Department of Electrical and Computer Engineering
ECE 358 – Computer Networks

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

Group #45
Names: Shrey Kavi and Chris Huynh
User IDs: skavi (20613111) and cv2huynh (20616110)

Table of Contents

1. Mean = 0.013745426599965874
   Variance = 0.00018271183098679045

   1/75 = 0.0133333..
   (1/75)^2 = 0.00017778

   Here the expected mean and variance are 1/lamda and (1/lamda)^2 respectively. They are quite close to the values given by the 1000 variables generated by our code so we do accept these.

2. The first function "generate_events()" produces the packets. We used an exponential random distribution to produce arrival times until they accumulate to T. The random distribution is given a rate parameter = rho * C / L. The departure times are then produced based on each arrival time and the packets length (also an exponential random distribution but with average = L). The departure is arrival + service_time if queue is empty but if queue is not empty then it is last_departure + service_time. The observer events are created at a random distribution five times as often as the arrival. This function then returns the array of all events sorted by time.

   The second function "DES_simulator()" takes the sorted events and produces counts for the type of packets (Na, Nd, No). The function also pushes E_N and P_idle to the return array, this represents the actual observer event and what it should return. E_N = queue_size/ # of observers AND P_idle = # idle periods / # of observers.

   Notes:
   - We utilized special classes called DESEvent and ObserverRecord to keep things organized.
   - random_variable.py code is included below

   CODE:

```
from __future__ import division
import random_variable as rv
import numpy as np

class DESEvent:
    """
    Class for classifying DES events
    class members:
        type: The type of event (arrival, departure, or observer).
        time: The time at which the event occurs.
    """
    def __init__(self, e_type, e_time):
        self.type = e_type
        self.time = e_time

    def __repr__(self):
        return """
```

```python
{{
    DESEvent.type: {},
    DESEvent.time: {}
}}
""".format(self.type, self.time)

class ObserverRecord:
    """
    Class for recording data at observer events
    class members:
        E_N: The time-average number of packets in the buffer E[N].
        P_idle: The proportion of time the server is idle.
    """
    def __init__(self, E_N, P_idle):
        self.E_N = E_N
        self.P_idle = P_idle

    def __repr__(self):
        return """
{{
    E_N: {},
    P_idle: {}
}}
""".format(self.E_N, self.P_idle)

def generate_events(T, L, C, rho):
    """
    Generate Events for DES
    params:
        T: Time period for simluation
        L: Average package length in bits
        C: Service rate
        rho: Traffic intensity value
    """
    lam = rho*C/L
    arrivals = [rv.exponential(1/lam)]
    l = rv.exponential(L)
    departures = [arrivals[0] + l/C]
    while arrivals[-1] < T:
        a = arrivals[-1] + rv.exponential(1/lam)
        l = rv.exponential(L)
        s = l/C
        arrivals.append(a)
        if a < departures[-1]:
            departures.append(departures[-1]+s)
        else:
            departures.append(a+s)
    observers = [rv.exponential(1/(lam*5))]
    while observers[-1] < T:
        observers.append(observers[-1]+rv.exponential(1/(lam*5)))
    arrivals = [DESEvent(e_type="arrival", e_time=a) for a in arrivals[:-1]]
    departures = [DESEvent(e_type="departure", e_time=d) for d in departures[:-1]]
    observers = [DESEvent(e_type="observer", e_time=o) for o in observers[:-1]]
    events = arrivals+departures+observers
    return sorted(events, key=lambda x: x.time)

def DES_Simulator(events): #m/m/1
    """
    Discrete Event Simulator
    Iterate through events are calculate stats
    params:
        events: Sorted list of DESEvent objects to iterate over.
    """
    observer_records = []
    Na, Nd, No = (0, 0, 0)
    idle_count = 0
    avg_queuesize_sum = 0
    for e in events:
        # Conditional block to update Na and Nd
        if e.type == "arrival":
            Na += 1
        elif e.type == "departure":
            Nd += 1
        # If observer event record data and add to list
        if e.type == "observer":
            No += 1
            queue_size = (Na - Nd)
            idle_count += 1 if not queue_size else 0
            avg_queuesize_sum += queue_size
            observer_records.append(
                ObserverRecord(E_N=avg_queuesize_sum/No, P_idle=idle_count/No)
            )
    print("Na={}, Nd={}, No={}".format(Na, Nd, No))
    return observer_records
```

random_variable.py code:

```
from __future__ import division
import numpy as np
import math

def exponential(beta, size=1):

    collection = []
    for x in range(0, size):
        x = uniform(0,1)
        x = - beta * math.log(1-x)
        collection.append(x)

    if (len(collection) == 1):
        return collection[0]

    return collection
```

3.

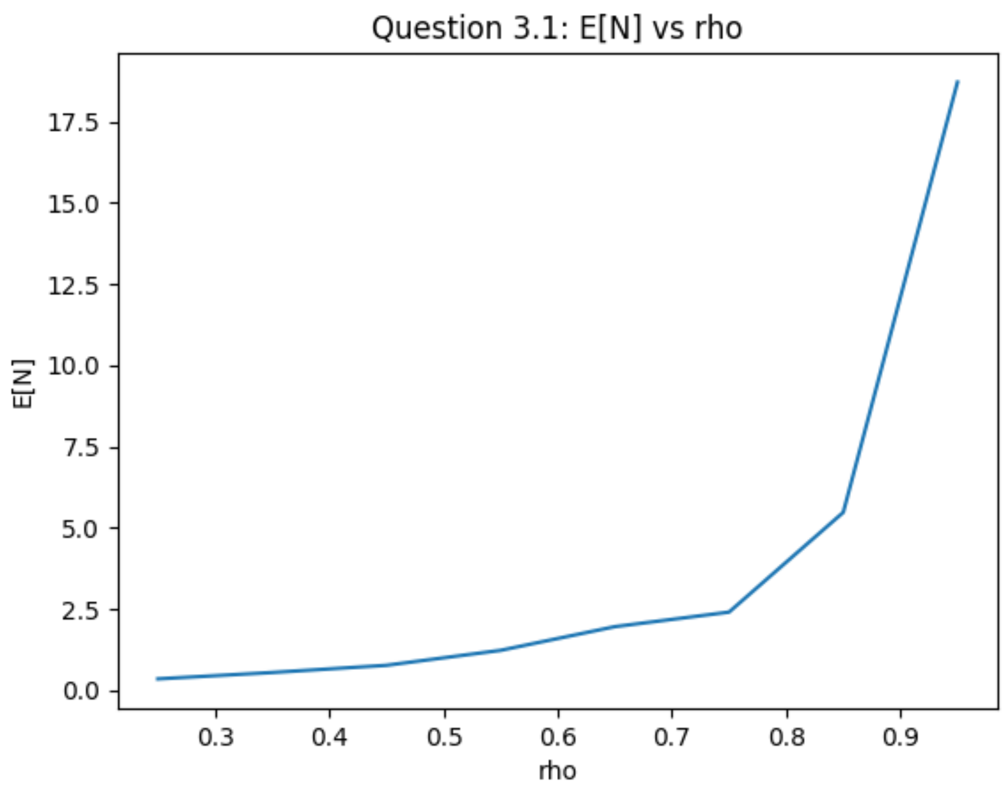

Figure 1: Plot of average number of packets in queue as a function of rho for M/M/1

Comment: Used matplotlib to plot tests with rho from [0.25, 0.9]. With the given C=1Mbps and L=2000 the lambda = rho * 1000000 / 2000 = 500*rho. We started with T = 0.1 and iterated to T = 10 because we wanted to increment time until the graph showed sufficient stability (consistent without noise). Since service rate is constant (C and L) we will see more packets (E[N]) as rho increases and rate of packets is proportional to rho.

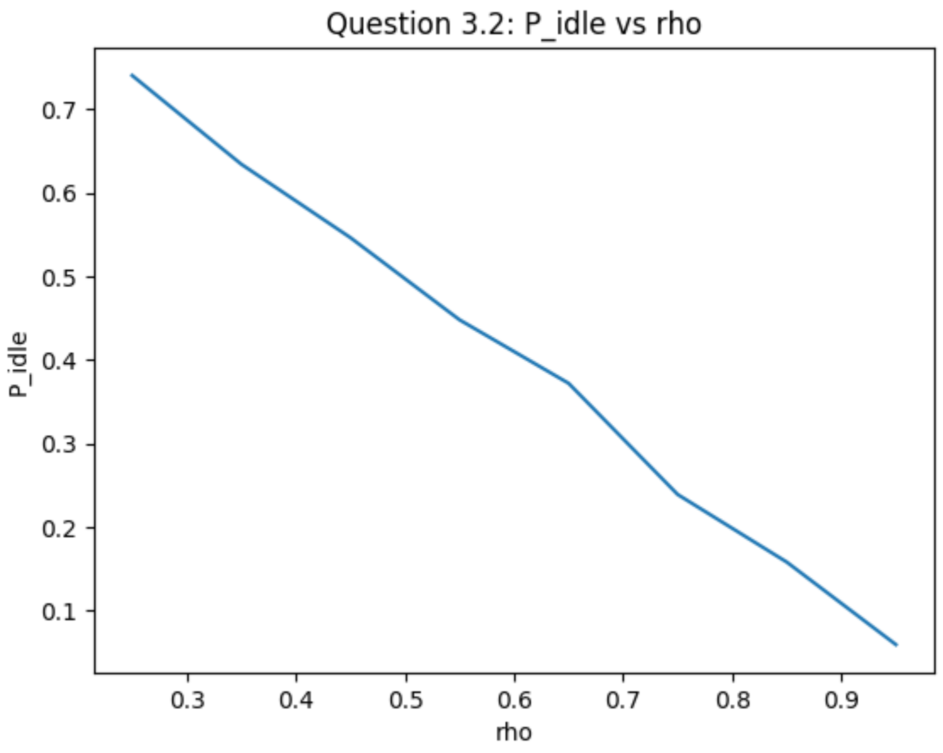

Figure 2: Proportion of time the system is idle as a function of rho for M/M/1

Comment: Used matplotlib to plot tests with rho from [0.25, 0.9]. With the given C=1Mbps and L=2000 the lambda = rho * 1000000 / 2000 = 500*rho. We started with T = 0.1 and iterated to T = 10 because we wanted to increment time until the graph showed sufficient stability (consistent without noise). Since service rate is constant (C and L) and we will see more packets (E[N]) as rho increases the buffer will be idle less often since it must spend more time servicing packets.

4. P_idle gets close to 0, and E[N] drastically increases. Since rho is input rate / service rate so when it is increased to above 1 it means more packets can come in than be served. Therefore, the queue will almost always be populated; driving down P_idle and increasing E[N].

5. This implementation was similar to question 2 but instead of calculating all of arrival, departure and observation times in the "generate_events()" function it excludes the departures. Instead these departure times are calculated on the fly during simulation and pushed into a separate list called "buffer". Since the buffer is limited to K we only allow a K departure times to exist at once and if any further arrivals exist in the time the buffer is full we do not process the departure for these. The simulation also pops the most recent of the events and buffer so we still have the events processing in order of time. The P_loss is the number of packets that are loss due to the buffer being full, we calculate this by keeping count of # of packets lost and using it as follows:

   P_loss = num_packet_loss/Na

CODE:

```python
from __future__ import division
import random_variable as rv
import numpy as np

class DESEvent:
    """
    Class for classifying DES events
    class members:
        type: The type of event (arrival, departure, or observer).
        time: The time at which the event occurs.
    """
    def __init__(self, e_type, e_time):
        self.type = e_type
        self.time = e_time

    def __repr__(self):
        return """
{{
    DESEvent.type: {},
    DESEvent.time: {}
}}
""".format(self.type, self.time)

class ObserverRecord:
    """
    Class for recording data at observer events
    class members:
        E_N: The time-average number of packets in the buffer E[N].
        P_idle: The proportion of time the server is idle.
    """
    def __init__(self, E_N, P_idle, P_loss):
        self.E_N = E_N
        self.P_idle = P_idle
        self.P_loss = P_loss

    def __repr__(self):
        return """
{{
    E_N: {},
    P_idle: {},
    P_loss: {}
}}
""".format(self.E_N, self.P_idle, self.P_loss)

def generate_events(T, L, C, rho):
    """
    Generate Events for DES
    params:
        T: Time period for simluation (s)
        L: Average package length (bits)
        C: Service rate (bps)
        rho: Traffic intensity value
    """
    lam = rho*C/L
    arrivals = [rv.exponential(1/lam)]
    while arrivals[-1] < T:
        a = arrivals[-1] + rv.exponential(1/lam)
        arrivals.append(a)
    observers = [rv.exponential(1/(lam*5))]
    while observers[-1] < T:
        o = observers[-1]+rv.exponential(1/(lam*5))
        observers.append(o)
    arrivals = [DESEvent(e_type="arrival", e_time=a) for a in arrivals[:-1]]
    observers = [DESEvent(e_type="observer", e_time=o) for o in observers[:-1]]
    events = arrivals+observers
    return sorted(events, key=lambda x: x.time)

def DES_Simulator(events, L, C, K): #M/M/1/K
    """
    Discrete Event Simulator
    Iterate through events are calculate stats
    params:
        events: Sorted list of DESEvent objects to iterate over.
        L: Average package length (bits)
        C: Output link rate (bps)
        K: Max buffer size
    """
    buffer = []
    observer_records = []
    Na, Nd, No = (0, 0, 0)
    idle_count = 0
    avg_queuesize_sum = 0
    num_packet_loss = 0
    while events or buffer:
        # Pop event that occurs first
        if buffer:
            if events:
                e = buffer.pop(0) if buffer[0].time < events[0].time else
events.pop(0)
            else:
                e = buffer.pop(0)
        else:
            e = events.pop(0)
        # Conditional block to update Na and Nd
        if e.type == "arrival":
            Na += 1
            package_length = rv.exponential(L)
            service_time = package_length/C
            if len(buffer) < K:
                if buffer:
                    next_departure = buffer[-1]
                    dept_time = next_departure.time+service_time
                else:
                    dept_time = e.time+service_time
                buffer.append(DESEvent(e_type="departure", e_time=dept_time))
            else:
                num_packet_loss += 1
        elif e.type == "departure":
            Nd += 1
        # If observer event record data and add to list
        if e.type == "observer":
            No += 1
            queue_size = len(buffer)
            idle_count += 1 if not queue_size else 0
            avg_queuesize_sum += queue_size
            P_loss=num_packet_loss/Na if Na else 0
            observer_records.append(
                ObserverRecord(E_N=avg_queuesize_sum/No, P_idle=idle_count/No,
P_loss=P_loss)
            )
    return observer_records
```

6.

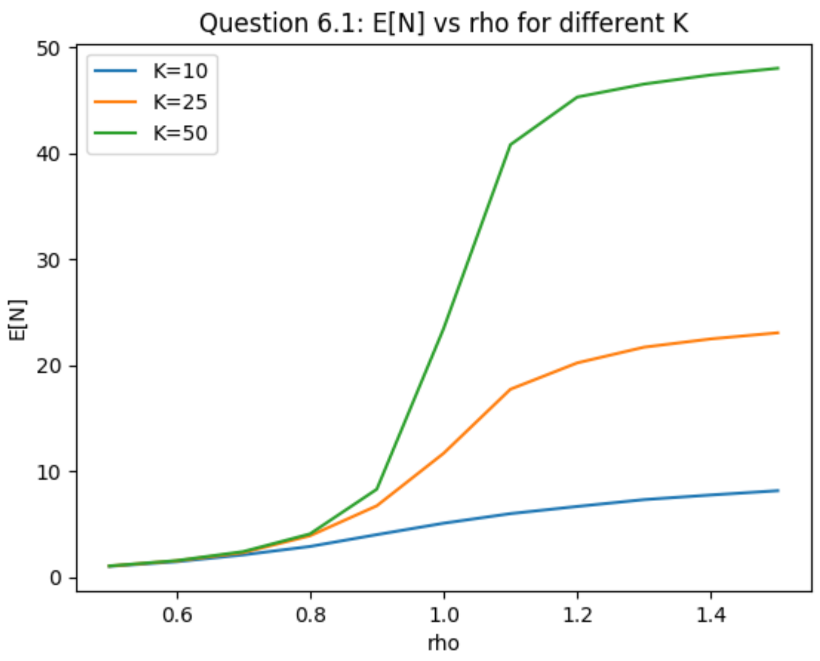

Figure 3: Plot of average number of packets in queue as a function of rho for M/M/1/K

The average number of packets is similar to the M/M/1 except it plateaus and has a peak because it is limited by the buffer size. This is shown in the next figure which describes packets loss increasing over time (P_loss).

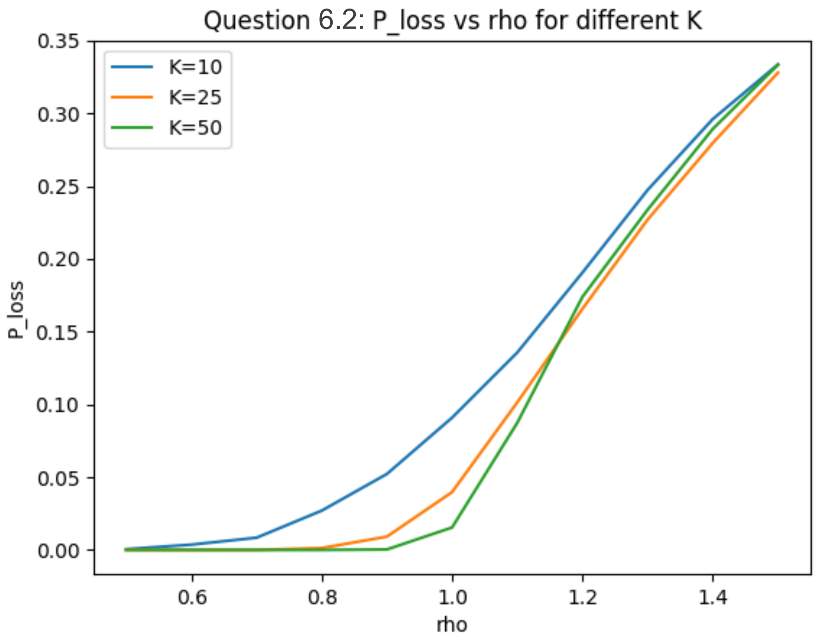

Figure 4: Proportion of time the system is idle as a function of rho for M/M/1/K

As rho increases so does the number of packets, this translates to more packets needing to be process but since the buffer can’t hold all these packets it increases P_loss.