ECE 358 - Computer Networks
Lab 1: M/M/1 and M/M/1/K Queues

Andrew Zhao, Orson Marmon

Table of Contents

Page 1	Title Page
Page 2	Table of Contents
Page 3	Source Code
Page 4	High Level Overview
Page 11	Question 1
Page 12	Question 2
Page 15	Question 3
Page 20	Question 4
Page 21	Question 5
Page 25	Question 6
Page 33	References (IEEE)

Source Code

Github

To view the source code for this repository, click on the above link.

To try the code out, try these steps:

1. Open a terminal instance on a computer
2. Navigate in the terminal to where the project will reside
3. Type `git clone https://github.com/andrew2002zhao/ECE358_Lab1.git`
4. Type `cd ECE358_Lab1`
5. Type `make run`

For something that's an easy copy-and-paste after finding where to put the simulator in step 2:

```
git clone https://github.com/andrew2002zhao/ECE358_Lab1.git; cd ECE358_Lab1; make run
```

High Level Overview

This project is a simulator designed to handle both M/M/1 and M/M/1/K Queues [1].

Resources

- All code snippets were taken from a code repository on Github [2]
- Diagrams were made and taken from draw.io [3]
- Graphs are generated from both Matplotlib and Desmos [4]
- Charts are generated using a python notebook [5]
- Libraries used to make the simulator are Numpy, Matplotlib and Pandas

Main Function

The control flow in the program is as follows.

1. Each question is answered with a function call in the main() function.
2. Inside each function call, either the question is directly solved or a DiscreteEventSimulator instance is constructed.
3. If a DiscreteEventSimulator instance is created, the simulation of the queue is done inside DiscerteEventSimulator.runSimulation()
4. Results for simulation are then saved as .csv files
5. Charts are then plotted

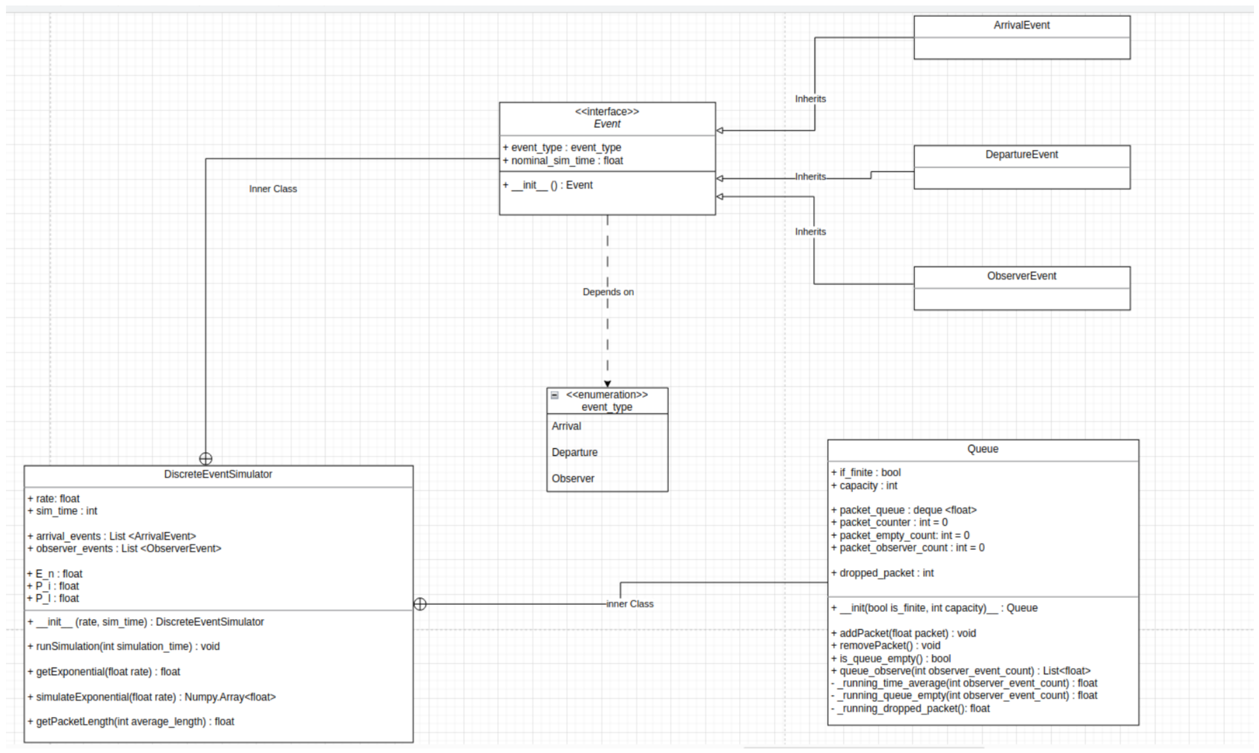
Code Snippet 1) Main Function

```
def main():
    # Question 1

    print("-----------------------------------------Question 1
START------------------------------------------\n\n")

    exp75 = simulateExponential(rate=75)
    print("Q1 mean: {} and var: {} of exponential distribution with rate:
75".format(exp75.mean(), exp75.var()) + "\n\n")

    print("-----------------------------------------Question 1
END------------------------------------------\n\n")

    print("-----------------------------------------Question 2 & 3
START------------------------------------------\n\n")

    # Question 2, 3
    simulateM_M_1()

    print("-----------------------------------------Question 2 & 3
END------------------------------------------\n\n")

    # Question 4
    print("-----------------------------------------Question 4
START------------------------------------------\n\n")

    discreteEventSimulatorQ4 =
DiscreteEventSimulator(rate=exponentialRateParameter(rho=1.2), sim_time=1000)
    discreteEventSimulatorQ4.runSimulation(transmission_rate=1e6)
    print("Q4 E[N]: {}, Pidle: {}".format(discreteEventSimulatorQ4.E_n,
discreteEventSimulatorQ4.P_i) + "\n\n")

    print("-----------------------------------------Question 4 END
------------------------------------------\n\n")

    # Question 6
    print("-----------------------------------------Question 5 & 6
START----------------------------------------- \n\n")

    simulateM_M_1_K()

    print("-----------------------------------------Question 5 & 6
END------------------------------------------\n\n")
```

DiscreteEventSimulator

The code for the Discrete Event Simulator was written with reusability and abstraction of implementation as the two key focuses. As such, the code is written mainly in an object oriented manner. The class structure of the simulator is shown below.

Figure 1) UML Diagram of DiscreteEventSimulator Class

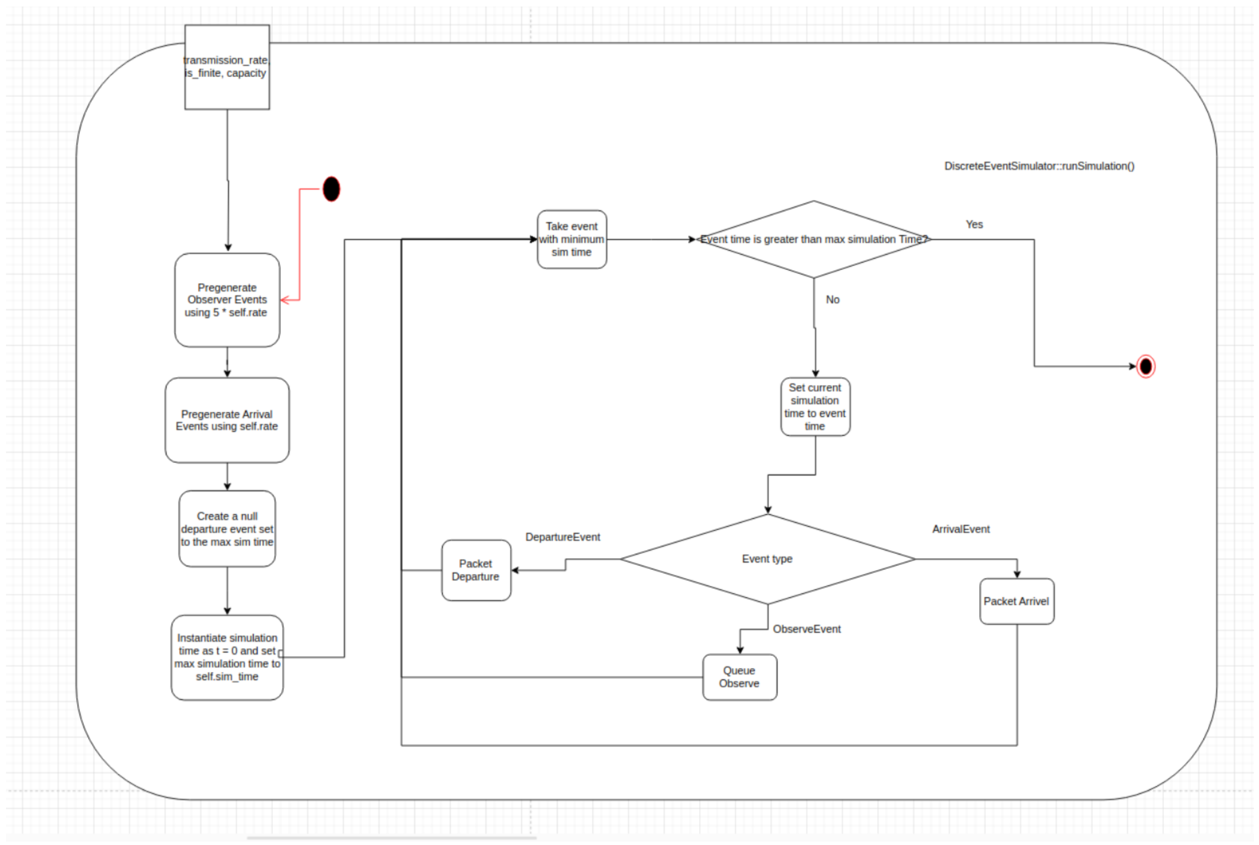

ArrivalEvent
Inherits
<<interface>>
Event
+ event_type : event_type
+ nominal_sim_time : float
+ __init__ () : Event
DepartureEvent
Inherits
Inner Class
Inherits
ObserverEvent
Depends on
<<enumeration>>
event_type
Arrival
Departure
Observer
Queue
+ if_finite : bool
+ capacity : int
+ packet_queue : deque <float>
+ packet_counter : int = 0
+ packet_empty_count: int = 0
+ packet_observer_count : int = 0
+ dropped_packet : int
+ __init(bool is_finite, int capacity)__ : Queue
+ addPacket(float packet) : void
+ removePacket() : void
+ is_queue_empty() : bool
+ queue_observe(int observer_event_count) : List<float>
- _running_time_average(int observer_event_count) : float
- _running_queue_empty(int observer_event_count) : float
- _running_dropped_packet(): float
DiscreteEventSimulator
+ rate: float
+ sim_time : int
+ arrival_events : List <ArrivalEvent>
+ observer_events : List <ObserverEvent>
+ E_n : float
+ P_i : float
+ P_l : float
+ __init__ (rate, sim_time) : DiscreteEventSimulator
+ runSimulation(int simulation_time) : void
+ getExponential(float rate) : float
+ simulateExponential(float rate) : Numpy.Array<float>
+ getPacketLength(int average_length) : float
inner Class

To try and maximize code reuse, both the buffered and unbuffered network queues were implemented using the same DiscreteEventSimulator object. Since both buffered and unbuffered cases are very similar, most of the explanation of class code will be done in this section to highlight both algorithmic and architectural design. Each question will then highlight how it used the DiscreteEventSimulator code and any additional modifications it may have made.

Running the simulator

The simulator runs for a maximum simulation time. This is not an actual time but a virtual time that tries to imitate the buffer if it was implemented as a router.
The unbuffered queue and the buffered queue are only different when DiscreteEventSimulator.runSimulation() and is_finite is passed as true or false depending on if we are using a finite sized buffer or not.

Figure 2) Pseudocode of DiscreteEventSimulator.runSimulation()


transmission_rate, is_finite, capacity
DiscreteEventSimulator::runSimulation()
Pregenerate Observer Events using 5 * self.rate
Pregenerate Arrival Events using self.rate
Create a null departure event set to the max sim time
Instantiate simulation time as t = 0 and set max simulation time to self.sim_time
Take event with minimum sim time
Event time is greater than max simulation Time?
Yes
No
Set current simulation time to event time
Event type
DepartureEvent
ArrivalEvent
ObserveEvent
Packet Departure
Packet Arrivel
Queue Observe

Events

Events for the Simulator are created as follows:

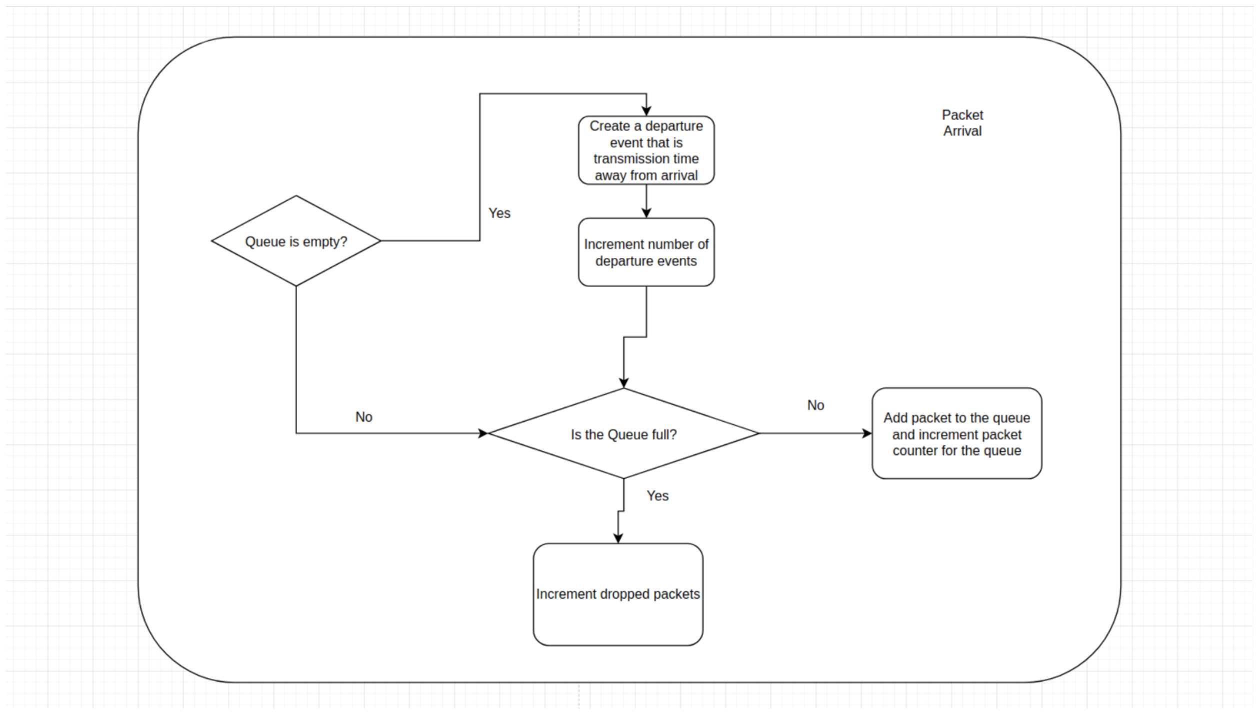
1. Packet Arrival

Packet arrivals are pre generated from the start of simulation to end of simulation all at once. This is because packet arrivals are independent events. For our purposes, we define packet arrivals as just an incoming packet, whether the queue accepts it or not is handled differently based on if we have a buffered or unbuffered queue. Because of this distinction, both queues can use the same simulator. If the queue has sufficient capacity, the packet is accepted and the packet arrival counter is incremented. This is formally defined as:

$$Time\ of\ Arrival_{Packet\ i} = Time\ of\ Arrival_{Packet\ i-1} + exp^{-1}(x,\ \lambda)$$

*arrival time does not mean the packet has been accepted for our report. This definition is used inorder to reduce unneeded code. Acceptance is still determined by if the queue is full.

Figure 3) Pseudocode of a packet arrival

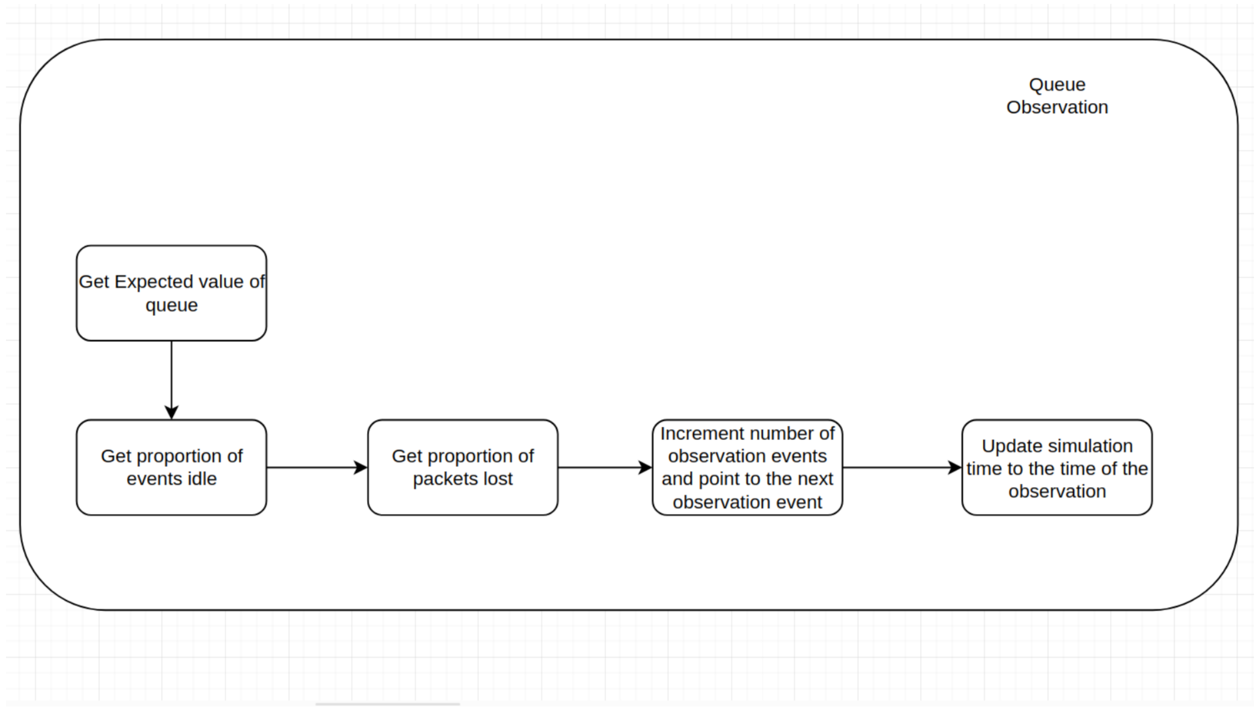

Packet Arrival
Queue is empty?
Yes
Create a departure event that is transmission time away from arrival
Increment number of departure events
No
Is the Queue full?
No
Add packet to the queue and increment packet counter for the queue
Yes
Increment dropped packets

2. Queue Observations

Queue observations are pre generated from the start of simulation time to the end of simulation time preemptively. Queue observations are independent events and are generated at 5 times the rate of Packet Arrivals (Incoming Packets). This is defined formally as

$$Observation\ Time_i = Observation\ Time_{i-1} + exp^{-1}(x,\ 5\lambda)$$

Figure 4) Pseudocode of a Queue Observation

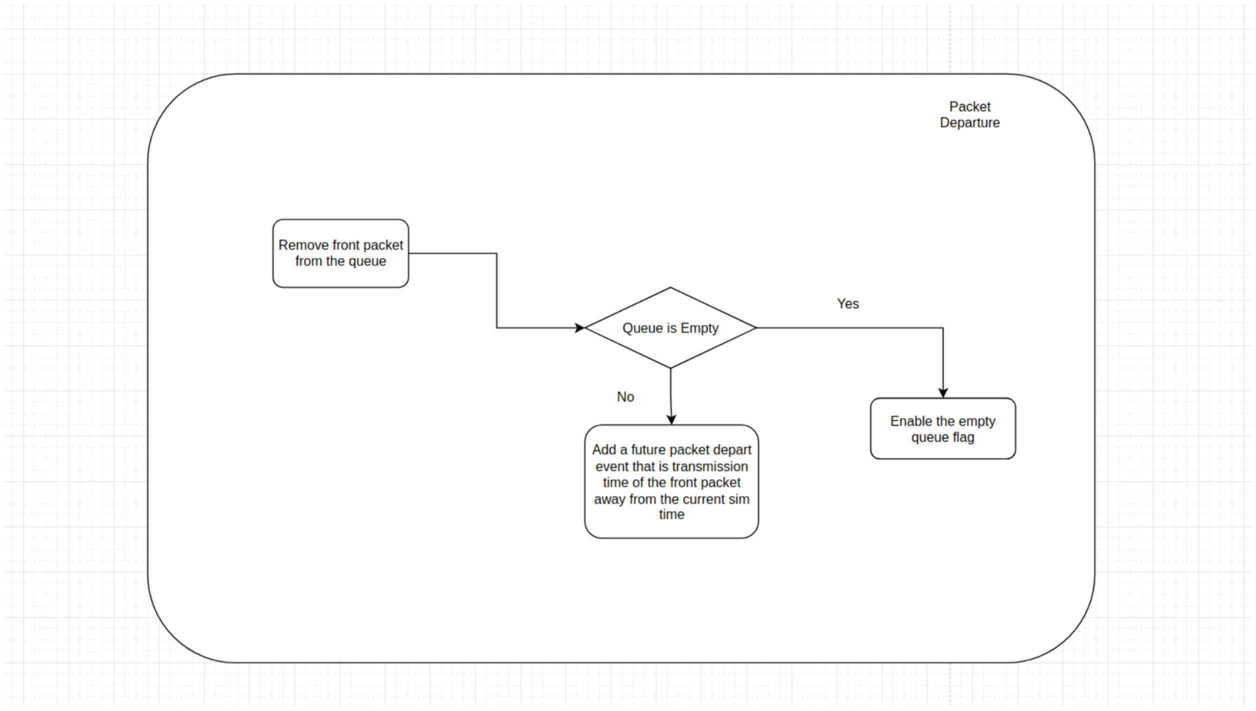

Queue Observation
Get Expected value of queue
Get proportion of events idle
Get proportion of packets lost
Increment number of observation events and point to the next observation event
Update simulation time to the time of the observation

3. Packet Departures

Packet departures are events that are dependent on the state of the queue. Thus, packet departures are generated as the simulator progresses. If the queue is empty, avoid adding a packet departure event until an arrival comes. When the arrival comes, add the departure time to be transmission time away from the arrival of the packet. Otherwise, if the queue is non empty, create a packet departure event based on the previous packet departure plus the transmission time. The formal definition is as follows:

$$\text{Time of Departure}_{\text{Packet } i} = \begin{cases} \text{Time of Departure}_{\text{Packet } i-1}, & \text{Queue is not empty} \\ \text{Time of Arrival}_{\text{Packet } i}, & \text{Queue is empty} \end{cases}$$

Figure 5) Pseudocode of a packet departure


Packet Departure
Remove front packet from the queue
Queue is Empty
Yes
No
Add a future packet depart event that is transmission time of the front packet away from the current sim time
Enable the empty queue flag

Question 1

Code Snippet 2) Code for Simulating an Exponential

```
def simulateExponential(rate):
        # rate must be bigger than 0
        if rate <= 0:
            return []


        # generate NUM_SMAPLE points from uniform distribution
        # NUM_SAMPLE is a constant defined as 1000
        U = array(uniform(low=0.0, high=1.0, size=NUM_SAMPLE))

        # vectorize ln function to apply to numpy array
        _vln = vectorize(log)

        # return numpy array of NUM_SAMPLE points using formula outlined in doc
        return array(-(1/rate)*_vln(1-U))
```

When passing a rate of 75 to simulateExponential, the code says:

Q1 mean: 0.013884896256971759 and var: 0.00019208701941861422 of exponential distribution with rate: 75

The formulas for mean and variance of an exponential distribution are

$E[X] \ = \frac{1}{\lambda} \ = \ \frac{1}{75} \ = 0.01333333333$, [6]

and

$Var(X) \ = \ \frac{1}{\lambda^2} \ = \ \frac{1}{75^2} \ = 0.00017777777$, [7]

The % error of both values are

$$E[X]\ \%error \ = |\frac{Actual - Expected}{Expected}| \ \times 100\% \ = |\frac{0.013884896256971759 - 0.01333333333}{0.01333333333}| \ \times 100\% \ = 4.12\%$$

$$Var(X)\ \%error \ = |\frac{Actual - Expected}{Expected}| \ \times 100\% \ = |\frac{0.00019208701941861422 - 0.00017777777}{0.00017777777}| \ \times 100\% \ = 8.47\%$$

These values agree with the actual expected value and variance of an exponential distribution

Question 2

The unbuffered queue function (simulateM_M_1()) creates an M/M/1 queue that is passed into the DiscreteEventSimulator. The behavior of the simulator is different than for a M/M/1/K queue since the queue that is passed is infinitely large. As a result, there are no packet losses.

Before the simulator is called, initialization is done to store P_idle, rho and E_n for plotting purposes. The simulator is run for multiple values of rho and for each value of rho, it is run for the $Max\ Simulation\ Time$ and 2 * $Max\ Simulation\ Time$. The results of the simulations are appended into lists and then using Pandas, put into a csv file. The deviation between results is calculated as a ratio between the two values with the following formula:

$$\%deviation = |\frac{val1 - val2}{val1}| * 100\%$$

Variables for the M/M/1 queue:

$0.25 \le \rho \le 0.95$ with a step size of 0.1
$L = 2000,$
$C = 1e6,$
$K \to \infty,$
$Max\ Simulation\ Time = 1000$

Code Snippet 3) Code for M/M/1 Queue

```
P_idle = []
rho = []
E_n = []

def simulateM_M_1():

    multiplier = [1, 2]

    data_frame = None
    data_frame_list = []

    for multiple in multiplier:
        E_n.clear()
        P_idle.clear()
        P_loss.clear()
        rho.clear()
        x = 0.25
        print('--------------------------------- START SIM_TIME*{}
-----------------------------------'.format(multiple))
        while x < 0.95:
            rate = exponentialRateParameter(rho=x)
            discreteEventSimulator = DiscreteEventSimulator(rate=rate,
sim_time=SIM_TIME*multiple)
            discreteEventSimulator.runSimulation(transmission_rate=1e6)
            print("##################################")
            print("rho: {}, rate_parameter: {}, E[N]: {}, P_idle: {}, P_Loss:
{}".format(x, rate, discreteEventSimulator.E_n, discreteEventSimulator.P_i,
discreteEventSimulator.P_l))

            rho.append(x)
            E_n.append(discreteEventSimulator.E_n)
            P_idle.append(discreteEventSimulator.P_i)
            print("##################################")

            x += 0.1

        # add simulation results to a data frame
        data_frame = pd.DataFrame({
            "rho" : rho,
            "E[N]" + "_"+ str(multiple) : E_n,
            "P_idle" + "_"+ str(multiple) : P_idle
        })
        data_frame_list.append(data_frame)
        print('--------------------------------- FINISHED SIM_TIME*{}
-----------------------------------'.format(multiple))

    def _f(col_1, col_2):
        return float(abs(col_1 - col_2)/col_1)*100

    # join the two dataframes on rho as the primary ID
    result = pd.merge(data_frame_list[0], data_frame_list[1], on='rho',
how='inner')

    # check if values are within 5% of each other
    result['Percent_Error'] = result.apply(lambda x: _f(x['E[N]_1'],
x['E[N]_2']), axis=1)
    result['Percent_Error_P_idle'] = result.apply(lambda x: _f(x['P_idle_1'],
x['P_idle_2']), axis=1)

    # output data to .csv
    result.to_csv('M_M_1_Simulation.csv', sep=",")
```

Average number of packets is calculated as the number of packets in the queue plus the number of observer events divided by the observer events.

Code Snippet 4) Code for Calculating Average Number of Packets

```
 # returns the current running time average of elements in the queue
        def _runningTimeAverage(self, observer_event_count):
            if observer_event_count > 0:
                self.packet_observer_count = self.packet_observer_count +
len(self.packet_queue)
                return
float(self.packet_observer_count/observer_event_count)
            return 0
```

Pidle is calculated as the number of times there has been an observer event where the queue was empty divided by the total number of observer events.

Code Snippet 5) Code for calculating pidle

```
# returns the current running Pidle
        def _runningQueueEmpty(self, observer_event_count):
            if observer_event_count > 0:
                self.queue_empty_count = self.queue_empty_count + 1 if
self.isQueueEmpty() else self.queue_empty_count
                return float(self.queue_empty_count/observer_event_count)
            return 0
```

Question 3

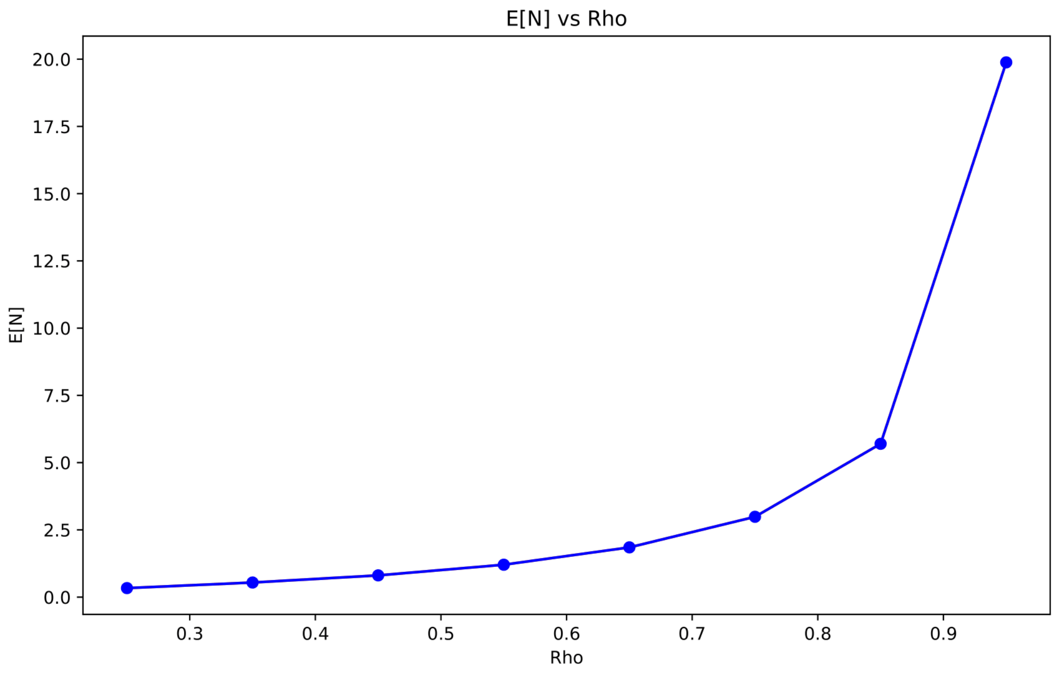
Table 1) Raw data from M/M/1 queue

rho (%)	E[N]_1 (#)	P_idle_1 (%)	E[N]_2 (#)	P_idle_2 (%)	Percent_Error_E[N] (%)	Percent_Error_P_Idle (%)
0.25	0.33673756776997577	0.7493022972981623	0.33176152025760686	0.7500521495609447	1.477722710098099	0.1000733970102811
0.35	0.5413599575184941	0.648705870504651	0.5379017420037415	0.6501748748383144	0.638801497363133	0.22645152455928616
0.44999999999999996	0.820306497737814	0.5497520843411206	0.8167974929008934	0.5509217838235928	0.4277675291610542	0.21276853981811475
0.5499999999999999	1.2074708548444972	0.45238914942073616	1.2250914498108818	0.4506760140210315	1.459297745837006	0.37868622664761686
0.6499999999999999	1.887788549485441	0.34840914500962933	1.8486208976165714	0.3506661661446058	2.074790202512115	0.6478076615681443
0.7499999999999999	3.0014548867769566	0.2490032318965793	2.9881469843251764	0.2494757909518826	0.4433817249897278	0.18978028987976073
0.8499999999999999	5.413402399250657	0.15292401620566295	5.670534256942601	0.1490903691236631	4.749912138944954	2.506896677918849
0.9499999999999998	20.351379835064847	0.048675491390715094	20.18769892574569	0.049121615090146675	0.8042742587760092	0.9165263394067766

Our results seem to be statistically valid as when comparing the results between the two simulation durations, $Max\ Simulation\ Time$ and $2 * Max\ Simulation\ Time$, there is less than a 5% deviation between expected number of packets and Pidle. Starting from the rho=0.44999999999999996, the values start to deviate from an exact step size of 0.1. This can be attributed to floating point error.

1)

Figure 6) Plot of experimental expected number of packets in unbuffered queue vs rho

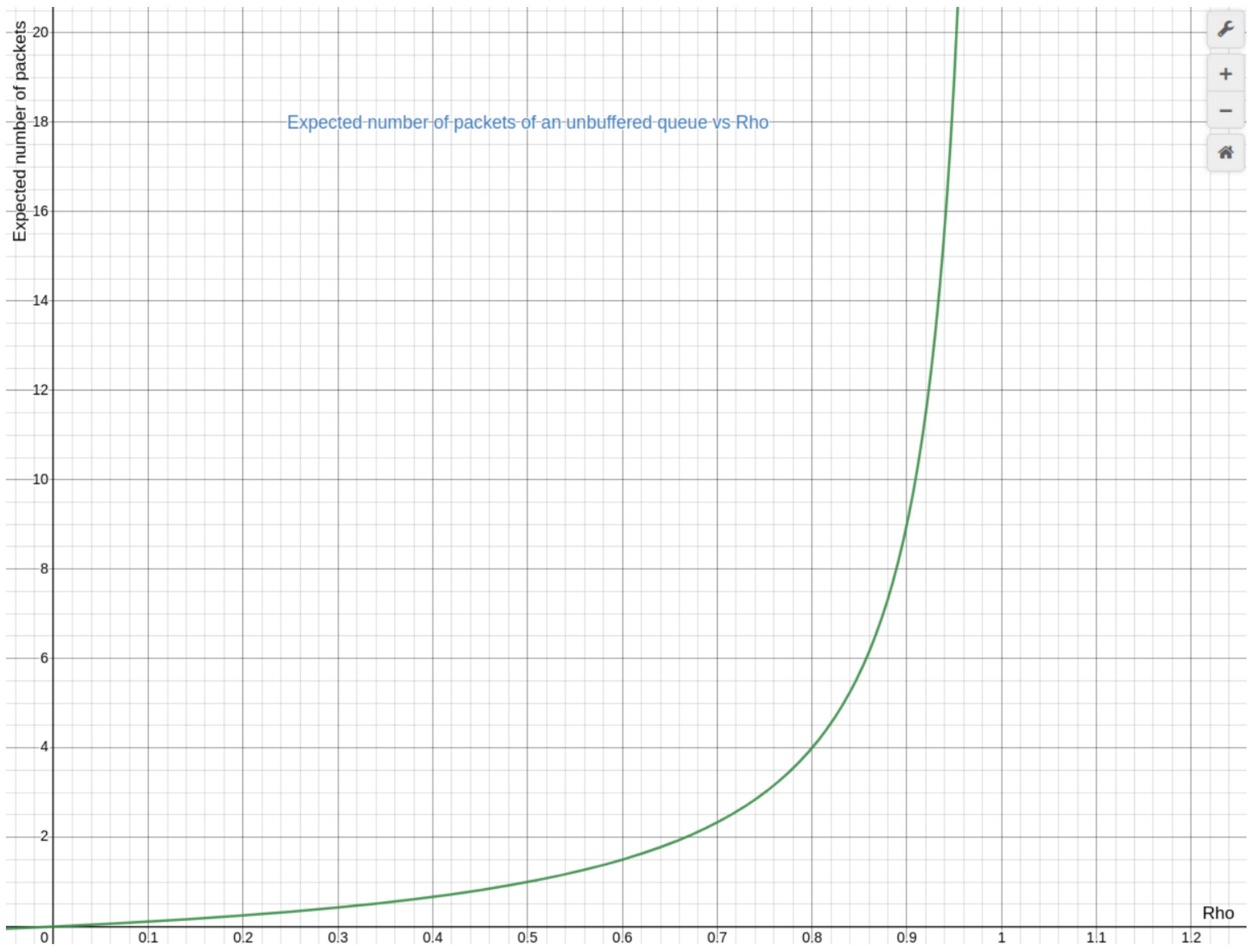


Compare these to the formula for the expected value of an M/M/1 queue [8] where

$\rho = utilization\ rate$
$K = capacity\ of\ queue$

$$E[N] = \frac{\rho}{1-\rho} = -1 - \frac{1}{1-\rho}$$

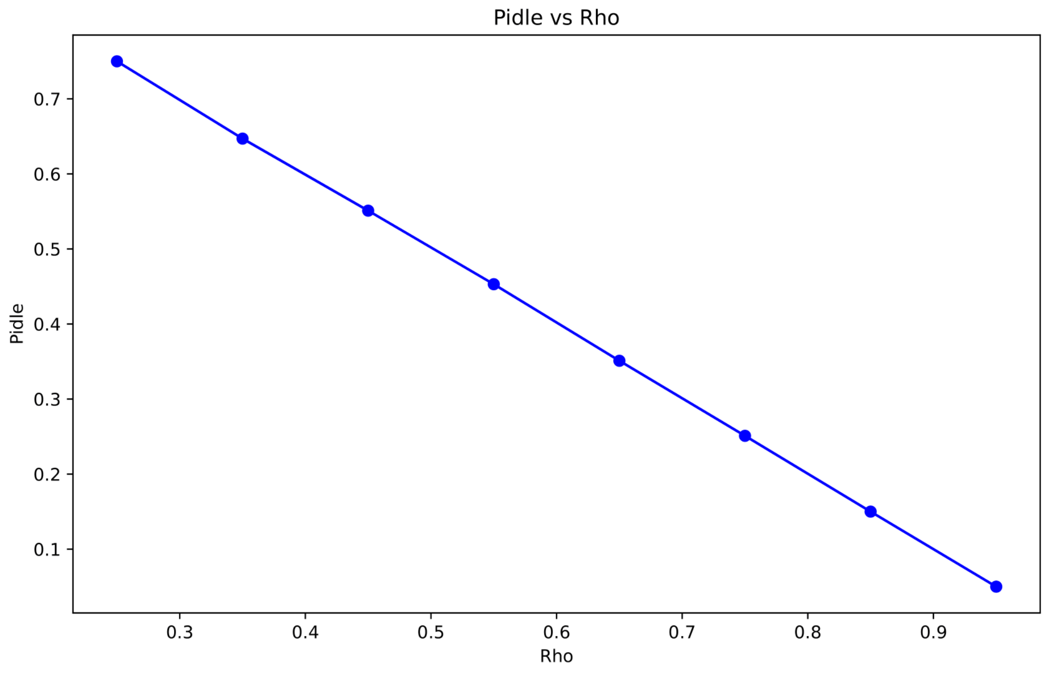
Figure 7) Plot of actual expected number of packets in an unbuffered queue vs Rho



The two figures (Figure 7 and Figure 8) follow very similar relationships. As utilization of the queue increases, the expected number of packets increases.

2)

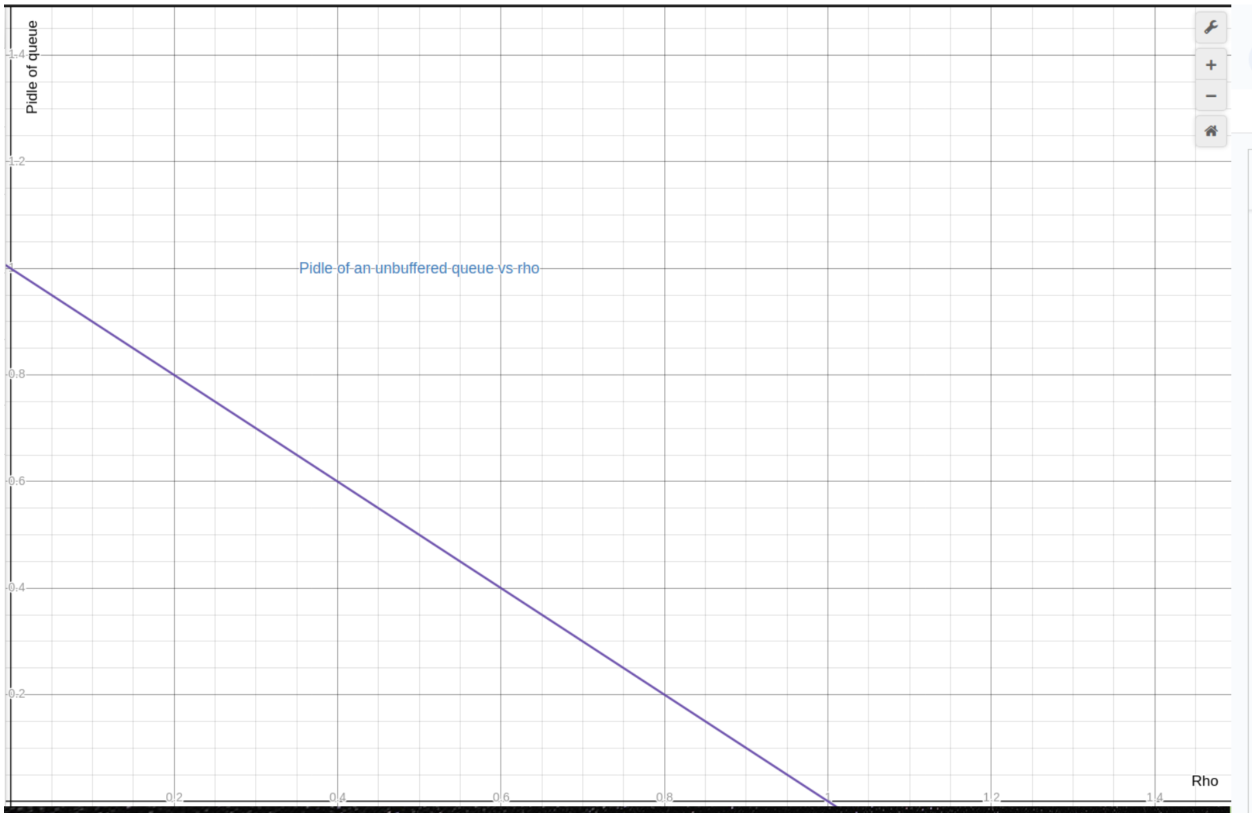
Figure 8) Plot of experimental pidle in an unbuffered queue vs rho



Compare these to the formula for the expected value of an M/M/1 queue [9] where

$P_{idle} = 1 - \rho$

Figure 9) Plot of actual pidle in an unbuffered queue vs rho



The graphs are almost identical from visual inspection. As utilization of the queue increases, the idle time decreases.

Question 4

Given a utilization rate greater than 1, the queue is now bottlenecked by the output. This means that the incoming packets are now coming at a greater rate than the outgoing rate. Because of this, it can be expected that there would be large Expected value and low Pidle values.

Code Snippet 6) Code for simulating an overloaded queue

```
# Question 4
discreteEventSimulatorQ4 =
DiscreteEventSimulator(rate=exponentialRateParameter(rho=1.2), sim_time=1000)
discreteEventSimulatorQ4.runSimulation(transmission_rate=1e6)
print("Q4 E[N]: {}, Pidle: {}".format(discreteEventSimulatorQ4.E_n,
discreteEventSimulatorQ4.P_i) + "\n\n")
```

The log gives:

```
Q4 E[N]: 49469.60799921069, Pidle: 1.899939201945538e-05
```

These results coincide with our initial predictions. This indicates that the router is being overloaded.

Question 5

The buffered queue function (simulateM_M_1_K()) creates an M/M/1/K queue that is passed into the DiscreteEventSimulator. The behavior of the simulator is different than for a M/M/1 queue since the queue that is passed is finitely large. As a result, there are now packet losses if the queue is full.

Before the simulator is called, initialization is done to store P_idle, rho and E_n for plotting purposes. The simulator is run for multiple values of rho and for each value of rho, it is run for the $Max\ Simulation\ Time$ and 2 * $Max\ Simulation\ Time$. The results of the simulations are appended into lists and then using Pandas, put into a csv file. The deviation between results is calculated as a ratio between the two values with the following formula:

$$\%deviation = |\frac{val1 - val2}{val1}| * 100\%$$

Variables for the M/M/1 queue:

$0.25 \leq \rho \leq 0.95$, with a step size of 0.1
$L = 2000,$
$C = 1e6,$
$K = 10, 25, 50$
$Max\ Simulation\ Time = 1000$

Code Snippet 7) Code for buffered queue

```
P_loss = []
P_idle = []
rho = []
E_n = []

def simulateM_M_1_K():

    E_n.clear()
    P_idle.clear()
    P_loss.clear()
    rho.clear()

    capacities = [10, 25, 50]

    multiplier = [1, 2]

    data_frame_list = []
    data_frame = None
    for multiple in multiplier:
        print('---------------------------------- START SIM_TIME*{}
------------------------------------'.format(multiple))

        for cap in capacities:

            x = 0.50
            while x < 1.5:
                rate = exponentialRateParameter(rho=x)
                discreteEventSimulator = DiscreteEventSimulator(rate=rate,
sim_time=SIM_TIME*multiple)
                discreteEventSimulator.runSimulation(transmission_rate=1e6,
is_finite=True, capacity=cap)
                print("##################################")
                print("capacity: {}, rho: {}, rate_parameter: {}, E[N]: {},
P_idle: {}, P_Loss: {}".format(cap,x, rate, discreteEventSimulator.E_n,
discreteEventSimulator.P_i, discreteEventSimulator.P_l))

                rho.append(x)
                E_n.append(discreteEventSimulator.E_n)
                P_loss.append(discreteEventSimulator.P_l)
                print("##################################")

                x += 0.1

            print("size rho {}, size E[N] {}, size P_loss {}".format(len(rho),
len(E_n), len(P_loss)))
            data_frame = pd.DataFrame({
                "rho" if multiple == 1 and cap == 10 else "rho"+"_"+str(cap) +
"_" + str(multiple): rho,
                "E[N]" + "_"+str(cap)+"_"+str(multiple) : E_n,
                "P_loss" + "_" + str(cap) + "_" + str(multiple) : P_loss
            })
            data_frame_list.append(data_frame)

            E_n.clear()
            P_idle.clear()
            P_loss.clear()
            rho.clear()

        print('---------------------------------- FINISHED SIM_TIME*{}
------------------------------------'.format(multiple))

    # concatenate all data frames from various simulations
    result = pd.concat(data_frame_list, axis=1, join='inner')
    # drop all redundant columns with the name rho_cap_multiple i.e. rho_10_2
    column_names_to_drop = []
    for multiple in multiplier:
        for cap in capacities:
            if multiple == 1 and cap == 10:
                continue
            column_names_to_drop.append("rho"+"_"+str(cap) + "_" +
str(multiple))

    result = result.drop(columns=column_names_to_drop)

    # check if values are within 5% of each other
    def _f(col_1, col_2):
        return float(abs(col_1 - col_2)/col_1)*100

    result['Percent_Error_E[N]_cap_10'] = result.apply(lambda x:
_f(x['E[N]_10_1'], x['E[N]_10_2']), axis=1)
    result['Percent_Error_P_Loss_cap_10'] = result.apply(lambda x:
_f(x['P_loss_10_1'], x['P_loss_10_2']), axis=1)

    result['Percent_Error_E[N]_cap_25'] = result.apply(lambda x:
_f(x['E[N]_25_1'], x['E[N]_25_2']), axis=1)
    result['Percent_Error_P_Loss_cap_25'] = result.apply(lambda x:
_f(x['P_loss_25_1'], x['P_loss_25_2']), axis=1)


    result['Percent_Error_cap_50'] = result.apply(lambda x: _f(x['E[N]_50_1'],
x['E[N]_50_2']), axis=1)
    result['Percent_Error_P_Loss_cap_50'] = result.apply(lambda x:
_f(x['P_loss_50_1'], x['P_loss_50_2']), axis=1)

    # save results to a csv
    result.to_csv("M_M_1_K_Simulation.csv", sep=",")
```

Ploss is calculated as the number of dropped packets divided by the total number of packets.

Code snippet 8) Code for ploss

```
# compute packet dropped ratio using dropped packets and total number of
packets sent
        def _runningDroppedPacketRatio(self):
```

```
        # if queue is M/M/1 then return None
        if not self.is_finite:
            return None
        else:
            if self.packet_counter > 0:
                return float(self.dropped_packet/self.packet_counter)
            else:
                return 0.0
```

Question 6

Table 2) Raw data for single time run from M/M/1/K queue

rho	E[N]_10_1	P_loss_10_1	E[N]_25_1	P_loss_25_1	E[N]_50_1	P_loss_50_1
0.5	0.990233797	0.000441338	0.993461161	0	1.004864027	0
0.6	1.463463129	0.002376944	1.501824833	6.66E-06	1.491821871	0
0.7	2.091982817	0.008274813	2.30758862	7.44E-05	2.326934619	0
0.8	2.967473773	0.024080687	3.962317869	0.000784843	3.985447853	0
0.9	4.015862415	0.051775969	7.254835306	0.007785469	8.439821315	0.000443726
1	4.961001782	0.089678997	12.57078075	0.03866599	25.0170051	0.019625481
1.1	5.935718166	0.139777589	17.20195528	0.097068824	40.24809389	0.092552266
1.2	6.718644405	0.190838483	20.1253467	0.166275639	44.93507281	0.164477303
1.3	7.342960063	0.246168298	21.65267593	0.230898153	46.68760099	0.231304222
1.4	7.771464252	0.29223458	22.50255853	0.285557518	47.47771439	0.284524341

Table 3) Raw data for double time run from M/M/1/K queue

rho	E[N]_10_2	P_loss_10_2	E[N]_25_2	P_loss_25_2	E[N]_50_2	P_loss_50_2	E[N]_10_2
0.5	0.992488224	0.000422826	1.008407635	0	1.001650695	0	0.992488224
0.6	1.455312232	0.002406845	1.504339902	6.42E-06	1.506060589	0	1.455312232
0.7	2.11060982	0.008518637	2.331396115	7.57E-05	2.332561369	0	2.11060982
0.8	2.976069247	0.024361805	3.979803902	0.000754834	4.06156536	7.50E-06	2.976069247
0.9	3.968286144	0.051099071	7.320877965	0.007781674	8.650197815	0.000451195	3.968286144
1	4.98374412	0.09061191	12.5815935	0.03877029	24.7173151	0.01920296	4.98374412

	2	5	3	7	4	4	2
1.1	5.927089791	0.138847776	17.45976503	0.099433989	40.56868781	0.092735072	5.927089791
1.2	6.712164011	0.192600304	20.16360289	0.166784835	44.93907726	0.16580522	6.712164011
1.3	7.304818375	0.244277281	21.72239023	0.231994196	46.62985793	0.230964369	7.304818375
1.4	7.782997204	0.292583503	22.49888334	0.284509607	47.4891599	0.28544808	7.782997204

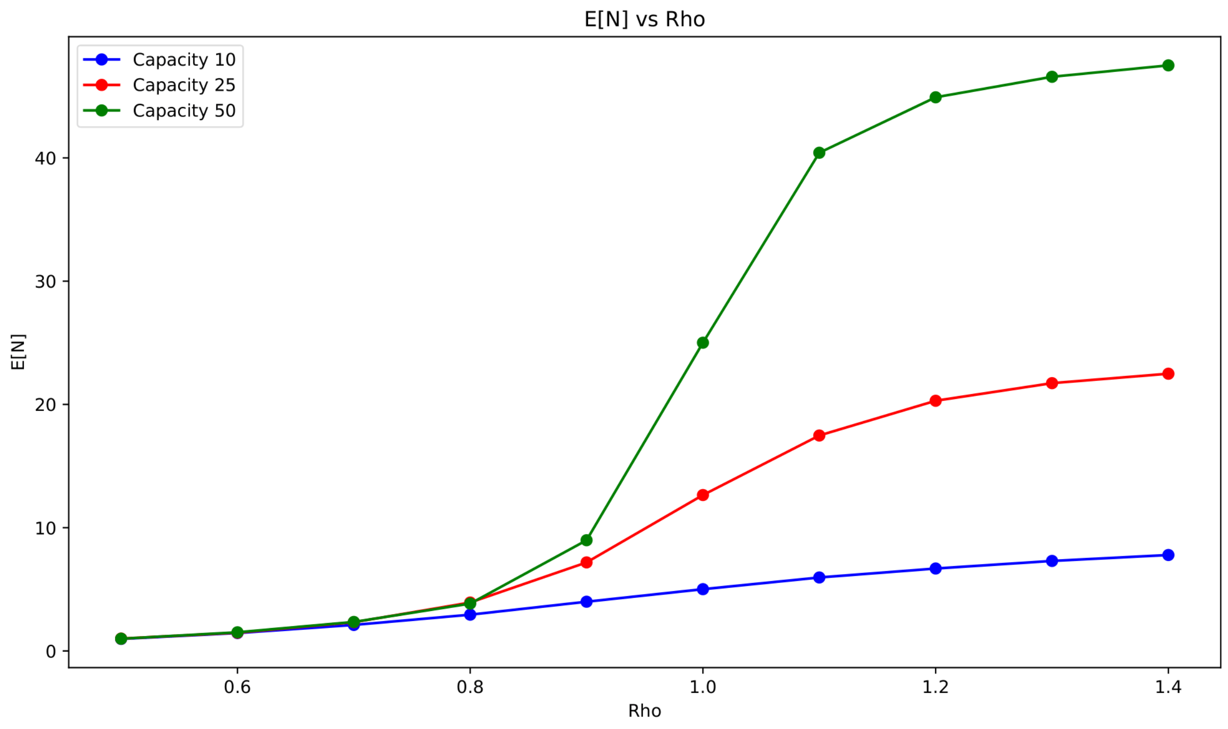
Table 4) Percent error measurements for M/M/1/K queue

rho	Percent_Error_E[N]_cap_10	Percent_Error_P_Loss_cap_10	Percent_Error_E[N]_cap_25	Percent_Error_P_Loss_cap_25	Percent_Error_E[N]_cap_50	Percent_Error_P_Loss_cap_50
0.5	0.227666037	4.19450151	1.504485007	0	0.319777731	0
0.6	0.556959507	1.257971223	0.167467494	3.61E+00	0.95445162	0
0.7	0.890399417	2.946580379	1.031704459	1.75E+00	0.241809553	0
0.8	0.289656251	1.167400204	0.441308178	3.82E+00	1.909885902	0
0.9	1.184708684	1.307360176	0.910326092	4.87E-02	2.492665333	1.683145469
1	0.458422332	1.040286308	0.086015164	2.70E-01	1.197944977	2.15289634
1.1	0.145363624	0.665208637	1.498723528	2.44E+00	0.796544376	0.197516796
1.2	0.09645388	0.923199995	0.190089586	3.06E-01	0.008911638	0.807355729
1.3	0.519432056	0.768180808	0.321966224	4.75E-01	0.123679648	0.14692896
1.4	0.148401283	0.119398324	0.016332304	3.67E-01	0.02410712	0.324660696

Our results seem to be statistically valid as when comparing the results between the two simulation durations, $Max\ Simulation\ Time$ and $2 * Max\ Simulation\ Time$, there is less than a 5% deviation between expected number of packets and Pidle.

1)

Figure 9) Plot of experimental expected number of packets for a M/M/1/K queue (K = 10, 25, 50) vs Rho



Comparing these to the formula for the expected value of an M/M/1/K queue [10] where

$\rho = \textit{utilization rate}$
$K = \textit{capacity of queue}$

$$E[N] = \frac{\rho}{1-\rho} - \frac{(K+1)\,\rho^{K+1}}{1-\rho^{K+1}}$$

Plots for the actual functions are generated below. Graphs for actual values were plotted using desmos.

Figure 10) Plot of actual expected number of packets in a M/M/1/10 queue vs Rho

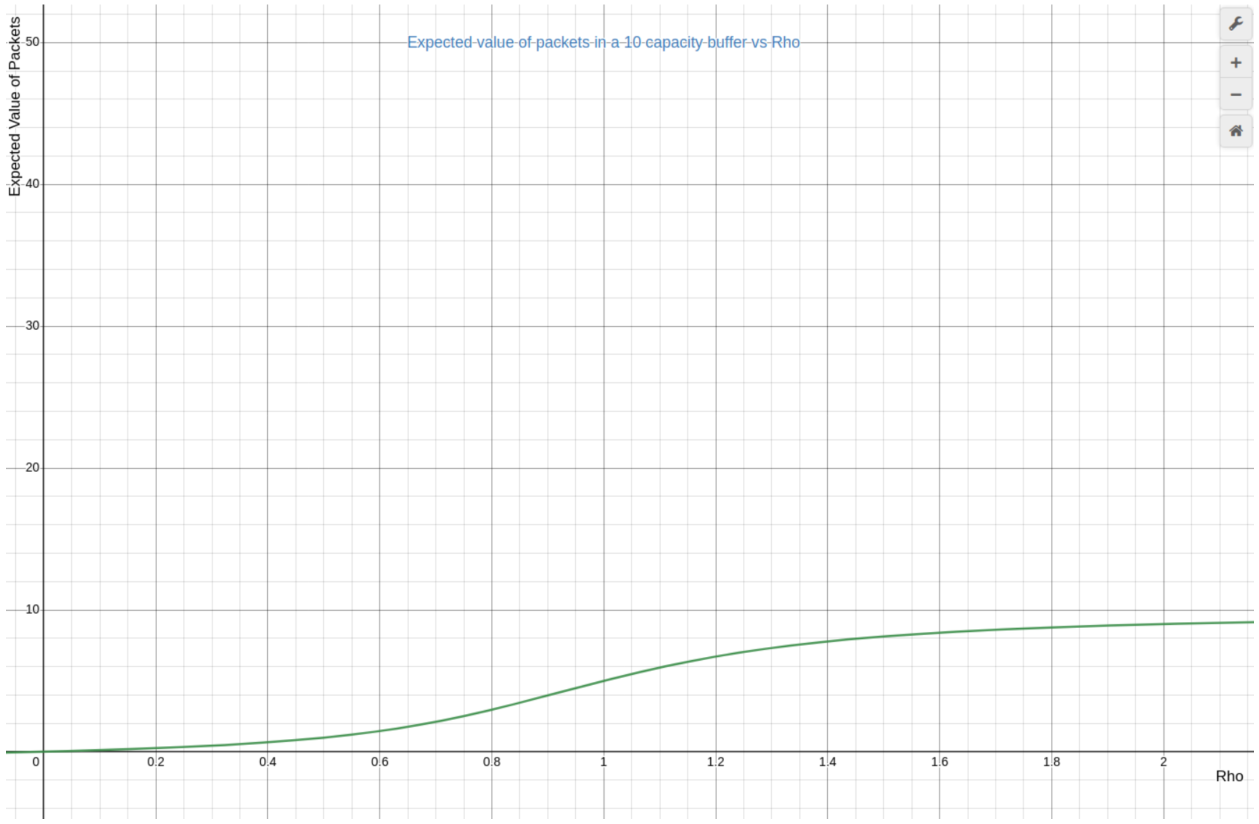


Figure 11) Plot of actual expected number of packets in a M/M/1/25 queue vs Rho

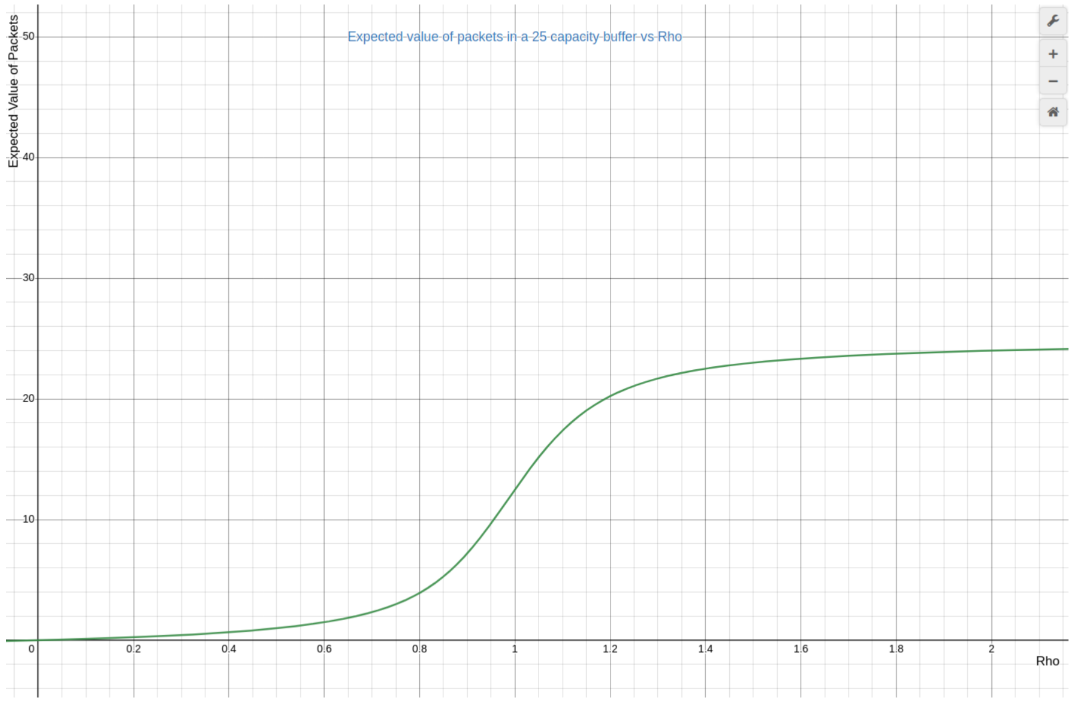
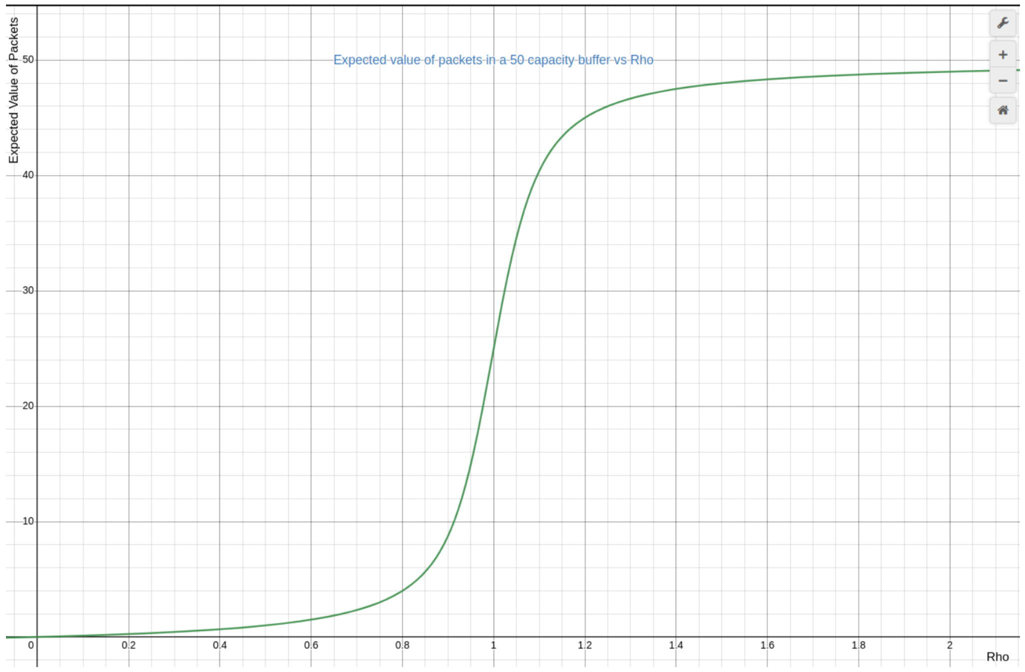


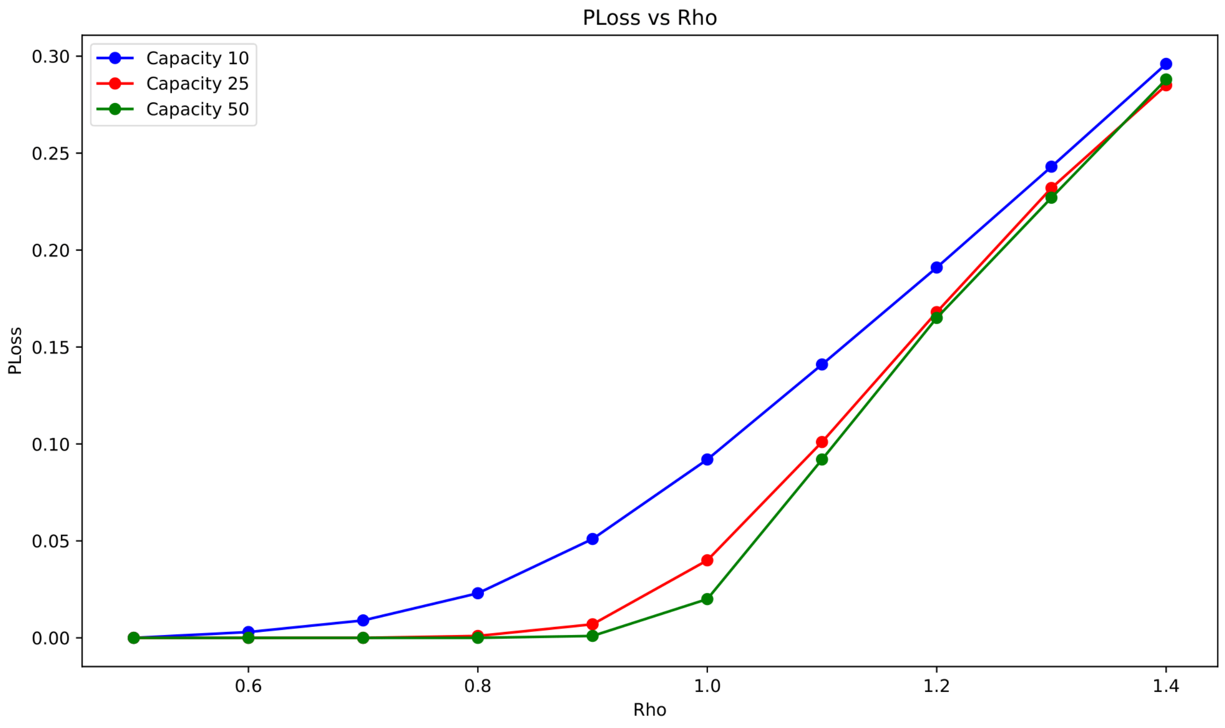
Figure 12) Plot of actual expected number of packets in a M/M/1/50 queue vs Rho



The actual plot seems very similar to what the expected plots look like. As utilization increases for the same K value, the expected number of packets also increases until the capacity K where further utilization does not increase expected number of packets. As K increases, the expected number of packets in the queue becomes larger for similar rho values, indicating that there are more packets in the queue at once.

2)

Figure 13) Plot of ploss for a M/M/1/K queue (K = 10, 25, 50) vs Rho



Ploss goes down as K increases. Compare these to the formula for the percentage of loss of an M/M/1/K queue [10] where

$\rho \; = \; utilization\, rate$
$K \; = \; capacity\, of\, queue$

$$P_{Loss} \; = \; \frac{(1-\rho)\,\rho^{K}}{1-\rho^{1+K}}$$

Plots for the actual functions are generated below. Graphs for actual values were plotted using desmos.

Figure 14) Plot of actual ploss of a M/M/1/10 queue vs Rho

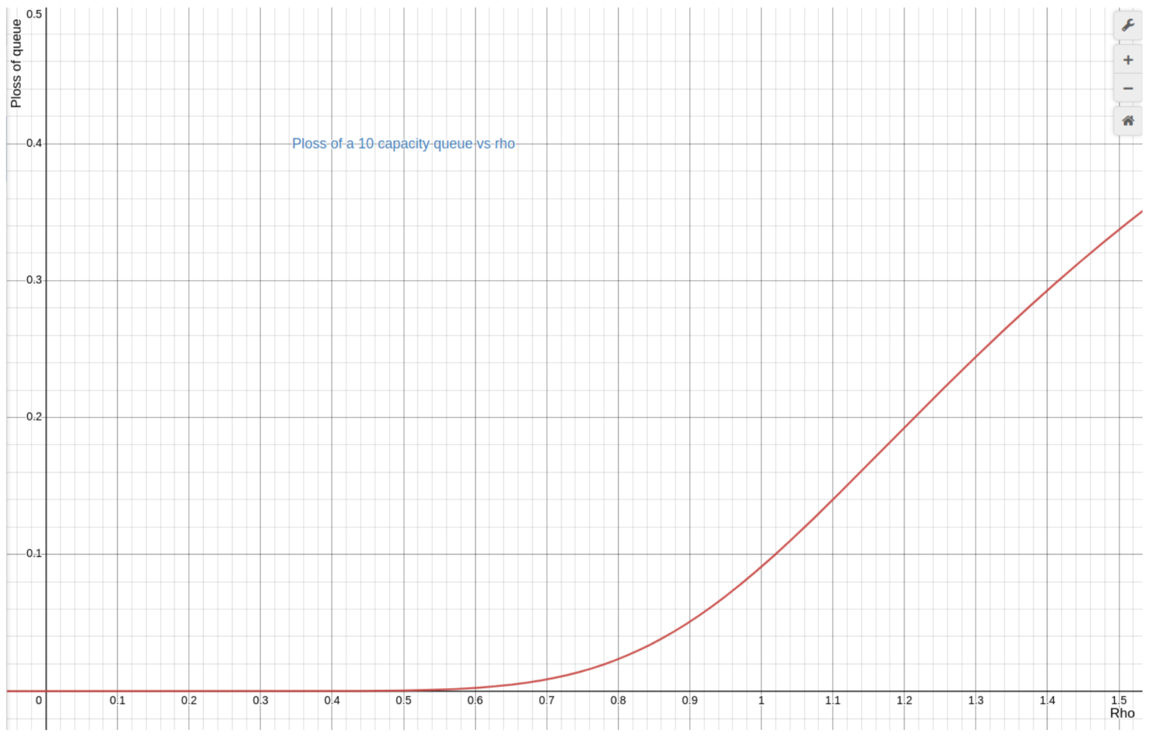


Figure 15) Plot of actual ploss of a M/M/1/25 queue vs Rho

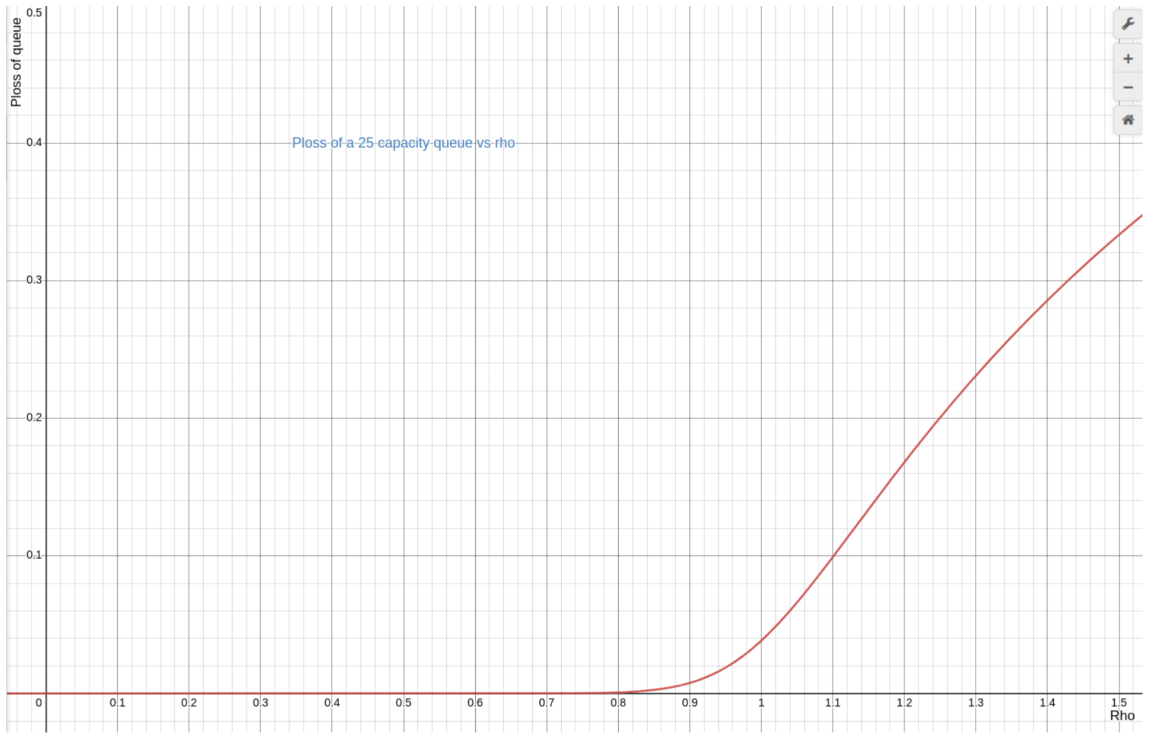
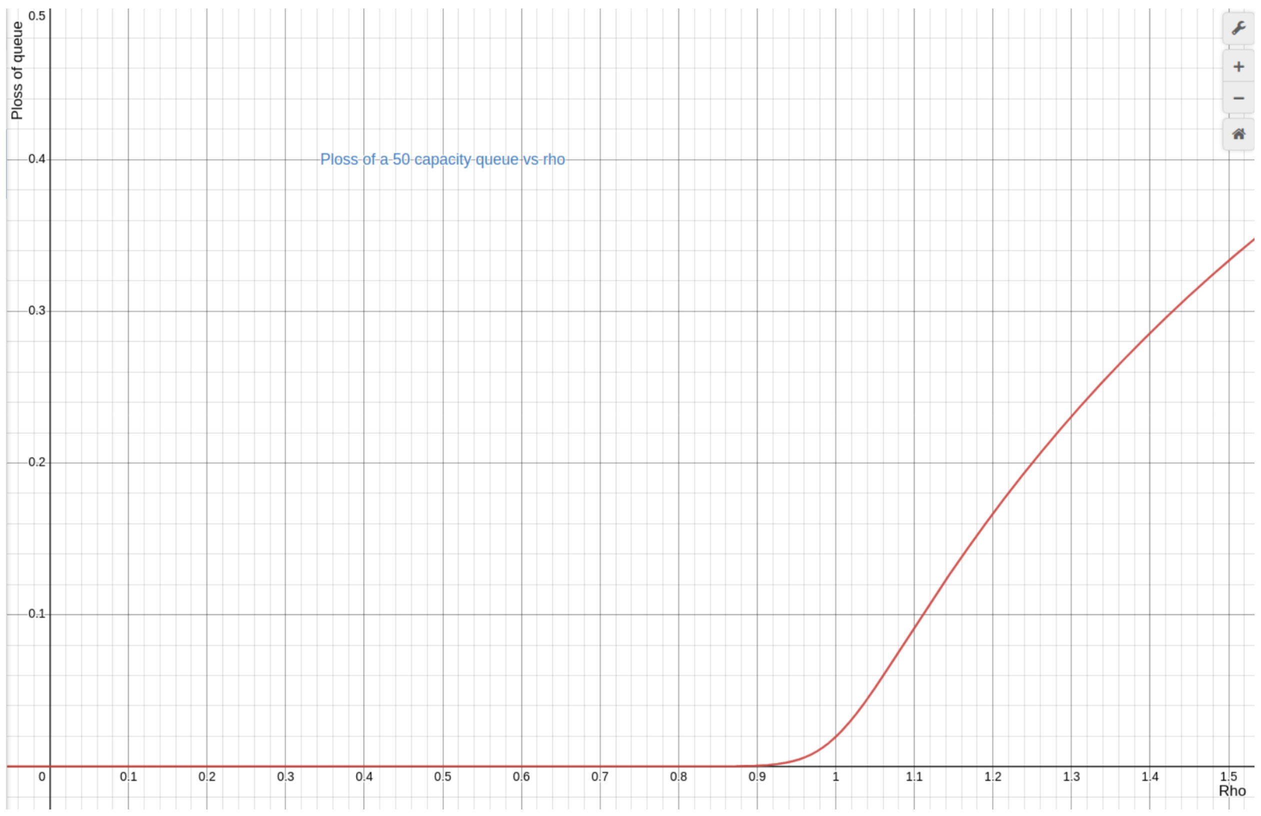


Figure 16) Plot of actual ploss of a M/M/1/50 queue vs Rho



The actual plot seems very similar to what the expected plots look like. As utilization increases given a constant K, the proportion of packet loss also increases. As K increases, ploss values get smaller for similar rho values, indicating that there is less packet loss for larger network queues.

References (IEEE)

[1] “ECE 358 - Computer Networks - Lab 1 ,” Learn, https://learn.uwaterloo.ca/d2l/le/content/948180/viewContent/5045401/View (accessed Oct. 6, 2023).

[2] A. Zhao and O. Marmon, “Andrew2002zhao/ece358_lab1,” GitHub, https://github.com/andrew2002zhao/ECE358_Lab1/tree/master (accessed Oct. 6, 2023).

[3] A. Zhao and O. Marmon, “Draw.io - free flowchart maker and diagrams online,” Flowchart Maker & Online Diagram Software, https://app.diagrams.net/#G1jXOLx1ElB8Eep2doT7c5DvFG1hP35Bw_ (accessed Oct. 6, 2023).

[4] “Graphing calculator,” Desmos, https://www.desmos.com/calculator (accessed Oct. 6, 2023).

[5] A. Zhao and O. Marmon, “DiscreteEventSimulatorGraphCreation.ipynb,” GitHub, https://github.com/andrew2002zhao/ECE358_Lab1/blob/master/DiscreteEventSimulatorGraphCreation.ipynb (accessed Oct. 6, 2023).

[6] Mean of an exponential distribution Koch, Karl-Rudolf (2007): "Expected Value" ; in: *Introduction to Bayesian Statistics* , Springer, Berlin/Heidelberg, 2007, p. 39, eq. 2.142a ; URL: https://www.springer.com/de/book/9783540727231 ; DOI: 10.1007/978-3-540-72726-2 .

[7] Variance of an exponential distribution Taboga, Marco (2023): "Exponential distribution" ; in: *Lectures on probability theory and mathematical statistics* , retrieved on 2023-01-23 ; URL: https://www.statlect.com/probability-distributions/exponential-distribution .

[8] Expected value of packets in the queue Guillemin, F.; Boyer, J. (2001). "Analysis of the M/M/1 Queue with Processor Sharing via Spectral Theory" (PDF). *Queueing Systems*. **39** (4): 377. doi:10.1023/A:1013913827667. Archived from the original (PDF) on 2006-11-29.

[9] Probability a queue is idle Harrison, Peter; Patel, Naresh M. (1992). *Performance Modelling of Communication Networks and Computer Architectures*. Addison–Wesley.

[10] m/m/1/k formulas Sztrik, J. (2021) *Basic queueing theory* https://yzr95924.github.io/pdf/book/Basic-Queueing-Theory.pdf