CECS 474: Computer Network Interoperability

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

Due Date: March 7, 2022

Team Members: Denise Paredes and Matthew Zaldana

Contents

Overview

This lab assignment was completed by Denise Paredes and Matthew Zaldana. The lab report will consist of code snippets and full explanation regarding the design of two basic types of queues, M/M/1 and M/M/1/K. The programming code used for this lab was C++. Both team members contributed to the research part of this lab. We relied heavily on C++ documentation and the guidance of Professor Peng. In addition, we used a simulator template open source created by https://github.com/thanujann/Queue-Simulator.

Matthew worked on debugging the code and adding functions to meet the requirements for this lab. Exporting data to excel for the graphs needed for question 6 was also completed by Matthew.

Denise worked on the README.txt file that shows how to run our source code. Adding proper documentation to the code and providing a satisfactory explanation to the questions given in the lab instructions was completed by Denise.

Both team members worked on completing the lab report and re-learning Poisson Distribution. As well as completing the short code to generate exponential random variables, as presented in the following page.

Question 1: Exponential Random Variable

What is the mean and variance of the 1000 random variables you generated?

```
Mean of exponential random variables: 0.013264
Variance: 1.00104
```

Do they agree with the expected value and the variance of an exponential random variable with λ =75?

Both values generated in the code are the expected values, given the Poisson Distribution formula.

Question 2: M/M/1 Simulator

```
175 // M/M/1 simulation
176 void simulateMM1Queue() {
177     cout << "\n==========MM1 Queue:==========" << "\n\n";
178     double simulationTime = 2000;
179     double alpha = 0;
180     double lambda = 0;
181     double L = 2000;
182     double C = 1000000;
183     int queueSize = INT_MAX; // Represents a M/M/1 queue
    (infinite queue size)
184 
185     // Display the effects of traffic intensity on the average
    number of
186     // packets in the system and the proportion of system idle
    time
187     int rho = lambda * L / C;
188 
189     for (double rho = 0.25; rho <= 1.05; rho += 0.1) {
190         lambda = (1.2 * C)/L;
191         alpha = 5 * lambda;
192         simulate(simulationTime, alpha, lambda, L, C, queueSize);
193 
194         cout << "-----Traffic Intensity: " << rho << "-----\n\n";
195         cout << "Average Number of Packets in System = " <<
    numberOfPacketsInQueue/observationCounter << "\n";
196         cout << "Proportion of system idle time = " <<
    idleCounter/observationCounter << "\n\n";
197     }
198     cout << "Done simulating MM1 Queue!\n";
199 }
```

As seen in the code, the way we calculate the traffic intensity and the average number of packets in our system is in our *for loop* which is inside our simulation.

The code seen is the function that creates our simulation. We created a function with simulation time set to 2000 seconds, alpha is our observer events which is set to 0 and the number of packets that arrive in packets per second, lambda is 0, the length of the packet L set to 2000 bits and the speed transmission rate C is 1000000 bits per second or 1MBps. It's important to note that the queue size is set to the max of the INT class, representing an infinite queue size.

We calculate rho given the formula on line 187. Using those values for rho, we make a for loop that incrementally adds 0.1 after each run of the simulation so that queue has more values before simulating again.

We call our simulate function, passing in all required variables to make it run successfully. Our simulate function looks like the following:

```
void simulate (double simulationTime, double alpha, double lambda, double L, double C, int queueSize) {
    // Reset global counters
    observationCounter = 0;
    arrivalCounter = 0;
    departureCounter = 0;
    idleCounter = 0;
    numberOfPacketsInQueue = 0;
    droppedPacketsCounter = 0;

    // Set random time arrays
    double observerTimes[defaultSize];
    double arrivalTimes[defaultSize];

    for (int i = 0; i < defaultSize; i++) {
        observerTimes[i] = getExponentialRandomVariable(alpha);
        arrivalTimes[i] = getExponentialRandomVariable(lambda);
    }

    // Generate observer and arrival events
    generateObserverAndArrivalEvents(simulationTime, observerTimes, arrivalTimes, alpha, lambda);
    // Handle events in the queues
    while (observerQueue.size() > 0 || arrivalQueue.size() > 0 || departureQueue.size() > 0) {
        Event *event = getNextEvent();
        if (event->getType() == OBSERVER) {
            handleObserverEvent();
        } else if (event->getType() == ARRIVAL) {
            handleArrivalEvent(event, L, C, queueSize);
        } else {
            handleDepartureEvent();
        }
    }
}
```

Notice that our counters in the simulate function are reset to make sure that each simulated run is different. Initially, as you can tell from the rest of the code that is being submitted with this lab, they were not initialized. Given the nature of global variables, we can call these variables at any point in the program and change them accordingly. We also have two arrays here, one for the observer times and the other for the arrival times. For each array, we populate their elements with a random variable generator that is shown in problem 1 given either alpha or lambda, depending on the array that must be used. For observer times array, we populate those with exponential random variables given alpha, our observer events per second, initially 0, while for arrival times, we populate it with lambda, our length of the packet in bits, initially also 0. We then generate the observer and arrival events by passing the required times. After this, we handle the events in our queue using a while loop that utilizes an if-else structure to choose between observer, arrival, or departure events.

These events and functions are seen also in our code and either increments the idle counter of the system or increments the number of packets given the event. In the case of departure, it decreases the number of events.

Finally, when all the events have been taken care of and the global variables have all been updated, those variables are passed back to the original function that simulates the run and outputs those variables with their values.

For example, we calculate the average number of packets in the system given the global variables *numberOfPacketsInQueue and observationCounter* that are saved in the observer events function. We then divide those two values to calculate the number of packets in the system and display it to the screen.

We also calculate the proportion of system idle time in the system given two other variables called *idleCounter* and *observationCounter* which are saved in the observer

events function as well. We then divide these two values which results in our system idle time metric and output that value to the screen.

Question 3: Simulator Graphs

Output for MM1 Queue

==========MM1 Queue:==========

-----Traffic Intensity: 0.25-----

Average Number of Packets in System = 0.3303

Proportion of system idle time = 0.751018

-----Traffic Intensity: 0.35-----

Average Number of Packets in System = 0.538225

Proportion of system idle time = 0.649851

-----Traffic Intensity: 0.45-----

Average Number of Packets in System = 0.819435

Proportion of system idle time = 0.549876

-----Traffic Intensity: 0.55-----

Average Number of Packets in System = 1.22073

Proportion of system idle time = 0.450785

-----Traffic Intensity: 0.65-----

Average Number of Packets in System = 1.84713

Proportion of system idle time = 0.350756

-----Traffic Intensity: 0.75-----

Average Number of Packets in System = 3.02575

Proportion of system idle time = 0.247775

-----Traffic Intensity: 0.85-----

Average Number of Packets in System = 5.69964

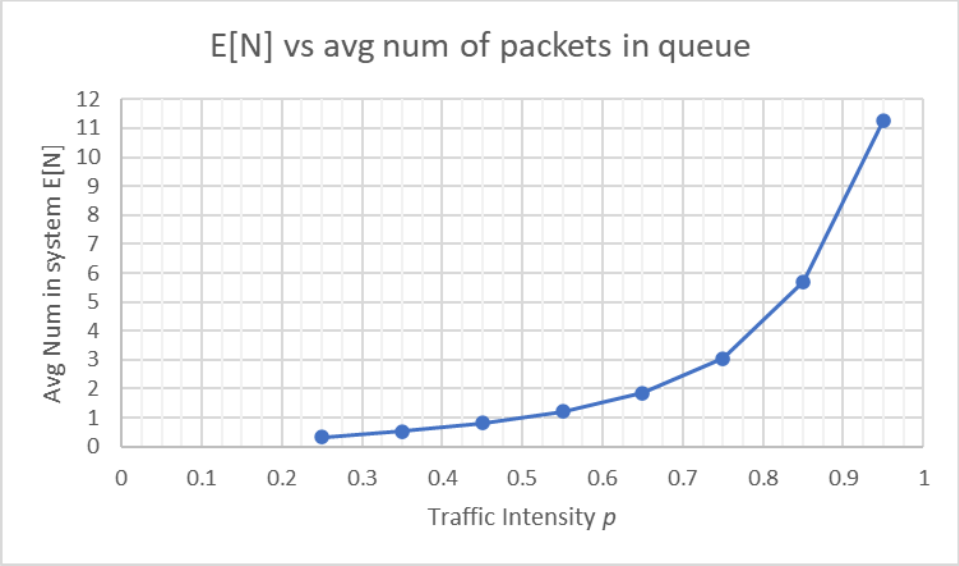
Proportion of system idle time = 0.14917

-----Traffic Intensity: 0.95-----

Average Number of Packets in System = 11.249

Proportion of system idle time = 0.04571

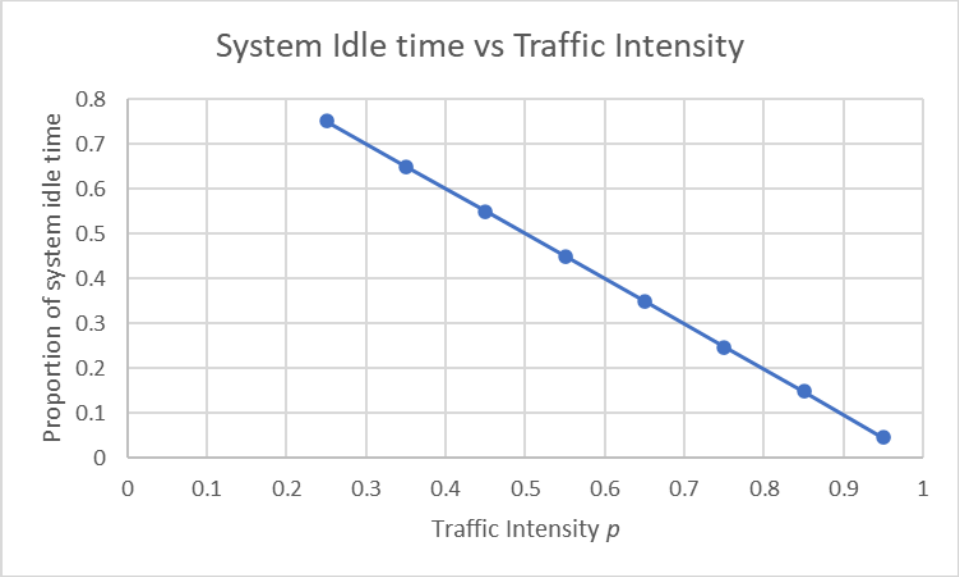
Avg number of packets in the system vs traffic intensity



Explain: As the traffic intensity, rho, increases, the average number of the system increases as well. As seen in the graph, we start the values of rho at 0.25 seconds and stop at 0.95 seconds. We notice that up until 0.65 seconds, the values increased at a steady rate in the form of a line. But as soon as the higher values are passed in the random generator function, those values increase drastically. After 0.65 and to the end, the average number of packets in the system increases. This is because the number of

events that are generated are higher than before, both observer and arrival events. For this, the simulate function takes longer to complete the while loop that handles these events and therefore there is more system usage. Nonetheless, as we see in the next graph, since there is more utilization, the packets are all being taken care of which decreases the system idle time.

System idle time vs traffic intensity



Explain: As the traffic intensity, rho, increases, the proportion of the system idle time decreases. This is because as time goes on, the observer and arrival events are being handled in our while loop inside our simulate function. As this happens, the amount of packets in the system queue is large, but decreases steadily, until reaching the stable average amount of the idle time in the system were there no events to handle. If we were to continue the rho values for more than 0.95 and not simulate any other events, then we would see a straight line at system idle time or the y values at 0.05%. We also notice how this line represents the derivative of the graph above which is in the form of a parabola or exponential value. Given the derivative of a parabola is a line, it would make sense how while the system is used more as more events are generated and departed, the number of packets in the system decreases steadily over time to a point where there are no more packets to take care of in the system.

Question 4: Simulator with rho = 1.2

```
~/Queue-Simulator$ ./DES

==========MM1 Queue:==========

-----Traffic Intensity: 0.25-----

Average Number of Packets in System = 100451
Proportion of system idle time = 1.16647e-06

-----Traffic Intensity: 0.35-----

Average Number of Packets in System = 100341
Proportion of system idle time = 1.00035e-06

-----Traffic Intensity: 0.45-----

Average Number of Packets in System = 101042
Proportion of system idle time = 2.66782e-06

Killed
~/Queue-Simulator$
```

We observed two events. First, the number of packets in the system decreases, which could mean that there are fewer packets being sent at that time. Second, as the number of packets decreases the system idle time increases. We noticed, however, that the number of packets in the system is large, in the hundreds of thousands and the system idle time is very small in the millionths of a second. We think that between the rho values of 0.95 and 1.2 the graph breaks, there is discontinuity for the values of the packets in the system idle time and that the observation counters are large. Since it is the divisor for the proportion of system idle time, that number is very small. We also noticed that when we ran the simulation, our RAM usage jumped a lot. This could also correlate to the same fact that there is so much system usage that there is not enough RAM to handle all the events necessary and the program breaks for lack of physical memory.

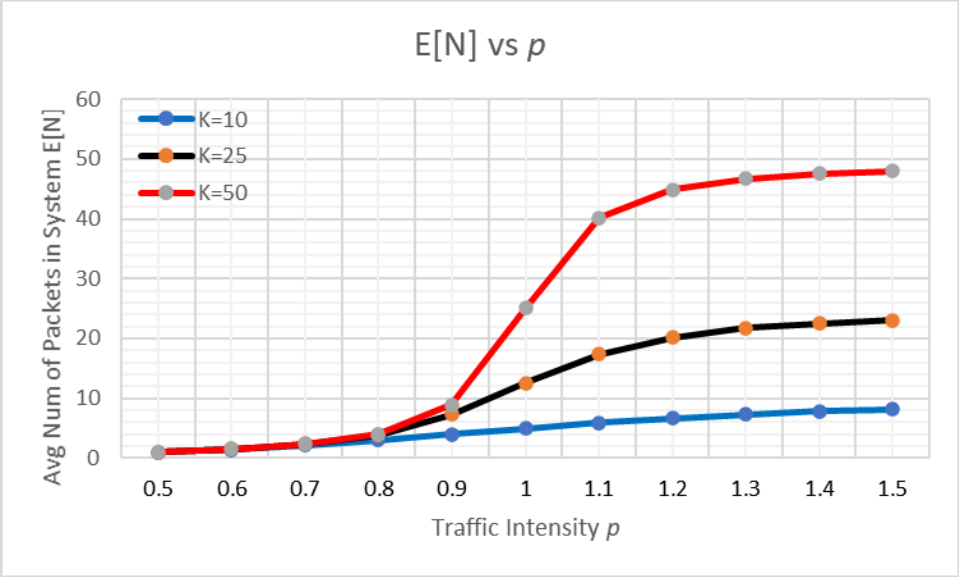
Question 5: M/M/1/K Simulator

This function relies on the functions provided in Question 2. We created a similar function for the MM1K Queue.

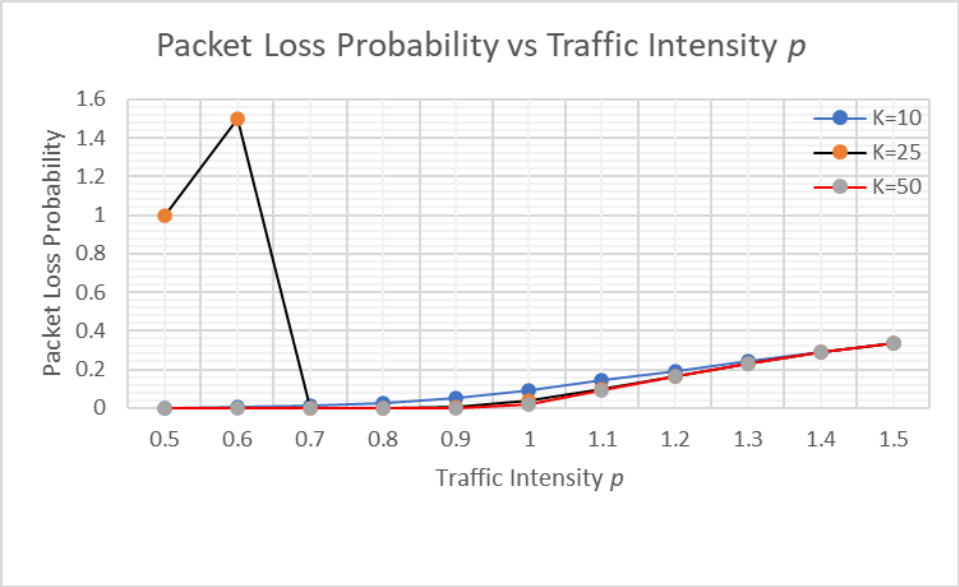
```
201
202 // M/M/M/1/K Simulation
203 void simulateMM1KQueue() {
204     cout << "\n=====MM1K Queue=====" << "\n\n";
205     double alpha = 0;
206     double lambda = 0;
207     double simulationTime = 2000;
208     double L = 2000;
209     double C = 1000000;
210     // Display the effects of traffic intensity on the average number of
211     // packets in the system and the probability of dropping a packet,
212     // with queue sizes of 10, 25, and 50
213     for (double rho = 0.5; rho <= 1.6; rho += 0.1) {
214         lambda = (rho * C)/L;
215         alpha = 5 * lambda;
216         cout << "Traffic Intensity = " << rho << "\n\n";
217         simulate(simulationTime, alpha, lambda, L, C, 10);
218
219         cout << "Queue Size = " << 10 << "\n";
220         cout << "Average Number of Packets in System = " << numberOfPacketsInQueue/observationCounter << "\n";
221         cout << "Probability of Dropping a Packet = " << droppedPacketsCounter/arrivalCounter << "\n\n";
222
223         simulate(simulationTime, alpha, lambda, L, C, 25);
224
225         cout << "Queue Size = " << 25 << "\n";
226         cout << "Average Number of Packets in System = " << numberOfPacketsInQueue/observationCounter << "\n";
227         cout << "Probability of Dropping a Packet  = " << droppedPacketsCounter/arrivalCounter << "\n\n";
228
229         simulate(simulationTime, alpha, lambda, L, C, 50);
230
231         cout << "Queue Size = " << 50 << "\n";
232         cout << "Average Number of Packets in System = " << numberOfPacketsInQueue/observationCounter << "\n";
233         cout << "Probability of Dropping a Packet  = " << droppedPacketsCounter/arrivalCounter << "\n\n";
234     }
235 }
```

As noted, the same functions are used. In this simulation, we add a couple of cout statements that will allow us to change the number of packets in the system. Using the same simulate function will allow us to pass in different values, in this case, 10, 25, 50 packets per run of the simulation. We then display those values to our terminal to show the results of the simulation. Another for loop is used to calculate lambda and increment the rho value after each run of the simulation. For this simulation, as stated above, one of the differences is that we specify the number of packets that we will be simulating. We have a queue size of 10, 25, and 50 packets. For each packet size and after calling the simulate functions that have already been explained, we display the same performance metrics. All other variables stay given the same values. The values of rho here change – they are now starting from the 0.5 value to the 1.5. You'll see in our code, however, that they we made the value of rho increased to 1.6 because during our runs, we would notice that the program would stop at rho=1.4. We wanted all the values, therefore, we bumped up the value of rho to 1.6 and stopped the program when it finished simulating for rho=1.5.

Question 6: M/M/1/K Simulator Graphs



These graphs show traffic intensity rho values versus the average number of packets in the system. There are three 3 graphs in this one graph. Each line, red, black, and blue represents a different number of packets: 10 is blue, 25 is black 50 is red as said in the legend. We notice that when the packet number is low, E[N] is also low no matter the traffic intensity. As the packet number increases, E[N] increases drastically. However, in all cases, we notice that around rho value of 1.1~1.2, the idle time stabilizes, and the packets transferred flows easier. What was interesting to note, however, was how when the packet size was 10, the average number of packets was steady and minimal compared to the other graph lines. This trend also is visible in the packet loss probability graph below, where the line is consistent, straight, and steady. As the number of packets increases, given their amounts of about more than 2 times the previous ones, we noticed that their average number of packets in the system also doubles in size roughly. This means that if more packets were sent to simulate, there is a proportional amount from its predecessor.



This graph shows the packet loss probability versus the traffic intensity *p* when transmitting packets. We notice that for the most part, the values climb between 0 and stay under 0.4%. However, it was interesting to note that for K=25, the value did spike up to roughly 1.5%. In the grand scheme of things, this value is still very small. For all K values, it is reassuring to note that, all things considered, the probability of a packet getting lost in a M/M/1/K queue is still very small. As mentioned in the previous graph, when packet K size = 10, the probability trend line stayed consistent, increasing steadily and minimal. The same goes for the other two. We think that the reason for the spike when K=25 is because of the previous simulations still in the queue, however, when K=50, the spike did not occur. This leads us to believe that it might be an internal system component, RAM, or CPU misconfiguration that could cause instability in the simulation However, again, the amount of probability is still very small, given that the values are represented using the percent scale.