Queue Simulation
Project - ECE 610

Filza Mazahir

20295951

fmazahir@uwaterloo.ca

Table of Contents

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 $f(x) = \begin{cases} \mu e^{-\mu x}, & x \geq 0 \\ 0, & x < 0 \end{cases}$

Cumulative Density Function $F(x) = \int_{-\infty}^{x} f(t)dt = \begin{cases} 1 - e^{-\mu x}, & x \geq 0 \\ 0, & x < 0 \end{cases}$

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

$$F(X) = U$$
$$X = F^{-1}(U)$$

$$1 - e^{-\mu X} = U$$
$$e^{-\mu X} = 1 - U$$
$$-\mu = \ln\,(1 - U)$$
$$X = \frac{-1}{\mu}\ln(1 - U)$$

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

$$X_i = F^{-1}(U_i)$$
$$X_i = \frac{-1}{\mu}\ln(1 - U_i)$$

Since U_i and $(1 - U_i)$ are both uniformly distributed random numbers between 0 and 1:

$$U_i = 1 - U_i$$

Therefore:

$$\boldsymbol{X_i = \frac{-1}{\mu}\ln U_i}$$

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.

```
# Generate Exponential Random Value
def generate_exponential_random_value(mu):
    uniform_rv = random.uniform(0, 1) # Python's built-in uniform random generator
    exponential_random_value = -(log(uniform_rv))/mu  # From equation shown in report
    return exponential_random_value
```

To generate 1000 exponential random variables:

```
# Generate 1000 Exponential Random Values to test
def generate_1000_exponential_random_values():
    mu = 0.1    # 1/mu = 10seconds -> mu = 0.1

    # Generate 1000 exponential random values
    exponential_rv_list = []
    for x in range(1000):
        exponential_rv = generate_exponential_random_value(mu)
        exponential_rv_list.append(exponential_rv)

    # Check with mean and variance - convert to numpy array and use np.mean and np.var
    exponential_rv_arr = np.array(exponential_rv_list)
    mean = np.mean(exponential_rv_arr)
    var = np.var(exponential_rv_arr)

    print('Exponential Random Value Generator with 1000 values:')
    print('Mean (exponential) - expected 10, calculated: {0:.1f}'.format(mean))
    f.write('Mean (exponential) - expected 10, calculated: {0:.1f}\n'.format(mean))
    print('Variance (exponential) - expected 100, calculated: {0:.1f}'.format(var))
    f.write('Variance (exponential) - expected 100, calculated: {0:.1f}\n'.format(var))
    return
```

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

```python
class Simulation:
  def __init__(self, arrival_process, service_process, n, K, T, L, C, rho, L1, L2, prob):

    #Input variables
    self.arrival_process = arrival_process  # Arrival process ('M' or 'D')
    self.service_process = service_process  # Service process ('M', 'D' or 'G')
    self.n = n # Number of servers in the queue
    self.K = K  # Size of buffer

    self.T = T  # Total time to run the simulation
    self.L = L # Average packet length in bits
    self.C = C # Transmission rate in bits/second
    self.rho = rho # Utilization factor of the queue

    self.L1 = L1  # For General service process, L1 is the packet length with probability of prob
    self.L2 = L2 # For General service process, L2 is the packet length with probability of (1-prob)
    self.prob = prob  # For General service process, Probability of the packet to have L1 length

    self.lambd = (self.n * self.rho * self.C )/ self.L  # Arrival rate (avg no. of packets per sec)
    self.alpha = self.lambd  # Observer rate - same as arrival rate so they are in the same order

    # State of the system
    self.Na = 0 # Number of packet arrival events
    self.Nd = 0 # Number of packet departure events
    self.No = 0 # Number of observation events
    self.Nt = 0 # Total number of packets in the system

    # Helper variables to keep track of things and determine the output variables
    self.Nt_observer = [] # No. of packets in system as seen by each observer event
    self.Nt_arrival = [] # No. of packets in system as seen by each arrival event
    self.Tsojourn = []  # Total sojourn time for each packet
    self.idle_counter = 0  # Number of times an observation event saw the system as idle
    self.total_packets_generated = 0 # Total number of packets generated in the system
    self.number_packets_dropped = 0  # Number of packets dropped by the queue

    print('Running simulation -> {0}/{1}/{2}/{3} (Rho = {4}, Lambda = {5})....'.format(self.arrival_process, self.service_process, self.n, self.K, self.rho, self.lambd), end='', flush=True)

    # Create Event Scheduler
    self.create_event_scheduler()
```

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 $\rho = {}^{L\lambda}/_{nC}$ 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 $\lambda = {}^{\rho\, nC}/_{L}$. 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()`.

```python
# Function to generate arrival time based on the type of arrival process
def generate_arrival_time(self):
    # Poisson Distribution for arrival process
    if self.arrival_process == 'M':
        arrival_time = generate_exponential_random_value(self.lambd)

    # Deterministic Distribution for arrival process (constant)
    elif self.arrival_process == 'D':
        arrival_time = (1.0/self.lambd)

    return arrival_time
```

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 $1/\lambda$.

```python
# Function to generate packet length based on type of service process
def generate_packet_length(self):

  # Poisson Distribution for service process
  if self.service_process == 'M':
    packet_length = generate_exponential_random_value(1.0/self.L)

  # General Distribution with bipolar length for service process
  elif self.service_process == 'G':
    # Generate uniform random number between 0 and 1
    uniform_random_value = random.uniform(0, 1)

    # Get packet length based on the number generated
    if uniform_random_value <= self.prob:
      packet_length = self.L1
    else:
      packet_length = self.L2

  # Deterministic Distribution for service process (constant length)
  elif self.service_process == 'D':
    packet_length = self.L

  return packet_length
```

The service rate of a process is given as nC/L , 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:

```python
# Function to create Event Scheduler
def create_event_scheduler(self):

  self.observation_times_list = []  # List of all observation times generated
  arrival_times_list = [] # List of all arrival times generated for each packet
  departure_times_list= []  # List of all depature times for each packet
  server_available_time = [0 for i in range(self.n)] # List of available time for all n servers

  # Generate set of random observation observation times with parameter alpha
  observation_time = 0  # Observation time initialization, starts at t = 0
  while (observation_time < T):
    observation_time += generate_exponential_random_value(self.alpha)
    self.observation_times_list.append(observation_time)


  # Generate packets, get packet length and its arrival and depature times
  arrival_time = 0  # Arrival time initialization, starts at t = 0
  while arrival_time < T:

    # Generate new packet - get arrival time and packet length
    arrival_time += self.generate_arrival_time()  # Generate arrival time
    packet_length = self.generate_packet_length() # Generate packet length
    self.total_packets_generated += 1 # Increment number of packets generated

    # Index of server that's available first
    i = server_available_time.index(min(server_available_time))

```

```python
            # Calculate the packet's departure time
            # If server is free when the packet arrived (no wait)
            if server_available_time[i] <= arrival_time:
                departure_time = arrival_time + (packet_length/self.C)

            # If server is not free when the packet arrived (packet has to wait in the queue)
            else:
                departure_time = server_available_time[i] + (packet_length/self.C)

            # Compute the soujorn time of the packet and save it
            sojourn_time = departure_time - arrival_time
            self.Tsojourn.append(sojourn_time)

            # Server will be available when this packet departs
            server_available_time[i] = departure_time

            # Add the arrival event and departure event to Event Scheduler
            arrival_times_list.append(arrival_time)
            departure_times_list.append(departure_time)


        # Combine the list of all observation times, arrival times, and departure times in one list
        events_list = []  # List of all events combined

        # Add all observation events
        for event_time in self.observation_times_list:
            events_list.append(('O', event_time))

        # Add all arrival events
        for event_time in arrival_times_list:
            events_list.append(('A', event_time))

        # Add all depature events
        for event_time in departure_times_list:
            events_list.append(('D', event_time))

        # Sort events list based on time after its completed
        events_list.sort(key=lambda event: event[1])

        # Create Event Scheduler as a double ended queue from the events_list
        # Event Scheduler created after events_list is already sorted with time so its more efficient
        self.Event_Scheduler = deque(events_list)

        return
```

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)

```python
# Function to run the simulation - goes through each event in the Event Scheduler
def run_simulation(self):

    # Loop through the Event Scheduler (continue looping as long as it has an element)
    # Update system metrics based on type of event
    while self.Event_Scheduler:
        # Dequeue the event from the Event Scheduler
        event_type, event_time = self.Event_Scheduler.popleft()

        # Observation Event
        if event_type == 'O':
            self.No += 1  # Increment number of observation events
            self.Nt_observer.append(self.Nt)  # Record system metric

            # If system is idle, then increment the idle_counter
            if self.Nt == 0:
                self.idle_counter += 1

        # Arrival Event
        elif event_type == 'A':
            self.Nt_arrival.append(self.Nt) # Record system metric (excluding this packet)
            self.Na += 1  # Increment number of arrival events
            self.Nt = self.Na - self.Nd  # Update current number of packets in the system

        # Departure Event
        elif event_type == 'D':
            self.Nd += 1  # Increment number of departure events
            self.Nt = self.Na - self.Nd  # Update current number of packets in the system

    # Calculate output variables after the simulation is run and complete
    self.EN = sum(self.Nt_observer)/len(self.Nt_observer) # avg. no. of packets seen by obsever
    self.EaN = sum(self.Nt_arrival)/len(self.Nt_arrival) # aavg. no. of packets seen by arrival packet
    self.ET = sum(self.Tsojourn)/len(self.Tsojourn) # average of the sojourn time of all packets
    self.Pidle = self.idle_counter/self.No  # Proportion of time observer saw system idle
    self.Ploss = self.number_packets_dropped/self.total_packets_generated  # Packet lost probability

    print('done')
    return
```

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 $E_a[N]$ (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. P_{IDLE} 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. P_{LOSS} 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

```
MM1 = Simulation('M', 'M', 1, inf, T, L, C, rho, L1, L2, prob)
MM1.run_simulation()
```

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.

```
# E[N] - Average No. of packets in the system as seen by observer
MM1.EN

# Ea[N] - Average No. of packets in the system as seen by arrival packet
MM1.EaN

# E[T] - Average sojourn time taken by all packets in the system
MM1.ET

# Proportion of time n servers are idle
MM1.Pidle

# Packet loss probability
MM1.Ploss
```

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] & P_{IDLE} ($0.35 \leq \rho \leq 0.95$)

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 P_{IDLE} (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

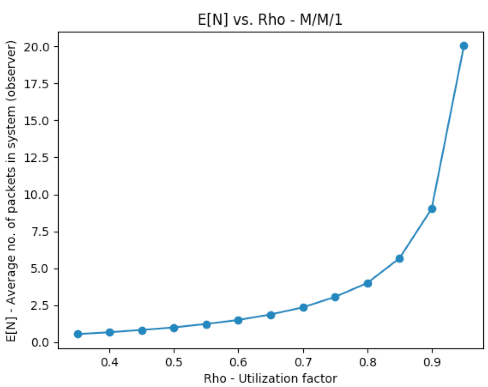


Fig 1: E[N] vs. Rho (M/M/1)

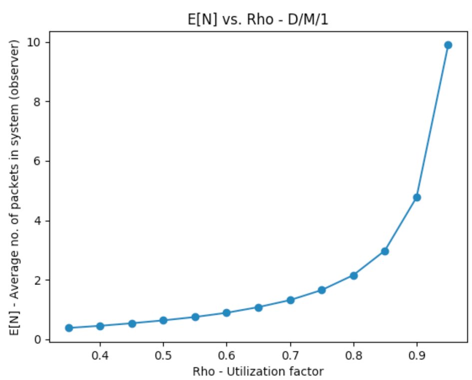


Fig 2: E[N] vs. Rho (D/M/1)

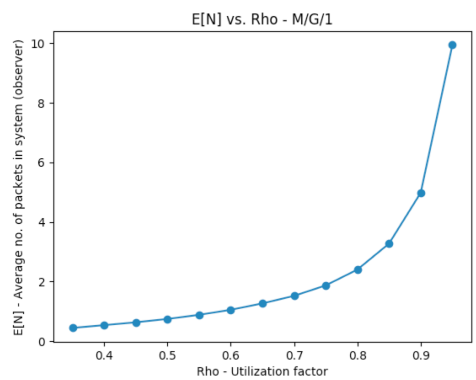


Fig 3: E[N] vs. Rho (M/G/1)

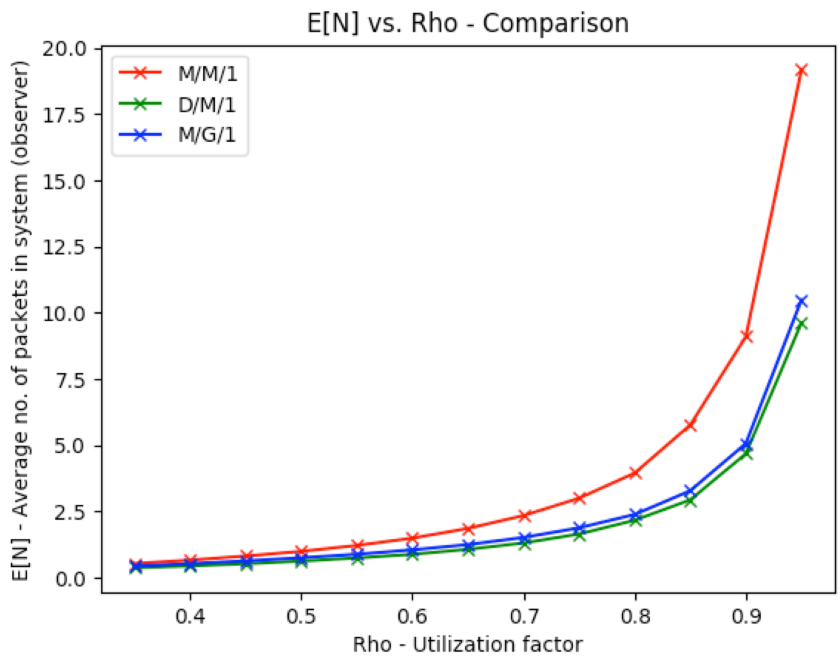


Fig 4: E[N] vs. Rho for M/M/1, D/M/1, and M/G/1

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_____

P_{IDLE} as a function of ρ

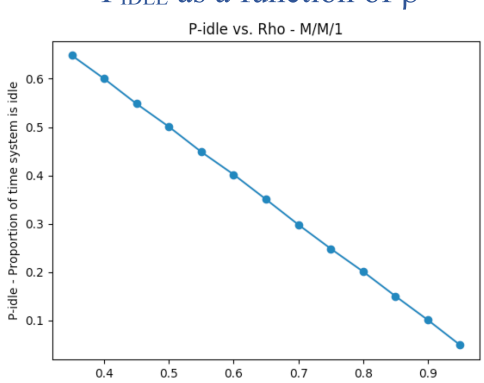


Fig 5: P_{IDLE} vs. Rho (M/M/1)

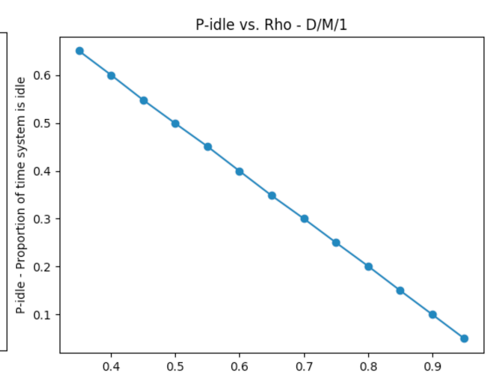


Fig 6: P_{IDLE} vs. Rho (D/M/1)

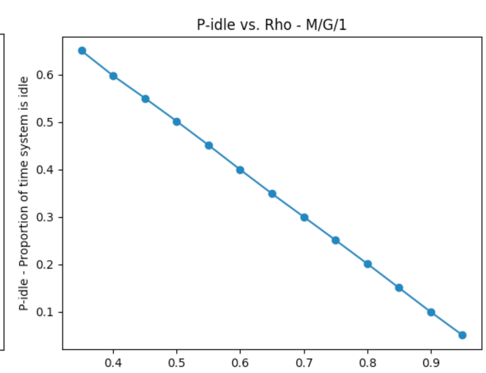


Fig 7: P_{IDLE} vs. Rho (M/G/1)

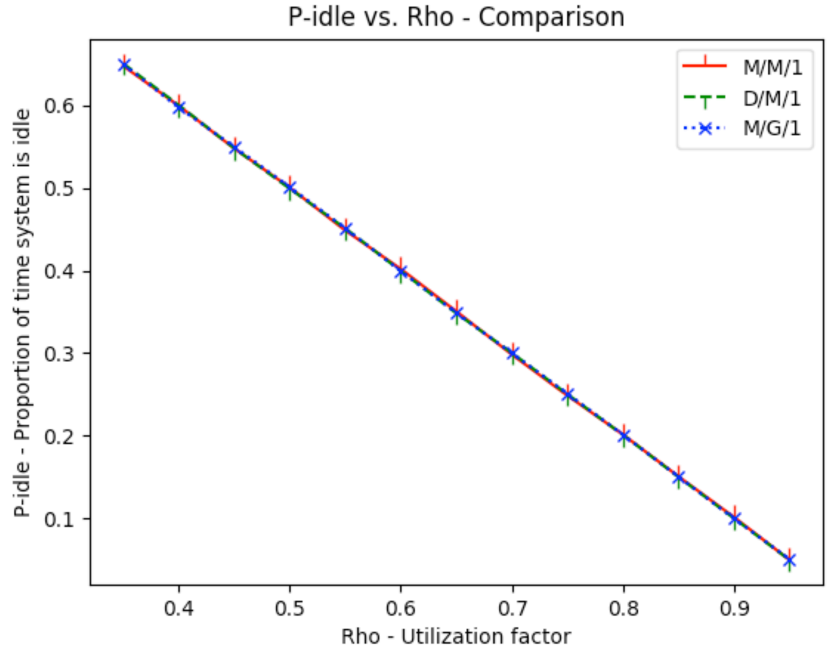


Fig 8: P_{IDLE} vs. Rho for M/M/1, D/M/1, and M/G/1

Figures 5, 6, and 7 show P_{IDLE} 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 P_{IDLE} and ρ, as ρ increases, P_{IDLE} 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 P_{IDLE} 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] & P_{IDLE} (ρ = 1.5)

For this question, the queue M/M/1 was simulated at ρ = 1.5, and E[N] and P_{IDLE} 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. P_{IDLE} 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 P_{IDLE} 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 P_{IDLE} of M/M/1 with other queues, D/M/1, and M/G/1 were also simulated at $\rho = 1.5$. It was found that the average number of packets in the system at $\rho = 1.5$ were very similar for all three queues, and were in the same order. All three of these queues had the expected P_{IDLE} 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
```

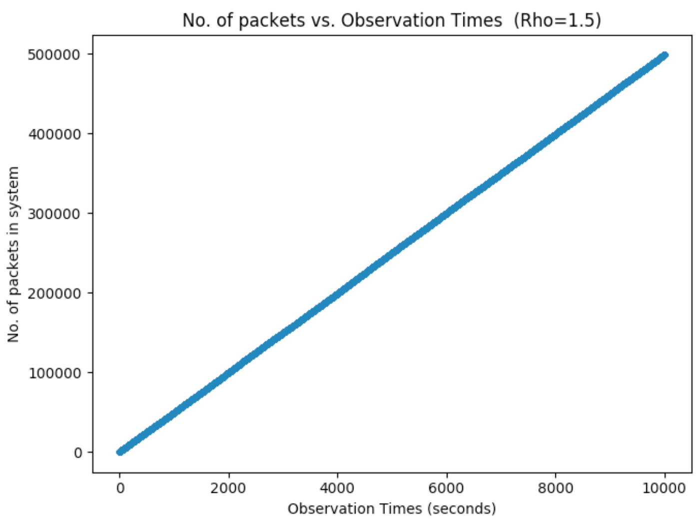


Fig 9: No. of packets vs. Observation times (Rho=1.5)

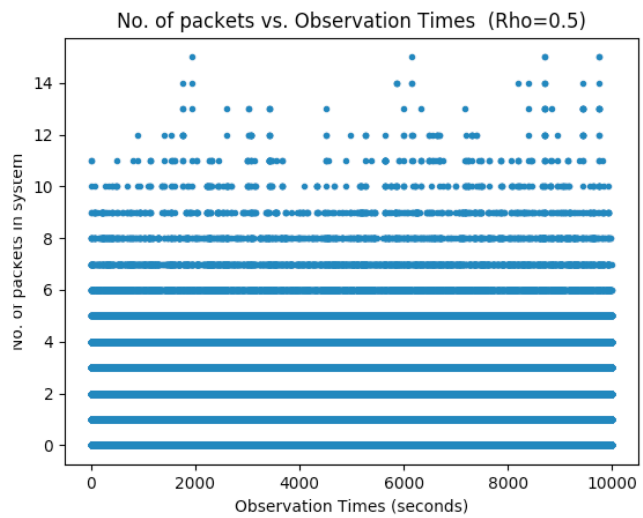
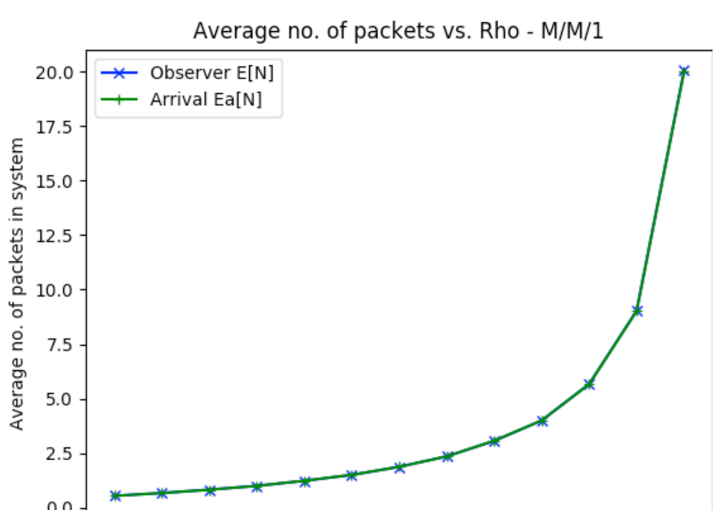


Fig 10: No. of packets vs. Observation times (Rho=0.5)

The queue M/M/1 was also simulated at $\rho = 0.5$, 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 $\rho = 1.5$, number of packets in the system increase linearly with time, whereas at $\rho = 0.5$, 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 E_a[N]



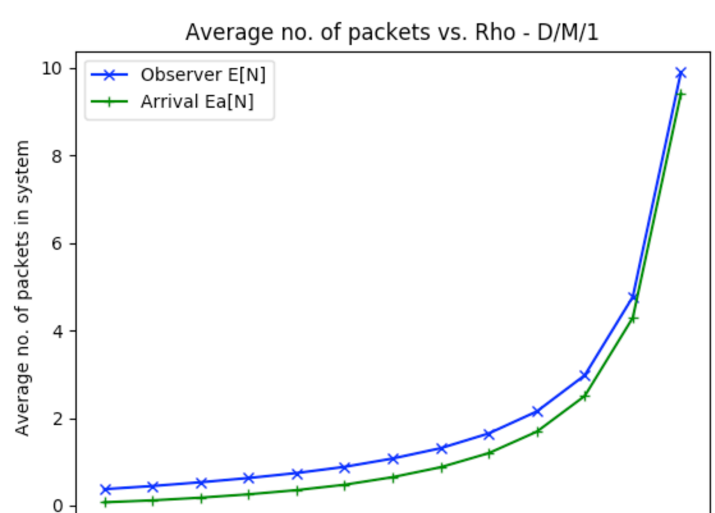


Fig 11: Comparison of E[N] and E_a[N] (M/M/1) **Fig 12:** Comparison of E[N] and E_a[N] (D/M/1)

Figure 11 and 12 show the comparison of E[N] and E_a[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]

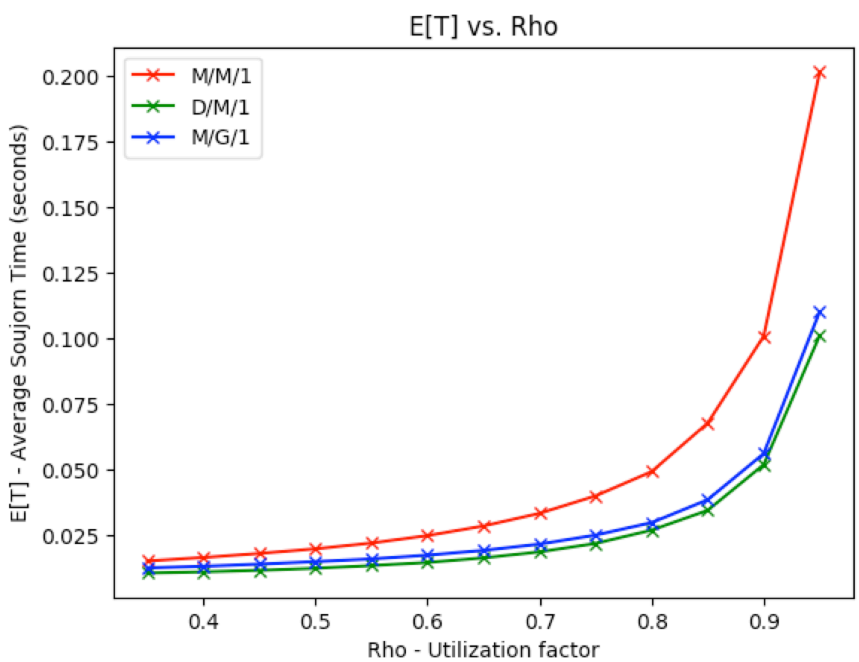


Fig 13: E[T] vs. Rho for M/M/1, D/M/1, and M/G/1

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:

```
self.total_packets_generated = 0 # Total number of packets generated in the system
self.number_packets_dropped = 0  # Number of packets dropped by the queue
```

Code Changed in `create_event_scheduler()` *function:*

```
171 # Generate packets, get packet length and its arrival and depature times
172 arrival_time = 0   # Arrival time initialization, starts at t = 0
173 while arrival_time < T:
174
175   # Generate new packet - get arrival time and packet length
176   arrival_time += self.generate_arrival_time()  # Generate arrival time
177   packet_length = self.generate_packet_length() # Generate packet length
178   self.total_packets_generated += 1 # Increment number of packets generated
179
180   # Index of server that's available first
181   i = server_available_time.index(min(server_available_time))
182
183   # Functionality added for M/D/1/K queue - check if buffer is full
184   # If not full - get departure time and sojourn time of this packet, and add to list
185   # Length of depature_times_list will always be less than K when K=inf
186   if len(departure_times_list) < self.K or departure_times_list[-self.K] < arrival_time:
187
188     # Calculate the packet's departure time
189     # If server is free when the packet arrived (no wait)
190     if server_available_time[i] <= arrival_time:
191       departure_time = arrival_time + (packet_length/self.C)
192
193     # If server is not free when the packet arrived (packet has to wait in the queue)
194     else:
195       departure_time = server_available_time[i] + (packet_length/self.C)
196
197     # Compute the soujorn time of the packet and save it
198     sojourn_time = departure_time - arrival_time
199     self.Tsojourn.append(sojourn_time)
200
201     # Server will be available when this packet departs
202     server_available_time[i] = departure_time
203
204     # Add the arrival event and departure event to Event Scheduler
205     arrival_times_list.append(arrival_time)
206     departure_times_list.append(departure_time)
207
208   # If buffer is full - drop packet (don't add the arrival and depature time to list)
209   else:
210     self.number_packets_dropped += 1  # Increment number of dropped packets
```

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 P_{LOSS} 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– P_{LOSS} ($0.4 \leq \rho \leq 3$)

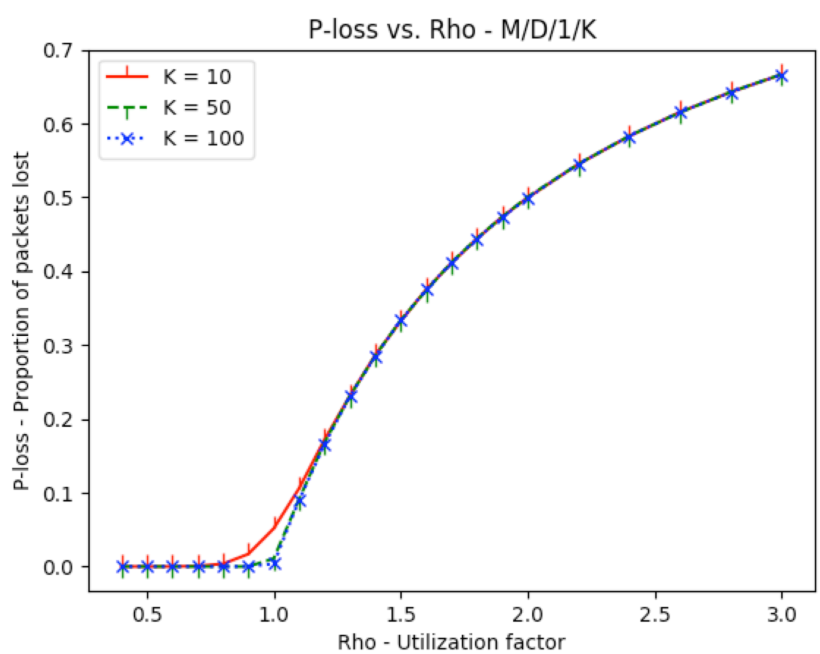


Fig 14: P_{LOSS} vs. Rho for M/D/1/K when K=10, 50, and 100

Figure 14 shows the relationship between P_{LOSS} and ρ for M/D/1/K. As seen, P_{LOSS} 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 P_{LOSS}. For the queue with K=10, P_{LOSS} increases more rapidly than the queue with K=50 and 100. However, after ρ=1.2, all three of these queues had the same P_{LOSS}, and had a logarithmic relationship between P_{LOSS} and ρ. This is because____

Question 9 – M/D/2 and M/D/1 (rate 2C) – E[N] ($0.35 \leq \rho \leq 0.95$)

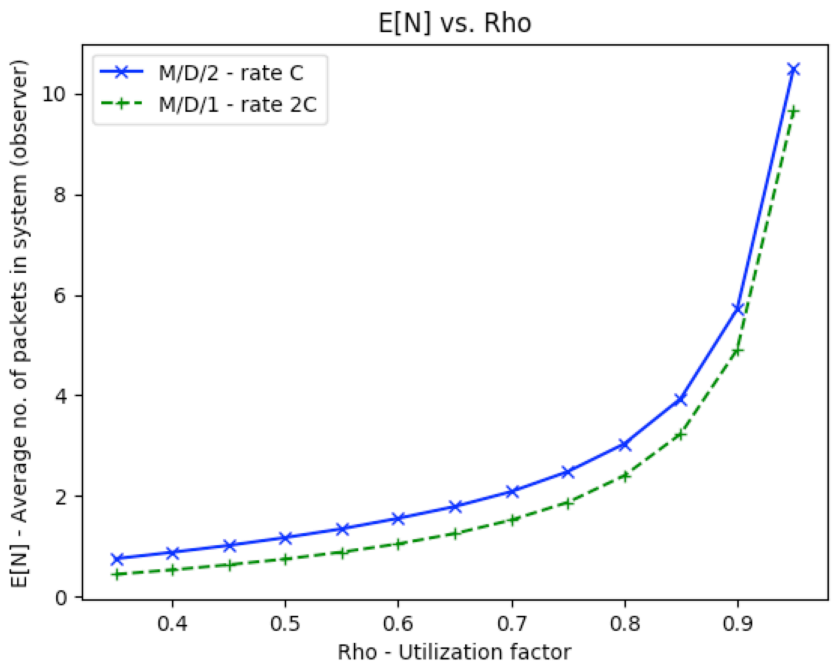


Fig 15: E[N] vs. Rho for M/D/2 and M/D/1

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.