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

Manthan Shah, 20658832
Heather D’Souza, 20671903

Faculty of Engineering
Department of Electrical and Computer Engineering

September 29, 2020
Course Instructor: Prof. A. Wasef

1 Exponential random variable generation

The mean and variance of the 1000 random variables, generated using the generateRandomVariable method were:

$$\mu = 0.0135$$
$$\sigma^2 = 0.000183$$

The expected value for $\lambda = 75$ can be computed as:

$$E[X] = \frac{1}{\lambda} = \frac{1}{75} = 0.0133$$

$$Var[X] = \frac{1}{\lambda^2} = \frac{1}{75^2} = 0.000178$$

Therefore, since the experimental mean and variance values (generated via code over 1000 samples) are within 5% of the expected values for $\lambda = 75$, the generateRandomVariable method is suitable for generating exponential random variables.

2 M/M/1 Queue Simulation

A discrete event simulator (DES) was developed for the infinite buffer scenario. The design of this simulator has two components; a method called buildEventsForInfiniteBuffer which generates arrival, departure, and observer events, and then there is an infiniteBufferDes method which performs the actual buffer simulation process.

2.1 Event generation

Each event is modelled as an object, that has two properties; time and event_type. The time property describes a relative time in seconds in the context of the event simulation program. The event_type property is an enumeration value which describes the type of event (arrival, departure, or observer).

```
# Enumeration that defines the different event types.
class EventType(Enum):
   ARRIVAL = 0
   DEPARTURE = 1
   OBSERVER = 2
```

```
# Data structure that describes an event based on it's simulation
time and type.
class Event:
   def __init__(self, time, event_type):
       self.time = time
       self.event_type = event_type
```

The event generation method sets up three variables, an event_queue which is a list of all the events, delta_t variable which tracks the change in time as arrival and departure events are created, and an obs_t variable which tracks the change in time as observer events are created. Event_queue is set up as an empty list, and delta_t and obs_t are both initialized to 0.

```
def buildEventsForInfiniteBuffer(T, l, L, C):
   event_queue = []
   delta_t, obs_t = 0, 0

   last_departure_time = 0
   while delta_t < T or obs_t < T:
       if delta_t < T:
           # Determine the next arrival event time, create the
arrival event, and
           # add it to the queue.
           delta_t += generateRandomVariable(l)
           arrival_event = Event(delta_t, EventType.ARRIVAL)
           event_queue.append(arrival_event)

           # Compute the service time as L / C, where L follows
exp. dist.
           service_time = generateRandomVariable(1 / L) / C

           # If arrival time occurs before last departure event
has exited queue,
           # then we compute departure event time as last time +
service time. Otherwise,
           # compute as arrival time + service time (no other
packets in queue).
           if arrival_event.time < last_departure_time:
               departure_event = Event(last_departure_time +
service_time, EventType.DEPARTURE)
           else:
               departure_event = Event(arrival_event.time +
service_time, EventType.DEPARTURE)
           last_departure_time = departure_event.time

           event_queue.append(departure_event)
```

```
        if obs_t < T:
            # Add observer events at a rate 5x that of arrival
events.
            obs_t += generateRandomVariable(5 * l)
            observer_event = Event(obs_t, EventType.OBSERVER)
            event_queue.append(observer_event)

    event_queue.sort(key=lambda x: x.time, reverse=True)
    return event_queue
```

Then, there is a loop which runs until a parameter-defined time threshold, T - the standard for this lab was T = 1000. This loop runs until delta_t or obs_t have not approached the T threshold. First, a check is performed to ensure that delta_t is within the threshold. This is done because it is possible that delta_t has been exhausted (reached the threshold), but obs_t has not; this is because obs_t has much smaller steps due to a 5x rate compared to arrival events. Then, a time calculation is done to determine the next time for an arrival event (using the exponential random variable generation method), which gets added to the event queue. The code associated with exponential random variable generation, mathematically, translates to:

$$\Delta t = (\frac{-1}{\lambda}) ln(1 - U)$$

The service time is then computed for a corresponding departure event (each arrival event must have a departure event). This is done by taking an exponential random variable that is centered around a parameter-value L (packet length) over the speed of the link. If the event queue is empty, then the departure time is computed as the sum of the corresponding arrival event time and service time. Now, it must be determined if the queue is empty or not to determine the departure time. This is done by checking if the last departure time (of the last departure event if there is one), occurs at a point after the current arrival time. This means the queue is not empty and the departure time computed as the sum of the last departure time and service time. If the queue is not empty (arrival time is at a point of time after the last departure time), then the departure event time is the sum of the time of the corresponding arrival event and service time. Finally, the last departure time variable is also updated for future iteration computations.

For observer event generation inside the loop, the observer event time is first computed at a rate that is 5 times that of the rate used for arrival events, using the exponential random variable generation method. The observer events then just get added to the event queue (at a much higher rate so there are a lot more of them).

Finally, the entire event queue is sorted, since the range of times between the observer and arrival/departure event times will be not in order.

2.2 Infinite buffer simulator

The infinite buffer simulator method sets up six different variables; num_arrivals to track arrival events, num_departures to track departure events, total_packets to track packets in the buffer during each observer event, observations to track the number of times we encounter an observer event, and an empty_counter to track the number of times the queue is empty during an observer event. All of these variables are initialized to 0.

```
def infiniteBufferDes(events, T, L, C):
   # Setup variables for computing e_n and p_loss.
   num_arrivals, num_departures, total_packets, observations,
empty_counter = 0, 0, 0, 0, 0

   while len(events) > 0:
       event = events.pop()
       if event.time >= T:
           break

       print(event.time, event.event_type)

       if event.event_type == EventType.ARRIVAL:
           num_arrivals += 1
       elif event.event_type == EventType.DEPARTURE:
           num_departures += 1
       else:
           # Determine the buffer length and increment total
packets that have
           # been observed.
           buffer_length = num_arrivals - num_departures
           total_packets += buffer_length
           observations += 1
           if buffer_length == 0:
               empty_counter += 1

   # e_n is average # of packets based on total # of observer
events.
   e_n = total_packets / observations

   # p_loss if average # of times empty buffer was observed based
   # on total # of observer events.
   p_loss = (empty_counter / observations) * 100
   return (e_n, p_loss)
```

The simulator method operates in an "event-based" manner as described in the lab manual. The method takes an input events list (generated using the buildInfiniteBufferEvents method), and loops through all the events inside. In each loop iteration, the event is first removed from the tail of the list. It is ensured that the time associated with the event is within the parameter threshold T as a safeguard. If the event is an arrival event, the num_arrivals counter is incremented. If the event is a departure event, the num_departures counter is incremented. In the case of an observer event, we compute the buffer length by taking the difference of num_arrivals and num_departures, and then adding that to the total_packets variable. The observations counter is incremented, and then it is checked that if the buffer is empty at this particular observer event. If it is, the empty_counter variable is incremented.

Thereafter, the time-average of the packets in the queue E[N] and the proportion of time the server is idle is computed, P_{idle}. E[N] is computed by dividing the total_packets variable by the observations variable. The observations variable is the time unit for every time total_packets is recorded. So taking the division yields the average in the system over time. P_{idle} is similarly computed by taking the empty_count variable and dividing it by the number of observations, and then multiplying by 100 (to get a percentage).

2.3 Infinite buffer simulator verification

To ensure that the simulator performance is stable, the lower and upper thresholds of ρ (traffic intensity) were tested at simulation times, T = 1000 s and T = 2000 s. The average number of events in the system and the percentage of time the system was idle were computed. Table 1 described the results that were obtained as a result.

Table 1: changes in E[N] and P_{idle} during lower and upper thresholds of ρ with T and 2T

Traffic intensity (ρ)	Simulation time (T)	Average number of events in the system (E[N])	Proportion of time system is idle (P_{idle})
0.25	1000 s	0.334	74.99 %
0.25	2000 s	0.333	74.91%
0.95	1000 s	20.235	4.67 %
0.95	2000 s	19.815	4.80 %

It can be observed from table 1, that when the simulation time was doubled, the results (E[N] and P_{idle}) were consistently within 5% of each other (between the simulation times). The values of 0.25 and 0.95 were chosen as testing targets of ρ because they represent the lower and upper bounds presented in the lab manual. It allows us to approximate and extrapolate that if the performance is stable at the boundaries, it will likely be stable for values of rho in between. This comes with a tradeoff of not testing every single possible value of ρ, which in a real-world situation would be a very expensive process and unfeasible. Thus, only the extremities of rho were used for simulation stability testing. This sufficiently demonstrates the system is stable, and can be used as an infinite buffer discrete event simulator (DES).

2.4 Performance metrics when $0.25 < \rho < 0.95$

The infinite buffer simulator method was run with a packet length following an exponential distribution around L = 2000 bits, and C = 1 Mbps. The following E[N] graph was generated:

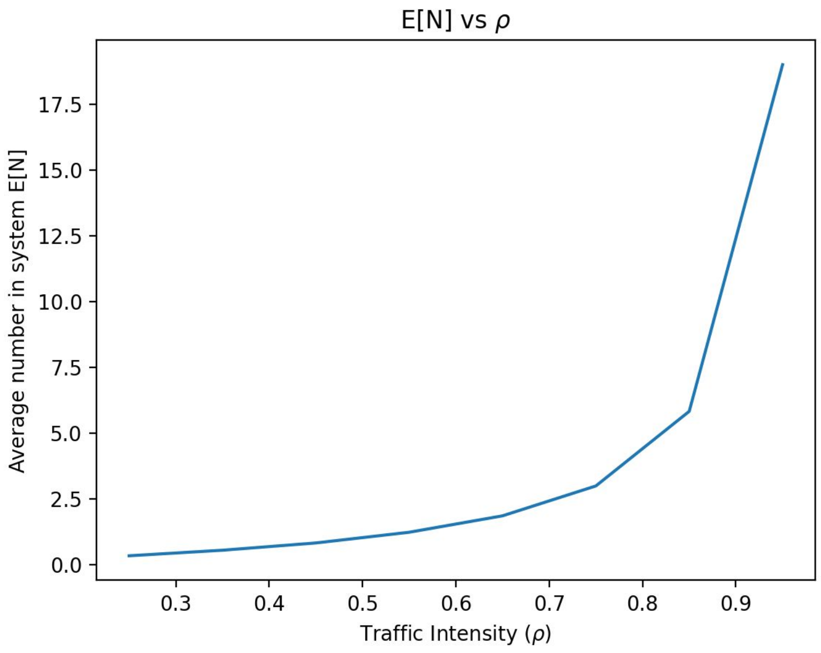

Figure 1: E[N] vs p with a simulation time of T = 1000 s

Figure 1 shows the average number of packets in the system as the traffic intensity parameter was sweeped from 0.25 to 0.95 with a step size of 0.1. It can be observed that at lower traffic intensities, the buffer is less full, thus lower average number of packets. As the traffic intensity is increased (specifically to 0.95), there is a significant increase in the number of packets in the buffer. This metric was computed by taking the total number of packets and dividing it by the number of observations, since packet count is only computed during an observer event.

The following represents the proportion of time the buffer was empty, P_{idle}:

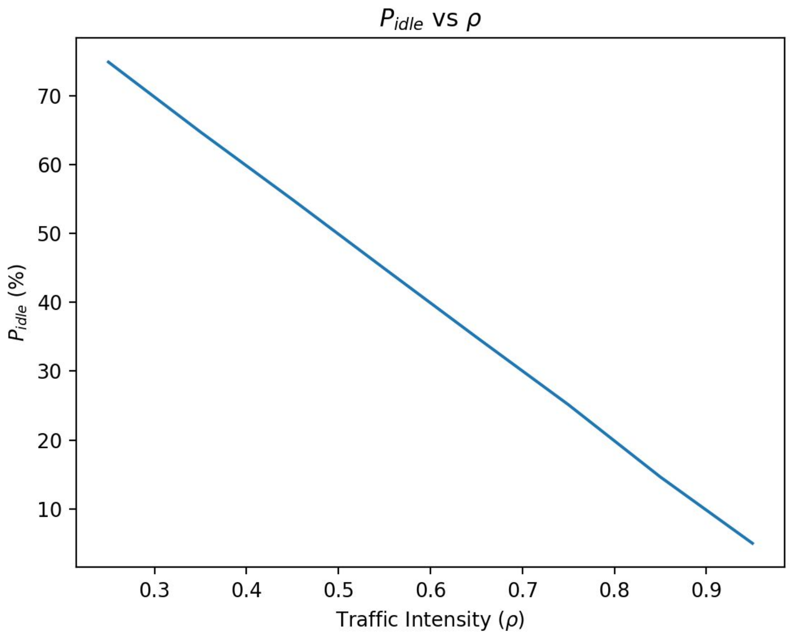

Figure 2: P_{idle} vs p with a simulation time of T = 1000 s

Figure 2 represents the proportion of time the buffer was empty as the traffic intensity parameter was sweeped from 0.25 to 0.95 with a step size of 0.1. It can be observed that P_{idle} decreases as the traffic intensity increases. This is because at a lower traffic intensity, the buffer is empty more because it can process the arrival and departure events without getting backed up. As traffic intensity increases, the number of events increases, and thus the P_{idle} decreases. This metric was computed by taking the total number of times the buffer was empty and dividing it by the number of observations (and multiplying by 100 to get a percentage) as well.

The code to compute E[N] and P_{idle} as functions of ρ were encapsulated in the `q3` method, which took a parameter `T` (the simulation time) which was defaulted to T = 1000 seconds unless specified. The method initializes two empty lists, E_N and P_idle to store the values at each step of ρ. Then, from the lab manual, the packet length (L), link speed (C), and list of rho values were initialized as variables.

The multiprocessing library in Python was used to improve the computation times. This meant first creating a list of arguments as tuples to be fed into the buildEventsForInfiniteBuffer method for each value of rho. Thereafter, the event generation method was called (using multiprocessing) for all values of rho, and stored into the event_list variable.

A similar process was also done for the infinite buffer simulator portion. A list of tuple arguments were created for the infiniteBufferDes method for multiprocessing based on the events associated with each value of ρ. Using the events generated earlier in the method, and the variables from the lab manual, the simulator results were stored into the results variable.

Then, the E_N and rho values that were returned from each infiniteBufferDes method call were extracted, and populated into the E_N and P_idle list variables in the method. Using matplotlib in Python, the plots were then generated, leading to figures 1 and 2.

```
def q3(T=1000):
   # Setup lists to append values to as: 0.25 < rho < 0.95.
   E_N = []
   P_idle = []

   # Total # of CPU cores we can use to multiprocess the
simulation.
   pool = Pool(cpu_count())

   C, L = 10 ** 6, 2000
   rho_list = [0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85, 0.95]
   events_list_args = []

   # For each value of rho, we compute the arrival rate, and
append to our
   # args list to be used for generating all events for all rho.
   for rho in rho_list:
       l = rho * (C / L)
       events_list_args.append((T, l, L, C))

   # List of arr/obs/dep events for each value of rho from 0.25 to
0.95.
   events_list = pool.map(buildEventsForInfiniteBufferWrapper,
events_list_args)

   # For each list of events, append to our args list for the DES.
   des_args = []
   for events in events_list:
       des_args.append((events, T, L, C))

   # Multiprocess the DES, and strip out the e_n and p_loss values
from each
   # simulation for each rho.
   results = pool.map(infiniteBufferDesWrapper, des_args)
   for result in results:
       E_N.append(result[0])
       P_idle.append(result[1])
```

```
    plt.plot(rho_list, E_N)
    plt.title(r'E[N] vs $\rho$')
    plt.xlabel(r'Traffic Intensity ($\rho$)')
    plt.ylabel('Average number in system E[N]')
    plt.show()

    plt.plot(rho_list, P_idle)
    plt.title(r'$P_{idle}$ vs $\rho$')
    plt.xlabel(r'Traffic Intensity ($\rho$)')
    plt.ylabel(r'$P_{idle}$ (%)')
    plt.show()
```

2.5 Traffic intensity $\rho = 1.2$

When the traffic intensity parameter was set to 1.2, with a T = 1000, the following values were observed:

$$E[N] = 49884.37$$
$$P_{idle} = 0.00043\ \%$$

As expected, the average number of events in the buffer significantly increased, since the effective rate of arrival and departure events significantly increased. Since there are more events overall, naturally, the proportion of time that the buffer was empty was significantly smaller compared to smaller traffic intensities.

3 M/M/1/K Queue Simulation

3.1 Event generation

To run the simulation, arrival and observer events were created in a similar way to the events created for the M/M/1 simulation. Arrival and observer events are created such that the timestamp of each event is less than or equal to *T*, the total simulation time. This is facilitated by *delta_t and obs_t*, which keep track of the timestamp of the last arrival and observer event, respectively. Additionally, the observer events are created at 5 times the rate of packet arrival events as described in the lab manual. However unlike the M/M/1 simulation, departure events cannot be created beforehand. This is because in a finite queue simulation, packet loss will occur if the queue is full. This means that not all events that arrive to the queue will depart from the queue. As a result, departure events are created during the simulation and depend on the current state of the queue.

```
def buildEventsForFiniteDes(T, l):
    event_queue = deque()
    delta_t, obs_t = 0, 0

    # only generate arrival and observer events since departure
      events will be created during the simulation
    while delta_t < T or obs_t < T:
        if delta_t < T:
            delta_t += generateRandomVariable(l)

event_queue.appendleft(Event(delta_t,EventType.ARRIVAL))

        if obs_t < T:
            obs_t += generateRandomVariable(5 * l)
            event_queue.appendleft(Event(obs_t,
EventType.OBSERVER))
    return event_queue
```

3.2 Finite buffer simulator

The simulation processes arrival and observer events in an ascending order, or in other words, from the smallest timestamp to the biggest timestamp until the *events* queue is empty. During each iteration, the size of the queue is determined by subtracting *num_departures* from *num_arrivals*. If the current event is an arrival event and the queue is full, no departure event will be created, and the *loss_counter*, which tracks the number of packets lost, and *lost_arrivals*, the number of arriving packets that did not make it into the queue, will be incremented. It is important to note that the *num_arrivals* counter is not incremented in the case of packet loss. In the event that the queue is not empty, a departure event will be created and added to the list of *departure_times* (more specifically, a Python deque). The departure time will depend on whether the queue is empty or if the packet must wait in the queue. Just like the M/M/1 simulation, if the queue is empty, the departure time is simply the arrival time of the packet plus the transmission time. If the queue is not empty, the departure time is the departure time of the previously serviced packet plus the transmission time.

```
def finiteBufferDes(T, l, L, C, K, events):
    # setup variables for computing e_n and p_loss
    # num_arrivals: number of arrival events of packets that
    # are not dropped
    # num_departures: number of departure events
    # total_packets: the total number of packets, dropped and
    # not dropped
    # observations: number of observation events
    # empty_counter: times during an observation the queue is
empty
    # last_departure_time: the departure time of the most recently
    # departed packet
    # loss_counter = number of packets dropped
    # lost_arrivals = number of arrival events of packets that
    # are dropped
    num_arrivals, num_departures, total_packets, observations,
empty_counter = 0, 0, 0, 0, 0
    last_departure_time, loss_counter = 0, 0
    lost_arrivals = 0

    departure_times = deque()
    while events:
        departure_time = departure_times[-1] if departure_times
else              float('inf')
        event = events[-1]

        # exit function if event time or departure time is greater
        # than the simulation time
        if event.time >= T or last_departure_time >= T:
            break

        # Note: num_arrivals only refers to packets that will have
a
        # corresponding departure
        buffer_length = num_arrivals - num_departures
        if event.time < departure_time:
            events.pop()
            if event.event_type == EventType.ARRIVAL:
                # print("ARRIVAL", event.time)
                # if buffer is full, the packet will be dropped
                if buffer_length == K:
                    loss_counter += 1
                    lost_arrivals += 1
                    continue

                # the service rate follows an exponential
               # distribution
                service_time = generateRandomVariable(1 / L) / C
                # if buffer is empty, departure time is the
                # service time + the arrival time
                # if buffer is not empty, departure time is
                # the service time + the departure time
                # of the last packet
                if buffer_length == 0:
                    last_departure_time = service_time +
event.time

departure_times.appendleft(last_departure_time)
```

```
                else:
                    last_departure_time += service_time

departure_times.appendleft(last_departure_time)

                num_arrivals += 1
            else:
                # print("OBSERVER", event.time)
                total_packets += buffer_length
                observations += 1
                if buffer_length == 0:
                    empty_counter += 1
        else:
            # print("DEPARTURE", departure_time)
            departure_times.pop()
            num_departures += 1

    e_n = total_packets / observations
    p_loss = (loss_counter / (num_arrivals + lost_arrivals)) * 100
    p_idle = (empty_counter / observations) * 100
    return (e_n, p_loss, p_idle)
```

E[n] is calculated by dividing the total number of packets by the number of observation events. P_{LOSS} is calculated by dividing `loss_counter`, the number of packets lost, by the number of packets that arrived at the queue. The latter metric includes both `num_arrivals` and `lost_arrivals`, the successful arrivals and non-successful arrivals, respectively.

3.3 Finite buffer simulator verification

The simulator performance was tested by comparing E[N] and P_{LOSS} when the simulation was run for 1000 seconds vs 2000 seconds. Since the simulator will be used for ρ, traffic intensity, where ρ is between 0.5 and 1.5, verification was done on these bounds. The size of the queue, K, for this experiment was set at 25 packets. Table 2 describes the results that were obtained.

Table 2: Changes in E[N] and P_{LOSS} for lower and upper thresholds of ρ with K = 25

Traffic intensity (ρ)	Simulation time (T)	Average number of events in the system (E[N])	Packet loss probability
0.5	1000 s	1.006	0.00000
0.5	2000 s	0.998	0.00000

1.5	1000 s	24.631	33.33058
1.5	2000 s	23.749	33.33196

It can be observed from Table 2, that when the simulation time was doubled, the results (E[N] and P_{LOSS}) were consistently within 5% of each other (between the simulation times).

2.4 Performance metrics

The performance of this simulator was tested by increasing the traffic intensity for various queue sizes. Figure 3 shows a graph displaying these results where E[N], the average number of packets in the queue, is graphed as a function of traffic intensity, ρ, where ρ lies within the range 0.5 to 1.5.

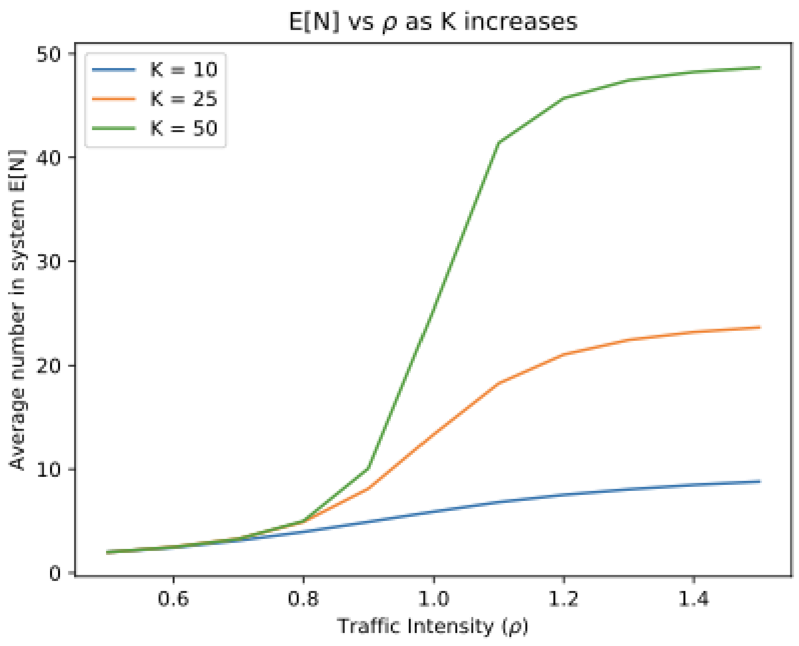

Figure 3: E[N] vs ρ for K = 10, 25, 50 packets with a simulation time of T = 1000 s

It can be observed that for each K, as traffic intensity increases, the average number of packets in the queue also increases. This is expected since a higher traffic intensity would result in an increased rate of arrival events, and this increase in rate would result in a more full queue. Each graph levels out at about the size of the queue since at a very high packet arrival rate, the queue would almost always be full.

P_{LOSS} was also calculated for traffic intensities between 0.5 and 1.5 for each queue size (Figure 4).

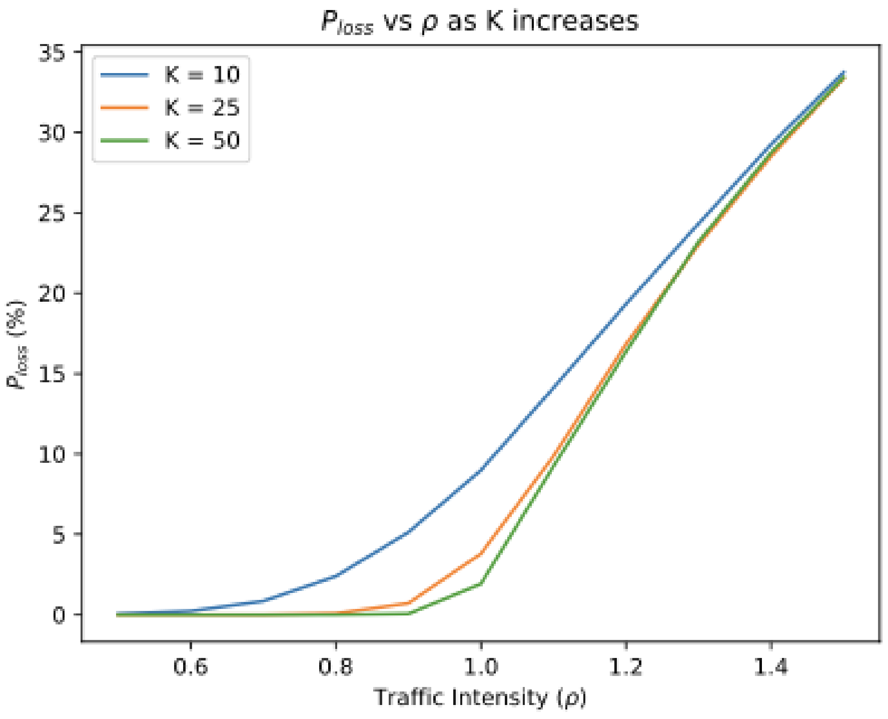

Figure 4: P_{LOSS} vs ρ for K = 10, 25, 50 packets with a simulation time of T = 1000 s

It is observed that P_{LOSS} increases with traffic intensity. At higher traffic intensities, the queue is utilized to a greater degree, and will often be full which results in dropped packets. It is also observed that the smaller the queue size, the faster the rate that P_{LOSS} increases. This is depicted by a larger slope as traffic intensity increases. This is also clear when comparing P_{LOSS} of each queue at a specific ρ. For example, at ρ = 0.9, the probability of packet loss is greatest for K = 10 and lowest for K = 50. The three graphs all reach a P_{LOSS} of about 34% at a traffic intensity of 1.5.

As described earlier, the packet loss probability is determined at the end of every finiteBufferDes run. The number of packets lost is divided by the number of packets that arrived at the queue. This result is multiplied by 100 to yield a percentage. To plot the relationship between ρ and P_{LOSS} and ρ and E[N], buildEventsForFiniteDes and finiteBufferDes was run for each ρ value between 0.5 and 1.5 (with a step of 0.1) for each K. The P_{LOSS} and E[N] for each run was extracted and the results were grouped depending on the K for which they were computed.

```
def q6(T=1000):
    # setup lists to append values to
    E_Ns = []
    P_LOSSes = []

    # from lab manual: traffic intensity (rho), queue size (K),
    # avg. length of packet (L), transmission rate (C)
    # Note: simulation time (T) was determined according to
    # the process described in the manual
    rho_steps = [0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4,
1.5]
    K_steps = [10, 25, 50]
    L, C = 2000, 10 ** 6

    # for each queue:
    # 1. calculate average number of packets arrived (lambda) for
    #    each value of rho
    # 2. generate events
    # 3. run finite buffer simulation (M/M/1/K) with generated
    #    events
    # 4. extract two metrics: average number of packets in queue
    #    (E[N]) and packet loss probability (Ploss) from each
    #    simulation result
    for K in K_steps:
        e_n = []
        p_loss = []
        for rho in rho_steps:
            l = rho * (C / L)
            events = buildEventsForFiniteDes(T, l)
            des = finiteBufferDes(T, l, L, C, K, events)
            e_n.append(des[0])
            p_loss.append(des[1])
        E_Ns.append(e_n)
        P_LOSSes.append(p_loss)

    # plot E[N] as a function of rho for each K
    f = plt.figure()
    for i in range(len(E_Ns)):
        plt.plot(rho_steps, E_Ns[i], label=f"K = {K_steps[i]}")
    plt.legend(loc="upper left")
    plt.title(r'E[N] vs $\rho$ as K increases')
    plt.xlabel(r'Traffic Intensity ($\rho$)')
    plt.ylabel('Average number in system E[N]')
    plt.show()
    f.savefig("en_q6_figure.pdf")

    # plot Ploss as a function of rho for each K
    f = plt.figure()
    for i in range(len(P_LOSSes)):
        plt.plot(rho_steps, P_LOSSes[i], label=f"K =
{K_steps[i]}")
    plt.legend(loc="upper left")
    plt.title(r'$P_{loss}$ vs $\rho$ as K increases')
    plt.xlabel(r'Traffic Intensity ($\rho$)')
    plt.ylabel(r'$P_{loss}$ (%)')
    plt.show()
    f.savefig("ploss_q6_figure.pdf")
```