue. Comparing the K values, we can see that as we increase the buffer capacity, K, the steepness of the exponential will increase. Furthermore, as rho continues to grow past 1, we observe that E[n] flattens out at the K value. This is because the finite queue is designed to drop the incoming packets that exceed the capacity K of the buffer. As a result, there is a horizontal asymptote at K.

For Figure 14, we notice an exponential increase in Ploss as rho increases. This is intuitive because as the arrival rate continues to exceed the service time, more and more packets will be

dropped from the buffer. We can see that the K value determines the growth rate of this exponential. For example, K=10 starts to drop packets earlier (due to its smaller capacity) than K=50, but ramps up slower. Interestingly, regardless of the buffer capacity size, the Ploss for every value of K converges as rho increases. This means that the ratio between packets dropped and total packets are the same for high values of rho.