ECE358: Computer Networks

Winter 2014

Project 1: Queue Simulation

Date of submission: February 3, 2014

Submitted by:

Student ID: 20390063

Student name: Cheung, Justin

Waterloo email address: j43cheun@uwaterloo.ca

Marks received:

Marked by:

Table of Contents

Question 1

How to generate an exponential r.v. with parameter λ from $U(0,1)$? Show all your calculations. Write a short piece of C code to generate 1000 exponential r.v.

To generate an exponential random variable X with parameter λ from a uniform random variable $U(0,1)$, the inverse of the exponential cumulative distribution function, $X = F^{-1}(U)$, must first be determined. In particular, the cumulative distribution function for an exponential distribution with parameter λ is given by the following formula:

$$F(a) = P\{X \leq a\} = 1 - e^{-\lambda a} \tag{1.1}$$

Given formula (1.1), the inverse of the exponential cumulative distribution function, $X = F^{-1}(U)$, can be evaluated as follows:

$$U = 1 - e^{-\lambda X}$$

$$1 - U = e^{-\lambda X}$$

$$\ln(1 - U) = -\lambda X$$

$$X = F^{-1}(U) = -\frac{\ln(1-U)}{\lambda} \tag{1.2}$$

Using formula (1.2), a uniform random variable $U(0,1)$ can be used as input to generate an exponential random variable X. This is ideally how exponential random variable generation should be implemented within the Queue Simulator.

Implementation-wise, the Queue Simulator uses a modified approach for generating exponential random variables. In particular, conventional workstations are incapable of non-determinism. Hence, the uniform random variable $U(0,1)$ is generated using a time-seeded pseudorandom number generator (i.e., gcc's rand() function) in place of an actual random number generator. In addition, rather than explicitly using formula (1.2) to generate an exponential random variable X for some given $U(0,1)$, the Queue Simulator computes X using the following formula:

$$X = F^{-1}(U) = -\frac{\ln(U)}{\lambda} \tag{1.3}$$

It turns out that it does not really matter whether we use formula (1.2) or (1.3) for the generation of exponential random variable X, as both $U(0,1)$ and $1 – U(0,1)$ are in theory, uniform on [0,1]. Since we use formula (1.3) rather than formula (1.2) for generating exponential random variable X, however, the Queue Simulator enforces that $U(0,1)$ must exist in the range of (0,1]. By formula (1.3), this constraint is necessary to ensure that exponential random variable X does not blow up to infinity.

To enforce that $U(0,1)$ exists in the range of (0,1], the Queue Simulator will always add 1 to the pseudorandom value generated through gcc's rand() function. That way, the result will always be greater than 0. This pseudorandom value is then normalized in the range of (0,1] by dividing it by RAND_MAX + 1, where RAND_MAX is the upper bound on the infinite set of all possible pseudorandom numbers generated by gcc's rand() function. The reason why we divide the pseudorandom value by RAND_MAX + 1 rather than RAND_MAX is because we previously added 1 to the pseudorandom value generated by gcc's rand() function. Hence, it is possible for the pseudorandom value to exceed RAND_MAX by 1. Given the normalized pseudorandom value, which we shall now refer to as $U(0,1)$, X is then computed using formula

(1.3) and the resulting floating point value is returned to the calling function.

For completeness, the C++ source code that implements the exponential random variable generator called by the Queue Simulator is defined in the Random.h header file and implemented in Random.cpp source file. In particular, the exponential random variable generator is implemented as a function called generateExponentialRandomVariable(), which accepts a floating point argument called “parameter”. In this context, “parameter” is equivalent to λ. In addition, access to a subroutine called QuestionOne() in the Questions.cpp source file is documented in the provided README.txt file. In particular, running this subroutine will generate 1000 exponential random variables for parameter λ = 75 using the aforementioned generateExponentialRandomVariable() function. The subroutine will then compute and output the ideal expected value and ideal variance (based on parameter λ), and the actual expected value and actual variance (based on the 1000 generated exponential random variables). You should notice that the actual values for expected value and variance do not differ by much from their ideal counterparts.

Question 2

Build your own simulator for this queue and explain in words what you have done. Show your code in the report. In particular, define your variables.

The Queue Simulator discussed in this report implements Discrete Event Simulation (DES) using the notion of an Event Scheduler (ES), as specified in the lab manual. To make things more intuitive, our discussion of the M/M/1 Queue Simulator implementation will be broken down into three parts: simulation of an event, simulation of an Event Scheduler and simulation of the queue. For completeness, all classes central to the architecture of the Queue Simulator have been defined in the Event.h and Event.cpp header and source files respectively. In particular, Event.h defines the interface to the Queue Simulator (i.e., important member variables), while Event.cpp defines the actual implementation of the Queue Simulator.

We first start our discussion with the simulation of an event. In particular, the Queue Simulator implements an Event class (see Event.h and Event.cpp) for simulating observer, arrival and departure events. The Event class contains two member variables of interest: m_eventType and m_time. m_eventType is an instance of the EventType enumerated type (see Event.h) that can take on one of three values: OBSERVER, ARRIVAL or DEPARTURE. It is used strictly for identifying whether the Event instance corresponds to an observer event, arrival event or departure event respectively. Meanwhile, m_time stores a floating point representation of the time assigned to the event. The Event class also contains member variables m_previousEvent and m_nextEvent; these member variables are pointers to events preceding and succeeding the event in question and are used by the EventQueue class (see Event.h and Event.cpp) for ordering events in FIFO order. It is important to note here that the EventQueue class alone does not represent the aforementioned simulated queue. Rather, the EventQueue class defines a queue data structure for organizing scheduled events as part of the Event Scheduler.

The next part of our discussion deals with the simulation of an Event Scheduler. In particular, the Queue Simulator implements an EventScheduler class (see Event.h and Event.cpp), which is responsible for simulating the event scheduler and queue. In terms of event scheduling, the EventScheduler class contains the following member variables of interest: m_T, m_lambda, m_L, m_alpha, m_C, m_Na, m_Nd, m_previousObserverTime, m_previousArrivalTime, m_previousDepartureTime, m_observerEventQueue, m_arrivalEventQueue and m_departureEventQueue. The definitions of these member variables are defined in Table 2-1 below.

In terms of generating events, the EventScheduler class has dedicated functions for generating observer and arrival events. An observer event is generated when the EventScheduler class's simulate() function calls the generateObserverEvents() function. Likewise, an arrival event is generated when the EventScheduler class's simulate() function calls the generateArrivalEvents() function. Functionality-wise, the two functions are identical. In particular, the generateObserverEvents() function will first check if m_previousObserverTime is less than or equal to m_T. If this is true, it will pass m_alpha as a parameter into the generateExponentialRandomVariable() function to generate an exponential random interarrival time. This exponential random interarrival time is then added to m_previousObserverTime and the resulting time value is assigned to a newly instantiated observer event. The observer event is then enqueued to m_observerEventQueue. The generateArrivalEvents() function performs the same process, except that it operates on m_previousArrivalTime, m_lambda and m_arrivalEventQueue as opposed to m_previousObserverTime, m_alpha and m_observerEventQueue respectively.

Table 2-1: "EventScheduler" class member variables of interest for event scheduling.

Member Variable	Definition
m_T	A floating-point type variable that stores the period in seconds over which the experiment is run.
m_lambda	A floating-point type variable that stores the average number of packets generated or arrived per second. It is used as a parameter for generating exponential random interarrival times of arrival events.
m_L	A floating-point type variable that stores the average length of a packet in bits. It is used as part of the parameter, 1.0/m_L, for generating exponential random packet lengths.
m_alpha	A floating-point type variable that stores the average number of observer events per second. It is used as a parameter for generating the exponential random interarrival times of observer events.
m_C	A floating-point type variable that stores the transmission rate of the output link in bits per second. It is used to compute departure times.
m_Na	An integer type variable that stores the number of arrival events that have occurred thus far.
m_Nd	An integer type variable that stores the number of departure events that have occurred thus far.
m_previousObserverTime	A floating-point type variable that stores the time corresponding to the previously generated observer event. When m_previousObserverTime exceeds m_T, generation of observer events stop.
m_previousArrivalTime	A floating-point type variable that stores the time corresponding to the previously generated arrival event. When m_previousArrivalTime exceeds m_T, generation of arrival events stop.
m_previousDepartureTime	A floating-point type variable that stores the time corresponding to the previously generated departure event. It is used to compute departure times when the queue is busy.
m_observerEventQueue	A pointer to an event queue for storing observer events in FIFO order.
m_arrivalEventQueue	A pointer to an event queue for storing arrival events in FIFO order.
m_departureEventQueue	A pointer to an event queue for storing departure events in FIFO order.

In contrast, departure events are generated by the EventScheduler class's arrivalEventHandler() function (see Event.h and Event.cpp). Whenever the simulate() function calls the arrivalEventHandler() function, the arrivalEventHandler() function will compute the number of packets in the system. For the M/M/1 queue, the number of packets in the system is given by the following formula:

$$Buffered\ Packets = m\_Na - m\_Nd \tag{2.1}$$

The arrivalEventHandler() function then proceeds to generate an exponential random packet length by passing the parameter 1.0/m_L into generateExponentialRandomVariable(). In turn, the exponential random packet length is used to compute the transmission delay of the packet corresponding to the argument arrival event. In particular, the transmission delay is given by the following formula:

$$Transmission\ Delay = Packet\ Length/m\_C \tag{2.2}$$

Finally, the arrivalEventHandler() function checks if the queue is busy or idle. If the queue is idle, the departure time will be computed using formula (2.3) below. Otherwise, the departure time will be computed using formula (2.4) below. Note that the queue is idle if and only if the number of packets in the system is zero.

$$Departure\ Time = arrival\ event\ time + transmission\ delay \tag{2.3}$$

$$Departure\ Time = m\_previousDepartureTime + transmission\ delay \tag{2.4}$$

Once the departure time has been computed, the arrivalEventHandler() function will then assign this departure time to a newly instantiated departure event. The newly instantiated departure event is then enqueued to m_departureEventQueue.

The final part of our discussion deals with the simulation of the queue. This part of the discussion brings together event simulation and event scheduler simulation from the previous two parts of our discussion. For queue simulation specifically, consider the member variables of interest for event scheduling in Table 2-1, as well as the member variables of interest for queue simulation in Table 2-2.

Table 2-2: "EventScheduler" member variables of interest for queue simulation

Member Variable	Definition
m_No	An integer type variable that stores the number of observer events that have occurred thus far.
m_Ni	An integer type variable that stores the number of observer events that have occurred when the system was idle. It is used to compute the proportion of time the server is idle.
m_aggregateBufferedPackets	An integer type variable that stores the aggregate sum of the number of packets observed in the system by each observer event. It is used to compute the average number of packets in the buffer or queue.

It is important to note here that the Queue Simulator discussed in this report rejects the strategy presented in the lab manual for simulating queues. Due to memory constraints, the Queue Simulator does not schedule observer, arrival and departure events ahead of time. Instead, observer, arrival and departure events are scheduled as queue simulation occurs. In particular, queue simulation starts when some subroutine instantiates an EventScheduler object and calls its simulate() function. The simulate() function will initially generate an observer event and an arrival event so that the event queues are not empty. The simulate() function will then enter a while loop that terminates when all three event queues are empty. Within the scope of this while loop, a single event will be dequeued in each iteration from one of the three event queues. The event that is dequeued is the event with the smallest m_time of the three events at the front of the each of the three event queues respectively.

Depending on the type of the event dequeued, the simulate() function will call one of three event handlers. If the event that was dequeued was an observer event, the simulate() function will call the observerEventHandler() function (see Event.h and Event.cpp). If the event that was dequeued was an arrival event, the simulate() function will call the arrivalEventHandler() function. Finally, if the event that was dequeued was a departure event, the simulate() function will call the departureEventHandler() function (see Event.h and Event.cpp). Conceptually, each of these event handlers represents a specific operation on the queue. The observerEventHandler() function represents a random observer. Meanwhile, the arrivalEventHandler() function represents the arrival of a packet at the back of the buffer. Finally, the departureEventHandler() represents the exiting of a packet from the queue after it has been serviced.

In the observerEventHandler() function, the state of the queue is recorded. In particular, the number of packets in the queue is computed using formula (2.1) for M/M/1 queues and added to m_aggregateBufferedPackets. Next, m_No is incremented to indicate that an observer event has occurred. Finally, if the system happens to be idle, m_Ni is incremented. In the arrivalEventHandler() function, a departure event for the packet corresponding to the argument arrival event is generated as mentioned previously. In addition to this, m_Na is incremented to indicate that an arrival event has occurred. In the departureEventHandler() function, m_Nd is incremented to indicate that a departure event has occurred. Other than this, nothing else happens in the departureEventHandler() function.

When the dequeued event has been properly handled, control returns back to the EventScheduler class's simulate() function. At this point, the simulate() function makes calls to the generateObserverEvents() and generateArrivalEvents() functions. Provided that m_previousObserverTime or m_previousArrivalTime have not exceeded m_T, generateObserverEvents() or generateArrivalEvents() will continue to generate observer or arrival events respectively. Conversely, if m_previousObserverTime and m_previousArrivalTime have exceeded m_T, generation of observer and arrival events will stop, after which only dequeuing will occur for each while loop iteration until all three event queues are empty. When all three event queues are empty, the simulate() function will exit the while loop and call the EventScheduler class's benchmark() function (see Event.h and Event.cpp). The benchmark() function will use formulas (2.5), (2.6) and (2.7) below to compute the queue utilization, the average number of packets in the buffer or queue and the proportion of time the server is idle respectively.

$$\rho = m\_L \times \frac{m\_lambda}{m\_C} \tag{2.5}$$

$$E[N] = \frac{m\_aggregateBufferedPackets}{m\_No} \tag{2.6}$$

$$P_{IDLE} = \frac{m\_Ni}{m\_No} \tag{2.7}$$

Question 3

Assume L = 12,000 bits, C = 1 Mbits/second and give the following figures using the simulator you have programmed. Provide comments on all your figures:

1. E[N], the average number of packets in the system as a function of ρ (for $0.25 < \rho < 0.95$, step size 0.1). Explain how you do that.

The average number of packets E[N] as a function of the queue utilization ρ (for $0.25 < \rho < 0.95$, step size 0.1) is determined using the questionThree() subroutine (see Questions.h and Questions.cpp) in the Queue Simulator project. In particular, questionThree() generates the function by iterating over ρ from 0.25 to 0.95 by a step size of 0.1. For each value of ρ, the questionThree() subroutine computes λ by isolating for λ in formula (2.5) (see Question 2):

$$\lambda = \frac{\rho C}{L} \tag{3.1}$$

where L and C are given above. In turn, α is computed by multiplying λ by a factor of 8. Formally, α is computed as follows:

$$\alpha = 8 \times \lambda \tag{3.2}$$

Afterwards, an EventScheduler object is instantiated using arguments T = 20,000 seconds, λ, L, α, C and K = -1. In this case, T = 20,000 seconds is the period over which the queue simulation should be run for, while K = -1 is the maximum buffer size (a negative K indicates that the buffer size is unbounded). The simulate() method is then called on the "EventScheduler" instance to simulate the queue for the given parameters. The simulation process occurs as described in Question 2 and an output E[N] is obtained for the current value of ρ. To reiterate, E[N] is computed using formula (2.6) (see Question 2). The resulting function is shown below in Figure 3-1.

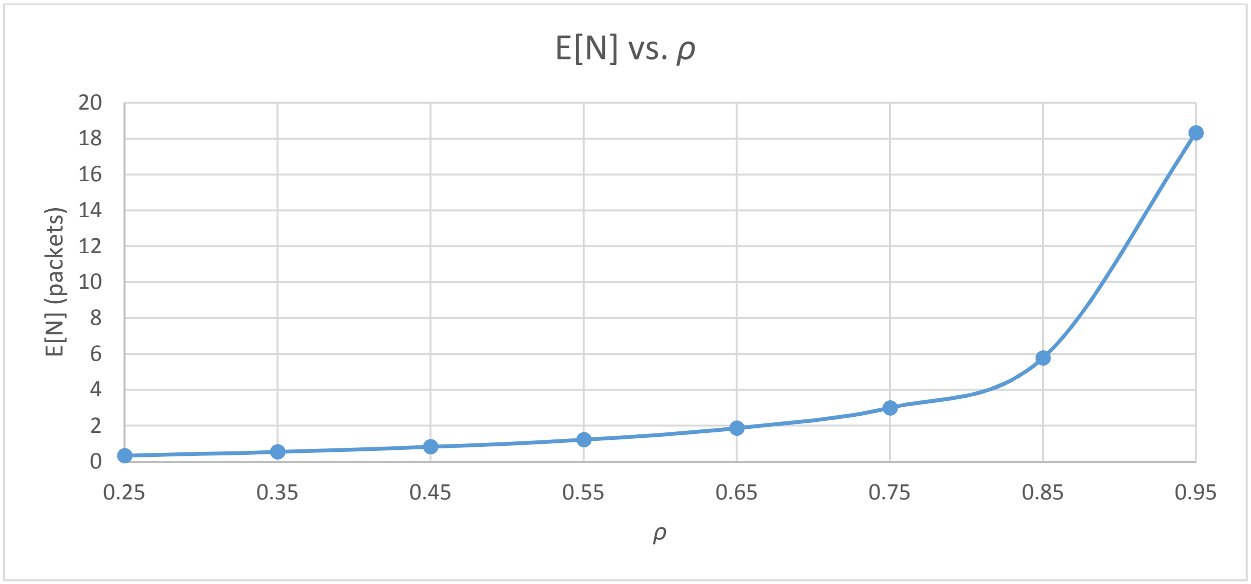

Figure 3-1: A plot of E[N] vs. ρ for M/M/1 queue

Based on Figure 3-1, it is evident that the average number of packets in the buffer E[N] increases exponentially for increasing queue utilization ρ from 0.25 to 0.95. This trend in E[N] vs. ρ makes sense by definition of ρ as the queue utilization. In particular, since m_L and m_C (corresponding to L and C respectively) are constant in formula (2.5), this implies that the queue utilization is directly proportional to λ (recall that λ is the average number of packets generated or arrived per second). At the same time, exponential random packet interarrival time X in formula (1.2) is inversely proportional to λ. Hence, an increase in ρ will result in a decrease in the packet interarrival time X. Since the exponential random packet interarrival time X shortens for increasing ρ, this implies that the number of simulated packet arrival events will increase for increasing ρ. Given that the number of arrival events increases exponentially faster than the rate at which packets are serviced for the M/M/1 queue, this would explain why E[N] as a function of ρ appears to be exponential.

To ensure that the simulation results are stable for the given period T, the method described in the lab manual is used. In particular, stability was verified by running the simulation described in Question 2 for T = 10,000, 2T = 20,000, 3T = 30,000, etc. and discrete values of ρ in the range of [0, 1]. In both circumstances, the values of E[N] for *2T*, 3T, 4T, etc. for each discrete ρ was noted to differ by less than 5%, thus indicating stability. Based on this result, T = 20,000 will be used to run the simulations described in Question 3, Question 4 and Question 6 of this report.

2. P_{IDLE}, the proportion of time the system is idle as a function of ρ (for $0.25 < \rho < 0.95$, step size 0.1). Explain how you do that.

The proportion of time the system is idle P_{IDLE} as a function of the queue utilization ρ (for $0.25 < \rho < 0.95$, step size 0.1) is computed using the same subroutine and parameters from part 1 of Question 3. During queue simulation, recall that formula (2.7) from Question 2 is used to compute P_{IDLE}. The details for how P_{IDLE} is computed are also provided in the explanation of the Queue Simulator implementation in Question 2 (as with E[N], P_{IDLE} is also computed in benchmark()). The resulting function is shown below in Figure 3-2.

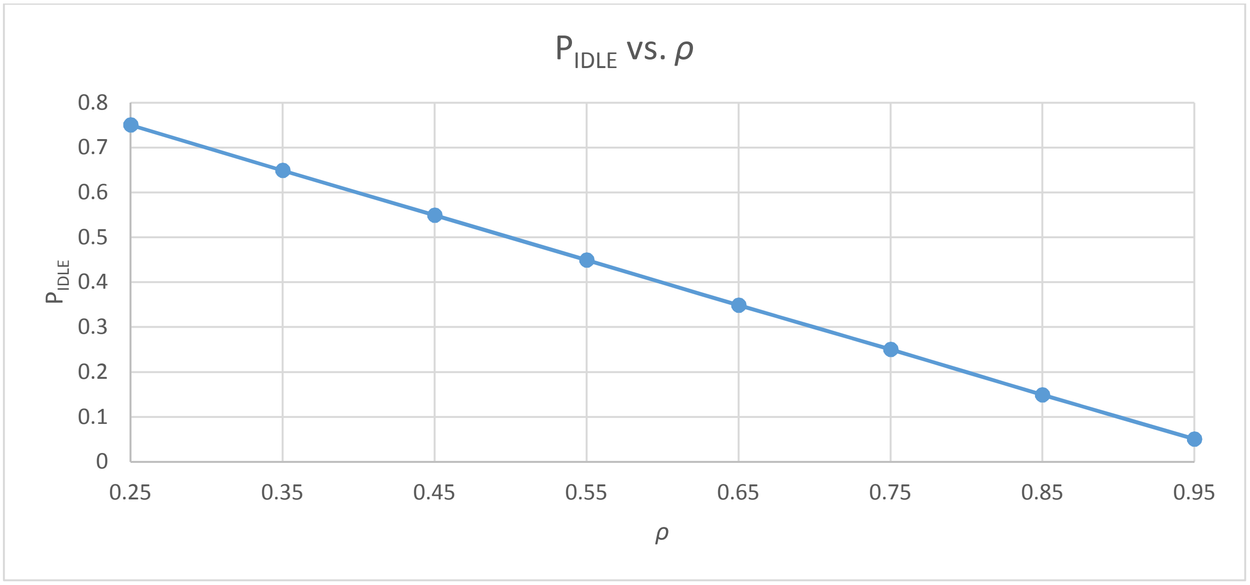

Figure 3-2: A plot of P_{IDLE} vs. ρ for M/M/1 queue

Based on Figure 3-2, P_{IDLE} as a function of ρ appears to be affine. In particular, Figure 3-2 suggests that P_{IDLE} is related to ρ by the following formula:

$$P_{IDLE} = 1 - \rho \tag{3.3}$$

Formula (3.3) suggests that ρ, as the queue utilization, actually represents the proportion of time (for ρ in the range of [0,1]) in which the queue is busy or being utilized by definition of P_{IDLE}.

Question 4

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

In Question 4, a subroutine in the Queue Simulator project called QuestionFour() (see Questions.h and Questions.cpp) is used to perform queue simulation for $\rho = 1.2$ using the same parameters defined in Question 3. In this case, the test process defined for QuestionFour() is similar to the test process defined for QuestionThree() with the exception that ρ is fixed to 1.2 (there is no while loop).

In order to observe what happens when queue simulation is performed for $\rho = 1.2$, the test process defined in QuestionFour() was run manually for T = 10,000 seconds, 20,000 seconds, 30,000 seconds, 40,000 seconds and 50,000 seconds. The results of the queue simulation for $\rho = 1.2$ are shown in Table 4-1 below.

Table 4-1: Simulation results for E[N] and P_{IDLE} when $\rho = 1.2$

***T* (seconds)**	**E[N] (packets)**	**P_{IDLE}**
10,000	83744.5	0.00000212549
20,000	165105	0.00000318775
30,000	247916	0.000000416513
40,000	332175	0.00000121894
50,000	417602	0.00000222489

Based on Table 4-1, it is evident that the results for E[N] are unstable (the values differ by more than 5% between T, $2T$, $3T$, $4T$ and $5T$) and blow up for $\rho = 1.2$. This makes sense, given the behaviour of E[N] vs. ρ and P_{IDLE} vs. ρ in Question 3. In particular, for P_{IDLE} vs. ρ, ρ in the range of [0,1] was found to represent the proportion of time in which the queue is busy. Meanwhile, for E[N] vs. ρ, E[N] was found to increase exponentially for increasing ρ in the range of [0,1]. Consequently, when ρ is set to a value that exceeds 1, the M/M/1 queue is no longer able to keep up with the influx of packet arrival events. That is, there are significantly more packets entering than leaving the queue when $\rho = 1.2$. As a result, for $\rho > 1$, the value of E[N] blows up and is no longer consistent for T, $2T$, $3T$, $4T$, $5T$, etc. Instead, it appears that E[N] becomes proportional to T for $\rho > 1$. In addition, P_{IDLE} is basically zero for $\rho = 1.2$, as the queue is "over-utilized".

Question 5

Build a simulator for an M/M/1/K queue. Explain what you had to do to modify the previous simulator and what new variables you had to introduce. Show your code in the report. (Hint: do not forget that dropping a packet is NOT considered an event)

To modify the previous Queue Simulator implementation such that it supports simulation of M/M/1/K (finite) queues, member variables m_K and m_Nl were added to the EventScheduler class. In particular, m_K is the maximum number of packets that can be queued. Meanwhile, m_Nl is an integer that keeps count of the number of packet arrival events that have occurred in which the packet was dropped.

In addition, modifications were also made to the arrivalEventHandler() function. To deal with the notion of dropped packets, the arrivalEventHandler() function now checks whether the queue is full or not. To check whether the queue is full or not, the arrivalEventHandler() function evaluates the condition below:

$$(m\_K < 0)\ OR\ (NOT(m\_Na \geq m\_Nd\ + m\_Nl + m\_K)) \tag{5.1}$$

In particular, the condition in (5.1) will check if m_K is less than 0 (indicating that the buffer is infinite) or if the number of buffered packets is less than m_K for the case where m_K ≥ 0. If the condition in (5.1) is satisfied, a departure event is generated as described in Question 2. Otherwise, m_Nl is incremented by 1 to indicate that the packet that has just arrived has been dropped. In both cases, m_Na is incremented by 1 at the end of the arrivalEventHandler() function to indicate that a packet arrival event has occurred (this is carried over from the Queue Simulator implementation described in Question 2).

Finally, the benchmark() function in the "EventScheduler" class has been modified to include an additional computation for P_{LOSS}. In particular, the following formula is used to compute P_{LOSS}:

$$P_{LOSS} = \frac{m\_Nl}{m\_Na} \tag{5.2}$$

It is important to note that the computation with P_{LOSS} will occur for both M/M/1 and M/M/1/K queues, but P_{LOSS} will always be zero for M/M/1 queues.

Question 6

Let L = 12000 bits and C = 1 Mbits/second. Use your simulator to obtain the following figures (provide comments for each figure):

1. E[N], the average number of packets in the system as a function of ρ (for 0.5 < ρ < 1.5, step size 0.1) for K = 5, 10 and 40 packets (One curve per value of K on the same graph). Compare it with the case K = ∞.

The average number of packets in the system E[N] as a function of the queue utilization *ρ* (for 0.5 < *ρ* < 1.5, step size 0.1) for K = 5, 10 and 40 packets, *T* = 20,000, *α* = 8*λ* (recall that λ is computed using formula (3.1) at each *ρ*), L = 12,000 bits and C = 1 Mbits/second is obtained using the questionSixPartOne() subroutine (see Questions.h and Questions.cpp). The process carried out by this function is similar to the process described in part 1 of Question 3, except for the fact that it implements three separate while loops for K = 5, K = 10 and K = 40 packets respectively. The resulting function is shown in Figure 6-1 below:

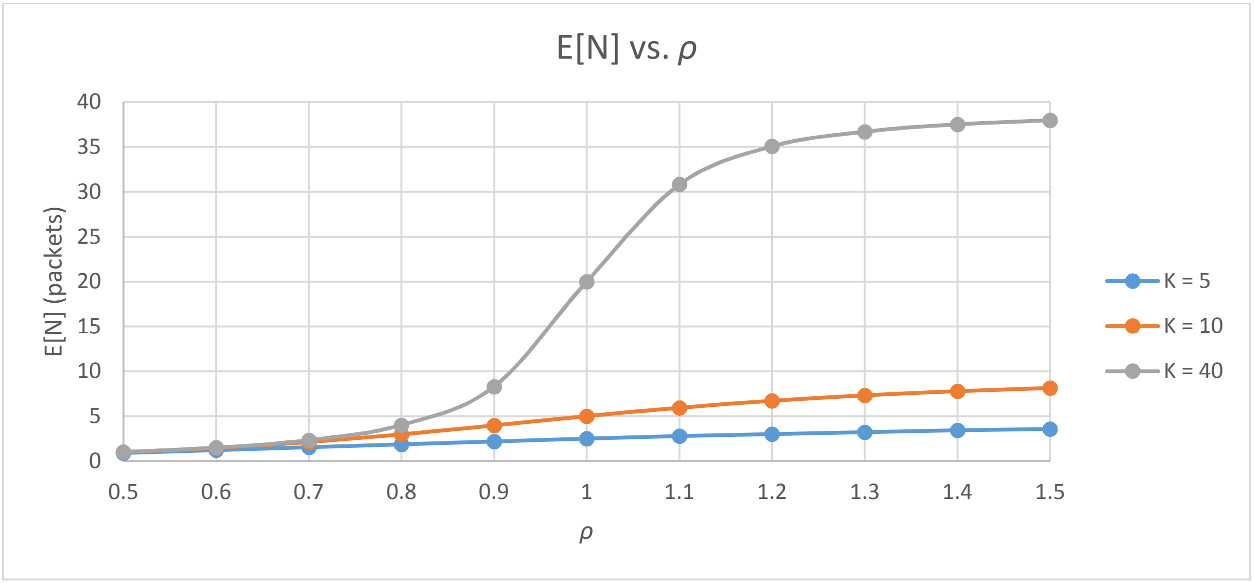

Figure 6-1: A plot of E[N] vs. ρ for M/M/1/5, M/M/1/10 and M/M/1/40 queues

As shown in Figure 6-1, the plots for E[N] vs. *ρ* for the M/M/1/5, M/M/1/10 and M/M/1/40 queues resemble phase-shifted hyperbolic tangent functions, especially in the case of the M/M/1/40 queue. In contrast, the plot for E[N] vs. *ρ* for the M/M/1 queue in Figure 3-1 resembles an exponential function. A hyperbolic tangent-shaped curve makes sense for finite queues. In particular, notice that for smaller values of *ρ*, the curves of E[N] vs. *ρ* in Figure 6-1 almost resemble the curve of E[N] vs. *ρ* in Figure 3-1. This resemblance for smaller values of *ρ* occurs because the queues in question are not full yet. As the value of *ρ* is increased to infinity, however, the curves of E[N] vs. *ρ* in Figure 6-1 converge towards each of their respective maximum buffer sizes, as the finite queues cannot store more than *K* packets. Had the buffer size been unbounded, the curves of E[N] vs. *ρ* in Figure 6-1 would have continued increasing exponentially for increasing *ρ*. Hence, this is why the curves for E[N] vs. *ρ* for the M/M/1/5, M/M/1/10 and M/M/1/40 queues in Figure 6-1 resemble hyperbolic tangent functions rather than exponential functions.

2. P_{LOSS} as a function of ρ (for $0.4 < \rho < 10$) for $K = 5$, 10 and 40 packets. (One curve per value of K on the same figure). Explain how you have obtained P_{LOSS}. Use the following step sizes for ρ:

- For $0.4 < \rho \leq 2$, step size 0.1
- For $2 < \rho \leq 5$, step size 0.2
- For $5 < \rho \leq 10$, step size 0.4

The packet loss probability P_{LOSS} as a function of the queue utilization ρ (for $0.4 < \rho < 10$) for K = 5, 10 and 40 packets, $T = 20{,}000$, $\alpha = 8\lambda$ (recall that λ is computed using formula (3.1) at each ρ), L = 12,000 bits and C = 1 Mbits/second is obtained using the questionSixPartTwo() subroutine in the Queue Simulator project (see Questions.h and Questions.cpp). As with the questionSixPartOne() subroutine in part 1 of Question 6, the questionSixPartTwo() subroutine implements three separate while loops for buffer sizes of $K = 5$, 10 and 40 packets respectively. In addition, the while loops have been modified such that the step size is changed from 0.1 to 0.2 for ρ in the range of (2,5] and from 0.2 to 0.4 for ρ in the range of (5,10]. Finally, P_{LOSS} is computed using the method described in Question 5 (see formula (5.2) in Question 5). The resulting function is shown in Figure 6-2 below.

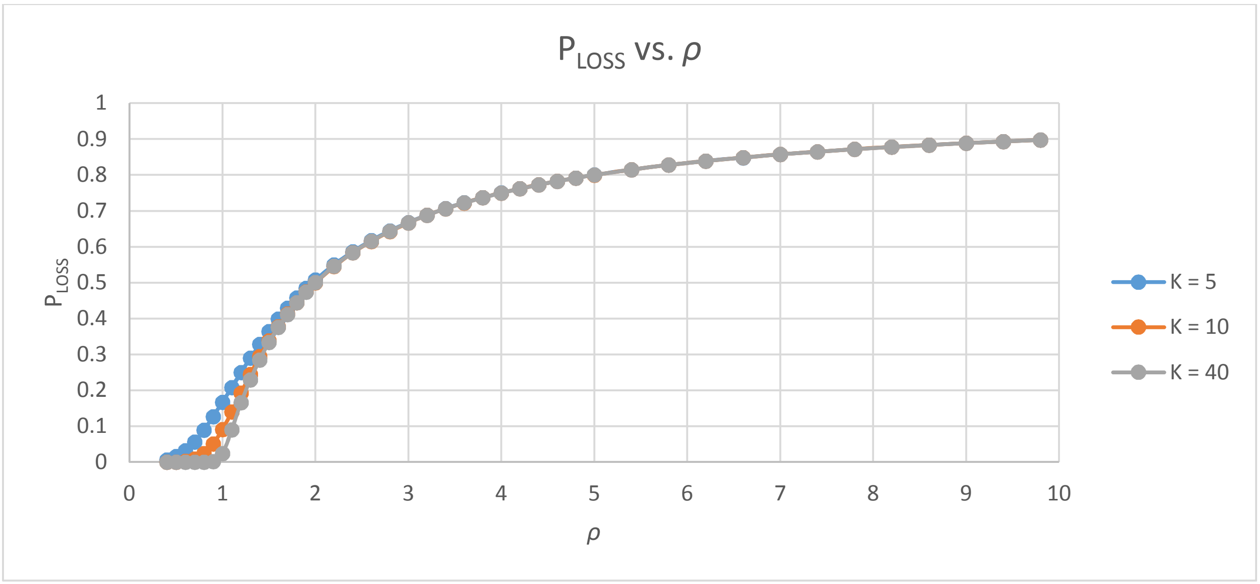

Figure 6-2: A plot of P_{LOSS} vs. ρ for M/M/1/5, M/M/1/10 and M/M/1/40 queues

As shown in Figure 6-2 above, the plots for P_{LOSS} vs. ρ for the M/M/1/5, M/M/1/10 and M/M/1/40 queues start at $P_{LOSS} = 0$ for lower values of ρ and converge towards $P_{LOSS} = 1$ as ρ is increased to infinity. This makes sense, given that the average number of packets generated or arrived per second E[N] increases with ρ. Since the buffer size is fixed at K, this implies that for increasing ρ, the number of dropped packets will increase such that P_{LOSS} converges to 1. In addition, P_{LOSS} stays at 0 for larger ρ for larger buffer size K, since the finite queue in question is able to store a larger number of packets before becoming full. This is evident in Figure 6-2, where the length of the curve where $P_{LOSS} = 0$ is longer for larger K.