**ECE 358 – Lab 1 Report**

Lucas Chevalier ljcheval 20632358

Umesh Dhurvas … …

**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 . 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) {

LinkedList<Event> eventList = new LinkedList<Event>();

double qDelay = 0.0;

double currentTime = 0.0;

while (currentTime < t) {

double delta = genarateRandom(lambda);

currentTime += delta;

double serviceTime = genarateRandom(1.0 / l) / c;

qDelay = Math.max(0, qDelay - delta);

double departureTime = currentTime + serviceTime + qDelay;

qDelay += serviceTime;

Event arrival = new Event("Arrival", currentTime);

eventList.add(arrival);

Event departure = new Event("Departure", departureTime);

eventList.add(departure);

}

currentTime = 0.0;

while (currentTime < t) {

currentTime += genarateRandom(alpha);

Event temp = new Event("Observer", currentTime);

eventList.add(temp);

}

Collections.sort(eventList, new timeComp());

double q = 0;

double qSum = 0;

double observerCount = 0;

long idleCount = 0;

for (Event e : eventList) {

if (e.type.equals("Arrival")) {

q++;

} else if (e.type.equals("Departure")) {

q--;

} else if (e.type.equals("Observer")) {

qSum += q;

observerCount++;

idleCount += (q == 0) ? 1 : 0;

}

}

double avgNumberOfElementsInQ = (qSum / observerCount);

double idle = (idleCount / observerCount);

double res[] = new double[2];

res[0] = avgNumberOfElementsInQ;

res[1] = idle;

return res;

}

mm1 code explanation:

the method takes alpha, lambda, l, c, t as the inputs. t is the total time of the simulation. the rest are variables as described in the lab manual. it returns an array of double of length 2 with average number of element in q and idle percentage.

the linked list eventList holds object so f type Event, which has a type(string) and time(double)

The first loop creates arrival event according to lambda and the random number generator function. then creates a random packet size and calculates service time according to l and c. then the departure event is calculated according to the q delay. qDelay is calculated by … . The departure event is added to the linked list,

The second while loop generates observer event according to alpha.

the linked list is sorted according the time in ascending order.

Now the simulation takes place.

variable q keeps track of elements in the queue.

variable qSum is used to calculates the average elements in the q. it adds the elements in the q on every observer event.

variable observerCount is counts observer events.

variable idleCount is counts number of times the q is empty when an observer even is processed.

The simulation takes place by iterating through the eventList. On arrival event we increment q.

on departure we decrement q. On observer sum is updated by adding q to qSum. observer count is incremented, idleCount is incremented by one if q is empty,

Finally, average number of element in q is calculated by qSum/observerCount.

idle percentage is calculated by idle count / observerCount. The results are put in an array and returned.

**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 < < 0.95, step size 0.1). Explain how you do that.*

Using the expression for provided in the lab manual and substituting the provided values yields the following:

= 0.002

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

To find the lower bound, upper bound and step sizes of , simply substitute the corresponding value of . Substituting yields , corresponding to the lower bound, upper bound and step sizes respectively. Using this result, we constructed a for loop that iterates the parameter from 125 to 475 with a step size of 5, and extracted the resulting simulation output for each value of

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 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.

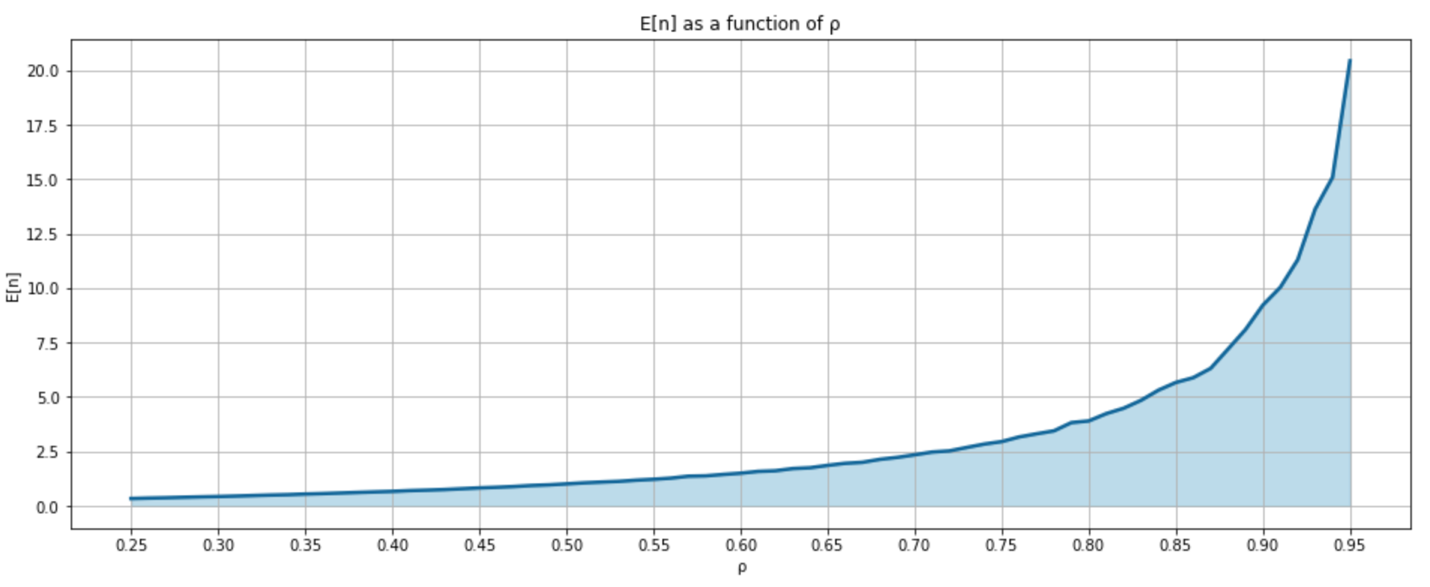


Figure 1. Average number of elements in queue as a function of ρ

1. *Pidle, the proportion of time the system is idle as a function of , (for 0.25 < < 0.95, step size 0.1). Explain how you do that.*

The logic for the selection of is identical to the procedure followed in Q3 part 1. In order to record Pidle, 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 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 is in the range of 0.90 to 0.95, the system is idle for less than 10% of execution time.

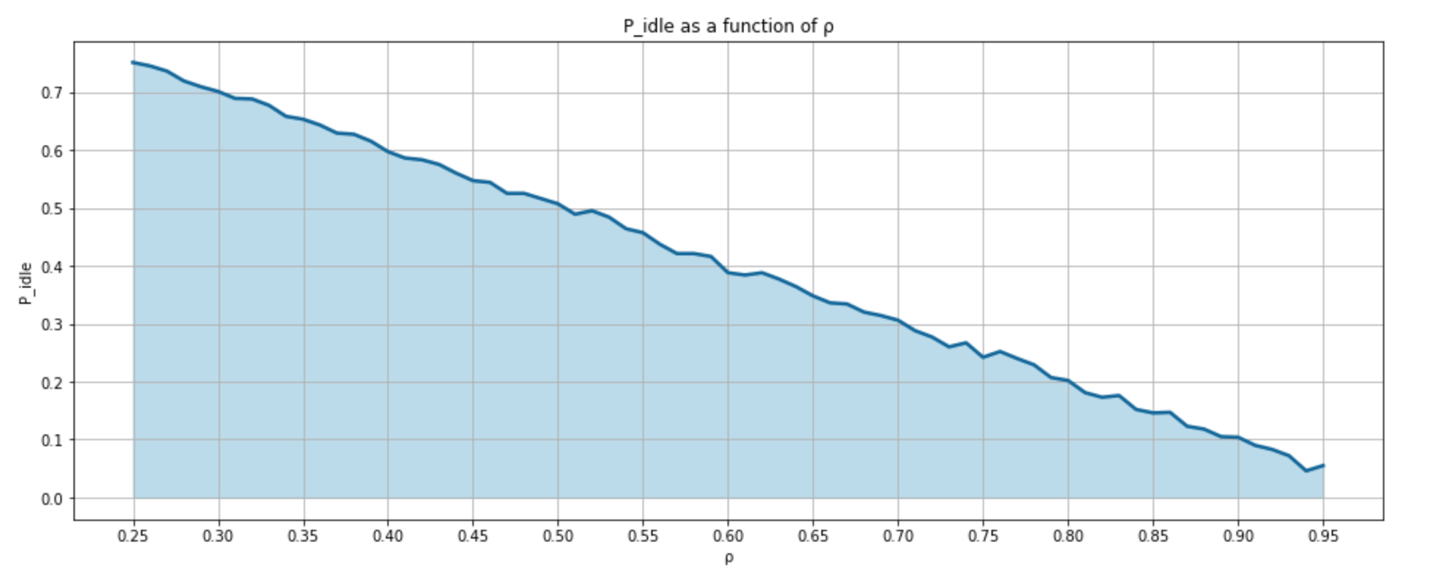


Figure 2. Idle time fraction as a function of ρ

**Question 4.** *For the same parameters, simulate for ρ=1.2. What do you observe? Explain.*

For =1.2, the resulting Pidle 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 Pidle, 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 >1, which describes a scenario where the inflow of requests is greater than the outflow. This result is therefore expected for any >> 1.

For E[n], the obtained result is consistent with the exponential growth observed as a function of 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) {

PriorityQueue<Event> eventList = new PriorityQueue<Event>(10000000, new timeComp());

double currentTime = 0.0;

while (currentTime < t) {

double delta = genarateRandom(lambda);

currentTime += delta;

Event arrival = new Event("Arrival", currentTime);

eventList.add(arrival);

}

currentTime = 0.0;

while (currentTime < t) {

currentTime += genarateRandom(alpha);

Event temp = new Event("Observer", currentTime);

eventList.add(temp);

}

LinkedList<Double> q = new LinkedList<Double>();

double qDelay = 0;

double qSum = 0;

double dropCount = 0;

double observerCount = 0;

long idleCount = 0;

for (int i = 0;; i++) {

Event e = eventList.poll();

if (e == null)

break;

if (e.type.equals("Arrival")) {

if (q.size() > k) {

dropCount++;

} else {

double serviceTime = genarateRandom(1.0 / l) / c;

q.addFirst(serviceTime);

double departureTime = e.time + serviceTime + qDelay;

qDelay += serviceTime;

Event departure = new Event("Departure", departureTime);

eventList.add(departure);

}

} else if (e.type.equals("Departure")) {

qDelay = Math.max(0, qDelay - q.removeLast());

} else if (e.type.equals("Observer")) {

qSum += q.size();

observerCount++;

idleCount += q.isEmpty() ? 1 : 0;

}

}

double avgNumberOfElementsInQ = (qSum / observerCount);

double idle = (idleCount / observerCount);

double res[] = new double[3];

res[0] = avgNumberOfElementsInQ;

res[1] = idle;

res[2] = dropCount;

return res;

}

mm1k explanation:

the method takes alpha, lambda, l, c,,k, t as the inputs. t is the total time of the simulation. the rest are variables as described in the lab manual. it returns an array of double of length 3 with average number of element in q and idle percentage and drop count.

A priority queue is used to store the events, The same comparator used to sort in mm1 is used as the comparator sent to the priority queue for maintaining an assorted queue.

The first while loop generated random arrival avenues according to lambda.

The second while loop generated random observer avenues according to alpha.

Simulation:

a linked list called q is used to keep track of the events in the q. variable qDelay is the sum of doubles in the linked list q. dropCount keeps track of arrival events dropped due to queue overflow.

q sum, observerCount and idle count have the same purpose as mm1.

A for loop is used to iterate through the eventList. it breaks when the queue is empty.

on arrival event, the event is dropped if the queue if full. If not, the service time of the

current arrival event is calculated by generated a packet size using l, then divided by c. the service time is added to the linked list q. its departure time is calculated by adding its own service time and the sum of all the service times already in the q. then the event is added to the eventList(note it remains sorted because its a priority queue).

On departure event. the tail of the linked list q is pope and q delay is decreased by the value of the popes element. qDelay remains non-negative.

On observer the same variables as mm1 are calculated.

Finally, average number of element in q is calculated by qSum/observerCount.

idle percentage is calculated by idle count / observerCount.

The results are put in an array and returned, including the drop count.

**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 is similar to the procedure followed in question 3 part 1, only the bounds and step sizes have been changed. Using the following relationship:

and inserting the values = 0.5, 1.5, 0.1 yields = 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 parameter from 250 to 750 with a step size of 50, and extracted the resulting simulation output for each value of An outer loop would iterate through the three possible values of k, allowing for the results to be collected for each permutation of 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 is increased. At roughly , the average number of elements in the queue begins to increase dramatically, up until . 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 is increase and will scale with the capacity of the queue with a linear relationship.

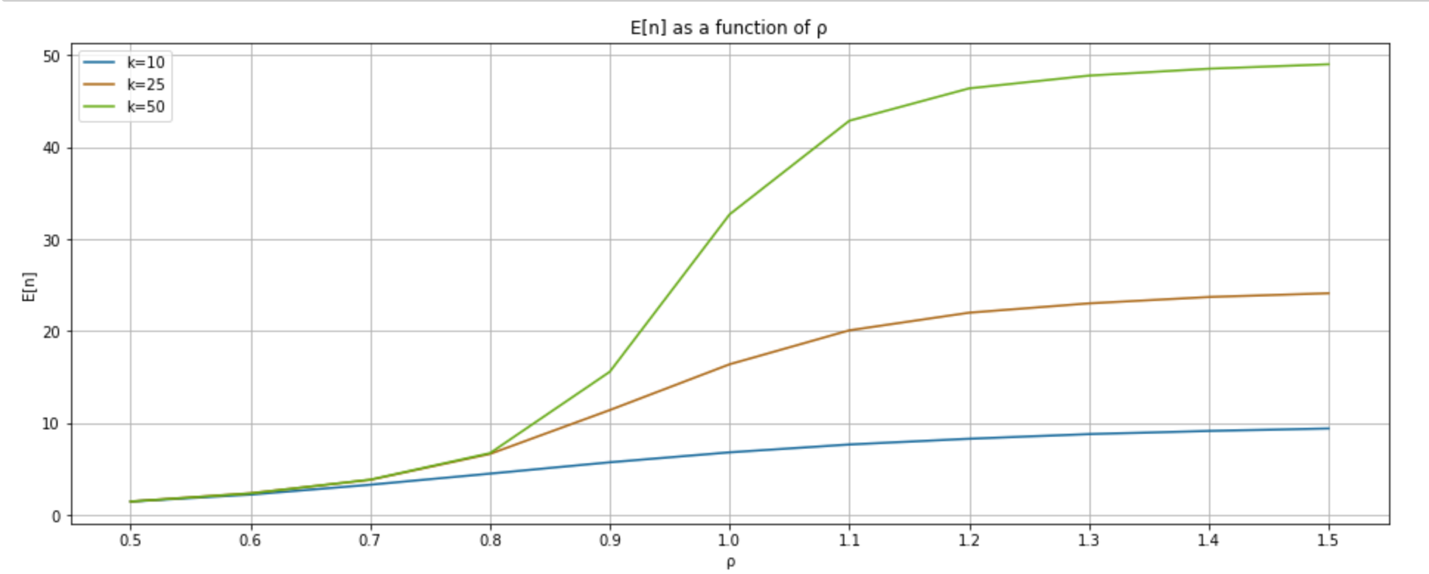


Figure 3. Average number of elements in queue as a function of ρ

1. *Ploss 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 Ploss. Use the following step sizes for ρ:*

The selection of is identical to the procedure followed in part 1. The results of the simulation can be seen in Figure 4. As 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 , compared to the nearly non-existant packet loss seen in the other two curves until . This can be tied into the results obtained in Figure 3. For lower values of , 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 and onwards, the packet loss increases steadily for each queue size at roughly the same rate. This is because for , 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 is reached, because the queues are at capacity.

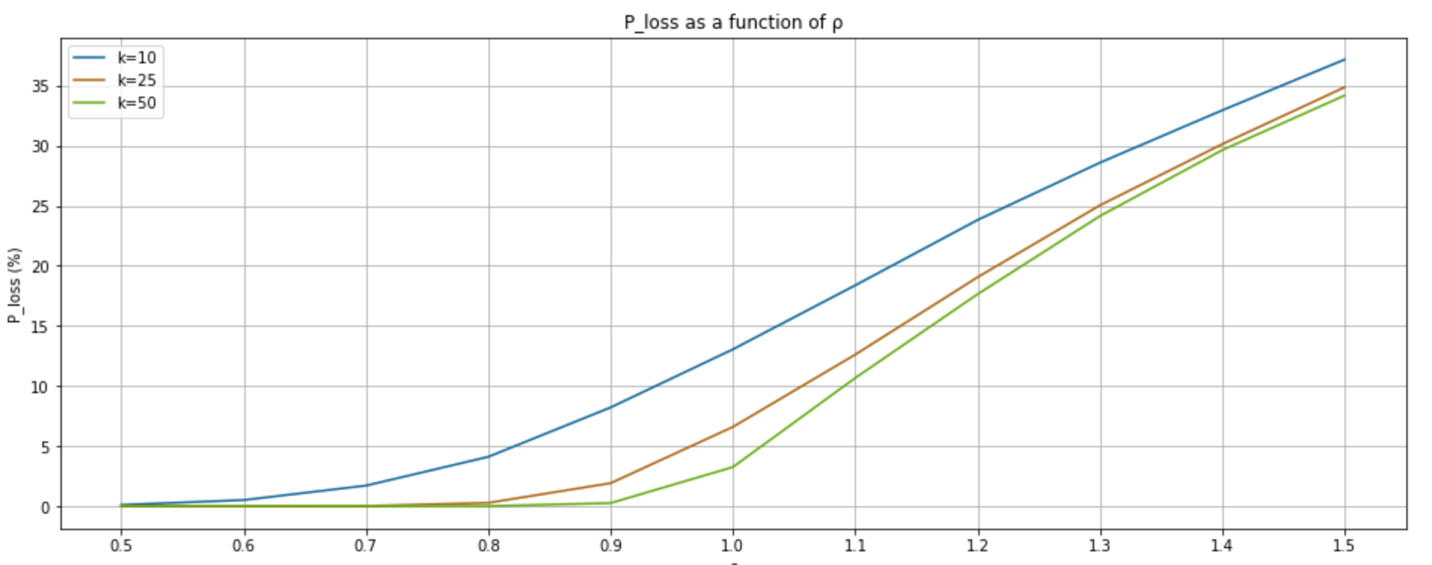


Figure 4. Packet loss as a function of ρ