UNIVERSITY OF WATERLOO

ECE 358: Computer Networks

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

Jiahao Chen (j2435che@uwateerloo.ca)

Gonzalo Carretero (gcarrete@uwaterloo.ca)

Lab Group: 103
Date: October 6 2023

Table of Contents

All the code done for this lab can be found in the appendix of this document, with comments defining each variable and function.
We have created a Makefile in order to help execute the code for each of the questions. Start by typing ‘make’ and the output will explain how to execute each part.
Additionally, there is also a Readme file for further clarification.

1. Question 1

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

Python

```
import random
import math


def generate_exponential_random_variables(lmbda, num_variables):
    exponential_random_variables = []

    # Generate num_variables exponential random variables
    for _ in range(num_variables):
        # Generate a uniform random variable between 0 and 1
        uniform_random_variable = random.uniform(0, 1)

            # Calculate the exponential random variable using the inverse
transform method
              exponential_random_variable = -(1 / lmbda) * math.log(1 -
uniform_random_variable)

        # Add the exponential random variable to the list
        exponential_random_variables.append(exponential_random_variable)

    return exponential_random_variables


# Generate 1000 exponential random variables with λ=75
random_variables = generate_exponential_random_variables(75, 1000)

# Calculate the mean of the generated random variables
mean = sum(random_variables) / len(random_variables)

# Calculate the variance of the generated random variables using the
unbiased formula
```

```
variance = sum((x - mean) ** 2 for x in random_variables) /
(len(random_variables) - 1)

# Print the calculated mean and variance
print(f"Mean: {mean}")
print(f"Variance: {variance}")
```

Output:
-> Mean: 0.013233279027044883
-> Variance: 0.00017540722175922105

These values are expected since:

Mean = 1/75 = 0.01333333333
Variance = 1/75^2 = 0.00017777777

2. Question 2

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

The code specific for this question, that is, for the simulator of a M/M/1, can be found in **Appendix A.2**.

Here's an explanation of what the code for our M/M/1 simulator:

1. It initializes various parameters, including simulation time, arrival rate, packet length, transmission rate, and buffer size (K) (inf in this case).

2. The code defines a function `generate_exponential(lambda_param)` to generate random interarrival times and service times, which follow an exponential distribution (same as question 1).

3. It then generates packet arrivals based on exponential interarrival times until the simulation time (T) is reached. For each arrival, it calculates the packet's arrival time and adds it to a list of arrival events.

4. The code creates packet departures. For each arrival event, it generates a departure event based on the service time, which is determined by the packet length and transmission rate. Departure events are added to a list.

5. Observer events are generated at a rate of 5 times the arrival rate. These events represent moments when observations are made in the queuing system.

6. All events (arrival, departure, and observer events) are sorted by their timestamps.

7. The code then goes through the sorted events to calculate various performance metrics depending on the type of event, including:
- Na (Number of Arrivals) - when there is an arrival event
- Nd (Number of Departures) - when there is a departure event
- No (Number of Observers) - when there is an observer event
- Total idle times observed - when observer arrives and Na equals No
- Sum of observed queue lengths - every time a observer arrives it records the difference between Na and Nd

8. Finally, it calculates and prints two performance metrics:
- Average Queue Length: The average queue length observed during the simulation = Sum of observed queue lengths / No
- Idle Ratio: The ratio of time the server is idle to the total simulation time = Total idle times observed / No

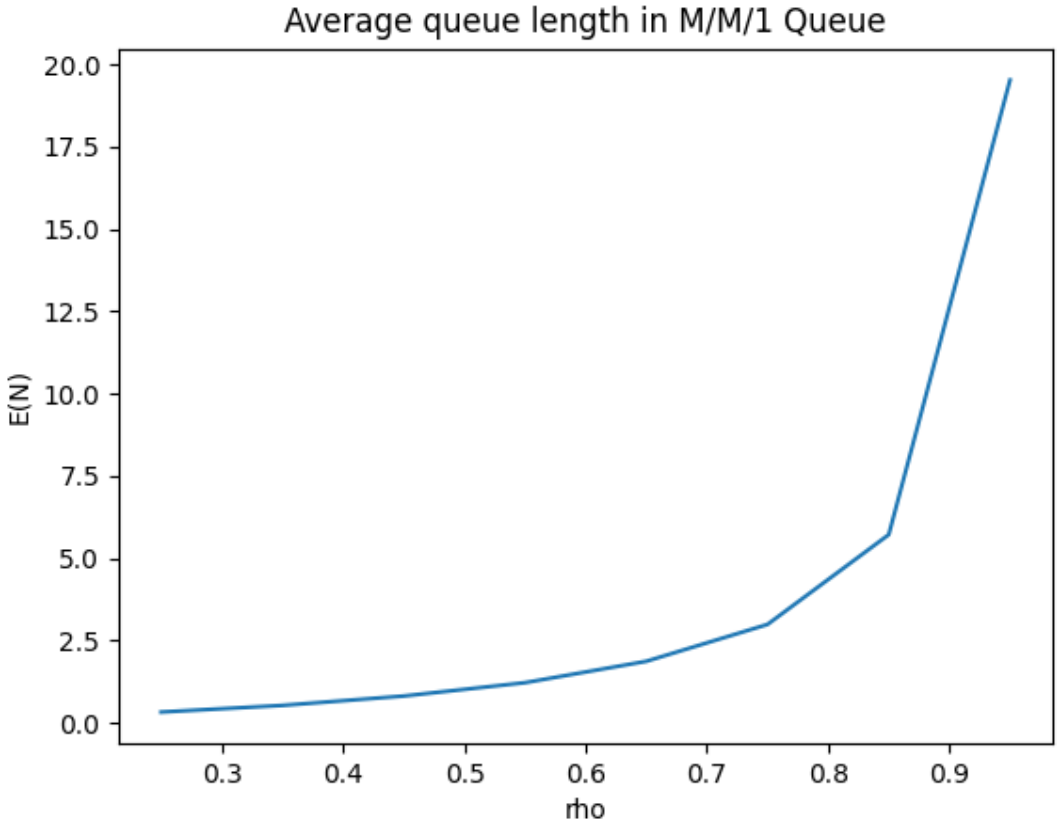
The code essentially models the behaviour of a queuing system, tracks arrivals, departures, and observer events, and computes performance metrics related to the system's behaviour. It is designed to analyze and understand the queuing behaviour in a network or system where packets arrive, are processed, and potentially wait in a queue when the server is busy. The code allows for experimentation with different parameters such as arrival rates, packet lengths, and buffer sizes to study the impact on system performance.

3. Question 3

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

3.1. Graph of E[N] as a function of ρ

E[N], the average number of packets in the queue as a function of (rhos 0.25 < 0.95, step size 0.1). Explain how you do that.



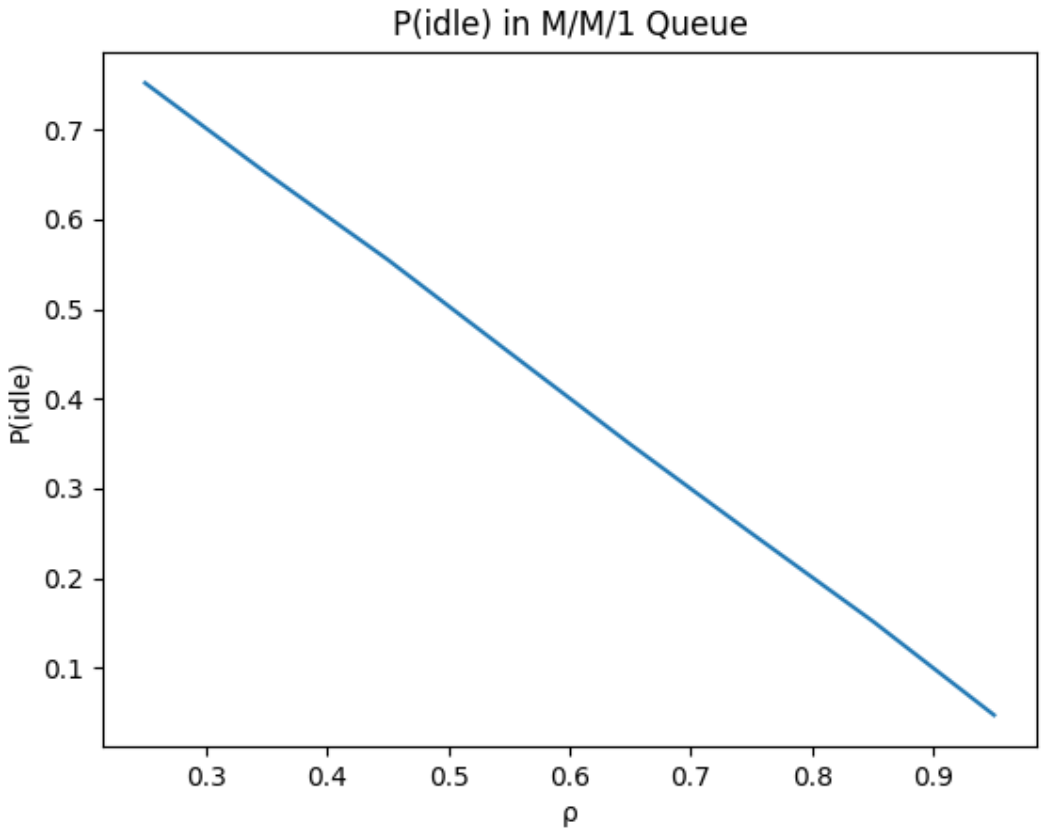
It can be seen in the graph that as the utilization of the queue increases (ρ), the average number of packets in the queue does so accordingly in an exponential way. This makes sense because a higher utilization means a higher arrival rate, which means that there will be more packets waiting in the queue to be served from the very beginning, and basically more packets in the current queue will also imply that future arrival packets will have to wait longer in the queue, making the average length of the queue increasing continuously.

The approach used to generate the graph is using Python library matplotlib.pyplot. We iterate over the values of rho and obtain the value of arrival rate from the equation *arrival_rate = rho * transmission_rate / average_packet_length, and save the E(N) calculated running the simulation in a list. Finally, we call the library function to plot the graph.*

The code to generate this graph can be seen in Appendix A.3.1.

3.2. Graph of P_{IDLE} as a function of ρ

PIDLE, the proportion of time the system is idle as a function of, (for 0.25 < 0.95, step size 0.1). Explain how you do that.



As expected, as the utilization of the queue increases (ρ) the probability that the queue is idle decreases accordingly. Utilization is the ratio of time that the queue is populated by at least one packet, while p_idle is the rest of the time when the queue is empty. They are inversely proportional.

The approach used is the same as for E(N).

The code to generate this graph can be seen in **Appendix A.3.2**.

We are going to check that the simulation is stable for every ρ in an MM1 with values of T = 1000 and T = 2000:

```
C:\Users\JIAHAO\Documents\GitHub\ECE358\venv\Scripts\python.exe C:\Users\JIAHAO\Documents\GitHub\ECE358\Stability_Check.py
Which simulator do you want to check the stability for (Enter MM1 or MM1K): MM1
Select the first value of T to compare with (for example T = 1000): 1000
Select the second value of T to compare with (for example T = 2000): 2000
Difference between simulations for 0.25 is E[n]: 1.544%, P(idle): 0.337%
Difference between simulations for 0.35 is E[n]: 0.48%, P(idle): 0.169%
Difference between simulations for 0.45 is E[n]: 0.446%, P(idle): 0.003%
Difference between simulations for 0.55 is E[n]: 0.273%, P(idle): 0.114%
Difference between simulations for 0.65 is E[n]: 2.315%, P(idle): 1.123%
Difference between simulations for 0.75 is E[n]: 0.315%, P(idle): 0.436%
Difference between simulations for 0.85 is E[n]: 2.97%, P(idle): 1.256%
Difference between simulations for 0.95 is E[n]: 4.428%, P(idle): 1.262%

Process finished with exit code 0
```

This output has been generated by a file we have created to check the stability of simulations, both for MM1 and MM1K, with any K given and any two T values to compare. The code can be seen in **Appendix A.7**.

The formulas that have been used to calculate the difference between values of both simulations for very ρ are:

$$\Delta E[n]\ \% = (abs(E[n]_{T=1000} - E[n]_{T=2000}) / E[n]_{T=1000}) * 100$$

$$\Delta P_{idle}\ \% = (abs(Pidle_{T=1000} - Pidle_{T=2000}) / Pidle_{T=1000}) * 100$$

As can be seen in the photo, all the differences differ in values smaller than 5%, so T = 1000 is a good enough amount of time to measure our statistics.

4. Question 4

For the same parameters, simulate for ρ = 1.2. What do you observe? Explain.

Result: (50147.46678475495, 4.997153288343407e-06)

We see that the queue is not idle ratio is really low and the average number of packets is really large and not stable (it continually increases as we use bigger T). This happens because the number of rho is bigger than 1, which means that packets are arriving faster than the speed that our server can handle, creating buffer overflows if we were using a finite queue. So the packets just keep accumulating and there are more and more packets waiting as we run longer.

```
C:\Users\JIAHAO\Documents\GitHub\ECE358\venv\Scripts\python.exe C:\Users\JIAHAO\Documents\GitHub\ECE358\Stability_Check.py
Which simulator do you want to check the stability for (Enter MM1 or MM1K): MM1
Select the first value of T to compare with (for example T = 1000): 1000
Select the second value of T to compare with (for example T = 2000): 2000
Difference between simulations for 0.25 is E[n]: 0.238%, P(idle): 0.071%
Difference between simulations for 0.35 is E[n]: 0.211%, P(idle): 0.062%
Difference between simulations for 0.45 is E[n]: 0.554%, P(idle): 0.094%
Difference between simulations for 0.55 is E[n]: 0.06%, P(idle): 0.136%
Difference between simulations for 0.65 is E[n]: 0.651%, P(idle): 0.027%
Difference between simulations for 0.75 is E[n]: 2.175%, P(idle): 0.229%
Difference between simulations for 0.85 is E[n]: 0.446%, P(idle): 0.011%
Difference between simulations for 0.95 is E[n]: 2.189%, P(idle): 0.261%

Process finished with exit code 0
```

This is the stability check for the MM1 simulator. We see that both E(n) and P(idle) are stable (<5%) for all the values asked of rho.

5. Question 5

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

The code specific for this question, that is, for the simulator of a M/M/1/K, can be found in **Appendix A.5**.
The design for the M/M/1/K queue has several similarities to the design of the M/M/1 queue. Therefore, we are going to explain the main differences that can be found in this new design:

1. K is now an integer number, defining the size of the queue.
2. Our events list is now a min-heap. This means that whenever we add an event to it, it will be sorted directly (in ascending order) according to its time value.
3. All arrivals are not inserted into their own list, just into the events heap.
4. Now when we start the simulation, only arrival and observer events are in the heap. During the simulation, when we extract an arrival event, we check if it fits into the queue. That is, the number of arrivals minus the number of departures is smaller than K. Then the queue has space and we generate a departure event corresponding to that arrival and insert it in the min heap. If the queue is full, a departure event is not generated, and a counter for dropped packages (dropped_counter) is increased.
5. The rest of the metrics are computed similarly to the M/M/K queue, except for the **P_{Loss}. This parameter is computed as the dropped counter divided by all the packets generated (Na).**

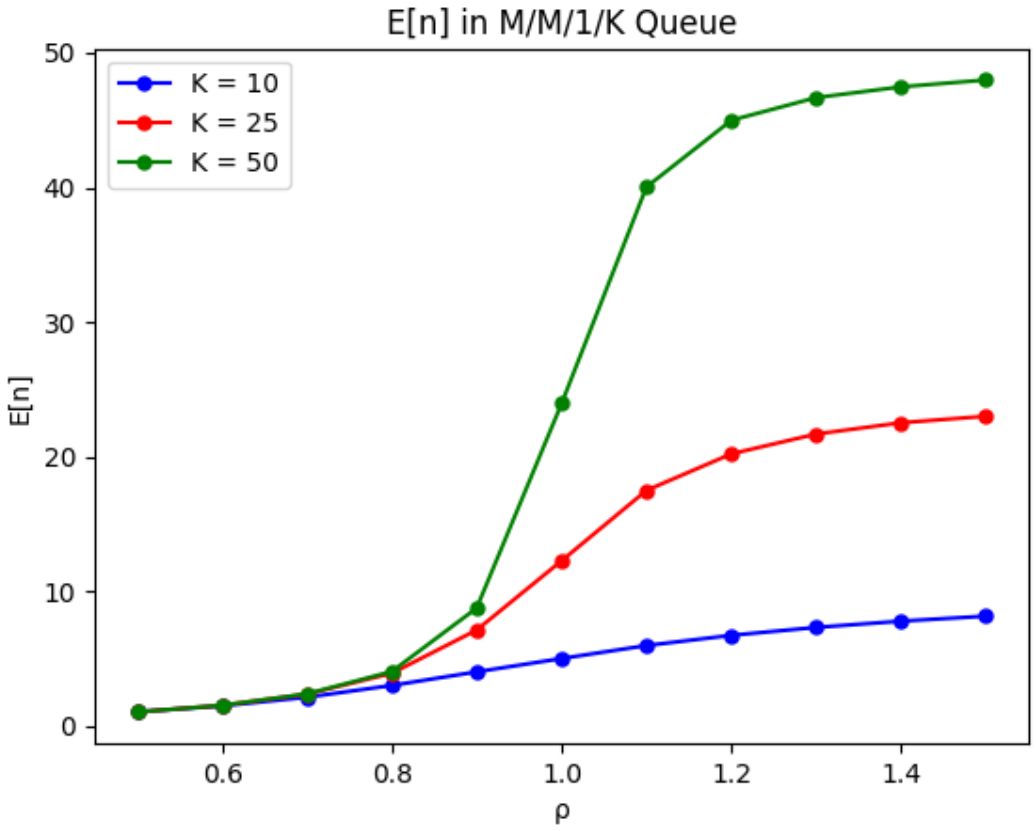
   In conclusion, the main characteristics of this new queue are that we need to be careful in deciding when it is full and generate departure events only if it has space. This is also important to compute the probability of packets being dropped in that simulation.

6. Question 6

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

6.1. Graph of E[N] as a function of ρ

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.

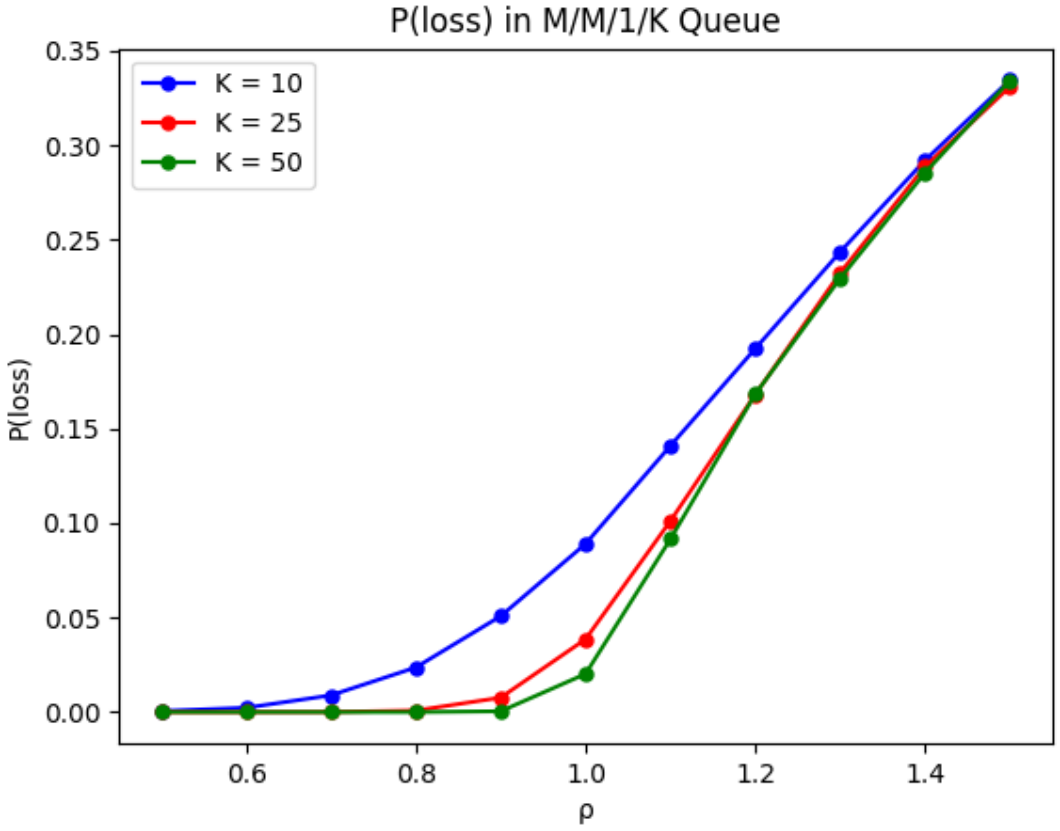


As the utilization of the queue increases (ρ), it is expected that the average number of packets in the queue (E[n]) increases. The correlation can be seen in the graph for all three different values of K. However, an interesting behaviour occurs. All of the lines tend to stabilize for larger values of ρ. But for smaller values of ρ, the larger the queue, a higher the difference in E[n] can be seen. This is because, as ρ increases, small queues will begin to drop more packages, while larger queues will still be able to allocate them, increasing the overall average number of packets in the queue.

The code to generate this graph can be seen in **Appendix A.6.1**.

6.2. Graph of P_{LOSS} as a function of ρ

P_{LOSS} as a function of ρ (for 0.5 < 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 PLOSS.



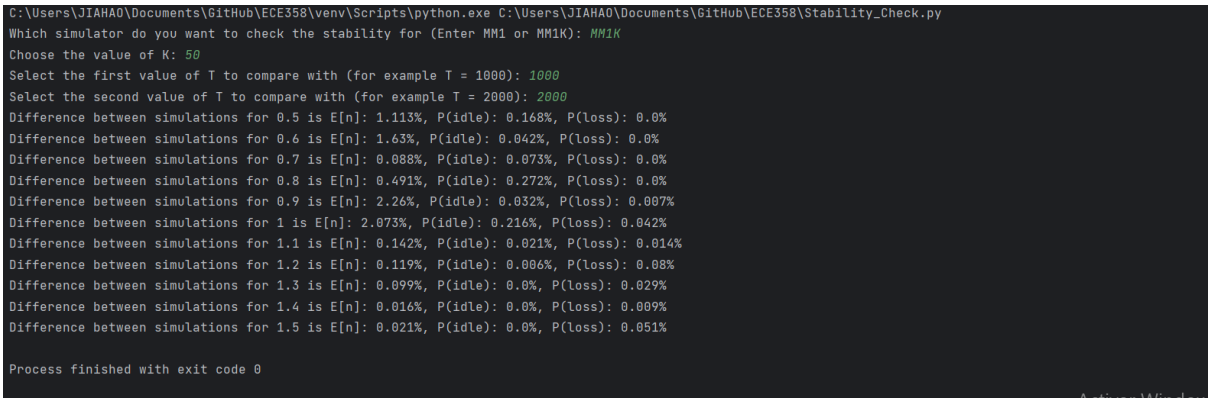
For this graph, T = 1000 was used.

As the utilization of the queue increases (ρ), it is normal that the probability of having packets dropped increases. We can see this correlation well represented in the graph above. It is important to point out that the smaller the size of the queue (K), the more probable it is that there will be packets dropped with a low value of ρ. However, for very large values of ρ, independently of the size of the queue (10, 25 or 50), they all tend to have the same probability of having packet losses, since the extra space that the queue provides is no longer relevant compared to the high arrival rate.

The process to obtain the probability of packet loss (P_{Loss}) is the following:
We have a counter (dropped_counter) initialized at 0, to count the number of dropped packets during the simulation. We also store the total number of arrival packets generated.
When the simulation starts, each time we receive an arrival event, we check if the queue is full or not (if full then abs(#arrivals - #departures - dropped_counter) >= K). If it is full, then we increase the dropped_counter by 1.
At the end of the simulation, we compute P_{Loss} by dividing the dropped_counter by Na (Number of arrivals packets)

The code to generate this graph can be seen in **Appendix A.6.2**.

> We are going to check that the simulation is stable for every ρ in an MM1K with K = 50 and values of T = 1000 and T = 2000:



This output has been generated by a file we have created to check the stability of simulations, both for MM1 and MM1K, with any K given and any two T values to compare. The code can be seen in **Appendix A.7**.

The formulas that have been used to calculate the difference between values of both simulations for very ρ are:

Δ E[n] % = (abs($E[n]_{T=1000}$ - $E[n]_{T=2000}$) / $E[n]_{T=1000}$) * 100

Δ P_{idle} % = (abs($Pidle_{T=1000}$ - $Pidle_{T=2000}$) / $Pidle_{T=1000}$) * 100

Δ P_{Loss} % = (abs($Ploss_{T=1000}$ - $Ploss_{T=2000}$) / $Ploss_{T=1000}$) * 100

As can be seen in the photo, all the differences differ in values smaller than 5%, so T = 1000 is a good enough amount of time to measure our statistics.

7. Appendix

A.1. Code for Question 1

Python

```python
import random
import math


def generate_exponential_random_variables(lmbda, num_variables):
    exponential_random_variables = []

    # Generate num_variables exponential random variables
    for _ in range(num_variables):
        # Generate a uniform random variable between 0 and 1
        uniform_random_variable = random.uniform(0, 1)

            # Calculate the exponential random variable using the inverse transform method
            exponential_random_variable = -(1 / lmbda) * math.log(1 - uniform_random_variable)

        # Add the exponential random variable to the list
        exponential_random_variables.append(exponential_random_variable)

    return exponential_random_variables


# Generate 1000 exponential random variables with λ=75
random_variables = generate_exponential_random_variables(75, 1000)

# Calculate the mean of the generated random variables
mean = sum(random_variables) / len(random_variables)

# Calculate the variance of the generated random variables using the unbiased formula
variance = sum((x - mean) ** 2 for x in random_variables) / (len(random_variables) - 1)

# Print the calculated mean and variance
print(f"Mean: {mean}")
print(f"Variance: {variance}")
```

A.2. Code for Question 2

Python

```
import random
import math
import queue

class MM1QueueSimulator:
    def __init__(self, T, arrival_rate, average_package_length,
transmission_rate, K=None):
        # Simulation time
        self.T = T
        self.arrival_rate = arrival_rate
        self.average_package_length = average_package_length
        self.transmission_rate = transmission_rate
        # Queue size
        self.K = K
        # Track time when generating packets
        self.clock = 0
        self.arrival_events = queue.Queue()
        # Queue
        self.events = []

    def generate_exponential(self, lambda_param):
        u = random.random()
        return -math.log(1 - u) / lambda_param

    def generate_packet_arrivals(self):
        """Function to generate all arrival events"""
        # Generate arrival events under simulation time
        while self.clock < self.T:
            interarrival_time = self.generate_exponential(self.arrival_rate)
            # Arrival time is prev arrival time + interarrival time
            self.clock += interarrival_time
            # Create arrival event
            packet = {
                'type': "arrival",
                'time': self.clock,
            }
             # Put arrival event at the end of the queue
            self.events.append(packet)
            self.arrival_events.put(packet)

    def generate_departures(self):
        """Function to generate all departure events"""
        self.clock = 0
        # One deparrture event for each arrival event
        while not self.arrival_events.empty():
```

```
            # Get an arrival event
            curr_event = self.arrival_events.get()

            if self.clock < curr_event["time"]:
                # The packet doesn't need to wait to be processed
                self.clock = curr_event["time"]

            package_length = self.generate_exponential(1 /
self.average_package_length)
            service_time = package_length / self.transmission_rate
             # Update the clock until the time current packet finishes being
served
            self.clock += service_time


            # Create departure event
            packet = {
                "type": "departure",
                'time': self.clock
            }
            # Put departure event at the end of the queue
            self.events.append(packet)



    def generate_observers(self):
        """Function to generate all observer events"""
        self.clock = 0
        # Observer average rate is five times grater than arrival's
        observer_average = 5 * self.arrival_rate
        # Generate observer events under simulation time
        while self.clock < self.T:
            observer_time = self.generate_exponential(observer_average)
            # Observer time is prev observer time + new difference computed
            self.clock += observer_time
            # Create observer event and put it at the end of queue
            self.events.append({
                "type": "observer",
                'time': self.clock
            })

    def run_simulation(self):
        """Run the simulation"""
        # Generate all events
        self.generate_packet_arrivals()
        self.generate_departures()
        self.generate_observers()
         # Sort events by time
```

```
        self.events.sort(key = lambda x: x["time"])

        Na = 0  # Number of Arrivals
        Nd = 0  # Number of departures
        No = 0  # Number of observers

        total_idle_count = 0
        sum_queue_length_observed = 0
        # Loop through all events in queue
        for event in self.events:
            if event["type"] == "arrival":
                # New arrrival
                Na += 1
            elif event["type"] == "departure":
                # New departure
                Nd += 1
            else:
                """Observer event"""
                if Na == Nd:
                    # Queue is empty
                    total_idle_count += 1
                # Add packets observed in the queue to counter
                sum_queue_length_observed += abs(Na - Nd)
                No += 1
        # Compute statistics
        average_queue_length = sum_queue_length_observed / No
        idle_ratio = total_idle_count / No

        return average_queue_length, idle_ratio

if __name__ == "__main__":
    # Usage example
    T = 1000
    average_package_length = 2000
    # average_package_length = 2000
    transmission_rate = 1000000
    arrival_rate = 1.2 * transmission_rate / average_package_length
    # Infinite queue
    K = None

    simulator = MM1QueueSimulator(T, arrival_rate, average_package_length,
transmission_rate, K)
    print(simulator.run_simulation())
```

A.3.1 Code for Question 3.1

Python

```
from MM1 import MM1QueueSimulator

# Simulate MM1 Queue for different values of rho
rhos = [0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85, 0.95]
average_packet_counts = []

# Simulation time
T = 1000
average_packet_length = 2000
transmission_rate = 1000000
# Infinite queue
K = None

print("MM1 Plot E[N] vs. rho")

for rho in rhos:
    # Compute arrival rate for given rho
    arrival_rate = rho * transmission_rate / average_packet_length
    simulator = MM1QueueSimulator(T, arrival_rate, average_packet_length,
transmission_rate, K)
    # Get E[n] from current simulator
    e_n = simulator.run_simulation()[0]
    average_packet_counts.append(e_n)
    print("rho = " + str(rho) + ", E[N] = " + str(e_n))

if __name__ == "__main__":
    import matplotlib.pyplot as plt
    # Plot E[N] vs. rho
    plt.plot(rhos, average_packet_counts)
    plt.xlabel('rho')
    plt.ylabel('E(N)')
    plt.title('Average queue length in M/M/1 Queue')
    plt.show()
```

A.3.2 Code for Question 3.2

```python
from MM1 import MM1QueueSimulator

# Simulate M/M/1 Queue for different values of rho
rhos = [0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85, 0.95]
average_packet_counts = []

# Simulation time
T = 1000
arrival_rate = 200.0
transmission_rate = 1000000
average_packet_length = 2000
# Infinite queue
K = None

print("MM1 Plot P(idle) vs. rho")

for rho in rhos:
    # Compute arrival rate for given rho
    arrival_rate = rho * transmission_rate / average_packet_length
    simulator = MM1QueueSimulator(T, arrival_rate, average_packet_length,
transmission_rate, K)
    # Get Pidle from current simulator
    p_idle = simulator.run_simulation()[1]
    average_packet_counts.append(p_idle)
    print(f"rho = {rho}, P(idle) = {p_idle}")

if __name__ == "__main__":
    # Plot P(idle) vs. rho
    import matplotlib.pyplot as plt
    plt.plot(rhos, average_packet_counts)
    plt.xlabel('rho')
    plt.ylabel('P(idle)')
    plt.title('P(idle) in M/M/1 Queue')
    plt.show()
```

A.5. Code for Question 5

```python
import heapq
import random
import math
```

```
class MM1KQueueSimulator:
    def __init__(self, T, arrival_rate, average_package_length,
transmission_rate, K=None):
        # Simulation time
        self.T = T
        self.arrival_rate = arrival_rate
        self.average_package_length = average_package_length
        self.transmission_rate = transmission_rate
        # Queue size
        self.K = K
        # Track time when generating packets
        self.clock = 0
        # Queue
        self.events = []

    def generate_exponential(self, lambda_param):
        u = random.random()
        return -math.log(1 - u) / lambda_param

    def generate_packet_arrivals(self):
        """Function to generate all arrival events"""
        # Generate arrival events under simulation time
        while self.clock < self.T:
            interarrival_time = self.generate_exponential(self.arrival_rate)
             # Arrival time is prev arrival time + interarrival time
            self.clock += interarrival_time
             # Generate arrival event
            packet = (self.clock, "arrival")
            # Put event in the "queue" (it is a heap so it is inserted in the
position sorted by time)
            heapq.heappush(self.events, packet)

    def generate_service_time(self):
        """Auxiliary function for departures during simulation"""
        package_length = self.generate_exponential(1 /
self.average_package_length)
        service_time = package_length / self.transmission_rate
        return service_time

    def generate_observers(self):
        """Function to generate all observer events"""
        self.clock = 0
        # Observer average rate is five times grater than arrival's
        observer_average = 5 * self.arrival_rate
        # Generate observer events under simulation time
        while self.clock < self.T:
```

```
            observer_time = self.generate_exponential(observer_average)
            # Observer time is prev observer time + new difference computed
            self.clock += observer_time
            # Create observer event
            observer = (self.clock, "observer")
            # Put observer event into the "queue"
            heapq.heappush(self.events, observer)

    def run_simulation(self):
        """Run the simulation"""
        # Generate arrival and observer events
        self.generate_packet_arrivals()
        self.generate_observers()

        Na = 0  # Number of Arrivals
        Nd = 0  # Number of departures
        No = 0  # Number of observers
        # Counters
        total_idle_count = 0
        sum_queue_length_observed = 0
        dropped_counter = 0
        prev_depart_time = 0
        # Loops through all events in order
        while self.events:
            # Take event with smallest time out of the queue
            event = heapq.heappop(self.events)
            if event[1] == "arrival":
                if abs(Na - Nd - dropped_counter) >= self.K:
                    Na += 1
                    # Buffer is full
                    dropped_counter += 1
                else:
                    # There is space in the queue
                    Na += 1

                    if prev_depart_time < event[0]:
                        # There is no other packet being serviced
                        prev_depart_time = event[0]

                    service_time = self.generate_service_time()
                    # Current departure time is service time + prev
departure time
                    current_departure_time = prev_depart_time + service_time
                    # Create departure event
                    departure = (current_departure_time, "departure")
                    # Put departure event in correspinding place in queue
                    heapq.heappush(self.events, departure)
```

```
                    prev_depart_time = current_departure_time

            elif event[1] == "departure":
                # Departure event
                Nd += 1

            else:
                if Na - dropped_counter == Nd:
                    # server is idle
                    total_idle_count += 1
                # Add observed packets in queue to counter
                sum_queue_length_observed += abs(Na - dropped_counter - Nd)
                No += 1
        # Compute statistics
        average_queue_length = sum_queue_length_observed / No
        idle_ratio = total_idle_count / No
        packet_loss_probability = dropped_counter / Na

        return average_queue_length, idle_ratio, packet_loss_probability


if __name__ == "__main__":
    # Usage example
    T = 2000
    arrival_rate = 1.5 * 1000000 / 2000
    average_package_length = 2000
    transmission_rate = 1000000
    K = 50

    simulator = MM1KQueueSimulator(T, arrival_rate, average_package_length,
transmission_rate, K)
    print(simulator.run_simulation())
```

A.6.1 Code for Question 6.1

Python

```
from MM1K import MM1KQueueSimulator

# Simulate M/M/1/K Queue for different values of rho
rhos = [0.5, 0.6, 0.7, 0.8, 0.9, 1, 1.1, 1.2, 1.3, 1.4, 1.5]
average_packet_counts1 = []
```

```
average_packet_counts2 = []
average_packet_counts3 = []
# Simulation time
T = 1000
average_packet_length = 2000
transmission_rate = 1000000
K = 10

print("MM1K Plot E[n] vs. rho")

for rho in rhos:
    # Compute arrival rate for given rho
    arrival_rate = rho * transmission_rate / average_packet_length
    simulator1 = MM1KQueueSimulator(T, arrival_rate, average_packet_length,
transmission_rate, K)
    # Get E[n] from first simulator
    e_n1 = simulator1.run_simulation()[0]
    average_packet_counts1.append(e_n1)
    print(f"rho = {rho}, K = 10, E[N] = {e_n1}")
    simulator2 = MM1KQueueSimulator(T, arrival_rate, average_packet_length,
transmission_rate, K + 15)
    # Get E[n] from second simulator
    e_n2 = simulator2.run_simulation()[0]
    average_packet_counts2.append(e_n2)
    print(f"rho = {rho}, K = 25, E[N] = {e_n2}")
    simulator3 = MM1KQueueSimulator(T, arrival_rate, average_packet_length,
transmission_rate, K + 40)
    # Get E[n] from third simulator
    e_n3 = simulator3.run_simulation()[0]
    average_packet_counts3.append(e_n3)
    print(f"rho = {rho}, K = 50, E[N] = {e_n3}")

if __name__ == "__main__":
    import matplotlib.pyplot as plt
    # Plot E[N] vs. rho
    plt.plot(rhos, average_packet_counts1, label = 'K = 10', color = 'blue',
marker = 'o',
             markerfacecolor = 'blue', markersize = 5)
    plt.plot(rhos, average_packet_counts2, label = 'K = 25', color =
'red',marker = 'o',
             markerfacecolor = 'red', markersize = 5)
    plt.plot(rhos, average_packet_counts3, label = 'K = 50', color =
'green', marker = 'o',
             markerfacecolor = 'green', markersize = 5)

    plt.legend()
    plt.xlabel('rho')
    plt.ylabel('E[n]')
```

```
    plt.title('E[n] in M/M/1/K Queue')
    plt.show()
```

A.6.2 Code for Question 6.2

Python

```python
from MM1K import MM1KQueueSimulator

# Simulate M/M/1/K Queue for different values of rho
rhos = [0.5, 0.6, 0.7, 0.8, 0.9, 1, 1.1, 1.2, 1.3, 1.4, 1.5]
average_packet_counts1 = []
average_packet_counts2 = []
average_packet_counts3 = []
# Simulation time
T = 1000
average_packet_length = 2000
transmission_rate = 1000000
K = 10
print("MM1K Plot p_loss vs. rho")

for rho in rhos:
    # Compute arrival rate for given rho
    arrival_rate = rho * transmission_rate / average_packet_length
    simulator1 = MM1KQueueSimulator(T, arrival_rate, average_packet_length,
transmission_rate, K)
    # Get Ploss from first simulator
    p_loss1 = simulator1.run_simulation()[2]
    average_packet_counts1.append(p_loss1)
    print(f"rho = {rho}, K = 10, P(loss) = {p_loss1}")
    simulator2 = MM1KQueueSimulator(T, arrival_rate, average_packet_length,
transmission_rate, K + 15)
    # Get Ploss from second simulator
    p_loss2 = simulator2.run_simulation()[2]
    average_packet_counts2.append(p_loss2)
    print(f"rho = {rho}, K = 25, P(loss) = {p_loss2}")
    simulator3 = MM1KQueueSimulator(T, arrival_rate, average_packet_length,
transmission_rate, K + 40)
    # Get Ploss from third simulator
    p_loss3 = simulator3.run_simulation()[2]
    average_packet_counts3.append(p_loss3)
    print(f"rho = {rho}, K = 50, P(loss) = {p_loss3}")
```

```
if __name__ == "__main__":
    import matplotlib.pyplot as plt
    # Plot P(loss) vs. rho
    plt.plot(rhos, average_packet_counts1, label = 'K = 10', color = 'blue',
marker = 'o',
             markerfacecolor = 'blue', markersize = 5)
    plt.plot(rhos, average_packet_counts2, label = 'K = 25', color =
'red',marker = 'o',
             markerfacecolor = 'red', markersize = 5)
    plt.plot(rhos, average_packet_counts3, label = 'K = 50', color =
'green', marker = 'o',
             markerfacecolor = 'green', markersize = 5)

    plt.legend()
    plt.xlabel('rho')
    plt.ylabel('P(loss)')
    plt.title('P(loss) in M/M/1/K Queue')
    plt.show()
```

A.7. Code to check stability of simulations for MM1 and MM1K

Python

```
from MM1K import MM1KQueueSimulator
from MM1 import MM1QueueSimulator

simulator = input("Which simulator do you want to check the stability for
(Enter MM1 or MM1K): ")

k = None

if simulator == "MM1":
   rhos = [0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85, 0.95]
else:
   rhos = [0.5, 0.6, 0.7, 0.8, 0.9, 1, 1.1, 1.2, 1.3, 1.4, 1.5]
   K = int(input("Choose the value of K: "))


# Values of T you want to compare
T1 = int(input("Select the first value of T to compare with (for example T =
1000): "))
```

```
T2 = int(input("Select the second value of T to compare with (for example T
= 2000): "))

average_packet_length = 2000
transmission_rate = 1000000

# Store the computed values for each simulation
values1 = {}
values2 = {}

for rho in rhos:
    # Compute arrival rate for given rho
    arrival_rate = rho * transmission_rate / average_packet_length
    if simulator == "MM1":
       # Infinite buffer with T1
       simulator1 = MM1QueueSimulator(T1, arrival_rate,
average_packet_length, transmission_rate)
       # Infinite buffer with T2
       simulator2 = MM1QueueSimulator(T2, arrival_rate,
average_packet_length, transmission_rate)
    elif simulator == "MM1K":
       # Finite buffer with T1
       simulator1 = MM1KQueueSimulator(T1, arrival_rate,
average_packet_length, transmission_rate, K)
       # Finite buffer with T2
       simulator2 = MM1KQueueSimulator(T2, arrival_rate,
average_packet_length, transmission_rate, K)
    else:
         print("Wrong simulator name!")
         exit()

    # Store values computed in both simulations
    values1[rho] = simulator1.run_simulation()
    values2[rho] = simulator2.run_simulation()

for r in rhos:
  # Store the values computed for each rho
  t1_E = values1[r][0]
  t2_E = values2[r][0]
  t1_Pidle = values1[r][1]
  t2_Pidle = values2[r][1]
  if simulator == "MM1K":
      # Only P(loss) for finite queue
      t1_Ploss = values1[r][2]
      t2_Ploss = values2[r][2]
  # Compute the difference in percentage between both simulations' values
  diff_E = (abs(t1_E - t2_E) / max(1, t1_E)) * 100
  diff_Pidle = (abs(t1_Pidle - t2_Pidle) / max(1,t1_Pidle)) * 100
```

```
  if simulator == "MM1K":
        diff_Ploss = (abs(t1_Ploss - t2_Ploss) / max(1,t1_Ploss)) * 100
  # Print the results
  if (simulator == "MM1"):
      print(f"Difference between simulations for {r} is E[n]: {round(diff_E,
3)}%, P(idle): {round(diff_Pidle,3)}%")
  else:
      print(f"Difference between simulations for {r} is E[n]:
{round(diff_E,3)}%, P(idle): {round(diff_Pidle,3)}%, P(loss):
{round(diff_Ploss,3)}%")
```