University of Waterloo
Faculty of Engineering
Department of Computer and Electrical Engineering

ECE-358 Lab 1 Report

Prepared by
Nicholas Shields
nwshield@uwaterloo.ca
20626940
Eric Tweedle
20605022
emtweedl@uwaterloo.ca

25 September, 2019

Table of Contents

Question 1 **3**

Question 2 **4**

Question 3 **7**

Question 4 **9**

Question 5 **9**

Question 6 **13**

Question 1

Write a short piece of C code to generate 1000 exponential random variable with l=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 l=75? (if not, check your code, since this would really impact the remainder of your experiment)

```
def generateVariables(L, s):
    sum = 0
    values = []

    for i in range(s):
        U = random.random()
        x = -(1/L) * math.log(1 - U)
        values.append(x)
        print(i, ": ", x)
        sum += x

    average = sum/s
    print("average: ", average, 1/L)

   stdev = 0
    for num in values:
        stdev += (num-average)**2

    variance = math.sqrt(stdev/(len(values)-1))

    print("variance: ", variance)
```

The code above outputs the following mean and variance for L=75:
Mean: 0.01322892787756283 (Expected = 0.01333333333)
Variance: 0.012608903911106719
The mean of the 1000 variables generated is very close to the expected value as shown above. For an exponential distribution, the variance is mean value (ie: 1/75), so it follows that the variance we obtained is very close to the expected value as well.

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.

```
import random
import math


class Packet(object):
    def __init__(self, packet_type, time):
        self.type = packet_type
        self.time = time


class SimData(object):
    def __init__(self, idle_counter, rho, EN):
        self.idle = idle_counter
        self.rho = rho
        self.EN = EN


class MM1Simulator(object):
    def __init__(self):
        self.event_scheduler = []
        self.queue_time = 0
        self.num_arrivals = 0
        self.num_departures = 0
        self.num_observers = 0
        self.idle_counter = 0
        self.queue_packets = 0

    def runSimulation(self, llama, T):
        self.generateEvents(T, llama)
        self.generateObservers(T, llama)
        self.event_scheduler.sort(key=lambda x: x.time)
        for event in self.event_scheduler:
            if event.type == "Arrival":
                self.num_arrivals += 1
            elif event.type == "Departure":
                self.num_departures += 1
            elif event.type == "Observer":
                self.num_observers += 1
            if self.num_arrivals == self.num_departures:
                self.idle_counter += 1
            else:
                self.queue_packets += self.num_arrivals - self.num_departures

    def generateProcessTime(self, L=2000, C=1000000):
        """
        :param L: average packet size in bits
        :param C: transfer rate in bps
        :return: service time for arrival packet
        """
        bit_length = self.generateVariables(1 / L)
        service_time = bit_length / C

        return service_time

    def generateObservers(self, time, llama):
        """
        :param time: duration of the simulation
        :param llama: average number of events generated/arrived per second
        :return: None
        """
        i = 0
        while i < time:
            observer_time = i + self.generateVariables(5 * llama)
            observer_object = Packet("Observer", observer_time)
            if observer_time > time:
                break
            self.event_scheduler.append(observer_object)

            i = observer_time

    def generateEvents(self, time, llama):
        """
        :param time: duration of the simulation
        :param llama: Average number of packets generated/arrived per second
        :return: None
        """
        i = 0
        while i < time:
            step_time = self.generateVariables(llama)
            arrival_time = i + step_time

            if arrival_time > time:
                break

            arrival_packet = Packet("Arrival", arrival_time)
            self.event_scheduler.append(arrival_packet)

            if self.queue_time > step_time:
                self.queue_time -= step_time
            else:
                self.queue_time = 0

            processing_time = self.generateProcessTime()

            departure_time = arrival_time + self.queue_time + processing_time
            departure_packet = Packet("Departure", departure_time)

            if departure_time < time:
                self.event_scheduler.append(departure_packet)

            self.queue_time += processing_time
            i = arrival_time

    def generateVariables(self, llama):
        """
        :param llama: Average number of packets generated/arrived per second
        :return: random number with average value of 1/lambda
        """
        return -(1 / llama) * math.log(1 - random.random())
```

The above code is our implementation of an M/M/1 Simulator. When you create an instance of MM1Simulator(), the following member variables are initialized:

- self.event_scheduler = [] → This is our Discrete Event Simulator
- self.queue_time = 0 → Needed to determine if packets are in the queue at a given time
- self.num_arrivals = 0 → Indicates the number of arrival events created
- self.num_departures = 0 → Indicates the number of departure events created
- self.num_observers = 0 → Indicates the number of observer events created
- self.idle_counter = 0 → Counts how many times an observer event observes an idle queue
- self.queue_packets = 0 → counts the difference between the arrival and departure events when an observer event is being processed.

The way the simulator is executed is in the following order:

1) We create the Simulator object, which initializes all the member variables mentioned above.
2) We call the runSimulator(llama, T), which takes as input a lambda value (we referred to it as llama in this lab since lambda is a reserved keyword in Python) and then generates a bunch of arrival departure events based on this criteria. These events are places into the event_scheduler (Which is our representation of the Discrete Event Simulator)

3) After all the events are generated, we generate observer events, using llama = llama*5, so that we have a lot of observer events that allow us to really ‘see’ what’s going on during the simulation. We add these observer events to the event_scheduler list.
4) We then sort the event_scheduler list by times associated with each event in ascending order.
5) Lastly, we iterate through all the events and increment the following counters, based on the event:
    - num_arrivals
    - num_departures
    - num_observers
    - idle_counter

   We also report on the number of packets in the queue when an observer event is being serviced and there are packets in the queue at that time.

So that’s a high level overview of how the Simulator works, but it’s worth going a bit more detail on some of the other functions that make this Simulator work:

- generateVariables(llama): Takes a lambda value and returns an exponential random variable centered around 1/llama.
- generateProcessTime(L, C): Takes as input a length in bits L, and a transfer rate, C. A random variable centered around the bit length is generated and the service time needed to determine the deptature time is returned.
- generateEvents(time, llama): This is the core of the simulator. We take the simulation time as input and then we proceed to generate arrival/departure event pairs and increment a counter until we reach the time specified. We use a queue_time variable, which is our way of knowing if there’s a packet in the queue at a given time. On each arrival event generation, we call the generateProcessTime() function, which returns the service time needed to determine the time of the departure event.

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 < ρ < 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 < ρ < 0.95, step size 0.1). Explain how you do that.*

The average number of packets in the queue (E[N]) is calculated using the observer events. When iterating through the event scheduler the number of arrival and departure packets are recorded. When an observer event occurs it checks if there are more arrival packets than departure packets and if so it adds the difference to a running total as this represents how many packets are currently in the queue. After the entire event schedule is parsed the total is divided by the number of observers to give a value for average number of packets in the queue.

The relationship between the average number of packets in the queue compared to the traffic intensity is shown in figure 1. The relationship can be seen as exponential meaning the queue becomes exponentially more full as the traffic intensity increases.This makes sense as when packets arrive more frequently the queue is less likely to have the other packets processed before a new one is added and this continues to add over the length of the simulation.

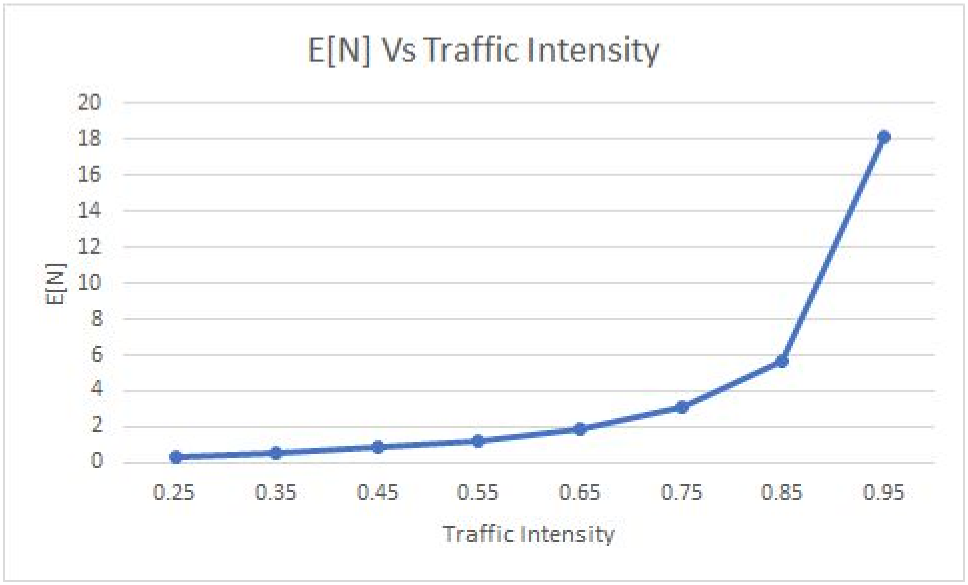

Figure 1: Relationship between average number of packets in the queue and traffic intensity

The portion of time idle is calculated similarly to the average number of packets in the queue but instead of adding up the number in the queue it only increments when an observer event occurs and the number of arrival packets is equal to the number of departed packets. The total is again divided by the total number of observer events.

The relationship between idle time and traffic intensity is seen in figure 2. The graph shows a linear relationship that has the idle time decrease as the traffic intensity increases. This is expected as the more packets there are the more likely there is one in the queue so the idle time decreases. The relationship is inversely proportional to the average number in the queue but since it only increments by one instead of a difference it is not exponential. The portion of idle time starts around 75% of the time but when the intensity is increased it drops to around 5% of the time.

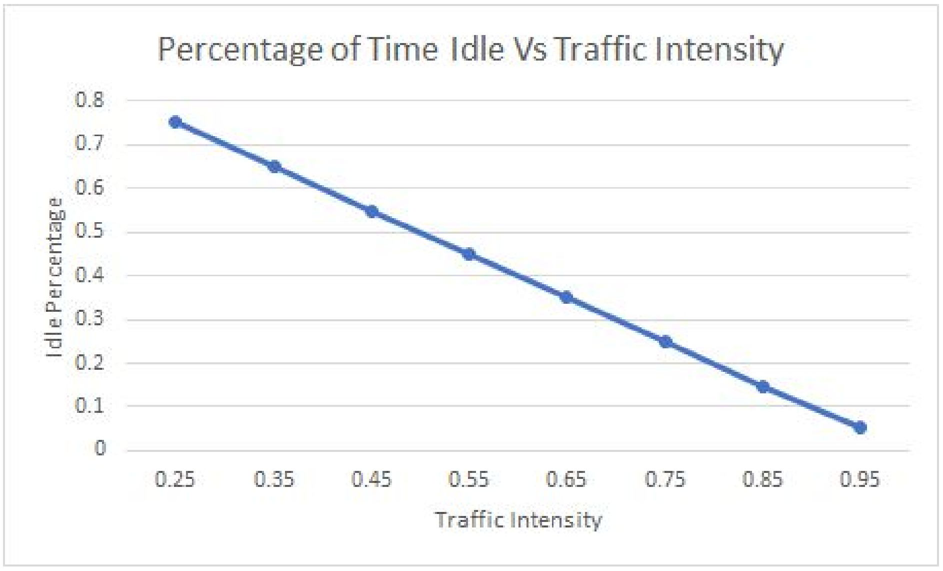

Figure 2: Relationship between percentage of idle time and the traffic intensity

Question 4

For the same parameters, simulate for ϱ=1.2. What do you observe? Explain.
With ϱ=1.2 with the same parameters as used in question 3, the following observations were made:

E[N] = 50057.3
Idle Percentage = 0

With a ϱ value greater than one this means the average rate at which packets arrive is always greater than the average rate they are processed. Therefore the queue will continue to grow for the entire simulation. This means that the queue is never idle, shown by an idle percentage of 0 and that there is a very large value for number of packets in the queue. The average number in the queue is approximately 2800 times greater than the average number when ϱ is 0.95. This demonstrates that if packets are processed slower than they arrive the queue will grow indefinitely and without a limit on the queue, packets could be waiting a long time to depart.

Question 5

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

```
import random
import math


class Packet(object):
    def __init__(self, packet_type, time):
        self.type = packet_type
        self.time = time


class SimData_mm1k(object):
    def __init__(self, rho, K, EN, idle_counter, loss):
        self.idle = idle_counter
        self.rho = rho
        self.EN = EN
        self.loss = loss
        self.K = K


class MM1KSimulator(object):
    def __init__(self):
        self.event_scheduler = []
        self.queue_time = 0
        self.num_arrivals = 0
        self.num_departures = 0
        self.num_observers = 0
        self.idle_counter = 0
        self.queue_packets = 0
        # For MM1K Specific
        self.queue = []
        self.total_packets_generated = 0
        self.dropped_packets = 0

    def runSimulation(self, llama, T, K):
        """
        :param llama: The lambda value used for generating events
        :param T: The duration of the simulation
        :param K: The size of the buffer
        :return: None
        """
        self.generateEvents(T, llama, K)
        self.generateObservers(T, llama)
        self.event_scheduler.sort(key=lambda x: x.time)
        for event in self.event_scheduler:
            if event.type == "Arrival":
                self.num_arrivals += 1
            elif event.type == "Departure":
                self.num_departures += 1
            elif event.type == "Observer":
                self.num_observers += 1
                if self.num_arrivals == self.num_departures:
                    self.idle_counter += 1
            else:
                self.queue_packets += self.num_arrivals - self.num_departures

    def generateProcessTime(self, L=2000, C=1000000):
        """
        :param L: average packet size in bits
        :param C: transfer rate in bps
        :return: service time for arrival packet
        """
        bit_length = self.generateVariables(1 / L)
        service_time = bit_length / C

        return service_time

    def generateObservers(self, time, llama):
        """
        :param time: duration of the simulation
        :param llama: the lambda value
        :return: None
        """
        i = 0
        while i < time:
            observer_time = i + self.generateVariables(5 * llama)
            observer_object = Packet("Observer", observer_time)
            if observer_time > time:
                break
            self.event_scheduler.append(observer_object)

            i = observer_time

    def generateEvents(self, time, llama, K):
        """
        :param time: duration of the simulation
        :param llama: Average number of packets generated/arrived per second
        :param K: BufferSize
        :return: None
        """
        i = 0
        while i < time:
            step_time = self.generateVariables(llama)
            arrival_time = i + step_time

            # Remove packets that have departed
            x = 0
            while x < len(self.queue):
                if self.queue[x].time <= arrival_time:
                    self.queue.pop(x)
                x += 1

            if arrival_time > time:
                break

            arrival_packet = Packet("Arrival", arrival_time)
            self.total_packets_generated += 1

            if self.queue_time > step_time:
                self.queue_time -= step_time
            else:
                self.queue_time = 0

            processing_time = self.generateProcessTime()

            departure_time = arrival_time + self.queue_time + processing_time
            departure_packet = Packet("Departure", departure_time)

            if len(self.queue) < K:
                self.event_scheduler.append(arrival_packet)
                self.queue.append(departure_packet)
                self.queue_time += processing_time
                if departure_time < time:
                    self.event_scheduler.append(departure_packet)
            else:
                self.dropped_packets += 1

            i = arrival_time

    def generateVariables(self, llama):
        """
        :param llama: Average number of packets generated/arrived per second
        :return: random number with average value of 1/lambda
        """
        return -(1 / llama) * math.log(1 - random.random())
```

The M/M/1/K queue has a similar design to that of the M/M/1 queue that was explained in detail in question 2. The big difference occurs in our generateEvents(self, time, llama, K) function, which now includes the K variable, indicating the buffer size. We also implemented a buffer using a Python list (self.queue) which can store up to K packets. In our design, the buffer only holds the departure events and on every arrival event generation, we check to see if there are departure events that finished that can be removed from the buffer. If the buffer is at capacity (ie: there are K events in the buffer) and an arrival event comes in, we drop the arrival event and keep track of this using a counter (dropped_packets variable). We also included a counter that counts the total number of packets generated (total_packets_generated) which is used after we generate all the arrival/departure events, so we can compare how many packets we generated vs how many packets we dropped.

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 < ρ < 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(for0.5<ρ<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. Use the following step sizes for ϱ:*

  For 0.4 < ϱ ≤2, step size 0.1
  For 2 < ϱ ≤5, step size 0.2
  For 5 < ϱ <10, step size 0.4

The average number of packets in the queue versus traffic intensity for different queue sizes is seen in Figure 3. The three different queue sizes have a similar behaviour up to around ϱ=0.9 and they start to diverge after that point when the traffic intensity is high. Around ϱ=1 they are each around half of the respective queue size. After which point they quickly jump up to around the queue size and settle since any addition packets are dropped. The lower queue sizes are always lower since there range is smaller and hence end up with a smaller average.

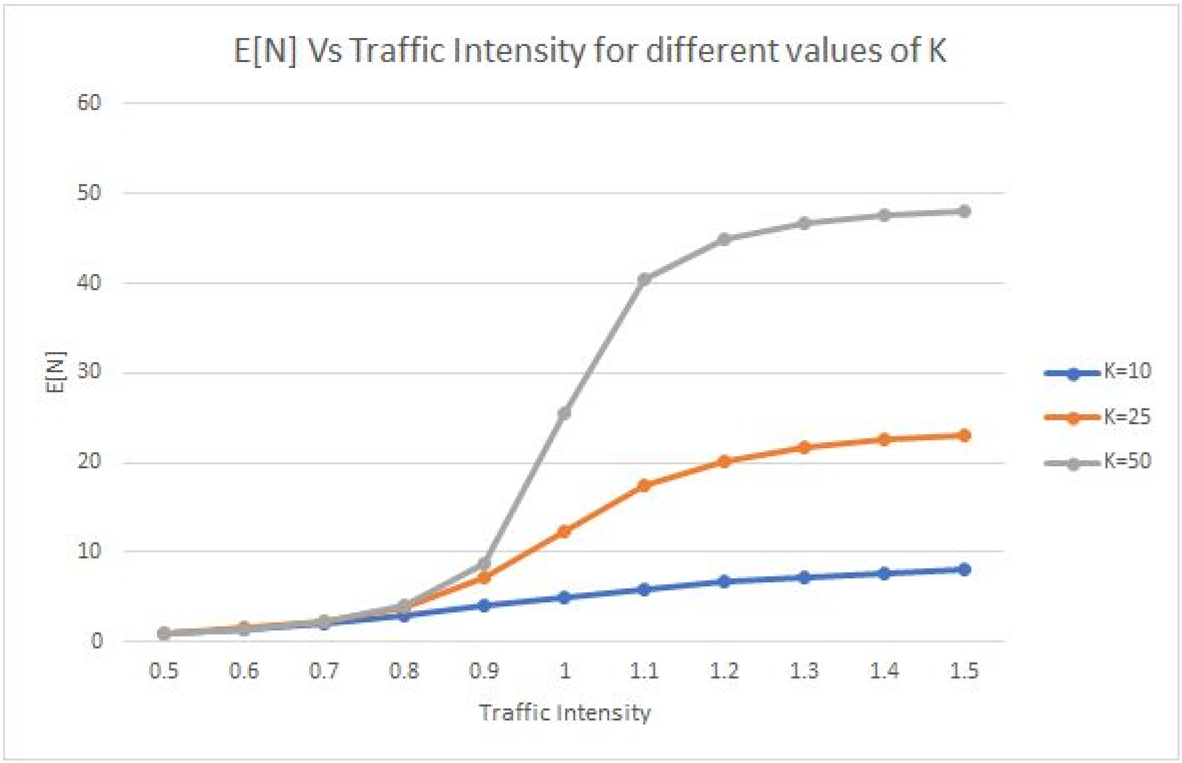

Figure 3: Relationship for the average number of packets in the queue and the traffic intensity for different queue sizes

The percentage of dropped packets is calculated by incrementing a counter every time a packet is generated but the queue is already full. The relationship between the percent of dropped packets and traffic intensity for different queue sizes is shown in Figure 4. The three lines converge around the same point the average number in the queue from Figure 3 is very close to the capacity. After this point all three have full queues for most of the simulation and therefore drop packets at the same rate. The lower queue sizes are always higher than the larger queue sizes as their queues fill up quicker and therefore have more time where packets must be dropped.

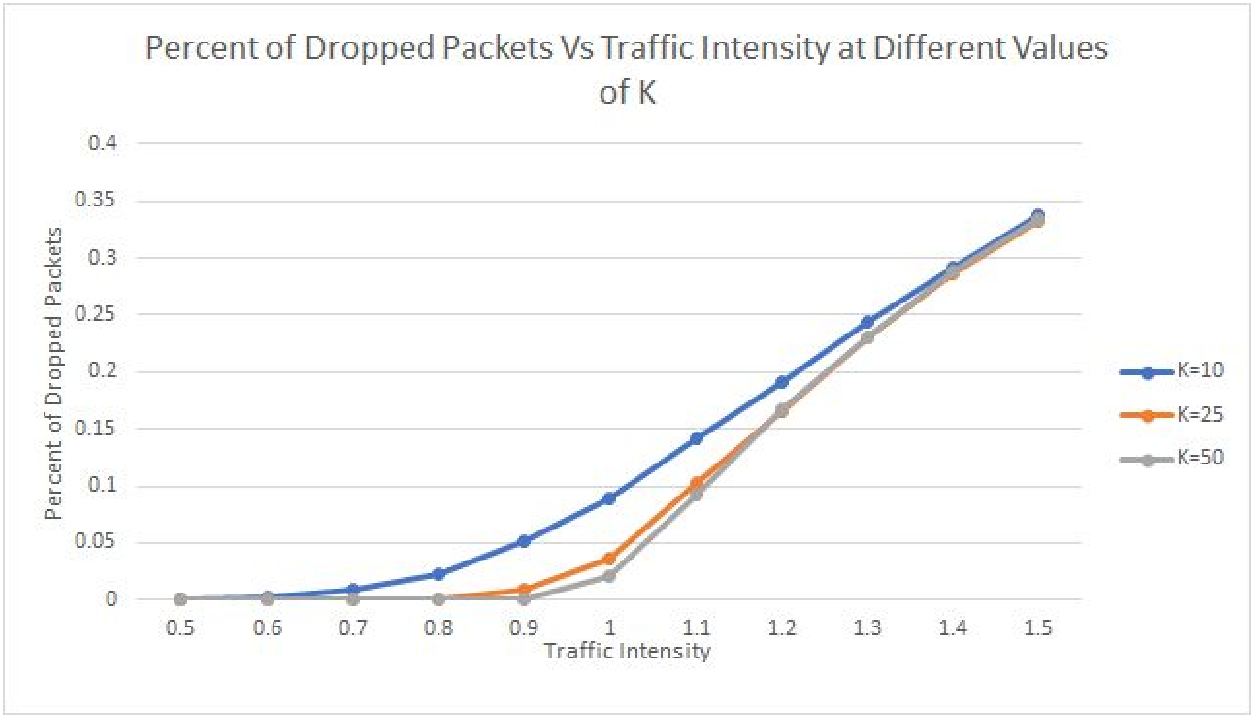

Figure 4: The relationship between the percent of dropped packets and the traffic intensity for different queue sizes

NOTE: To run our simulations run: python3 test.py

This will run all of our simulations as required by Questions 1,3,4,5 and 6