Faculty of Engineering
Department of Electrical and Computer Engineering

ECE 458 Lab 1
M/M/1 and M/M/1/K Queue Simulation

Jiwoo Jang (20617044)
Joel Ruhland (20607779)

September 25, 2019

Table of Contents

Question 1: Random Exponential Variable Generation Code

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

// Generates and prints 1000 random numbers in an exponential distribution with
lambda=75
int main(void) {
    int lambda = 75;

    // Initialize/seed the random number generator with the system time
    // Not a good idea from a security perspective but acceptable for simulations
    srand(time(0));

    for (int i=0; i<1000; i++) {
        // Generate and print the random value
        // rand() generates uniformly from 0 to RAND_MAX
        double uniformRandomVariable = 1.0f * rand() / RAND_MAX;

        // Transform to exponential distribution
        double randVal = -(1.0f / lambda) * log(1.0 - uniformRandomVariable);
        printf("%f\n", randVal);
    }
    return 0;
}
```

We expected mean of 1/λ=0.01333 and measured a mean of 0.0131 based on the output data. We also expected variance of 1/λ^2= 0.000178 according to the properties of an exponential distribution, and measured variance of 0.000169 according to the output data. Thus the results of the random number generator were consistent with an exponential distribution.

Question 2: MM1 Queue Design

A short utility class was made to generate random variables according to an exponential distribution. Of particular note is the `mt19937` generator, which is a random generator designed to be portable across hardware implementations. The class definition and implementation is shown below. The `random_device` was used to seed the generator, and a C++ standard `uniform_real_distribution` was used to generate random uniformly distributed numbers. This generator was tested using the same methods as Question 1 and was verified to have valid exponential distribution behaviour.

Random Number Generator Definition

```
#pragma once
#include <cmath>
#include <random>

using namespace std;

class RandomNumberGenerator
{
public:
    RandomNumberGenerator();
    ~RandomNumberGenerator() {};

    // Generates a random value according to the poisson distribution
    double GenerateRandomValue(double lambda);

private:
    random_device rd;
    mt19937 generator;
    uniform_real_distribution<double> dis;
};
```

Random Number Generator Implementation

```
#include "RandomNumberGenerator.hpp"

RandomNumberGenerator::RandomNumberGenerator()
{
    // Seed the generator
    generator.seed(rd());

    // Initialize the uniform distribution
    dis = std::uniform_real_distribution<double>(0, 1.0f);
}

double RandomNumberGenerator::GenerateRandomValue(double lambda)
{
    // Generate a uniform random variable
    double uniformRandomVariable = dis(generator);

    // Crunch the inverse function for a poisson distribution
    return -(1.0f / lambda) * log(1.0 - uniformRandomVariable);
}
```

The MM1 Queue simulator was built using two basic parts: an Event class that contained event specific data, and an EventQueue class that initialized a queue with events, and processed the simulation.

The Event class was a basic class that contained data about it's type and its timestamp in simulation time. The class definition and implementation is shown below.

Event Class Definition

```
class Event
{
public:
    enum EventType
    {
        Invalid,
        Arrival,
        Observer,
        Departure
    };

    Event(EventType type, double newProcessTime);
    ~Event(){};

    double GetProcessTime() const { return processTime; }
    EventType GetEventType() const { return eventType; }

private:
    EventType eventType = Invalid;
    // TODO: Figure out if there are better portable versions of floats
    double processTime = 0.0f;
};
```

Event Class Implementation

```
#include "Event.hpp"

Event::Event(EventType type, double newProcessTime)
{
    eventType = type;
    processTime = newProcessTime;
}
```

A base class was created for use with the MM1 queue and MM1K queue, which contained all queue parameters, such as lambda (for arrival events), the mean packet length, alpha (for observer events) and the link rate.

The queue was initialized by using the random number generator to generate random interarrival times for events. The rate parameter lambda was used to generate times for arrival events, and the rate parameter alpha was used to generate times for observer events. These events were added to a list of events stored in a member variable.

For the MM1 queue, departure events were also initialized. If a departure was calculated to occur at a time greater than the time of the last departure, it was added to the list and set to depart after the service time. If the departure was found to occur before the last departure, then it was set to depart after the last departure plus the service time.

All events were then sorted by timestamp to make the queue processable by time.

A struct also held the results for processing once the queue had been simulated. The class definition and implementation is shown below.

Event Queue Definition

```
#include "Event.hpp"
#include "RandomNumberGenerator.hpp"

class EventQueue
{
public:
    virtual ~EventQueue();

    struct ProcessResults
    {
        int numArrivals = 0;
        int numDepartures = 0;
        int numObservations = 0;
        int idleCount = 0;
        int numDropped = 0;
        int queueSum = 0;

        float GetAveragePacketsInQueue() const { return queueSum /
(float)numObservations; }
        float GetIdleTimePercent() const { return 100.0f * idleCount /
(float)numObservations; }
        float GetDroppedPacketPercent() const {return 100.0f * numDropped /
(float)numArrivals; }
    };

    void InitalizeQueue(double simulationTime, bool generateDepartures=true);
    const list<Event>& GetEventList() { return eventList; }
    virtual void ProcessQueue();

protected:
    double lambda;
    int L;
    double alpha;
    int C;
    int K; // Queue Size

    RandomNumberGenerator numGen;
    list<Event> eventList;
};
```

Event Queue Implementation

```
#include <iostream>

#include "EventQueue.hpp"
#include "Event.hpp"

EventQueue::~EventQueue() {
    eventList.clear();
}

void EventQueue::ProcessQueue() {}

bool locSortEventByTime(const Event& firstEvent, const Event& secondEvent)
{
    return firstEvent.GetProcessTime() < secondEvent.GetProcessTime();
}

void EventQueue::InitalizeQueue(double simulationTime, bool generateDepartures) {
    double time = 0.0f;
    Event lastDeparture(Event::EventType::Arrival, 0);

    // Populate observer events
    while(time < simulationTime)
    {
        // Increase simulation time by random interarrival time
        double randomInterArrivalTime = numGen.GenerateRandomValue(alpha);
        time += randomInterArrivalTime;

        // Create and push an observer event
        Event observerEvent(Event::EventType::Observer, time);
        eventList.push_back(observerEvent);
    }
    time = 0.0f;

    while(time < simulationTime)
    {
        // Increase simulation time by random interarrival time
        double randomInterArrivalTime = numGen.GenerateRandomValue(lambda);
        time += randomInterArrivalTime;

        // Create and push an arrival event
        Event arrivalEvent(Event::EventType::Arrival, time);
        eventList.push_back(arrivalEvent);

        if (generateDepartures) {

            // Calculate time at which this arrival will be processed
            double packetLength = numGen.GenerateRandomValue(1.0f/L);
            double serviceTime = (packetLength / C);

            // If the last generated departure occurs later than our current time 
base our new departure off that time
            double departureTime = lastDeparture.GetProcessTime() > time ? 
lastDeparture.GetProcessTime() : time;

            // Create and push corresponding departure event
            Event departureEvent(Event::EventType::Departure, departureTime + 
serviceTime);
            lastDeparture = departureEvent;

            eventList.push_back(departureEvent);
        }
    }

    // Sort all events by time stamp
    eventList.sort(locSortEventByTime);
}
```

The MM1 Queue class derived from the Event Queue class to implement a specific method to process the results of the queue.

Processing checked the type of event, and incremented the appropriate counters. Arrival events incremented the arrival event counter, and departure events incremented the departure event counter. Observer events incremented the observer counter, but also checked if the queue was empty. If there was content in the queue, they were added to a rolling sum of packets in queue. If the queue was empty, the idle counter was incremented to find percent idle time after processing.

E[N] was computed by dividing the rolling sum of packets in queue by the number of observer events. Percent idle time (PIdle) was calculated by dividing the idle counter total by the number of observer events and multiplying by 100%. The class definition and implementation is shown below.

MM1 Queue Definition

```
#pragma once
#include <list>

#include "Event.hpp"
#include "EventQueue.hpp"
#include "RandomNumberGenerator.hpp"

using namespace std;

class MM1Queue : public EventQueue
{
public:
    MM1Queue(double newLambda, int newL, double newAlpha, int newC);
    void ProcessQueue() override;
};
```

MM1 Queue Implementation

```
#include <iostream>

#include "MM1Queue.hpp"
#include "Event.hpp"

MM1Queue::MM1Queue(double newLambda, int newL, double newAlpha, int newC)
{
    lambda = newLambda;
    L = newL;
    alpha = newAlpha;
    C = newC;
}

void MM1Queue::ProcessQueue()
{
    EventQueue::ProcessResults results;

    for (Event event : eventList)
    {
        switch(event.GetEventType())
        {
            case Event::EventType::Arrival:
            {
                ++results.numArrivals;
                break;
            }
            case Event::EventType::Departure:
            {
                ++results.numDepartures;
                break;
            }
            case Event::EventType::Observer:
            {
                ++results.numObservations;

                if (results.numArrivals > results.numDepartures)
                {
                    results.queueSum += results.numArrivals - results.numDepartures;
                }
                else
                {
                    ++results.idleCount;
                }

                break;
            }
            default:
                break;
        }
    }

    cout << (alpha/5.0)*2000/1000000 << ",";
    cout << results.GetAveragePacketsInQueue() << ",";
    cout << results.GetIdleTimePercent() << endl;
}
```

Question 3: MM1 Queue Analysis

To evaluate E[N] as a function of ρ, a short function was written to loop over ρ values. This loop initialized a queue with a lambda value calculated according to the following equation:

$$\lambda = \rho C / L$$

The queues were initialized with this lambda value, L value, C value and an alpha value set to five times the lambda value. Each queue was simulated fully for each ρ value, and processed to obtain E[N] and PIdle. Results were graphed to produce Figure 1, and the code for this analysis is shown below.

MM1 Queue Analysis Implementation

```
cout << "Rho,E[N],P idle" << endl;
for (float i = 0.25f; i < 1.05f; i += 0.1f)
{
    double lambda = i * LINKRATE / MEANPACKETLENGTH;

    MM1Queue queue(lambda, MEANPACKETLENGTH, lambda * 5, LINKRATE);
    queue.InitalizeQueue(2000.0f);
    queue.ProcessQueue();
}
```

The resulting graph was consistent with what theoretical analysis would point to. An MM1 Queue is a queue of infinite length, meaning that if packets are arriving faster than they can be serviced, they will always be queued. Service time is randomized independent of traffic intensity, which causes the queue to fill up exponentially as traffic intensity increases, leading to the behaviour outlined in Figure 1.

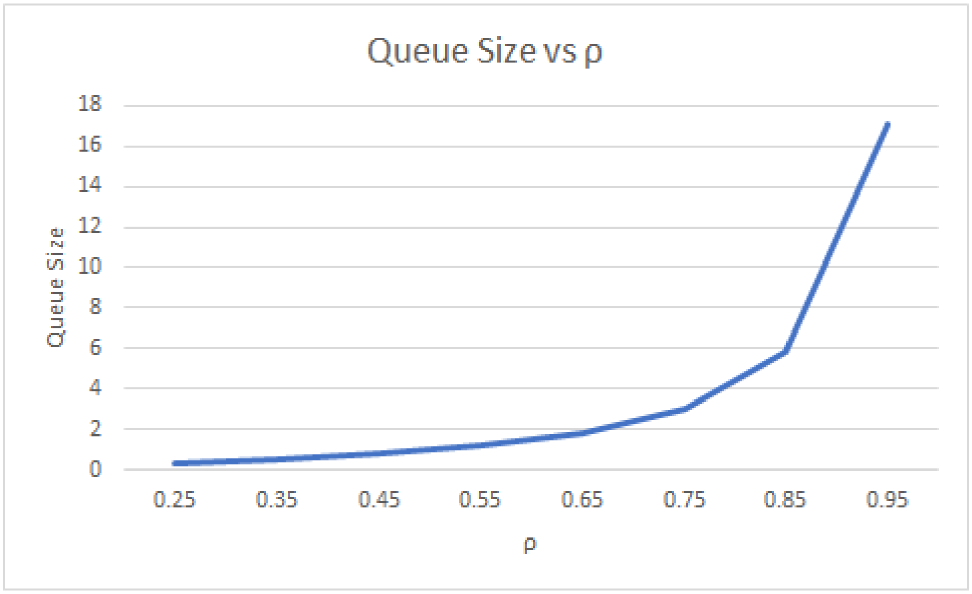

Figure 1: Queue Size vs ρ for MM1 Queue

The queue idle percentage relationship is also consistent with theoretical analysis of an MM1 queue. Intuitively, we find that the idle percentage should be negatively proportional to the frequency of arrivals (more arrivals means less idle time). Additionally, the frequency of arrivals is a function of λ, which is linearly related to rho according to the following equation:

$$\rho = \lambda L / C$$

This leads to a negative linear relationship between queue idle percentage and ρ, which is consistent with the results shown in Figure 2. Intuitively, we would expect the queue to never be idle at ρ=1 and 100% idle at ρ=0 which is shown. As the traffic intensity increases, the percent idle decreases linearly.

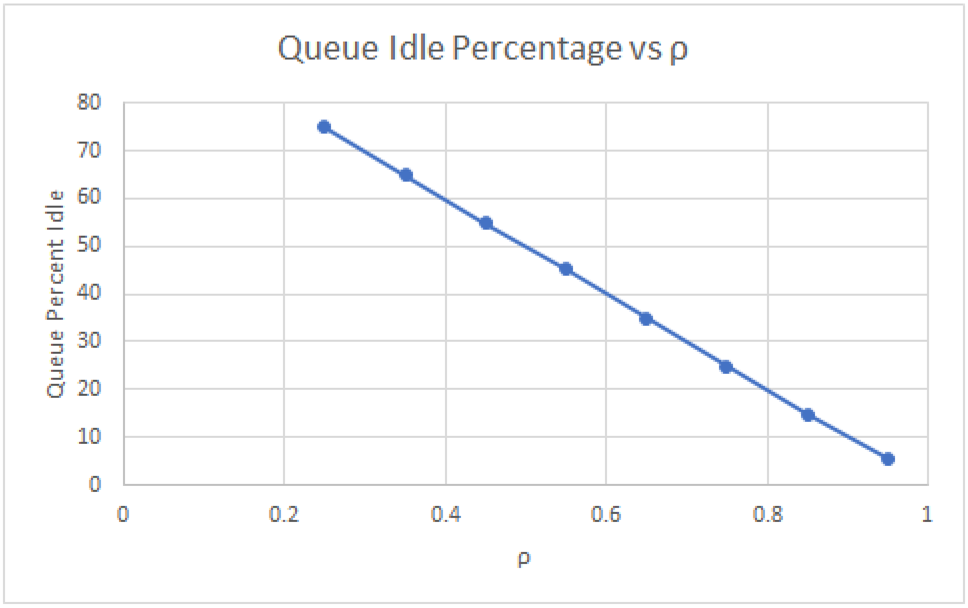

Figure 2: Queue Idle Percentage vs ρ for MM1 Queue

Question 4: MM1 Queue High Traffic Analysis

For ρ = 1.2 we find that that the average number of packets and the queue idle varies per simulation run, and the simulation becomes unstable. However E[N] remains consistently above 100 packets, and the idle percent remains below 0.001%. This result is expected for an MM1 queue because a value of ρ greater than 1 indicates that more packets are arriving than can be serviced, causing the average delay to be high and almost all arriving packets to be queued. If the simulation time were to be increased the queue size would grow linearly, since the queue size is roughly proportional to the difference between the number of generated packets and the number of packets that can be serviced. Because there is an overabundance of packets to service the queue is also active all the time, and the queue length will increase unbounded over time. This exponential growth of queue length is consistent with the graph found in Figure 1.

Question 5: MM1K Queue Design

Also see Question 2 as the MM1K queue utilized a shared base class with the MM1 queue.

The common `EventQueue` class was used to create and fill the queue in Arrival and Observer events, both using the arrival time generation described in Question 2. However, in this case, Departure events were not generated in the queue initialization process. This flag (to disable departure generation) in the queue initialization, and the MM1K class's implementation of the `ProcessQueue` function were the main differentiations between the MM1 and MM1K queue; the rest used shared code.

MM1K Queue Processing Implementation

```
void MM1KQueue::ProcessQueue()
{
    EventQueue::ProcessResults results;

    // Add departure events to both the list and here if applicable
    // if event in queue, base departure time off last event in queue
    std::queue<Event> packetQueue;

    for (auto it=eventList.begin(); it != eventList.end(); ++it)
    {
        Event event = *it;
        switch (event.GetEventType()) {
            case Event::EventType::Arrival:
            {
                ++results.numArrivals;

                if (packetQueue.size() >= K)
                {
                    // Drop packet
                    ++results.numDropped;
                    break;
                }
                else
                {
                    double packetLength = numGen.GenerateRandomValue(1.0f/L);
                    double serviceTime = (packetLength / C);
                    double departureTime = 0;

                    if (packetQueue.size() == 0)
                    {
                        departureTime = event.GetProcessTime() + serviceTime;
                    }
                    else {
                        departureTime = packetQueue.back().GetProcessTime() +
serviceTime;
                    }
                    Event departureEvent(Event::EventType::Departure, departureTime);
                    packetQueue.push(departureEvent);

                    // Insert departure event into current list at correct time
                    auto insertIt = std::next(it);
                    while (insertIt->GetProcessTime() < departureTime && insertIt !=
eventList.end())
                    {
                        insertIt = std::next(insertIt);
                    }
                    eventList.insert(insertIt, departureEvent);
                }
                break;
            }
            case Event::EventType::Departure:
            {
                packetQueue.pop();
                ++results.numDepartures;
                break;
            }
            case Event::EventType::Observer:
            {
                ++results.numObservations;

                if (packetQueue.size() == 0)
                {
                    ++results.idleCount;
                }
                else
                {
                    results.queueSum += packetQueue.size();
                }
            }
            default:
                break;
        }
    }
}
```

After the queue has been initialized, the `ProcessQueue` function above is called. Similar to the MM1 case, it iterates through events one-by-one and processes them. However, a separate explicit processing/packet queue was created. This seperate queue tracks the virtual nodes queue size, and is used to drop events when an event arrives and the queue is already at capacity. When an arival event occurs, if the packet queue is empty its departure time is initialized based on its random service time and the current time. The created departure event is then inserted into the main event queue based on its departure time, as well as being inserted at the end of the packet queue. If the packet queue has departure packets already in it, the new departure event is created based on the event time of the last departure event in the queue and the processing time, and then is inserted at the end of the packet queue, and into the main queue in time-order. If an arrival event occurs and the packet queue is full, it is dropped and the counter is incremented. When a departure event occurs, the departure event counter is increased, and the departure event is popped from the front of the packet queue.

Observer events either increment the idle counter, or add the size of the packet queue to a running counter. When all events have been processed, this counter is divided by the total number of observer events to determine the average number of events in the packet queue. In general, performance metrics were calculated similarly to Question 2.

Question 6: MM1K Queue Analysis

Analysis of the average number of packets in the MM1K queue is shown below in Figure 3. All queues of varying sizes show a S-curve type model. At low traffic (ρ<0.8), an abundance of queue slot resources are available to all arriving packets causing the number of packets in the queue, defined as E[N], to grow roughly linearly, similar to the low ρ<0.6 section of Figure 2 . From (0.8<ρ<1.2), the queue size grows exponentially more traffic is generated than can be processed, causing some to be dropped and some to be stored in a queue of greater size. However, after passing the point of inflection at ρ = 1.2, the asymptotically approaches the maximum queue capacity K due to an overwhelming amount of traffic. This result aligns with intuition, as no matter the level of traffic there cannot be more packets in the queue than it can carry, which is denoted as K.

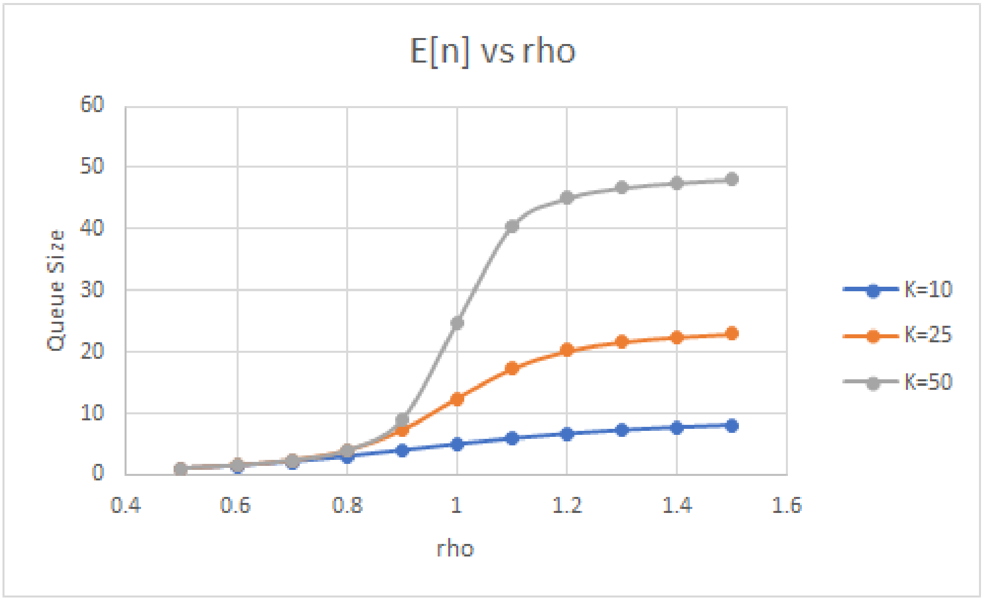

Figure 3: Average Number of Packets in Queue E[N] vs ρ

Percent packet loss behaviour converges into a mostly logarithmic curve with increasing traffic intensity for all values of K. This behaviour is caused by an upper limit of packet loss at 100%, meaning that loss will asymptotically approach this value as traffic intensity increases. This behaviour is outlined in Figure 4. However, upon further inspection of packet loss before traffic intensity is greater than 1, it is apparent that queue size does affect the rate at which packet loss increases. At traffic intensity levels under 1, percent packet loss is roughly inversely and directly proportional to ρ. This is to be expected, as the smaller the queue size, the faster it will reach maximum capacity as traffic intensity increases, causing more packet loss. This behaviour is outlined in Figure 5.

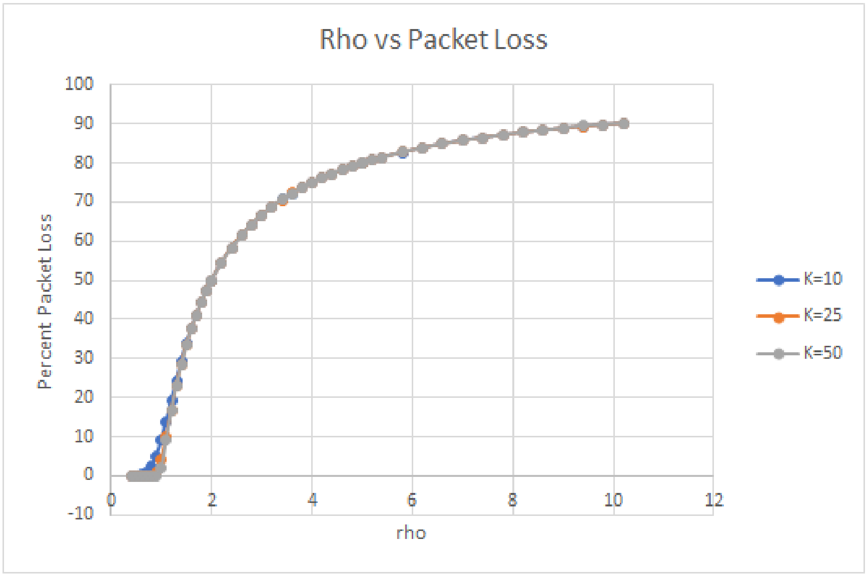

Figure 4: ρ vs Percent Packet Loss at Full ρ Scale

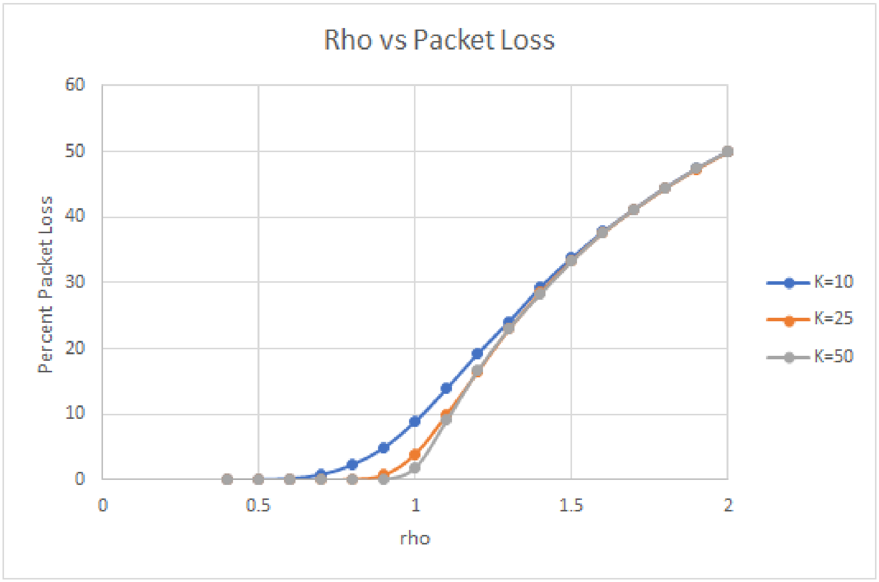

Figure 5: ρ vs Percent Packet Loss at ρ near 1

Simulation Time Tuning

Simulation time was tuned by running each simulation for each question for a total simulation time of 1000 seconds, and doubling simulation time until results were within 5% of the previous simulation. For most simulations a total time of 2000 seconds was sufficient to produce stable results.

Special care was noted for the high traffic MM1 simulation, as the simulation became unstable by nature of the infinitely long queue. This simulation was run at 2000 seconds, but was resimulated if integer overflow caused packet counts to become negative.

Integer overflow was also accounted for in the percent packet loss simulations run in Question 6. At high queue sizes and near the end of the high traffic intensity range integer overflow occurred and resimulation was done for 1000 seconds instead of the full 2000 seconds.