358 Project 1 Lab Report

By Zahin Mohammad (z5mohamm) and Brian Norman (b2norman)

Table of Contents

Question 1

The variance and mean for a Poisson distribution is determined in reference to λ as show below.

$$Mean = \frac{1}{\lambda}$$

$$Variance = \frac{1}{\lambda^2}$$

The comparison of this expected relationship and the results from 1000 randomly generated random variables are show in the table below.

	Mean	Variance
Expected	$\frac{1}{\lambda} = \frac{1}{75} = 0.0133$	$\frac{1}{\lambda^2} = \frac{1}{75^2} = 0.000177$
Actual	0.013541	0.000179

The results from the 1000 generated random variables are accurate to two significant digits. This is what we would have expected for a Poisson Distribution.

Question 2 & Question 5

To build the simulator, our group created a class called Event:

```
class Event {
    EventType type;
    double occurrenceTime;

    Event(EventType type, double occurrenceTime) {
        this.type = type;
        this.occurrenceTime = occurrenceTime;
    }
}
```

With an extension for the “Departure” event:

```
class Departure extends Event {
    Event arrival;
    Departure(EventType type, double occurrenceTime, Event arrival) {
        super(type, occurrenceTime);
        assert arrival.type == EventType.ARRIVAL;
        this.arrival = arrival;
    }
}
```

Where “EventType” is enumerated:

```
enum EventType {
    ARRIVAL, DEPARTURE, OBSERVE
}
```

The simulation events and processing happen in a “Simulation” class with a structure as show below.

```
public class Simulation {
    private double simulationTime;
    private double packetLength;
    private double packetGenerationAvg;
    private double linkCapacity;
    private int bufferSize = -1;

    Simulation(SimulationParams simulationParams){
        this.simulationTime = simulationParams.simulationTime;
        this.packetLength = simulationParams.packetLength;
        this.packetGenerationAvg =
                (simulationParams.queueUtilization *
simulationParams.linkCapacity)/simulationParams.packetLength;
        this.linkCapacity = simulationParams.linkCapacity;
        this.bufferSize = simulationParams.bufferSize;
    }

    public SimulationResponse runSimulation() {…}

    private  ArrayList<Event> generateEvents() {…}
}
```

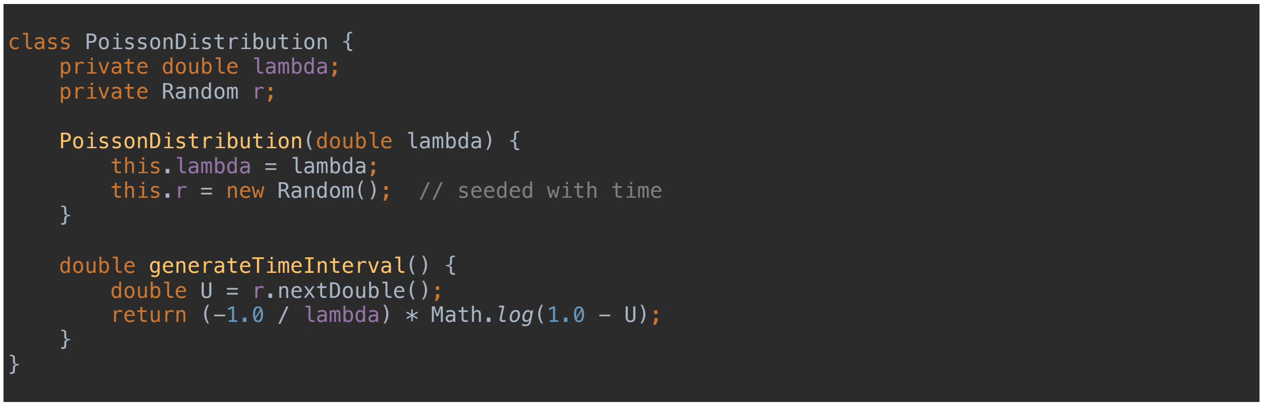
With “SimulationParams” as:

```
public class SimulationParams {
    double simulationTime;
    int packetLength;
    int linkCapacity;
    double queueUtilization;
    int bufferSize = -1;

    public SimulationParams(double simulationTime, int packetLength, int linkCapacity,
double queueUtilization, int bufferSize) {
        this.simulationTime = simulationTime;
        this.packetLength = packetLength;
        this.linkCapacity = linkCapacity;
        this.queueUtilization = queueUtilization;
        this.bufferSize = bufferSize;
    }
    public SimulationParams(double simulationTime, int packetLength, int linkCapacity,
double queueUtilization) {
        this.simulationTime = simulationTime;
        this.packetLength = packetLength;
        this.linkCapacity = linkCapacity;
        this.queueUtilization = queueUtilization;
    }
}
```

With “SimulationResponse” as:

```
public class SimulationResponse {
    double pIdle;
    double pLoss;
    double avgPacketsInBuffer;

    public SimulationResponse(double pIdle, double pLoss, double avgPacketsInBuffer) {
        this.pIdle = pIdle;
        this.pLoss = pLoss;
        this.avgPacketsInBuffer = avgPacketsInBuffer;
    }
}
```

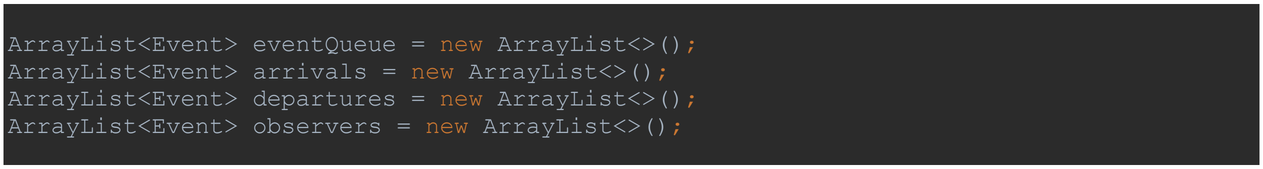
We also have a Poisson Distribution class which generates new time intervals based on lambda:



```java
class PoissonDistribution {
    private double lambda;
    private Random r;

    PoissonDistribution(double lambda) {
        this.lambda = lambda;
        this.r = new Random();  // seeded with time
    }

    double generateTimeInterval() {
        double U = r.nextDouble();
        return (-1.0 / lambda) * Math.log(1.0 - U);
    }
}
```

generatEvents:

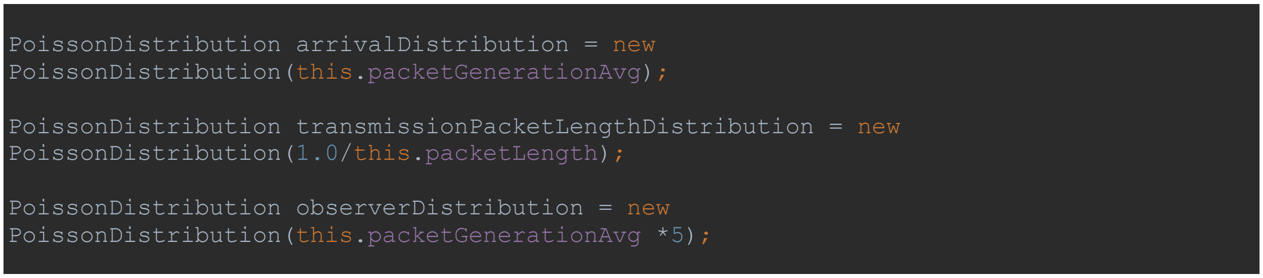
1. First we initialize lists to hold arrival, departure and observer events along with the final event queue.



```java
ArrayList<Event> eventQueue = new ArrayList<>();
ArrayList<Event> arrivals = new ArrayList<>();
ArrayList<Event> departures = new ArrayList<>();
ArrayList<Event> observers = new ArrayList<>();
```

2. Next we create new Poisson distributions for arrival time, transmission length and observer time. The variables are summarised in the table below.

packetGenerationAvg	λ
packetLength	L



```java
PoissonDistribution arrivalDistribution = new
PoissonDistribution(this.packetGenerationAvg);

PoissonDistribution transmissionPacketLengthDistribution = new
PoissonDistribution(1.0/this.packetLength);

PoissonDistribution observerDistribution = new
PoissonDistribution(this.packetGenerationAvg *5);
```

3. Arrival and departure events are created in a loop until the entire range of the simulation time has been reached. The loop variable is the "currentTime" which increments in steps of an exponential random variable with mean of,

$$Mean = \frac{1}{\lambda}$$

that is referenced as "arrivalDistribution".

Arrival events have an occurrence time equal to the "currentTime", which is the incrementing value in the loop.

Departure events have an occurrence time that is dependant on the previous departure event time if available and the packet length. If a previous timestamp is available, then the departure occurrence time will be the maximum of the previous departure time and the “currentTime” plus the transmission time, else it will just be the transmission time.

The transmission time is generated using the “linkCapacity” and the “transmissionLength” which is an exponential random variable with a mean of,

$$Mean = L.$$

The transmission time is defined below, where “L” represents the packet length in bits and “C” represents the link capacity in bits/s.

$$Transmission\ Time = \frac{L}{C}$$

```
double currentTime = 0.0;
// Generate events up until simulation time
while (currentTime < simulationTime) {
    double departureOccurrenceTime = currentTime;
    // Packet has to wait until previous packet has departed
    if (!departures.isEmpty() && departures.get(departures.size()-1).occurrenceTime >
currentTime){
        departureOccurrenceTime = departures.get(departures.size()-1).occurrenceTime;
    }
    double transitionTime =
transmissionPacketLengthDistribution.generateTimeInterval() / this.linkCapacity;
    departureOccurrenceTime += transitionTime;

    Event newArrival = new Event(EventType.ARRIVAL,currentTime);
    Event newDeparture = new Departure(EventType.DEPARTURE, departureOccurrenceTime,
newArrival);
    departures.add(newDeparture);
    arrivals.add(newArrival);
    currentTime += arrivalDistribution.generateTimeInterval();
}
```

4.

Observer events are generated in a separate loop, but in a similar manner to the arrival and departure events. Observer events happen until simulation time in increments of “observerDistribution” which is an exponential random defined with a mean of,

$$Mean = \frac{1}{\lambda * 5}.$$

```
currentTime = 0.0;
while (currentTime < simulationTime){
    double observationTime = observerDistribution.generateTimeInterval();
    Event newObserver = new Event(EventType.OBSERVE, currentTime);
    observers.add(newObserver);
    currentTime += observationTime;
}
```

5. Once all the events have been generated, they’re combined into one EventQueue and sorted by their occurrence time:

```
eventQueue.addAll(arrivals);
eventQueue.addAll(departures);
eventQueue.addAll(observers);
eventQueue.sort(Comparator.comparingDouble(s -> s.occurrenceTime));
return eventQueue;
```

runSimulation:

Now that the eventQueue is populated up to simulation time, we need code to run the simulation. A LinkedList was used for the buffer, which acts like an infinite buffer if the buffer size is a negative value. This enabled the simulation to be configured for both a M/M/1 and M/M/1/K queue based on buffer size input. All the events in the event queue are looped over with a switch case handling each event type (Arrival/Departure/Observe).

Logic of the arrival case:

- Increment number of arrivals counter
- Add event to a set of dropped packets if the buffer is full
- Add to the buffer if buffer is not full

Logic of the departure case:

- Remove the arrival event from the buffer if it has not been dropped

Logic of the observe case:

- Increment an idle counter if buffer is empty
- Increment a counter for the size of the buffer
- Increment number of observations counter

Average number of packets in the buffer/queue, E[N], is calculated as shown below:

$$E[N] = \frac{bufferSizeCounter}{observerCounter}$$

The proportion of time the server is idle, P_{idle} is calculated as shown below:

$$P_{idle} = \frac{idleCounter}{observerCounter}$$

The packet loss probability (for M/M/1/K queue), P_{loss} is calculated as show below:

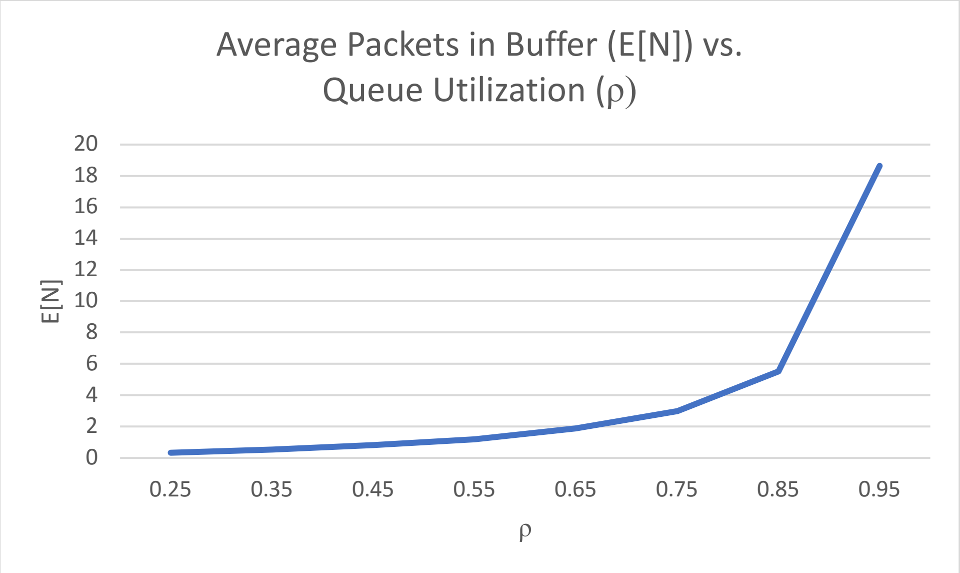
$$P_{loss} = \frac{|\{dropped\ arrival\ events\}|}{|\{arrival\ events\}|}$$

```
ArrayList<Event> eventQueue = generateEvents();
LinkedList<Event> buffer = new LinkedList<>();
HashSet<Event> packetsDropped = new HashSet<>();

double packetArrivalCounter = 0.0;
double idleCounter = 0.0;
double observerCounter = 0.0;
double bufferSizeCounter = 0.0;

for (Event event : eventQueue) {
    switch (event.type){
        case ARRIVAL:
            packetArrivalCounter++;
            if (this.bufferSize >-1 && buffer.size()>=this.bufferSize){
                packetsDropped.add(event);
            }else{
                buffer.add(event);
            }
            break;
        case DEPARTURE:
            if (!packetsDropped.contains(((Departure) event).arrival)){
                assert buffer.size() != 0;
                buffer.pop();
            }
            break;
        case OBSERVE:
            // Increment idle time whenever packet arrives to an empty buffer
            if (buffer.isEmpty()){
                idleCounter++;
            }
            bufferSizeCounter += buffer.size();
            observerCounter++;
            break;
    }
}
return new SimulationResponse(
        idleCounter / observerCounter,
        (double) packetsDropped.size() / packetArrivalCounter,
        bufferSizeCounter / observerCounter);
```

Question 3

3.1

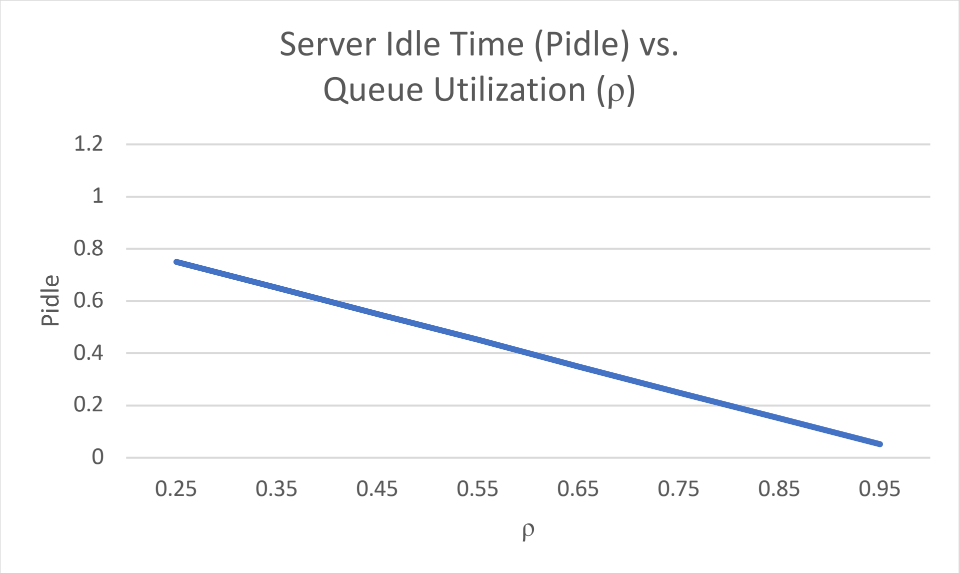
E[N] as we vary ρ (queue utilization) from 0.25 < ρ < 0.95 in increments of 0.1 can be seen in the graph below. This data was obtained by running the simulation in a for loop and keeping all the simulation parameters the same except for the queue utilization. The constant parameters are L=2000, and C=1Mbps. The response from each simulation is a “SimulationResponse” object which includes the average number of packets in the buffer. This data was then exported into a CSV.



The queue utilization is not linearly proportional to the average number of packets in the buffer. An increase in queue utilization results in a non-linear increase in average number of packets in the buffer.

3.2

P_{idle} as we vary ρ (queue utilization) from $0.25 < \rho < 0.95$ in increments of 0.1 can be seen in the graph below. This data was obtained by running the simulation in a for loop and keeping all the simulation parameters the same except for the queue utilization. The constant parameters are L=2000, and C=1Mbps. The response from each simulation is a "SimulationResponse" object which includes proportion of time the server spent in idle. This data was then exported into a CSV.



The queue utilization is linearly inversely proportional to the server idle time. An increase in queue utilization results in a linear decline of the server idle time. This makes sense given using more of the queue would obviously translate to less time when the queue is not being used!

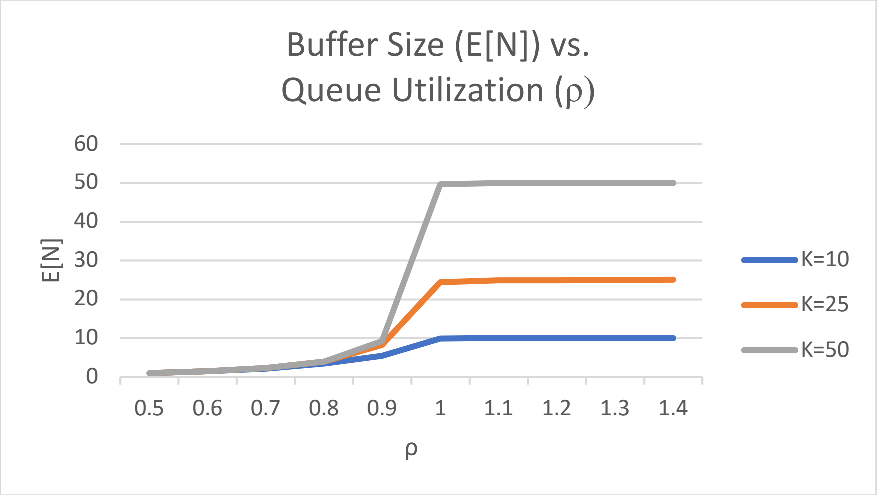
Question 4

The result of running the simulation with a queue utilization of 1.2 is shown below.

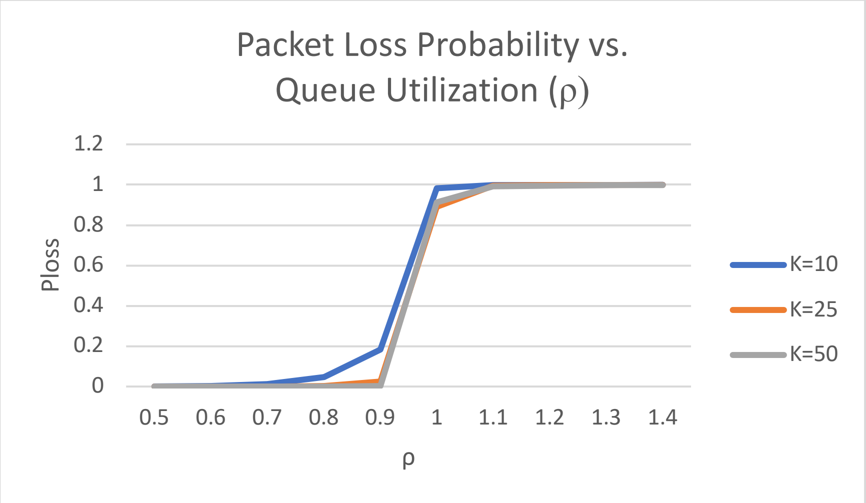
Simulation Time	Buffer	C	L	ρ	E[N]	P_{idle}	P_{loss}
1000	Infinite	1000000	2000	1.2	50114.43811	0.000003	0

We see almost no P_{idle} once the queue utilization crosses 1.0, which makes sense as the E[N] (average number of packets in queue) is extremely high at over 50 thousand packets. This is because the arrival rate into the network surpasses the service rate of each packet, meaning packets are arriving faster than they can leave. If this were a finite queue, we would be sure to see packets lost because the pace is continuous, meaning the longer the simulation time, the longer the queue would grow.

Question 6



As queue utilization increases so does the average buffer size. There is a plateau in each case at "K" (queue max size). This is because it is impossible to have more then K packets in the buffer. It seems the average number of packets in the buffer is not linearly proportional to queue utilization.



As queue utilization increases, so does the packet loss probability. Once queue utilization reaches and surpasses a value of one, probability of packet loss is almost 100%. This is due to the fact that once the queue is filled, and packets are arriving faster then are being sent out, there will always be packets that are dropped.

P_{loss} was obtained by the equation shown below.

$$P_{loss} = \frac{|\{dropped\ arrival\ events\}|}{|\{arrival\ events\}|}$$

Each simulation returns an object of "SimulationResponse" that includes P_{loss} as a field.