University of Waterloo
Faculty of Engineering
Department of Electrical and Computer Engineering

# ECE-358 – Lab 1 Report

Prepared by
Lucas Joseph Chevalier
2063235
ljcheval@edu.uwaterloo.ca
Umesh Dhurvas
udhurvas@edu.uwaterloo.ca
20640168

30 January, 2019

**Table of Contents**

## Question 1.

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

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

float calcErrPercent(float theoretical, float actual){
        return fabs((theoretical)-actual)/theoretical*100;
}

int main(){

        double lambda = 75;
        srand(time(0));
        double sum = 0;
        double nums[1000];

        for (int i = 0; i < 1000; i++) {
                nums[i] = -log(1.0 - ((double)rand())/RAND_MAX) / lambda;;
                sum += nums[i];
        }

        double avg = sum / 1000;
        double stdev = 0;
        for (int i = 0; i < 1000; i++) {
                stdev += pow(nums[i] - avg, 2);
        }

        double variance = sqrt(stdev / (1000-1));

        printf("Expected mean: %f\n", (1/lambda));
        printf("Measured mean: %f\n", avg);
        printf("Error for mean: %0.2f%%\n", calcErrPercent(1/lambda, avg));
        printf("Measured variance %f\n", variance);
        printf("Error for variance: %0.2f%%\n", calcErrPercent(1/lambda,
variance));

}
```

Which produces the output:

Expected mean: 0.013333

Measured mean: 0.013270

Error for mean: 0.48%

Measured variance 0.013440

Error for variance: 0.80%

The theoretical expected mean and variance for set of randomly generated numbers with a Poisson distribution correspond to $\frac{1}{l}$. As shown by the results, the set of randomly generated numbers agree with the expected mean and variance, with an error of 0.48% and 0.80% respectively.

## Question 2.

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

```
public static double[] simulatemm1(double alpha, double lambda, double l, double c,
double t) {
        // List containing arrival, departure and observer events
        LinkedList<Event> eventList = new LinkedList<Event>();
        double queueDelay = 0.0;        // Time to process the elements in queue
        double currentTime = 0.0;       // Current timestamp of simulation
        double delta = 0.0;             // Delay between arrival events
        double serviceTime = 0.0;       // Service delay for a given packet size
        double departureTime = 0.0;     // Timestamp for the departure event
        double observerCount = 0;       // Number of observer events

        // +++++++++++ EVENT GENERATION ++++++++++++++

        while (currentTime < t) { // Generate arrival and departure events
             // Perform Timing calculations for arrival/departure events
            delta = genarateRandom(lambda);
            currentTime += delta;
            serviceTime = generateRandom(1.0 / l) / c;
            queueDelay = Math.max(0, queueDelay - delta);
            departureTime = currentTime + serviceTime + queueDelay;
            queueDelay += serviceTime;

            // Add events with their corresponding timestamp
            Event arrival = new Event("Arrival", currentTime);
            eventList.add(arrival);
            Event departure = new Event("Departure", departureTime);
            eventList.add(departure);
        }
        currentTime = 0.0;
        while (currentTime < t) { //Populate event list with observer events
            currentTime += generateRandom(alpha);
            Event temp = new Event("Observer", currentTime);
            eventList.add(temp);
            observerCount++;
        }

        Collections.sort(eventList, new timeComp()); // Sort list on timestamps

        // +++++++++++ SIMULATION ++++++++++++++

        double queueSize = 0;           // Number of elements in the queue
        double queueSum = 0;            // Sum of number of elements in the queue
        long idleCount = 0;                     // Track number of times queue is idle
(queueSize = 0)
        for (Event e : eventList) {
            if (e.type.equals("Arrival")) {
                queueSize++;
```

```
            } else if (e.type.equals("Departure")) {
                queueSize--;
            } else if (e.type.equals("Observer")) {
                queueSum += queueSize;
                idleCount += (queueSize == 0) ? 1 : 0; //increment if idle
            }
        }

        double avgQueueSize = (queueSum / observerCount);   // Average number of
elements in the queue, E[n]
        double idleFraction = (idleCount / observerCount);  // Fraction of time the
queue is idle

        return new double[] { avgQueueSize, idleFraction };
    }
```

The method *simulatemm1* takes alpha, lambda, *l, c, t* as the inputs, and outputs the idle time and average number of elements in the queue. T corresponds to the total time of the simulation, while the other variables all correspond to the parameters described in the lab manual. Code execution can be broken into two logical components: event generation and simulation.

For event generation, a set of event objects are created and added to a linked list. First, arrival events are generated in chronological order, and a counter is used to record the current timestamp during event generation. Each event is generated with a timestamp that is equal to the previous timestamp plus an exponential random variable generated using lambda. Arrival events continue to be generated until the current timestamp exceeds the simulation time, in which case the loop breaks.

For each arrival event generated, a corresponding departure event is generated as well. The timestamp for the associated departure event is calculated as the sum of the arrival time, the service time and the queue delay. The service time is calculated using the parameter *l* whereas the queue delay represents the amount of time it would take to process the current number of elements in the queue. The queue delay is incremented by the service time of each newly added arrival event and is decremented by the time between successive arrival events.

After the arrival and departure events have been generated, the observer events are generated. The procedure for the generation of observer events is similar to that of arrival events. Observer events are continuously generated according to a random variable (generated using alpha) until the current timestamp exceeds the simulation time. We also track the number of observer events using the variable *observerCount*. After the observer events have been generated, the list is sorted in chronological order according to the timestamp of each element.

For simulation, we pop each event off the linked list and perform an action depending on the event type. During simulation, we keep track of the number of elements, the average number of elements in the queue as well as the number of times the queue is idle, using variables *queueSize* and *idleCount* respectively. For an arrival event, we increment the size of the queue, while on departure we decrement the size of the queue. For each observer event, we increment *queueSum* by the size of the queue and increment *idleCount* if the queue is empty. Once each element has been popped from the list, we calculate the performance metrics for the average queue size and the idle fraction by dividing *queueSum* and *idleCount,* respectively, by the number of observer events processed. The results are returned as an array containing two elements.

## Question 3.

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

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

Using the expression for $\rho$ provided in the lab manual and substituting the provided values for L and C yields the following:

$$\rho = \lambda * \frac{L}{C} = \lambda * \frac{2000}{1000000} = \lambda * 0.002$$

In order to determine the values of $\lambda$ as a function of $\rho$, the expression can be rearranged as follows:

$$\lambda = \frac{\rho}{0.002} = 500\rho$$

To find the lower bound, upper bound and step sizes of $\lambda$, simply substitute the corresponding value of $\rho$. Substituting $\rho = 0.25, 0.95, 0.1$ yields $\lambda = 125, 475, 50$, corresponding to the lower bound, upper bound and step sizes respectively. Using this result, we constructed a for loop that iterates the $\lambda$ parameter from 125 to 475 with a step size of 50, and extracted the resulting simulation output for each value of $\lambda$.

To compute the average number of packets in the queue, we created a counter variable to track the number of packets. At each observer event, the counter is incremented by the number of elements in the queue. At the end of the simulation, the average number of packets is computed by dividing the counter by the total number of observer events. The results of the simulation are captured in Figure 1.

The average number of elements in the queue follow an exponential curve as $\rho$ increases. It is expected that the values increase as lambda is increased, as the demand on the system is increasing while the processing speed remains constant.

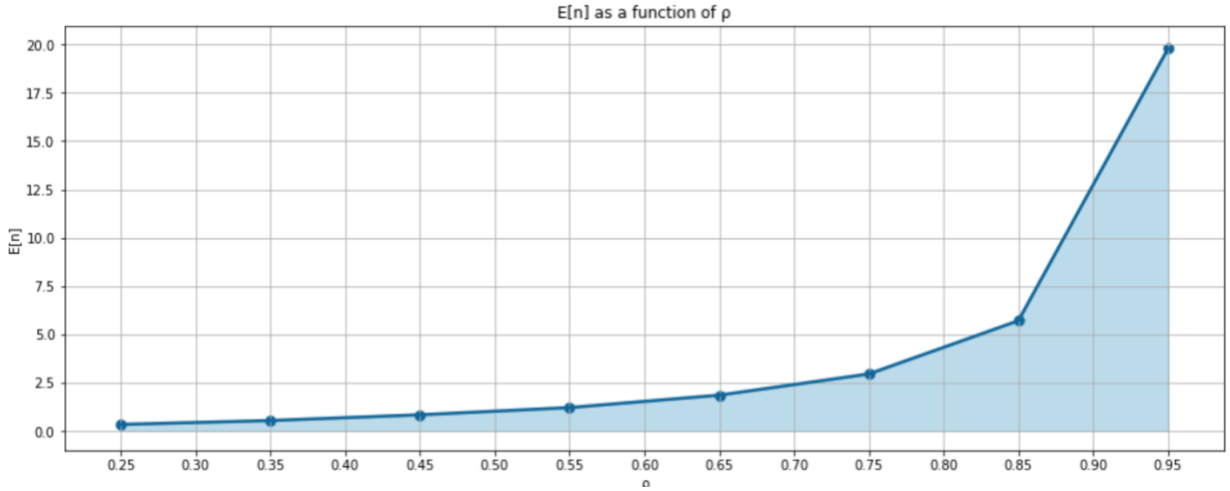


Figure 1. Average number of elements in queue as a function of ρ

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

The logic for the selection of $\rho$ is identical to the procedure followed in Q3 part 1. In order to record $P_{idle}$, we created a variable *idleCount* to keep track of the number of observer events where the system was idle. Whenever the queue was empty, the system was considered to be in an idle state. Finally, after the simulation is run, the idle fraction can be calculated by dividing the idle counter by the total number of observer events. The results can be seen in Figure 2.

The fraction of time where the system is idle is steadily decreasing as $\rho$ is increased. This is expected, as the system will be required to spend more time processing requests as the density of requests per unit of time is increased. The idle time decreases steadily from roughly 0.80, or 80% idle time percentage, all the way down to 0.05, or 5% idle time percentage. That is to say, when $\rho$ is in the range of 0.90 to 0.95, the system is idle for less than 10% of execution time.

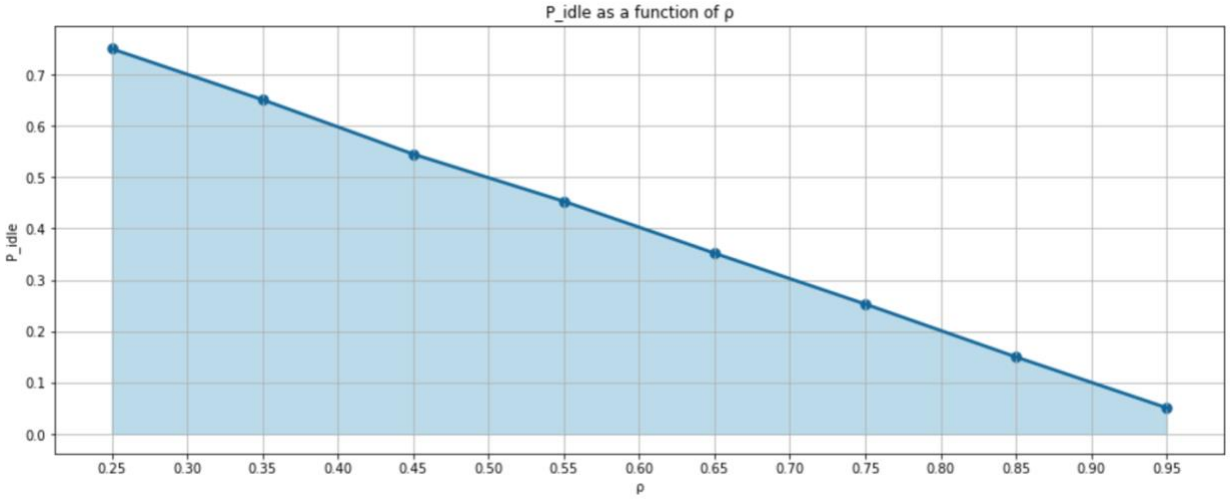


Figure 2. Idle time fraction as a function of ρ

## Question 4.

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

For $\rho$ =1.2, the resulting $P_{idle}$ is exactly 0 while the average number of elements in the queue is 4810.117. Each of these results correspond to the trends observed in question 3. For $P_{idle}$, the result tells us that the system is never idle from the beginning to the end of simulation. That is to say that the inflow of requests is so great that the queue is unable to process requests at the rate that they are arriving. This corresponds to the parameter $\rho$>1, which describes a scenario where the inflow of requests is greater than the outflow. This result is therefore expected for any $\rho >> 1$.

For E[n], the obtained result is consistent with the exponential growth observed as a function of $\rho$ in Figure 1. The number of elements in the queue will constantly be increasing over the course of the simulation, due to the inability of the system to process orders more quickly than they arrive.

## Question 5.

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

```
public static double[] simulatemm1k(double alpha, double lambda, double l, double c,
double t, double k) {
        // List containing arrival, departure and observer events
        PriorityQueue<Event> eventList = new PriorityQueue<Event>(1000000, new
timeComp());
        double currentTime = 0.0;       // Current timestamp of simulation
        int totalPacketCount = 0;       // Number of arrival events
        double observerCount = 0;       // Number of observer events

        while (currentTime < t) {
            currentTime += generateRandom(lambda);
            eventList.add(new Event("Arrival", currentTime));
            totalPacketCount++;
        }
        currentTime = 0.0;
        while (currentTime < t) {
            currentTime += generateRandom(alpha);
            Event temp = new Event("Observer", currentTime);
            eventList.add(temp);
            observerCount++;

        }

        // Packet queue
        LinkedList<Double> queue = new LinkedList<Double>();

        double dropCount = 0;           // Count number of dropped packets
        double queueDelay = 0.0;        // Time to process the elements in queue
        double delta = 0.0;             // Delay between arrival events
        double serviceTime = 0.0;       // Service delay for a given packet size
        double departureTime = 0.0;     // Timestamp for the departure event
        double queueSize = 0;           // Number of elements in the queue
        double queueSum = 0;            // Sum of number of elements in the queue

        while(true) {
            Event e = eventList.poll(); //pop event off the list
            if (e == null)       // break when the list is empty
                break;
            if (e.type.equals("Arrival")) {
                if (queue.size() > k) { // if the queue is full
                    dropCount++;        // the packet is dropped
                } else {
                    serviceTime = generateRandom(1.0 / l) / c;
                    queue.addFirst(serviceTime);
                    departureTime = e.time + serviceTime + queueDelay;
                    queueDelay += serviceTime;
                    eventList.add(new Event("Departure", departureTime));
                }
            } else if (e.type.equals("Departure")) {
                queueDelay = Math.max(0, queueDelay - queue.removeLast());
            } else if (e.type.equals("Observer")) {
                queueSum += queue.size();
            }
        }
        double avgQueueSize = (queueSum / observerCount);
        double packetLoss = (dropCount/totalPacketCount)*100;
```

```
        return new double[] { avgQueueSize, packetLoss };
    }
```

The method simulatemm1k takes as input the parameters alpha, lambda, *l*, *c*, *t* and *k*, and provides as output the values *avgQueueSize* and *packetLoss*. The input parameter t corresponds to the simulation time, whereas the other input parameters correspond to the parameters described in the manual. The output parameters correspond to the average number of elements in the queue and the fraction of packets lost during simulation, respectively. The execution flow for the M/M/1/K simulator is similar to that of the M/M/1 simulator, being broken into event generation and simulation.

Event generation logic, and the variables used, are identical to that of M/M/1 simulation, with the exception of timing calculation for the departure events. We populate the event queue with arrival and observer events during the event generation loops, departure events are to be generated during simulation. A priority queue is used to hold the events, and upon insertion events are sorted based on their timestamp such that the elements can be removed in order during simulation.

The primary difference between the M/M/1 simulation and the M/M/1/K implementation is that departure events must be generated during simulation. Furthermore, the linked list *queue* is used to keep track of the packet sizes for each element currently contained in the queue. During the simulation, elements are popped off the event queue and processed according to their event type. If the event is an observer event, the variable *queueSum* is incremented to track the average number of elements in the queue. If the event is an arrival event, we must generate the corresponding departure event. If the queue is at capacity, meaning queue.size() is greater than k, the incoming packet should be dropped, therefore we increment the *dropCount* variable. If the queue is not at capacity, we generate a new *serviceTime* using a randomly generated variable using *l*, and append the element to the queue. If the event is a departure event, we remove the topmost element from the queue and decrement the *queueDelay* variable accordingly. The simulation continues until the last element has been removed from the event queue.

The program them calculates the *avgQueueSize* and the *packetLoss*. The average queue size is simply the *queueSum* devided by the total number of observer events generated, while the *packetLoss* is the drop count divided by the total number of packets generated. The program outputs the two quantities as a double array.

## Question 6.

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

1) *E[N] as a function of ρ (for 0.5 < ρ < 1.5, step size 0.1), for K = 10, 25, 50 packets. Show one curve for each value of K on the same graph.*

The logic for selecting an appropriate $\rho$ is similar to the procedure followed in question 3 part 1, only the bounds and step sizes have been changed. Using the following relationship:

$$\lambda = \frac{\rho}{0.002} = 500\rho$$

and inserting the values $\rho$ = 0.5, 1.5, 0.1 yields λ = 250, 750, 50, corresponding to the lower bound, upper bound and step size respectively. Using the above results, we constructed a for loop that iterates the λ parameter from 250 to 750 with a step size of 50, and extracted the resulting simulation output for each value of $\rho$. An outer loop would iterate through the three possible values of k, allowing for the results to be collected for each permutation of λ and k.

The results of the simulation can be seen in Figure 3. As we can see, each curve follows the same general trend as $\rho$ is increased. At roughly $\rho = 0.8$, the average number of elements in the queue begins to increase dramatically, up until $\rho = 1.2$. At this point, the average number of elements in the queue begins to reach the capacity of the queue, which is the upper limit for the average number of elements. From this, we can conclude that the number of elements in the queue follow the same ratio as $\rho$ is increase and will scale with the capacity of the queue with a linear relationship.

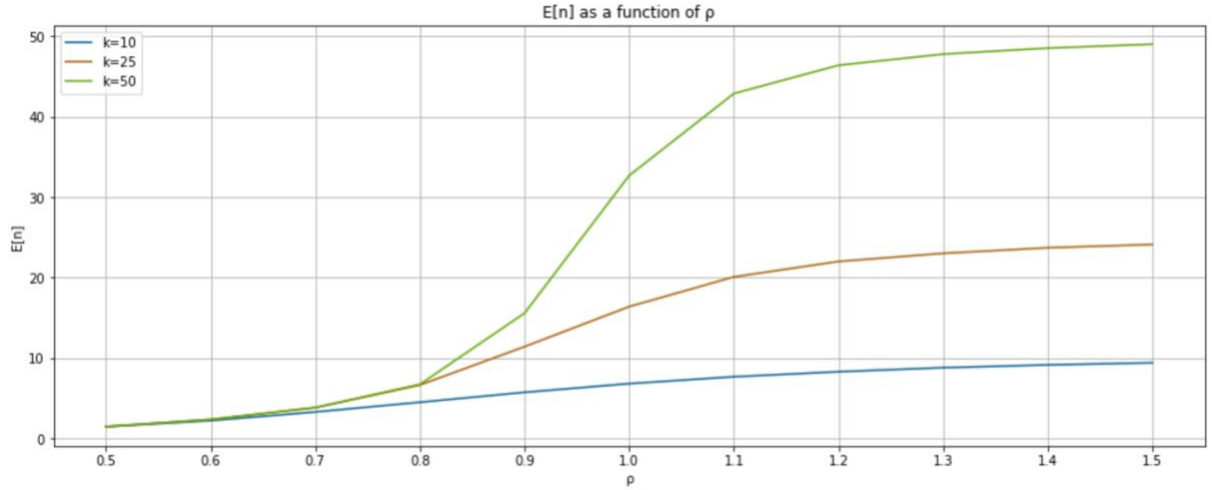


Figure 3. Average number of elements in queue as a function of ρ

2) *$P_{loss}$ as a function of ρ (for 0.5 < ρ < 1.5) for K = 10, 25, 50 packets. Show one curve for each value of K on the same graph. Explain how you have obtained $P_{loss}$. Use the following step sizes for ρ:*

The selection of λ is identical to the procedure followed in part 1. The results of the simulation can be seen in Figure 4. As $\rho$ is increased, the packet loss is increased accordingly. For the lower queue lengths k = 10 and 25, the packet loss is consistently higher. This is most noticeable in the blue curve for k=10. The packet loss is significantly higher for lower values of $\rho$, compared to the nearly non-existent packet loss seen in the other two curves until $\rho = 0.9$. This can be tied into the results obtained in Figure 3. For lower values of $\rho$, the average number of packets is similar to that of k=25,50, hovering around 4-6 packets until 0.9. However, for k=10, this corresponds to an average queue utilization of 40-60%, which is much more significant. Given a burst of packet arrivals, k=10 is much more susceptible to packet loss when compared against a larger queue size.

From $\rho = 1$ and onwards, the packet loss increases steadily for each queue size at roughly the same rate. This is because for $\rho > 1$, the rate of arrival is greater than the rate of departure. As a result, the queue is more likely to be at capacity within this range, as seen in Figure 3. The number of packet losses will continue to increase linearly as the additional packets will essentially be lost after the threshold of $\rho = 1$ is reached, because the queues are at capacity.

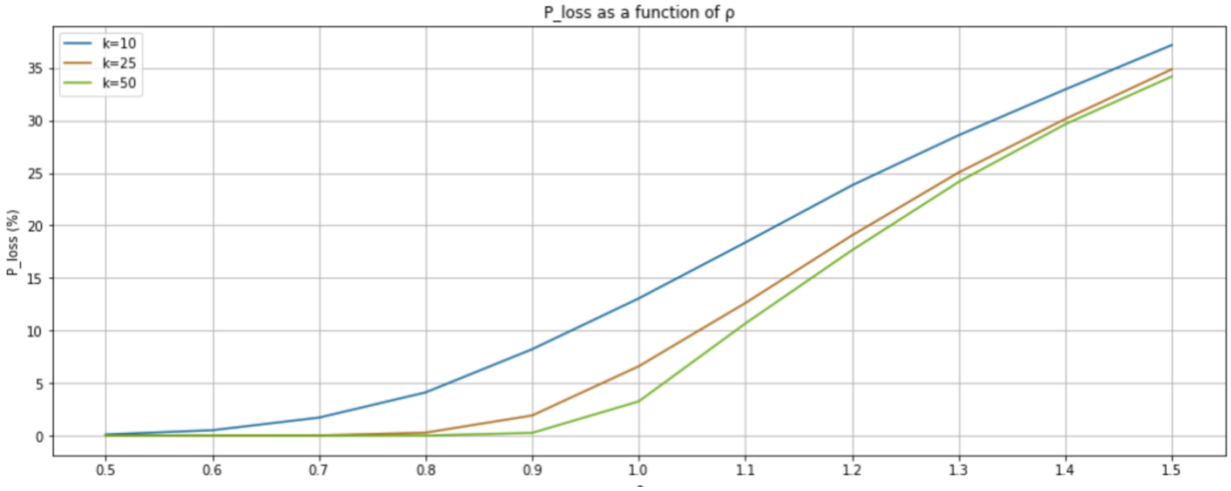


Figure 4. Packet loss as a function of ρ