

ECE 358 Computer Networks

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

By: Riddle Li and Jayson Yan

Table of Contents:

1.	Question 1	3
2.	Question 2	5
3.	Question 3	7
4.	Question 4	9
5.	Question 5	10
6.	Question 6	11

Question 1

Figure 1.0 shows a series of helper functions used to generate our set of exponentially distributed variables, and to verify the mean and variance of a data set. For the exponential distribution, mean is defined as $1/\lambda$, and the variance is defined as $1/\lambda^2$. Following these formulas, Figure 1.1 compares the expected values with the experimental values. In each test case, the experimental error is within 10% of the expected values.

```
double generateExp(double lambda) {
    double u = (double)rand() / RAND_MAX;
    return -(1 / lambda) * log(1 - u);
}

//Calculates the mean of a data set
//@param dis: The data set consisting of a vector of doubles
//@return: the mean of the data set
double getAvg(const vector<double>& dis) {
    double sum = 0;
    for (auto& v : dis) {
        sum += v;
    }
    return sum / dis.size();
}

//Calculates the variance of a data set
//@param dis: The data set consisting of a vector of doubles
//@return: the variance of the data set
double getVariance(const vector<double>& dis) {
    double avg = getAvg(dis);
    double var = 0;
    for (auto& v : dis) {
        var += pow(v - avg, 2);
    }
    return var / dis.size();
}

//Generates an exponential distribution and verifies the avg and variance
void verifyDistr() {
    double lambda = 75;
    vector<double> dis(1000);
    for (auto& v : dis) {
        v = generateExp(lambda);
    }
    double avg = getAvg(dis);
    double var = getVariance(dis);
    cout << "REAL AVG: " << avg;
    cout << " EXPECTED AVG: " << 1.0/lambda << endl;
    cout << "VAR: " << var;
    cout << " EXPECTED VAR:" << 1.0/pow(lambda, 2) << endl;
}
```

Figure 1.0: Code to generate 1000 exponentially distributed data points with $\lambda = 75$.

```
REAL AVG: 0.0128597 EXPECTED AVG: 0.0133333  
VAR: 0.000161308 EXPECTED VAR:0.000177778  
  
REAL AVG: 0.0131077 EXPECTED AVG: 0.0133333  
VAR: 0.000187713 EXPECTED VAR:0.000177778  
  
REAL AVG: 0.0139803 EXPECTED AVG: 0.0133333  
VAR: 0.000196031 EXPECTED VAR:0.000177778  
  
REAL AVG: 0.0123913 EXPECTED AVG: 0.0133333  
VAR: 0.000164667 EXPECTED VAR:0.000177778
```

Figure 1.1: Running the code in Figure 1.0 four times

Question 2

After verifying the exponential distribution generator in Question 1, we began constructing our simulation. We abstracted both packet arrival and observations using an Event object, which encapsulates a timestamp of when the event occurs (arrival time for packet events and observed time for observation events), as shown in Figure 2.0. Two helper functions are used to generate a vector of both observation and packet arrival events, sorted according to their timestamp. This vector is fed into a simulator object for packet simulation.

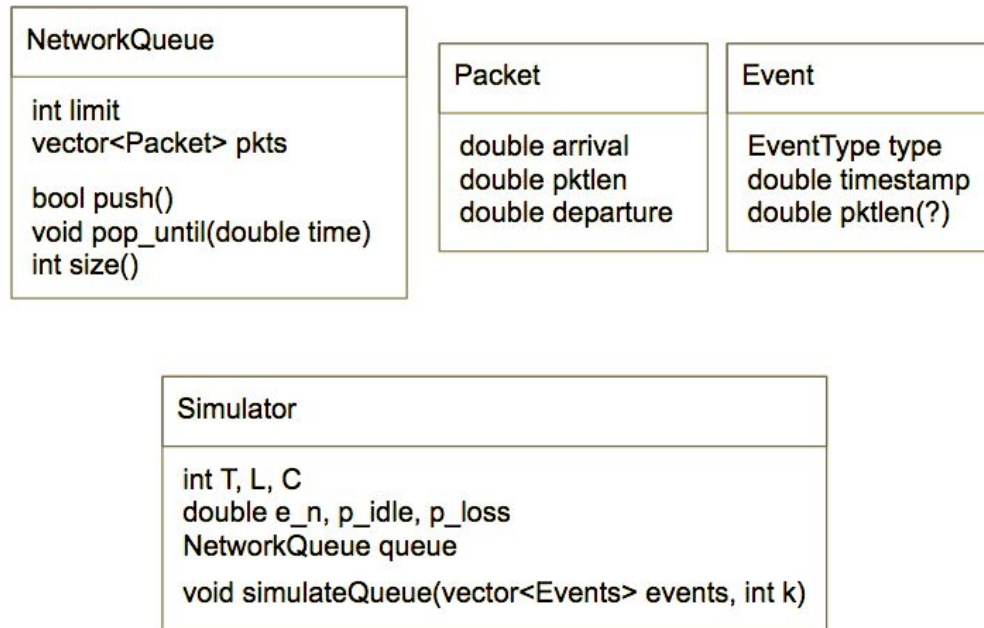


Figure 2.0 UML diagrams of Classes/Structs

The Packet object represents a single data packet, which holds 3 member variables

- arrival**: The timestamp of the packet arrival
- pktlen**: The length in bits of the packet
- departure**: The timestamp of the packet departure

The Event object represents either an Arrival or Observation event, which holds 3 member variables

- type**: The type of the event
- timestamp**: The timestamp of the event
- pktlen**: The length in bits of an arrival packet (Only populated for Arrival Events)

The NetworkQueue represents the packet buffer. It holds 2 member variables and 3 member functions

- limit**: Capacity of buffer (Only applicable for limited buffers)
- pkts**: Stores the actual packets in order of arrival
- push()**: Pushes a packet into buffer, returns false if there is no room
- pop_until()**: Pops all packets which have departed before the given timestamp
- size()**: Returns number of packets in buffer

The simulator object simulates a single execution of the network by processing every event. It holds on to 7 member variables and 1 member function

T: The total time simulated. All events will have their timestamp capped by this

L: The average packet length. Used by the packet arrival event generator to appropriate size each packet.

C: The transmission rate of output link in bits/s

e_n: The average number of packets in the queue

p_idle: The proportion of time the system is idle for

queue: A queue of packets which gets updated each event

simulateQueue(): Runs a single execution of the network

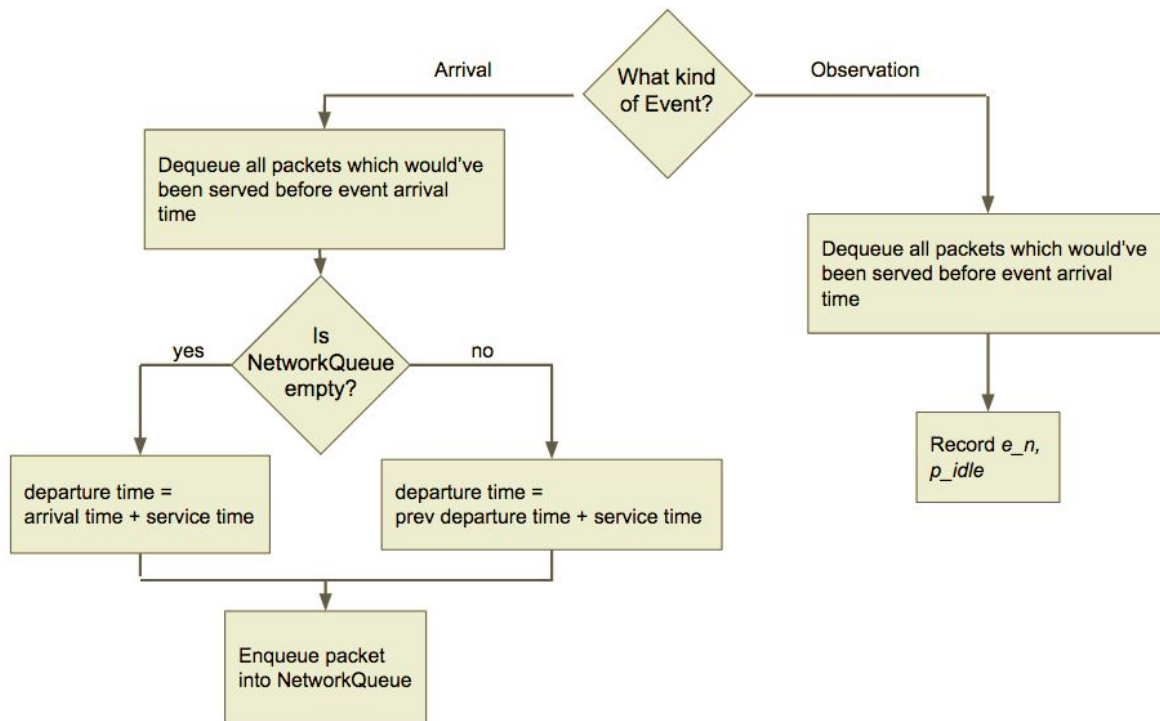


Figure 2.1 Flowchart for processing each event in an infinite buffer

As each event gets processed, if the event is a packet arrival event, a packet object is created. All packets with a departure time lower than the arrival time of the new packet are dequeued (to simulate packets that have been processed). This allows us to not have to generate departure events which is valuable when simulating a finite buffer which you can't pre-calculate the departure times of packets.

The service time of the new packet is calculated as $servTime = packet\ length / C$. Finally the departure time is calculated as $departure_time_of_last_packet + service_time_of_current_packet$ (if the queue isn't empty), or $arrival_time_of_current_packet + service_time_of_current_packet$ (if the queue is empty). The new packet object is then enqueued.

If the event is an observation event, the simulator simply records the current size of the NetworkQueue if it's not empty, or acknowledges it's empty.

Question 3

Using the simulator described in question 2 we arrived at a simulation time of $T=2000$. The following is a chart of observations for $T_1=1000$ and $T_2=2000$ and $\rho=0.15$.

	$T_1 = 1000$	$T_2=2000$	$ (T_1 / T_2 - 1) *100\%$
$E[n]$	0.332135	0.332561	0.12809680028 %
P_{IDLE}	0.749972	0.750477	0.06729053655 %

Since the observations are within 5% of each other we can use $T=2000$.

Using the simulator we were able to obtain a csv file for the relevant data. We then used a python script to parse and graph the results.

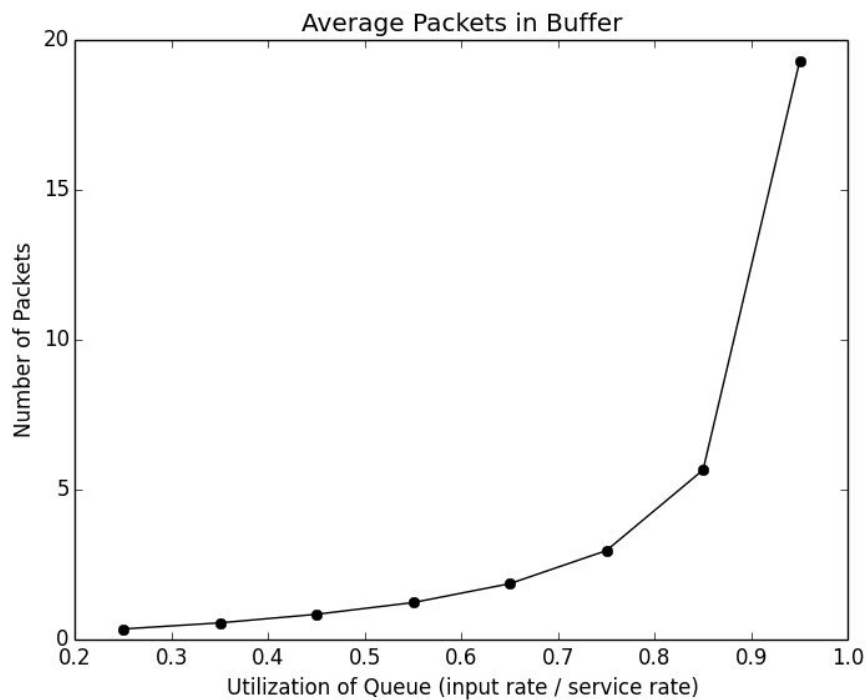


Figure 3.0 Average Number of Packets in Network Queue

The average number of packets was calculated by recording the number of packets in the queue and summing it in a variable. Then we divide the sum by the number of observations we've made which provides us the average number of packets in a queue. These observations are expected because as the utilization of the queue increases the speed the queue is able to service packets doesn't scale so the number of packets in the queue should increase.

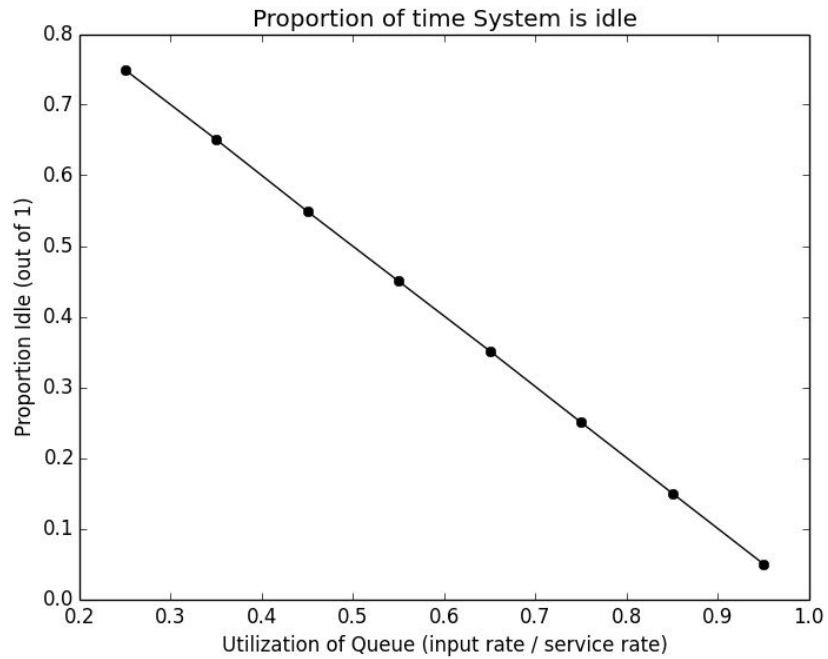
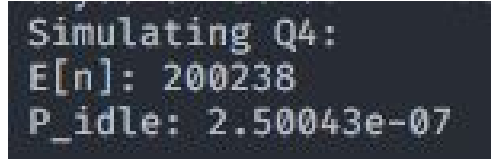


Figure 3.1 Proportion of time System is Idle

The proportion of idle time is calculated by recording whether the queue has zero packets at each observation and then dividing by the total number of observations. This way we're able to get a rough idea of the proportion of time the system is idle. The trend we are seeing is expected because as the utilization of the queue goes up the proportion that the system is idle for should go down.

Question 4

For $\rho = 1.2$ and $T=2000$ we observed the following output:

A screenshot of a terminal window with a dark background and light-colored text. The text is as follows:

```
Simulating Q4:  
E[n]: 200238  
P_idle: 2.50043e-07
```

Figure 4.0: Output of Terminal for $\rho=1.2$

We can observe that the average number of packets in the queue is much greater than any value observed in part 3. This is reasonable because ρ represents the ratio of input and service rate. So if $\rho=1.2$ then the input rate is larger than the service rate meaning on average more packets are arriving than can be serviced so the buffer will grow over time.

It would also be expected that the proportion of idle time is very low since the buffer grows over time. The amount of times we observe it to be empty will be very low.

Question 5:

The design for this section is identical to the description for Question 2, with one difference. When a packet arrival event gets processed, right before it gets enqueued, the NetworkQueue's size is checked. If it's full, the packet object is dropped instead of enqueued. The simulator keeps track of an integer p_loss to keep track of the number of packets lost.

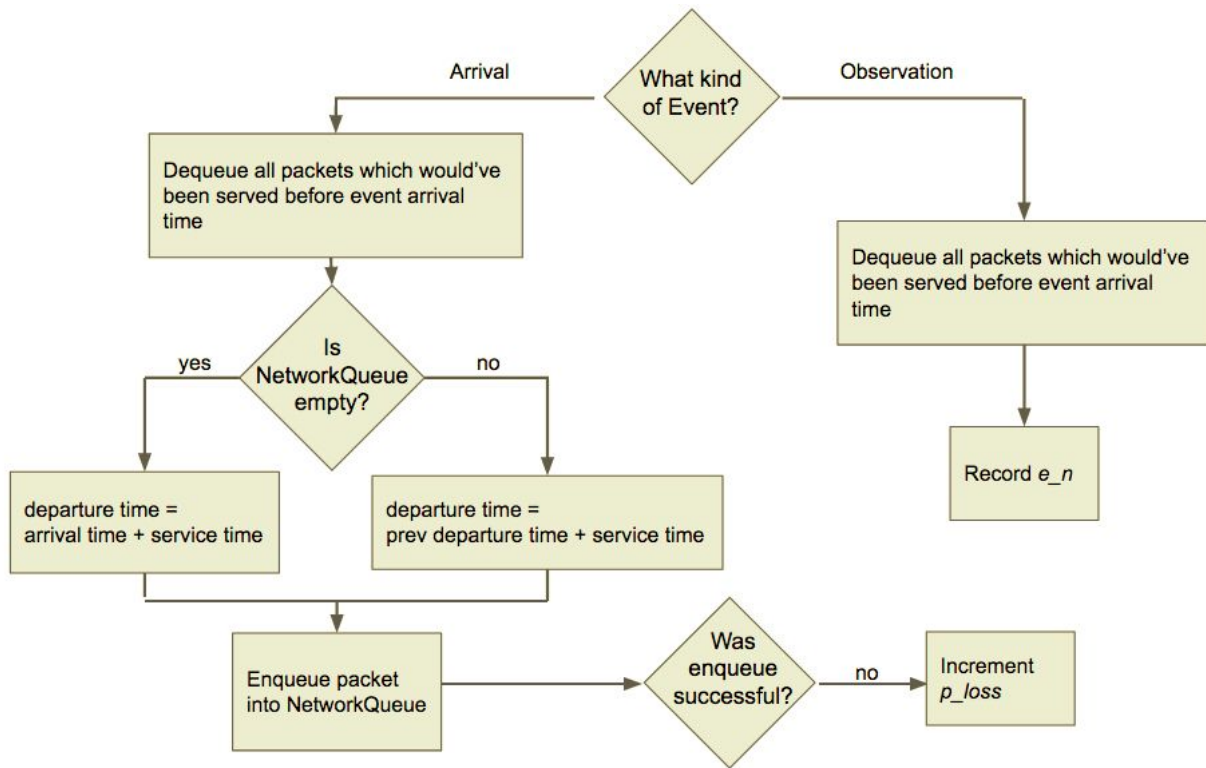


Figure 5.0: Flowchart of processing events in a finite buffer

Question 6:

Using the simulator described in question 5 and $T=2000$ as we decided in question 3, we were able to obtain a csv file for the relevant data. We then used a python script to parse and graph the following results.

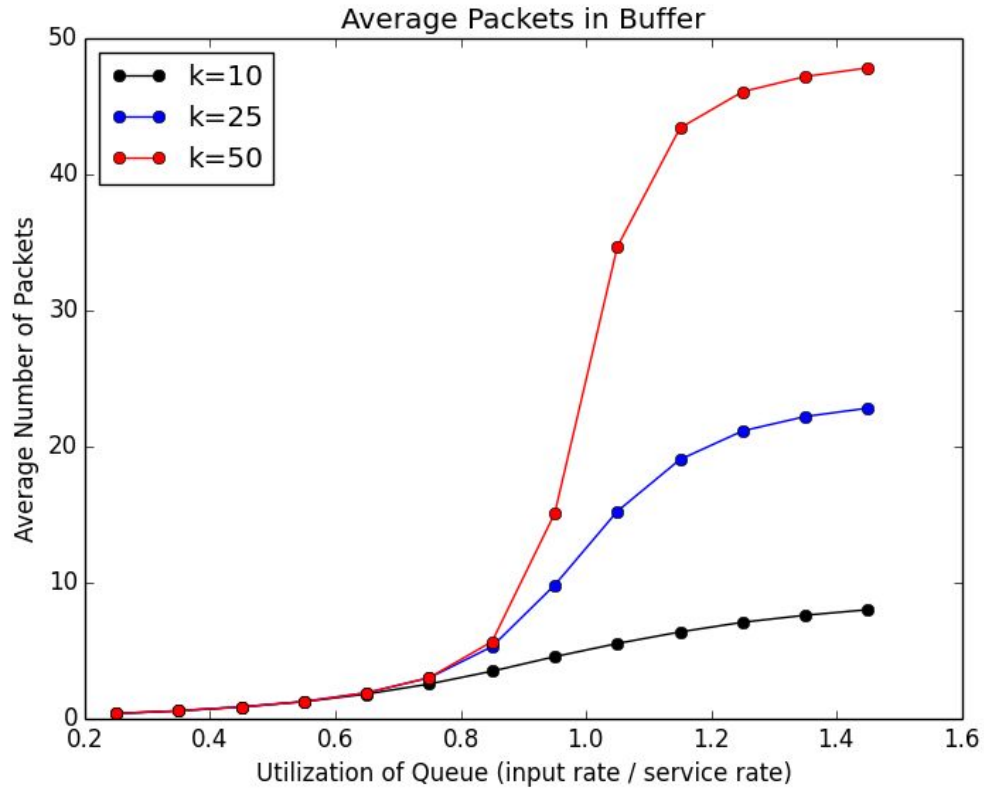


Figure 6.0: Average Number of Packets in Network Queue

These data points were gathered in the same way as question 3. Similar to our observations in question 3 the average number of packets in the queue goes up as a function of ρ . The only difference is that the growth shifts from exponential to logarithmic as we get close to the buffer size. This is expected as we get closer to the buffer limit more packets are likely to get rejected; and finally the average buffer size will max out at the buffer limit.

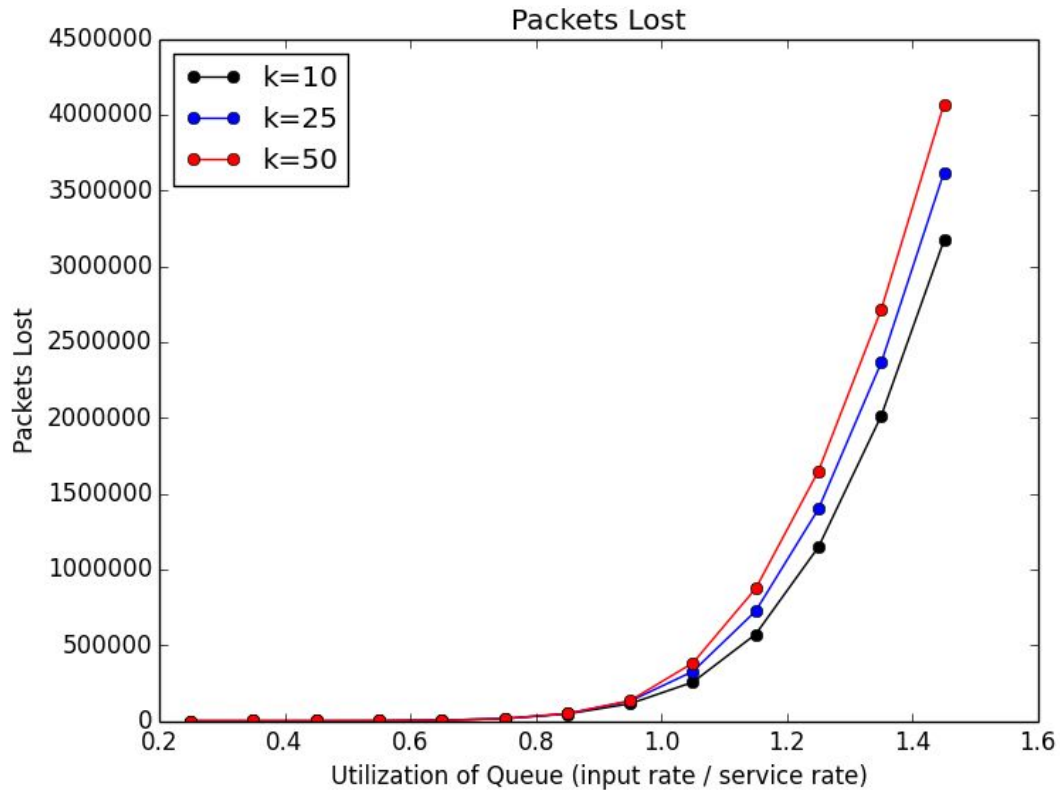


Figure 6.1: Total Number of Packets Lost

We were able to obtain P_{LOSS} by maintaining a set length queue of packets which would reject incoming packets when full. So, each time a packet was rejected we would increment the count of lost packets. These observations are consistent with our expectations as well as our observations from Figure 6.0. We see that as ρ increases the number of packets lost increase because we will more likely hit our buffer limit.