ECE358: Computer Networks

Winter 2018

Project 1: Queue Simulation

Date of Submission: 02/02/2018

Submitted by: Amish Patel

Student ID: 20511960

Student Name: Patel, Amish

a235pate@uwaterloo.ca

Mark received:

Marked by:

TABLE OF CONTENTS

Question 1 3
Question 2 4
Question 3 9
1.. 9
2.. 10
Question 4 10
Question 5 11
Question 6 13
1.. 13
2.. 14

Question 1

To generate a random exponential variable with parameter λ from U(0,1), we use the cumulative distribution function of an exponential random variable $f(x) = 1 - e^{-\lambda x}$.

$$f(x) = 1 - e^{-\lambda x}$$
$$f(x) - 1 = -e^{-\lambda x}$$
$$1 - f(x) = e^{-\lambda x}$$
$$\log(1 - f(x)) = -\lambda x$$
$$-\frac{\log(1 - f(x))}{\lambda} = x$$

In the inverse function above, we will input f(x) with a generated U(0,1) using the rand() function in C++.

The following code generates 1000 exponential random variables.

```
// Generate random number
double EventScheduler::GenerateExponentialRandomNumber(double parameter)
{
	double uniformRandomVariable = (double)rand() / double(RAND_MAX + 1.0);
	//printf("U=%f\n", uniformRandomVariable);
	double num =  log(1.0 - uniformRandomVariable) / (-1.0 * parameter);
	//printf("Num=%f\n", num);
	return num;
}
```

```
void EventScheduler::DisplayRandomNumbers()
{
	double sumMean = 0;

	for (int i = 0; i < 1000; i++)
	{
		double s = GenerateExponentialRandomNumber(75);
		sumMean += s;

		std::cout << "Rand Num: " << s << std::endl;
	}
}
```

The result were outputted to Excel to calculate mean and variance.

Mean: **MEAN**
Variance: **VARIANCE**

This is correct since the expected mean is $\frac{1}{\lambda} = \frac{1}{75} = 0.01333$ and the expected variance is $\frac{1}{\lambda^2} = \frac{1}{75^2} = 0.000177778$.

Question 2

In order to implement the simulator for M/M/1 queue, the simulator was broken down into three components. The first component is the main function. In this function the input parameters are set, such as the buffer utilization value (*rho*), transmission rate (*linkRate*), and average packet length (*avgPacktLength*), buffer size (*bufferSize*) and simulation time (*simulationLength*).

In order to represent a packet, an object called *Event* is used which has member variables for event type (*type*), time (*time*) and length of packet (*length*). The *Event* header file contains the definition for *EventType*. The event types are arrival packets (*Arrival*), departure packets (*Departure*) and observer (*Observer*).

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

	EventType type;
	double time;
	double length;

	Event(EventType type,
double time, double length);
	~Event();
};
```

```
#include "Event.h"

Event::Event(EventType type,
double time, double length)
{
	this->type = type;
	this->time = time;
	this->length = length;
}

Event::~Event()
{
}
```

The second component is the event scheduler which has its own class. The class contains member variables for simulation time (*totalRunTime*), packet length (*packetLength*), lambda (*lambda*) and alpha (*alpha*). The purpose of this class is to generate events of type arrival, departure and observer, where arrival and departures are packets. In order to generate a arrival packet we use a temporary variable to hold current time (*currentTime*) which is initially set to zero. While the current time is less than equal to the simulation time, a random number is generated using our random number generator (*GenerateExponentialRandomNumber(double parameter)*) with a parameter of lambda. This random number represents the time interval from current time to next packet. So to get the arrival time for the packet we add the time interval to the current time. The packet length is determined by using the random number generator with parameter 1/average packet length (*packetLength*). This continues until current time reaches the simulation time and is used to generated arrival packets and observers. Departure packets are generated after the arrival times are created. In order to calculate this a member variable for the furthest departure time (*furthestDepartureTime*) is used. This variable is used to keep track of the last departure time in the system and is updated when a new departure time is created. If the server is busy, then departure time of packet is the sum of *furthestDepartureTime* and server time. If server is not busy, then buffer is empty and there is no buffer delay so the departure time is the sum of the arrival time to the system and the service time. The service time is the packet length divide by the link rate. Also the departure event is only created if the departure time is less than the total simulation time. The event scheduler also creates observer event in the same way arrival packets are generated however the number generator uses the parameter alpha and there is not packet length, set to 0.

```
void EventScheduler::GenerateEvents(Event::EventType eventType, double
randomNumberParameter)
{
	double currentTime = 0;

	// generate event as long as current time is less than simulation time
	while (currentTime < (this->totalRunTime))
	{
		// generate time
		double eventTime = GenerateExponentialRandomNumber(randomNumberParameter);

		// increment current time
		currentTime += eventTime;

		// create new event and add to list
		if (eventType == Event::EventType::Arrival)
		{
			Event newEvent(Event::EventType::Arrival, currentTime,
GenerateExponentialRandomNumber(1 / (this->packetLength)));
			this->eventsList.push(newEvent);

			double departureTime = 0;
			double serviceTime = newEvent.length / this->linkRate;
			if (newEvent.time < furthestDepartureTime)
			{
				departureTime = furthestDepartureTime + serviceTime;
			}
			else
			{
				departureTime = newEvent.time + serviceTime;
			}

			// calculate departure time (arrival time + service time +  buffer
wait)
			// furthestDepartureTime = arrival time + buffer wait

			furthestDepartureTime = departureTime;

			if (departureTime <= this->totalRunTime)
			{
				// create departure event
				Event departureEvent(Event::EventType::Departure, departureTime,
newEvent.length);
				this->eventsList.push(departureEvent);
			}
		}
		else
		{
			Event newEvent(Event::EventType::Observer, currentTime, 0);
			this->eventsList.push(newEvent);
		}
	}

}
```

All of the packets are stored in a priority queue. This is used for fast sorting such that popping from the queue retries the minimum time in the list. When declaring the priority a comparer Comp is used to ensure that the event with the minion time in the queue is retired when popping.

```
#include <queue>
#include "Event.h"
#include <math.h>


struct Comp {
	bool operator()(const Event& event1, const Event& event2)
	{
		return event1.time > event2.time;
	}
};

class EventScheduler
{
public:
	EventScheduler(double totalRunTime, double packetLength, double lambda,
double alpha);
	~EventScheduler();

	// member variables
	double totalRunTime;
	double packetLength;
	double lambda;
	double alpha;
	std::priority_queue<Event, std::vector<Event>, Comp> eventsList;

	// generate events
	void GenerateEvents(Event::EventType eventType, double
randomNumberParameter);
	void GenerateArrivalEvents();
	void GenerateObserverEvents();

	// generate arrival/observer times
	double GenerateExponentialRandomNumber(double parameter);

	// Displaying purposes
	void DisplayRandomNumbers();
};
```

The third component in the simulator. This will remove each event from the priority queue and update counters or calculate measurement based on the type of the event. If the type is arrival the number of generated packets and the arrival counter are incremented. If the type is departure, the departure counter is incremented. If the type is observer, then the number of observers is incremented and if the system is empty, difference between number of arrival packets and number of departure packets is zero, increment the idle counter. As well as this measurements are made. The average number of packet in the system is calculated by dividing the running total number of packet in the system by the number of observers. The proportion of time that the server is idle is calculate by dividing the idle counter by the number of observers.

```
// Process events from buffer
void Simulation::processEvent(Event packet)
{
	if (packet.type == Event::EventType::Arrival)
	{
		// increment number of packets generated
		this->numOfGeneratedPackets++;
		this->numOfPacketsArrival++;
	}
	else if (packet.type == Event::EventType::Departure)
	{
		// Increment departure packets counter
		this->numOfPacketsDeparture++;
	}
	else
	{
		// increment number of observer
		this->numOfObservations++;

		// if server is empty (buffer is also empty) increment idle
counter
		if (isSystemEmpty())
		{
			this->idleCounter++;
		}

		// Get number of packets in system for every observer and
		// keep a running sum for this simulation
		double packetsInSystem = PacketsInSystem();
		this->sumPacketsInBuffer += packetsInSystem;

		// E[n] average number of packets in the system
		this->avgPacketsInBuffer = (double)sumPacketsInBuffer /
(double)numOfObservations;

		//Pidle - proportion of time server is idle
		this->pIdle = (double)idleCounter / (double)numOfObservations;
	}
```

Question 3

1.

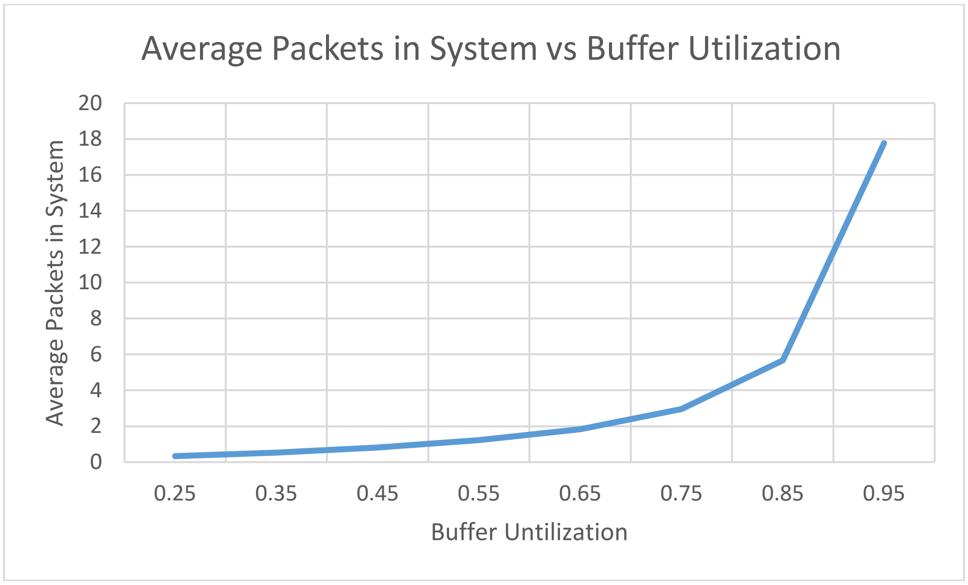

Figure 1: Showing relationship between average in system and buffer utilization

In Figure 1, shows the relationship between the average number of packets in the system and the buffer utilization value (rho). As the rho increases, so does the average number of packets. For the most part the increase is gradually, however once buffer utilization in around 0.75, the average number of packet in the system increase significantly. This is because the rho is proportional to the average number of packets generated per second (lambda), lambda = rho * transmissionRate / averagePacketLength. This means that with a higher buffer utilization values, more packets are going to be generated at a particular moment and since an infinite buffer is used, no packets are lost. Thus, the average number of packets in a system increases significantly at higher rho values.

To calculate the average number of packets in the system E[n], every time an observer is processed, the number of packets in the system at that moment is obtained. This can be done by subtracting the number of departure packets from the number of arrival packets. The obtained number of packets in the system is then added to a running total of the number of packets for the entire simulation. This sum is then divided by the number of observer events to get the average number of packets in the system for a particular buffer utilization value.

2.

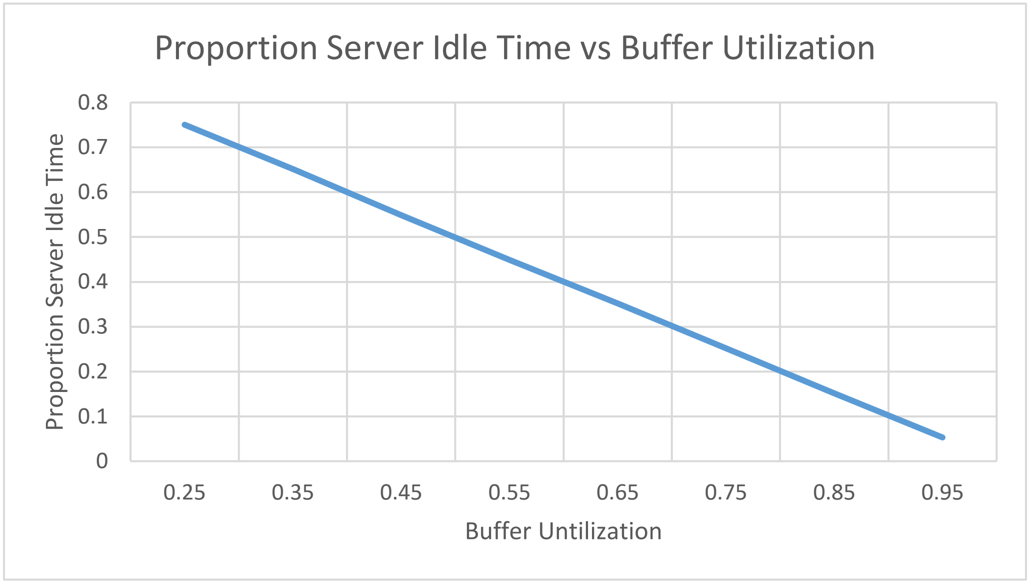

Figure 2: Showing relationship between server idle time and buffer utilization

In Figure 2, shows the relationship between the proportion server idle time and the buffer utilization value. From the graph, as the buffer utilization increases, the proportion time that the server is in idle decreases linearly. This is because as the buffer utilization increases, the average number of packets generated per second (lambda) also increases. Thus at higher buffer utilization values, more packets are generated making the server is most likely processing the packets rather than being in an idle state.

To calculate the service idle time, the number of times the system is empty is divided by the number of observer events. To determine the number of times the system is empty, a counter is incremented for an observer event when the difference between the number of arrival packets and the number of departure packets is zero. A counter is also used to keep track of how many observer events have been processed.

Question 4

Buffer Utilization	E[n]	Pidle	Number of Arrival Packets	Number of Departure Packets
1.200000	82771.031989	0.000008	999818	833432

From the observations above, when the buffer utilization value is 1.2 the average number of packets generated per second is 82771, which increases that there is a significant increase in packets generated as buffer utilization increases. This is correct as the lambda = buffer utilization * transmission rate /average packet length. Also, since there are so many packet generated, the system is busy in terms of processing the packets. This means that the server idle time would be close to zero which can be seen in the results above.

Question 5

In order to implement the M/M/1/K queue, the generation of the departure packets was removed from the event scheduler. Instead, it's moved to the simulator. As a result, the event scheduler only generates the arrival packets and observers. The code from the event scheduler for departure packet generation is copied and placed in the similar when an arrival packet is processed. Before the departure packet can be created, a new function *isBufferFull()* is used to determine in buffer is full. If so, packet drop counter is incremented to drop the current arrival packet being processed. If buffer has room, then increment arrival packet counter and generate the departure packet using the same code as in the event scheduler.

Simulation.cpp

```
// Check if buffer is full
// If buffer size is -1, buffer
size is infinite
bool Simulation::isBufferFull()
{
	if (bufferSize == -1)
	{
		return false;
	}
	return
(PacketsInSystem()) >
bufferSize;
}
```

```
// Process events from buffer
void Simulation::processEvent(Event packet)
{
	if (packet.type == Event::EventType::Arrival)
	{
		// increment number of packets generated
		this->numOfGeneratedPackets++;

		if (isBufferFull())
		{
			// Increment packet lost counter
			this->numOfDroppedPackets++;
		}
		else
		{
			// Increment packet arrival counter
			this->numOfPacketsArrival++;

			// calculate departure time (arrival time + service time + buffer
wait)
			// furthestDepartureTime = arrival time + buffer wait
			double serviceTime = packet.length / this->linkRate;
			double departureTime = 0;
			if (!isServerBusy())
			{
				departureTime = packet.time + serviceTime;
			}
			else
			{
				departureTime = this->furthestDepartureTime + serviceTime;
			}
			this->furthestDepartureTime = departureTime;

			if (departureTime <= this->simulationTime)
			{
				// create departure event
				Event departureEvent(Event::EventType::Departure,
departureTime, packet.length);
				this->eventsList->push(departureEvent);
			}
		}
	}
	else if (packet.type == Event::EventType::Departure)
	{
		// Increment departure packets counter
		this->numOfPacketsDeparture++;
	}
	else
	{
		// increment number of observer
		this->numOfObservations++;

		// if server is empty (buffer is also empty) increment idle counter
		if (isSystemEmpty())
		{
			this->idleCounter++;
		}
```

```
              // Get number of packets in system for every observer and
              // keep a running sum for this simulation
              double packetsInSystem = PacketsInSystem();
              this->sumPacketsInBuffer += packetsInSystem;

              // E[n] average number of packets in the system
              this->avgPacketsInBuffer = (double)sumPacketsInBuffer /
(double)numOfObservations;

              //Pidle - proportion of time server is idle
              this->pIdle = (double)idleCounter / (double)numOfObservations;

              //ploss - packet loss probability
              this->pLoss = (double)numOfDroppedPackets /
(double)numOfGeneratedPackets;
       }
```

Question 6

1.

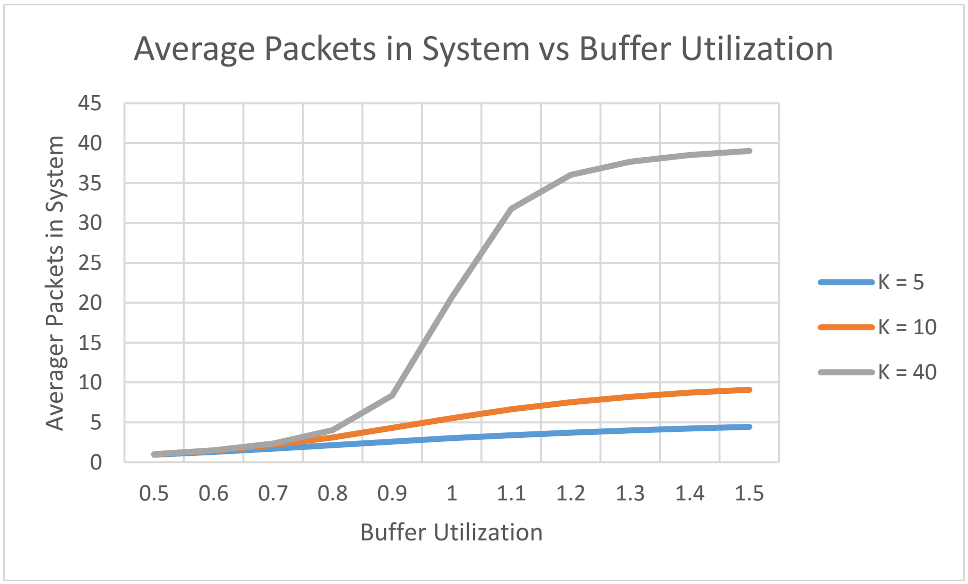

Figure 3: Showing relationship between average packet in system and buffer utilization

Figure 3 show the relationship between the average number of packets in the system and how the buffer is used for different buffer sizes (K). The buffer sizes are 5, 10 and 40. When the buffer size is 5, the average packets in the system increases slightly as buffer utilization increases and seems to saturates to E[n] = 5. This is also seen when the buffer size is 10, but the rate at which average packets in the system increases is slightly larger. The pattern is different for buffer size of 40 as rate of average buffers in system increase at larger rates from buffer utilization 0.5 to 1.1. Then the rate of increases decreases and continues to decrease as the graph approaches 40 packets in the system. This indicates saturation that there is a saturation point at 40. Thus we can conclude that as buffer utilization increases for a finite buffer size, the average number of packet in the system will saturate to the buffer size.

Comparing these results to the case with an infinite buffer size, we can see that the average number of packets in the buffer increases and buffer utilization increases. However, the rate of increase for a finite buffer size is samller than for an infinite buffer size. In the infinite case, the rate of increase for the average packet in the system seem exponential, whereas in the finite case, the rate increases seem more linear. Also, in the finite case the average number of packets saturates at the buffer size. This does not occur for an infinite buffer size.

2.

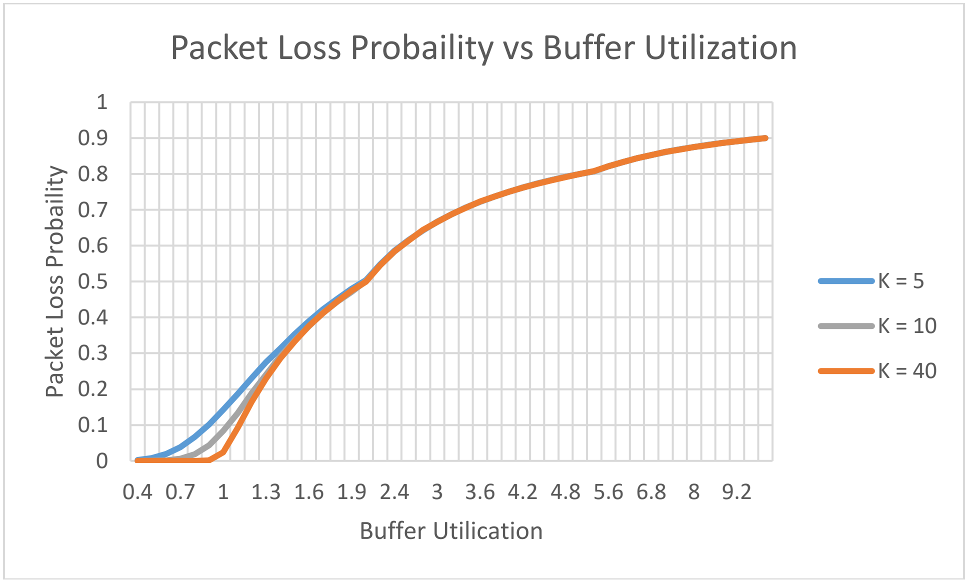

Figure 4: Showing the relationship between packet loss probability and buffer utilization

Figure 4 shows the relationship between the packet loss probability and buffer utilization for different buffer sizes. The buffer sizes that are used are 5, 10 and 40. As buffer utilization increases, the packet loss probability increases for all of the buffer sizes. The packet loss probability starts off really low. This is due to that fact that a low buffer utilization means that there are a small number of packets generated (indicated by lambda). This indicates that the server will likely process all the packets that enter the system, minimizing packet loss which can been seen in the graph for K = 40 where packet loss portability is nearly zero at the start. The packet loss probability starts to increase as buffer utilization increases. However, smaller buffer sizes with experience packet losses quicker as larger buffers can handle more packets. This is indicated in the graph by K = 5 increasing first, then K = 10 and lastly K = 40. As buffer utilization increases, the probability of packet loss increases quickly then slows done as probability approaches 1. This is due to the fact that it is takes longer to lose all the packets in the system.

To calculate the probability of packet loss, the number of dropped packets is divided by the number of packets generated.