<Cover Page>

ECE358: Computer Networks

Winter 2014

Project 1: Queue Simulation

Date of submission:

Submitted by: **Raunaq Sawhney**

Student ID: **20421357**

Student name **Sawhney, Raunaq**

Waterloo Email address **rsawhney@uwaterloo.ca**

Marks received: <Leave this blank>

Marked by: <Leave this blank>

Table of Contents

Question 1

PDF: $f(X) = \lambda * e^{(\lambda x)}$
CDF: $F(X) = y = 1 - e^{(\lambda x)}$
$= e(-\lambda x) = 1 - y$
$= -\lambda x = \ln(1 - y)$
$x = -(1 / \lambda) * \ln(1 - y)$

```
float uniform()
{

      return ((float)rand()/(float)(RAND_MAX + 1.0));
}

float exponential(const float lambda)
{
      float exp_value;
      exp_value = (float)((-1.0 / lambda) * log(1.0 - uniform()));
      return exp_value;
}

for (int i = 0; i < 1000; ++i)
{
      exponential(75);
}
```

```
Raw Mean:        0.013311
Expected Mean:   0.013333
Mean Difference: 0.168119 %

Raw Variance:          0.000178
Expected Variance:     0.000178
Variance Difference:   0.045338 %
```

The actual values for the mean and variance agrees with the expected theoretical values of an exponential random variable with $\lambda = 75$.

Question 2

Variables:

`struct event:`
A C-style struct used as an event for the event scheduler, containing the packet type, packet length, and packet time for all incoming events created

`float roh:`
Value used to determine the utilization rate of the server. It is calculated using the equation: L * (λ / C), where L is the average packet length in bits, λ is the average number of packets generated/arrived per second, and C is transmission rate of the output link in bits per seconds

`float buffer_size:`
The buffer size in number of packets. For an infinite buffer, the value of buffer_size is passed in, as an argument, as 1 and the corresponding interpretation of that is a buffer with size 0. It does not mean a buffer **is** created having a size of zero. Any non-negative value would result in a buffer being created of that size

`struct event *first, *last:`
Pointers to the first and last elements of the Singly-Linked List used to construct the time-ordered list of events for the Discrete Event Scheduler

`float current_time:`
A running counter used to keep track of the event (observers or arrivals) that was last created before the current event is created, to ensure events are not created at times greater than the duration of the simulation

`int num_packet_arrivals:`
A running counter used to keep track of the number of arrival events detected in the discrete event scheduler while running through the simulation

`int num_packet_departures:`
A running counter used to keep track of the number of departure events detected in the discrete event scheduler while running through the simulation

`int num_observations:`
A running counter used to keep track of the number of observer events detected in the discrete event scheduler while running through the simulation

`int num_generated:`
A running counter used to keep track of the number of observer events detected in the discrete event scheduler while running through the simulation

`long double num_packets_in_system:`
A running counter used to keep track of the number of packets in the system and the number of packets in the queue waiting to be served

`int num_idle:`
A running counter used to keep track of the number of instances of when the queue was empty, and the system was idle.

`long num_dropped:`
A running counter used to keep track of the number of packets that are dropped as a result of the queue being full with packets being serviced and the buffer having no more space to accommodate another packet. In such an event, the packet is dropped by the system

`float service_time:`
The service time is computed by dividing the arrival packet's length by the transmission to determine how long will the system take to service the packet. This time is later added to the departure time computation

`float departure_time:`
The departure time is determined by checking if the queue is empty, and the arriving packet can be served immediately. If so, the departure time is simply the arrival packet's length plus the service time for that packet. Otherwise, if the queue is not empty, the departure time is the previous arrival packet's departure time plus the current arrival packet's service time.

`struct event *highest_time:`
Pointer to the packet that currently has the highest time in the discrete event scheduler. This time is used for inserting the departure events in the event scheduler. Using this time, instead of traversing the entire linked list from the first element to the insertion point, the departure time can simply be inserted at a point very close to the highest_time event. This reduces the runtime of the insertion algorithm significantly, as with growing number of packets, the departure packets are always inserted after the arrival packet that generated it.

Functions:

`struct result simulator(int T, float r, int L, int C, int k):`
This is used to initiate the simulation for the queue. It takes in parameters T: duration, r: ρ, L: packet length, C: link rate, k: buffer size from a tester application. It then sets up the simulation and calls the `gen_observers`, `gen_arrivals`, and `run_system` functions. Finally, it returns a struct of holding all the information and computations to the tester application

`void gen_observers(int alpha, int duration):`
This function takes in the alpha value, and simply adds observer events to the discrete event scheduler using an exponential distribution with parameter `alpha`.

`void gen_arrivals(float lambda, int duration, int packet_len):`
This function takes in the lambda value, and simply inserts arrival events to the discrete event scheduler using an exponential distribution with parameter `lambda`. It does this by

using the `init_event` function and gives it the average packet length `packet_len` value, which then returns an event of type arrival with a length and time.

`void run_system(float lambda, int duration, int packet_len):`
This system runs through the discrete event scheduler looking at each event in it. If the event is an observer, a corresponding procedure is run (which computes the current system metrics). If the event is a departure, the departure event counter is incremented. If the event is an arrival, a departure event is created and inserted into the discrete event scheduler (more detailed description of this function will be discussed later).

`void compute_metrics(…):`
This function simply prints out the values of all the metrics calculated to the console.

`void add(struct event *):`
This function takes as argument an event struct and adds it to the end of a linked list used to define a discrete event scheduler

`void insert_event(struct event *, struct event *):`
This function takes two arguments, one, a pointer to the event struct to be inserted into the discrete event scheduler, and two, a pointer to event struct from which to start looking at spots to insert the new event, as opposed to traversing the linked list each time a new event is to be inserted

`void delete_event(struct event *):`
This function takes as argument the event struct to be deleted (or removed) from the discrete event scheduler, and deletes that event from the list

`struct event * init_event(char , float, int ):`
This function takes as argument the event struct to be initialized with a type, time, and length. The function allocated memory for this event, as well as, creates an actual pointer to a struct containing the information. It then returns the pointer to the event struct

`float uniform():`
This function returns a random, uniformly distributed number generated between 0 and 1

`float exponential(const float lambda):`
This function returns an exponentially distributed random number using the random, uniformly distributed number generated using the uniform function. The equation used to calculate this value is $-1 / \lambda * \ln(1-U)$

Description of System

The queue simulation works by having a tester file define several test parameters, such as, buffer size, ρ, duration, average packet length, and link rate. These parameters are then used to initialize a link list that would hold several “events.” This events contain three attributes, namely type, time, and length. First, using the value of lambda (obtained by the equation $\rho = \lambda * (L/C)$) alpha is calculated, where alpha = λ + 10. This is to ensure that there are always more

observers computing system metrics than the number of arrival packets. Using the alpha value, observers are created with a length of 0 and a time that is generated using the exponential distribution function. This observer event is then added to a time ordered list (using a linked list). Next, the arrival events are created in a similar way to the observers, however, instead of a length of 0, arrival events contain a length computed using the exponential function and parameter (1/L). The arrival event is then inserted into the linked list based on its time. Once arrival and observer events have been inserted into the discrete event scheduler, the system is run, where each event in the list is visited and depending on the type of event, an event procedure is run. After running each event's corresponding procedure, the pointer of the current event is moved to the next event in the list until the end of the list is reached.

If the event type is an **arrival**, the `num_generated` counter is incremented, signifying the presence of a arrival packet that was *generated*. Then the `buffer_size` is checked to ensure that packets can be put into the buffer if the queue of the system is busy servicing other packets, in the case of finite buffer (M/M/1/K), otherwise in the case of an infinite buffer (M/M/1), the `buffer_size` check is done for a size of zero (defined as an infinite buffer). If this check fails, the packet is dropped and the `num_dropped` counter is incremented by one, implying the queue was full and the buffer does not have enough space for another packet. If the check passes, the queue is checked to see if the current packet can be serviced immediately (i.e., `num_packet_arrivals - num_packet_departures == 0`), in which case the departure time is calculated as the current arrival packet's plus its service time (calculated as the packet's length divided by the link rate). If the queue is currently not empty, the departure time is calculated as the service time plus the previous arrival packet's departure time, to ensure that the current arrival packet is serviced *after* the previous arrival packet leaves the system. Then, a departure event with the departure time computed is initialized, created, and inserted into the discrete event scheduler at a time *after* the current arrival packet (to reduce insertion time complexity). Then, the `num_packet_arrivals` counter is incremented to imply that an arrival packet was handled by the system, either directly serviced or added to a buffer of size `K`.

If the event type is **departure**, only the `num_packet_departures` is incremented, implying a departure packet was seen in the discrete event scheduler

If the event type is **observer**, the `num_observers` counter is incremented, implying an observer packet was seen in the discrete event scheduler. In addition, the num_packets_in_system counter is incremented based on the existing packets in the system plus the number of packets currently in the queue. If the queue is currently empty, the `num_idle` counter is incremented, implying that the system was currently idle when the observer event was seen.

The pointer of the current event pointer is then moved to the next one in the list and the above steps are repeated until there are events in the list. At the end, the `compute_metrics` function is called that computes all of the required metrics, and the probabilities. It then prints the results to the terminal console and packages a result struct to return to the tester application that would print the results to a csv file in addition to the console.

Finally, the `cleanup` function is called that simply clears out the memory allocated by the linked list based discrete event scheduler and nulls the state of the system to be run again if a different set of input parameters. In the end, the total time taken by the system is calculated and printed to the console.

Question 3

Table 1: Raw metrics from simulation of an infinite buffer (M/M/1) queue

Question 3: Infinite Buffer								
alpha	lambda	Roh	N_o	N_a	N_d	N(t)	E[N]	P_idle
39.17	29.17	0.35	429218	320932	320932	232,233	0.5411	0.650
47.50	37.50	0.45	517155	412149	412149	421,241	0.8145	0.550
55.83	45.83	0.55	604845	503499	503499	734,752	1.2148	0.452
64.17	54.17	0.65	704755	596727	596727	1,321,071	1.8745	0.350
72.50	62.50	0.75	793062	688097	688097	2,374,013	2.9935	0.250
80.83	70.83	0.85	881124	779938	779938	5,004,650	5.6798	0.150
Question 4: Infinite Buffer (roh = 1.2)								
110.00	100.00	1.2	1210513	1099710	1099710	110888361501	91,604.44	0

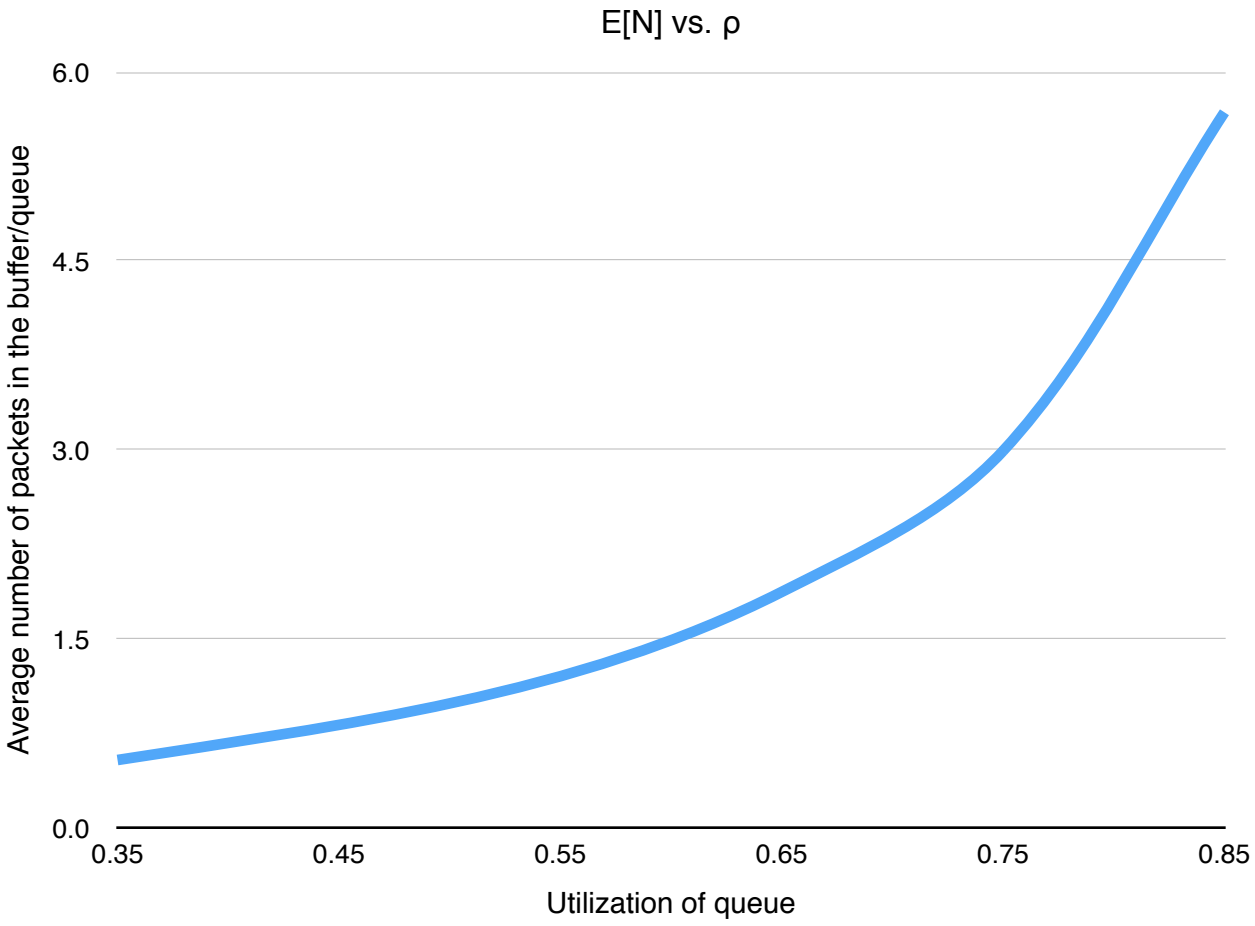

Figure 1: Average number of packets in buffer vs. Utilization of queue for $0.25 < \rho < 0.95$ (step size 0.1)

M/M/1 E[N] for 0.25 < ρ < 0.95 (step size 0.1)

In Figure 1, it can be seen that as the value of ρ is increased from 0.35 to 0.85, the average number of packets increases almost exponentially. This is because as ρ is increased, the value of lambda is also increased proportionally (i.e., increased average number of packets generated/second). There is also no loss of packets, since the queue is infinite. The plot was created by increasing the value of ρ using the step size of 0.1, and running the system for T = 11000 (i.e., the duration of simulation, for all simulations performed for this project). Then, the number of packets in the system N(t) and number of observers N_o value was used to determine the average number of packets in system E[N].

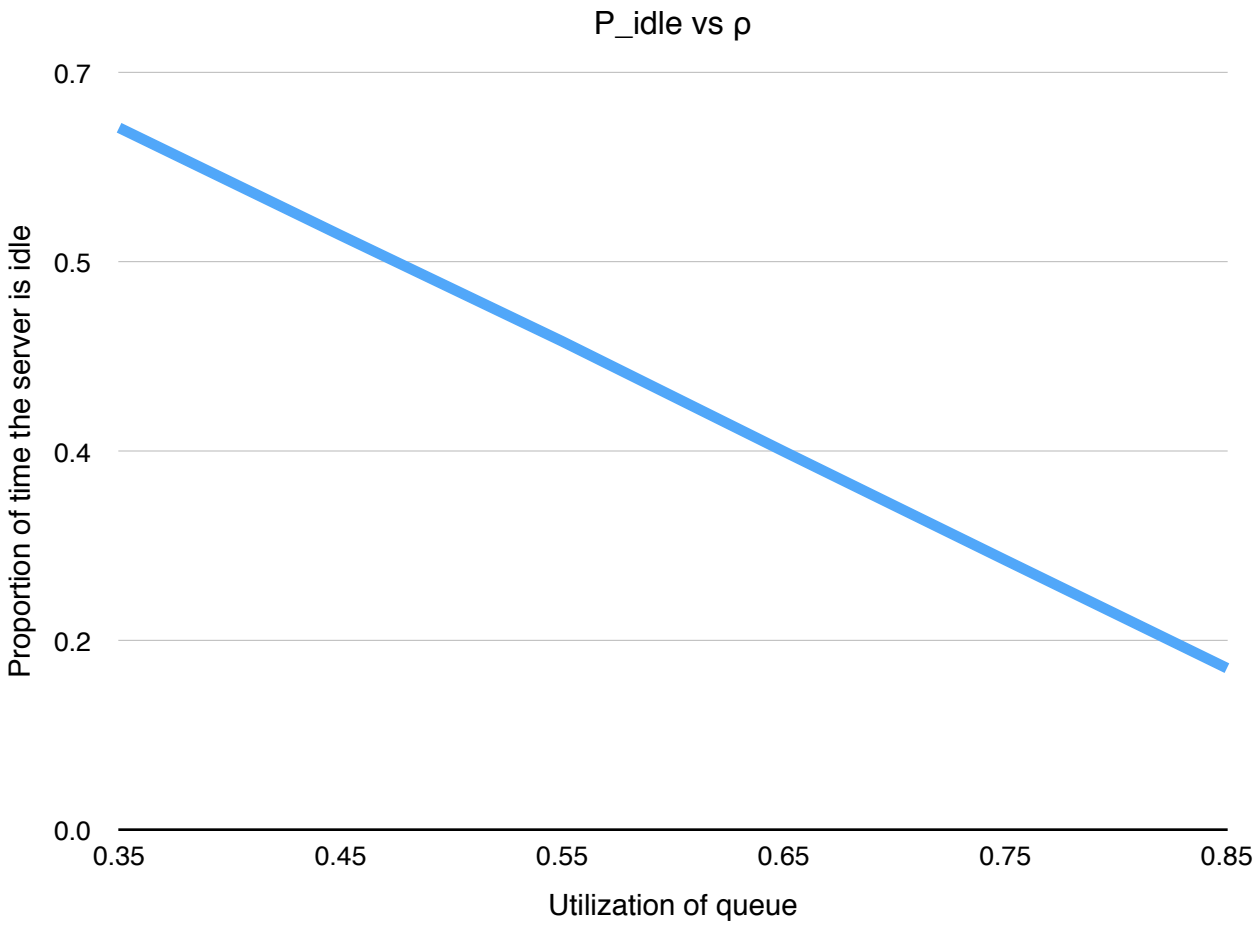

Figure 2: Proportion of time the server is idle vs. Utilization of queue for 0.25 < ρ < 0.95 (step size 0.1)

M/M/1 P_{IDLE} for 0.25 < ρ < 0.95 (step size 0.1)

In Figure 2, it can be seen that as the value of ρ is increased from 0.35 to 0.85, the probability of proportion of time the server is idle almost linearly decreases. This means that as there are more packets generated, the queue is used more, and the time the queue is *not* being used decreases. This is computed by dividing the `num_idle` value by the `num_observations` value to get the proportion of total time the server remains idle (and the queue, empty).

Question 4

For the same parameters as question 3, simulating for ρ significantly increased the number of packets in the system at a given time, resulting in the average number of packets in system to significantly increase. In addition, due to the extremely large number of packets in the system, the system is virtually never in ideal state anymore. This implies that the system is always running and there is a constant stream of continuous packets waiting to be served by the system. In such a situation, the output is not always stable, since ρ is a ratio between arrival rate and service rate, having ρ = 1.2, implies that there are more arrivals generated than the service rate of the system. In this case, the system would find it difficult to keep up with that stream of arrivals and as a result, could destabilize the output of the average number of packets in the system.

Question 5

Although already mentioned in Question 2, the `num_dropped` variable was introduced to realize a M/M/1/K queue system. This was used to count the number of times a packet was dropped by the system due to the buffer size (provided to the system as K) being unable to accept more packets. Other than that, no new variables were added for M/M/1/K that weren't already present in the M/M/1 queue system for a infinite buffer. In addition, now the buffer size was being checked when determine if a packet can be serviced or not. In such a case, a non-negative, integer value for the buffer size K must be provided to the system. As can be seen below in the code fragment:

```
if (num_packet_arrivals - num_packet_departures <= buffer_size || buffer_size == 0) {…} else { // Drop the packet }
```

If the system is able to service the packet, then the queue size must be less than or equal to the buffer size, otherwise, the system is unable to service the packet, since the queue is busy and the buffer is full. There is no way this packet can be service, and hence, must be dropped.

Question 6

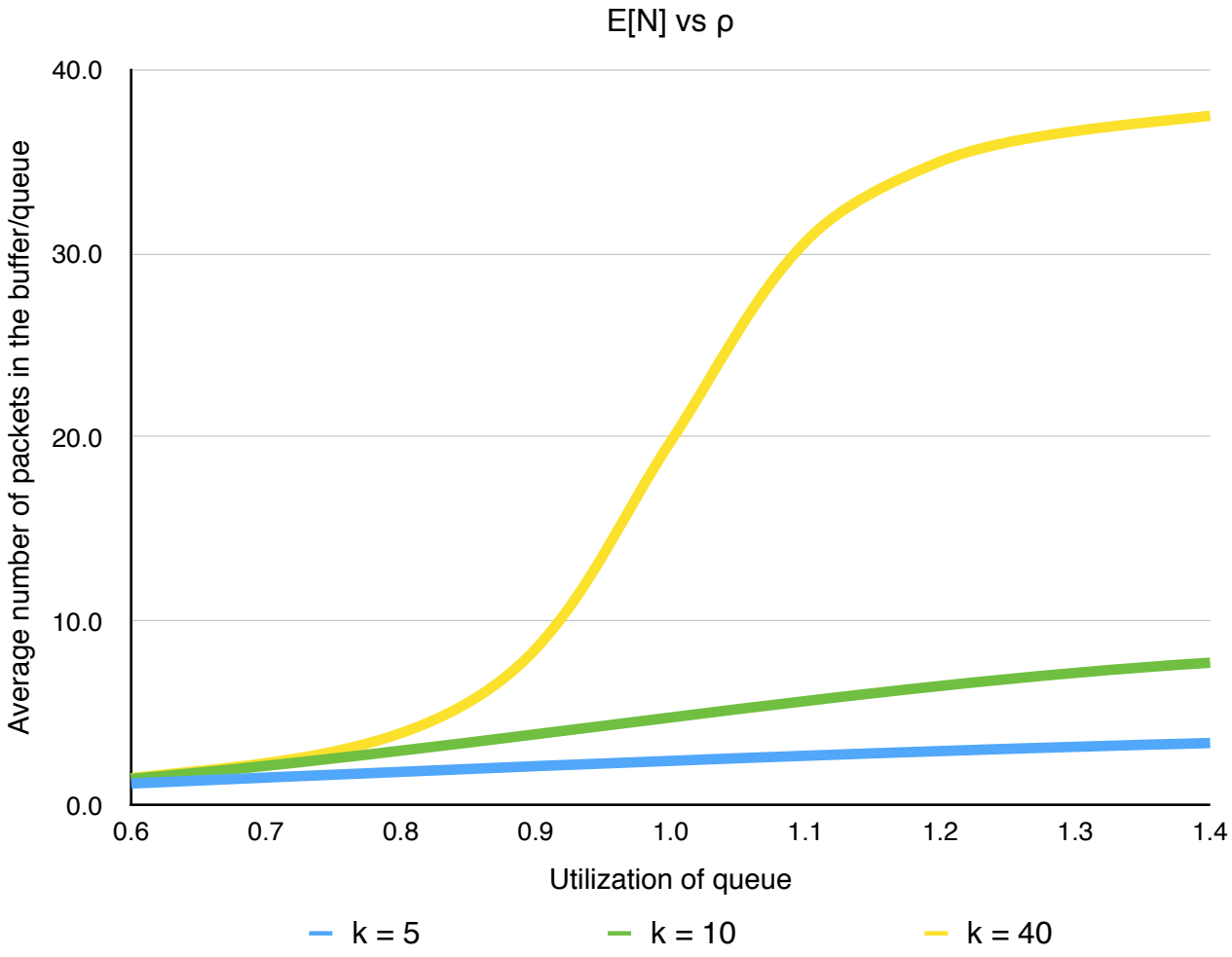

Figure 3: Average number of packets vs. Utilization of queue for $0.5 < \rho < 1.5$ (step size 0.1; k = 5, 10, 40)

In the E[N] vs. ρ plot in Figure 3 above, it can be seen that as the value of ρ is increased from 0.6 all the way to 1.4 (within the desired range of 0.5 and 1.5), the average number of packets in the system increases. Although this behavior was also observed in the $k = \infty$ case, the implication of this is different for the $k \neq \infty$ case. The value of E[N] for a finite buffer system increases much slower than in the infinite an infinite buffer system. The averages appear to asymptotically converge to (or saturate to) the buffer size. That is, for a k = 5 buffer system, the value of E[N] with increasing ρ appears to converge to E[N] = 5. This phenomenon is also observed for the k = 10 and k = 40 buffer size case. The k = 10 case exhibited a similar pattern as k = 5, where the average number of packets increased at a greater rate than the k = 5 case. However, for the k = 40 case, the pattern was slightly different from the k = 5 and k = 10 case. For k = 40, the general pattern of increasing E[N] with increasing ρ was observed, however, for $0.6 \leq \rho \leq 0.9$ the rate of increase for the average number of packets in the buffer/queue is low, with a significant increase in rate for $0.9 < \rho \leq 1.1$, followed by a decline in rate of increase of E[n] for $1.1 < \rho \leq 1.4$. Upon ρ reaching 1.4, the rate of increase decreases because the saturation point is approaching, for E[N], which is 40. Therefore, it can be concluded that for large ρ, the average number of packets in the buffer/queue saturates to the buffer size.

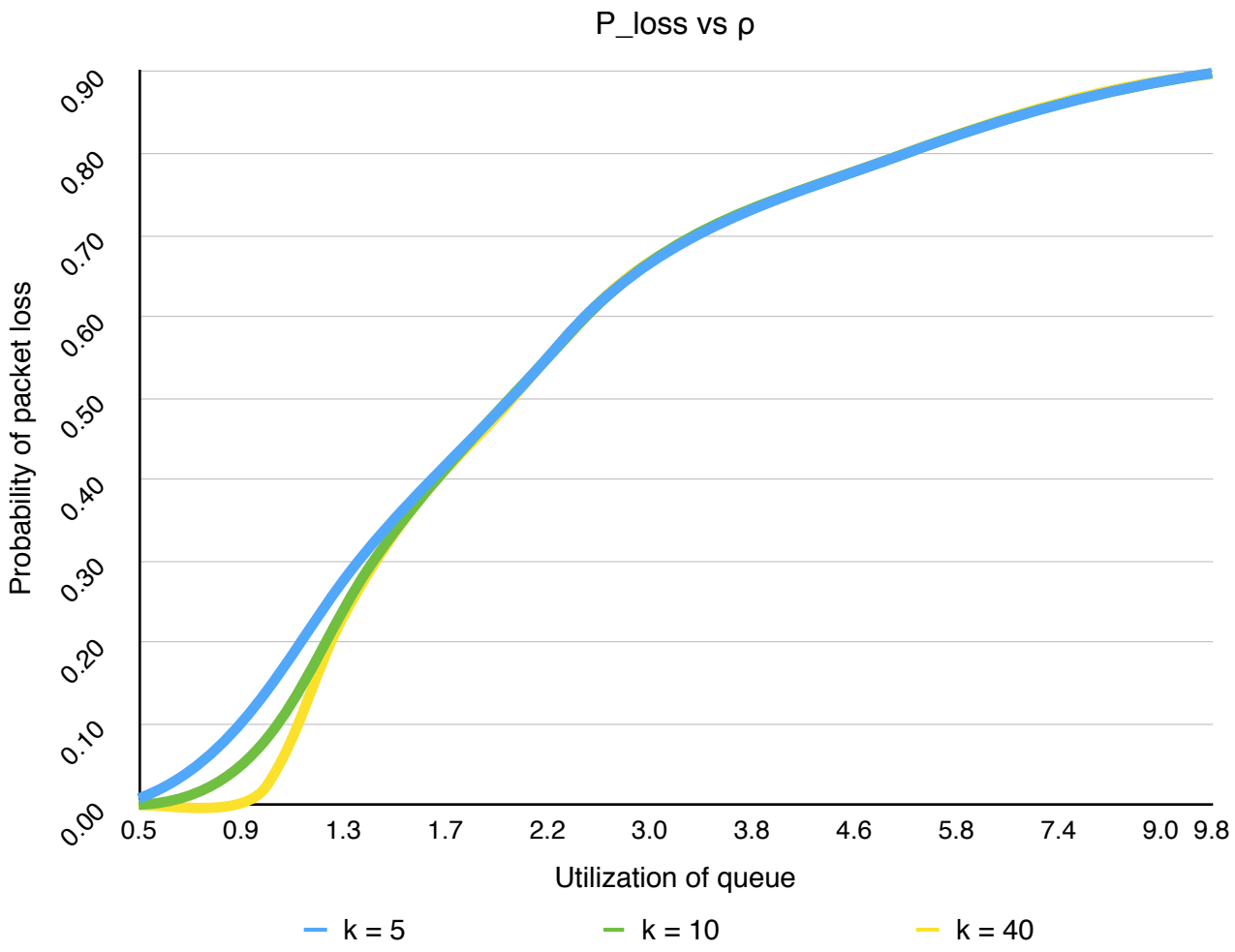

Figure 4: Probability of packet loss (for varying ρ; k = 5, 10, 40)

In the P_{LOSS} vs. ρ plot in Figure 4 above, it can be seen that as the value of ρ is increased from 0.6 all the way to 9.8, the probability of packet loss generally increases for all buffer sizes of interest (i.e. k = 5, k = 10, k = 40). The probably of packet loss initially starts extremely low at values generally near 0%. This is due to the value of ρ being low, implying that there aren't as many packets being generate to really stress the queue and the buffer system. There is packet loss, but that loss is negligible since the queue is able to handle most of the packets generated and service them in a timely manner. However, it can be observed that for k = 5, since the buffer size is small, there is an increase in packet loss probability much earlier (i.e., at a lower ρ value than a k = 40 buffer). In contrast, having a larger buffer for packets to wait while the queue is busy allows for less packet loss, when the system is being stressed by a continuous influx of packets arriving. For the k = 10 buffer size case, it is in the middle of k = 5 and k = 40 buffer size cases, since it is still a smaller sized buffer than a k = 40 buffer, it starts to lose packets fast than the k = 40 buffer but slower than the k = 5 buffer. In addition, for increasing ρ, it can be observed that the probability of packet loss generally increases smoothly and starts to slow down while approaching a packet loss probability of 100%. However, it will take much longer (if at all) for the system to drop every single packet, the system does converge to a packet loss probability of ~90% towards ρ = 9.8. Therefore, for low ρ, the probability a packet will be dropped is low which increases rapidly until slowing down reaching a high ρ and a packet loss probably of 90%.