ECE358: COMPUTER NETWORKS

Project 1: MD1 and MD1K Queue Simulation

Date of Submission: October 14th 2016

20460413: Ramie Raufdeen: mrmohame@uwaterloo.ca
20465897: Alex Ohn: amohn@uwaterloo.ca

Marks Received:

Marked By:

Table of Contents

MD1Queue Design

The software we designed and developed to run simulations for this report uses a linked-list based data structure to represent the queues. The main entity in the program running the simulations is a `QueueSession`. A `QueueSession` takes in the parameters `BufferSize`, `Ticks`, `PacketSize`, and `ServiceTime`. Using the `PacketSize` and `ServiceTime`, the Lambdas are calculated. Since the MD1K queue is an extension of an MD1 queue, it uses the `MD1Queue` class as a base class and overrides the `getLambda()` method to provide its own set of lambdas. A `QueueSession` is generated once with the properties. However, that same `QueueSession` is run 5 times to obtain an average for each lambda value (in the lab manual) – and eliminate any outliers. The `getLambda()` method for the `MD1QueueSession` generates 5 lambdas. For each Lambda, a `LinkBuffer` is created to simulate it, along with a `QueueSimulationReport`. The `QueueSimulationReport` acts as a subscriber to any metrics that the `LinkBuffer` and the `QueueSession` provides. Upon completion of the simulation of the lambdas, the `QueueSession` returns a list of `QueueSimulationReport` which contains metadata for every simulation run (based off of lambda value).

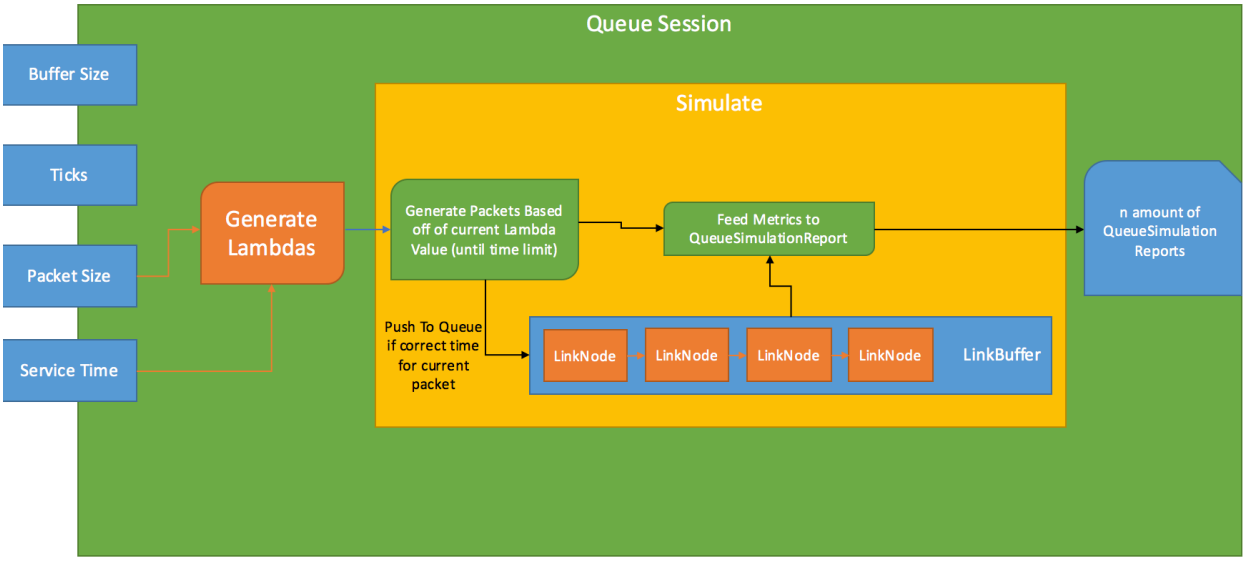
The `LinkBuffer` is constructed with a max size and acts as the queue that processes the packets. The `Append` method is used to add packets to the queue and `Check` is used to make sure the appropriate amount of process time has passed before considering the packet as processed. Within our `LinkBuffer` we also include a method that generates random numbers using the lambda value that determines when a packet should be placed into the queue (if space is available).

To generate the metrics measured for the report generated we have various measure within our code to make sure our results are valid. Idle time is measured by keeping track of how many ticks are spent where our buffer length is 0. We consider a packet to be sent when it has successfully been put through the queue and has been serviced. A packet that tries to be placed into the queue when the queue is full gets added into the packet lost counter. To determine when the report statistics should be printed out, we make sure the number of sent packets is roughly around the same amount that should be generated within the current time frame (`lambda` * `timeLength`).

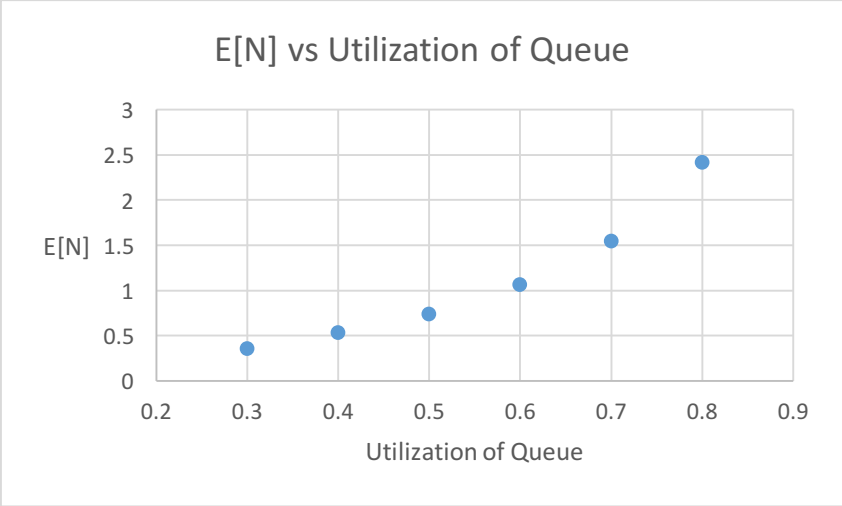
Here are the metrics calculated by the `QueueSimulationReport` entity.

Metric Name	Description
averagePacket	The average number of packets sent through the queue. *sentPackets / timeLength*
idleTime	The total amount of time that the queue has been idle. This is done in program by checking the queue every tick and converting the amount of times to seconds. *idleCount/1000000*
sojournTime	The average amount of time to process a packet. This is calculated from the buffer by subtracting the time a packet enters the queue from the current time (*cumulativeTime*). This is then divided by the number of packets to average out the sojourn time. *cumulativeTime/receivedPackets*
averagePacketsInBuffer	The amount of packets in the queue on average. This is calculated by adding the amount packets in the buffer for every tick and dividing it by the total ticks. *packetBuffer/(timeLength*1000000)*

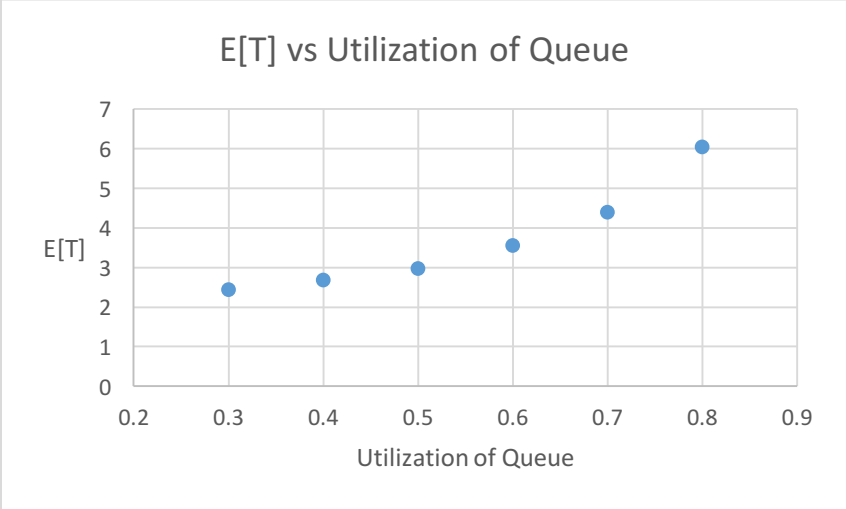
A diagram of our software design



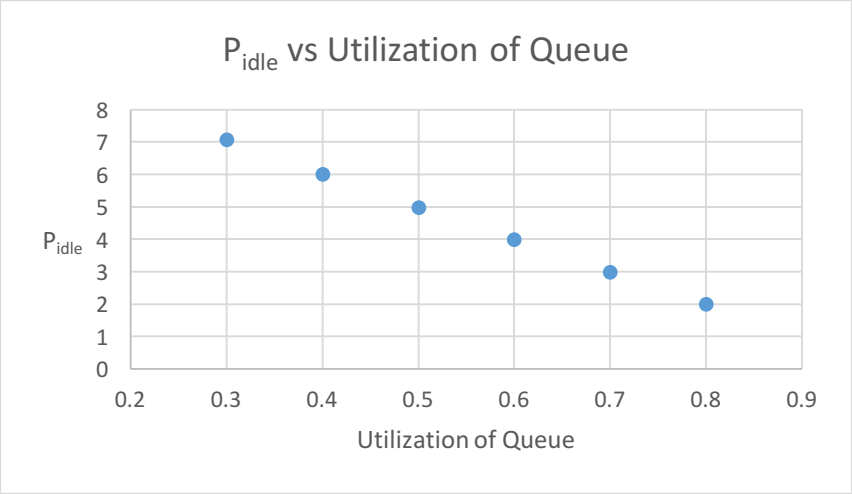
MD1 Queue Simulations



The average number of packets in a queue that has an infinite sized buffer increases exponentially as the utilization of the queue increases. The logic behind this follows as the more packets are added to an infinite queue the more packets will be in the queue.



This graph shows that much like the behavior that is present between average packets in queue vs utilization, the average sojourn time also increases exponentially as utilization increases. This makes sense since as more packets are generated there are more chances for a queue to form increasing the sojourn time.



This graph shows a negative linear relationship between P_{idle} and the utilization of the queue. The graph is in line with the programming logic as more packets are generated, there are more instances where packets will need to be serviced. Overall, this will result in a decrease in idle time.

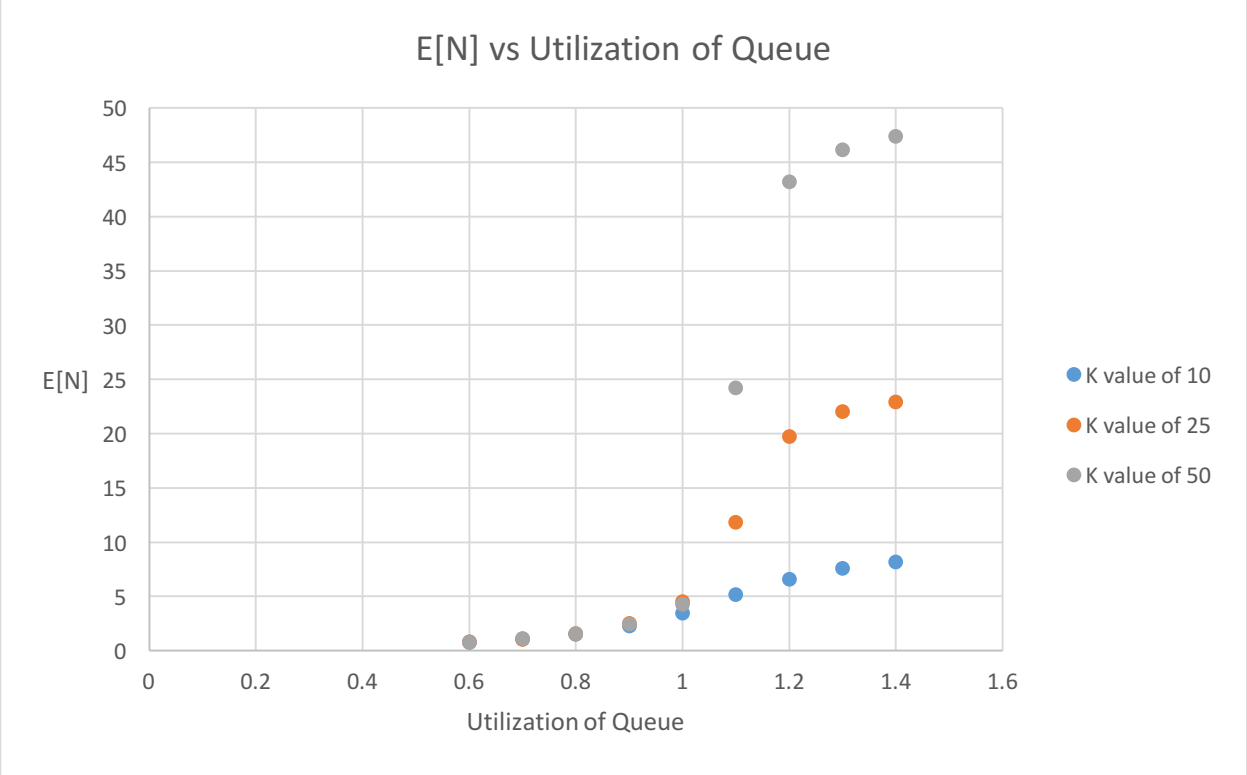
MD1K Queue Design

The simulation for the MD1K Queue was done using the MD1Queue data structure, but overriding the method for calculating lambdas. Also, there was a set size for the queue instead of an infinite one, which is set by the K value during our simulations. The main difference for the LinkBuffer when simulating for a `MD1KQueueSession` is the check to see if the queue is full when appending. The `append()` method would return false if the queue was full. As a result, the `MD1KQueueSession` would increment the counter for the `lostPackets` since the data was not added to the queue.

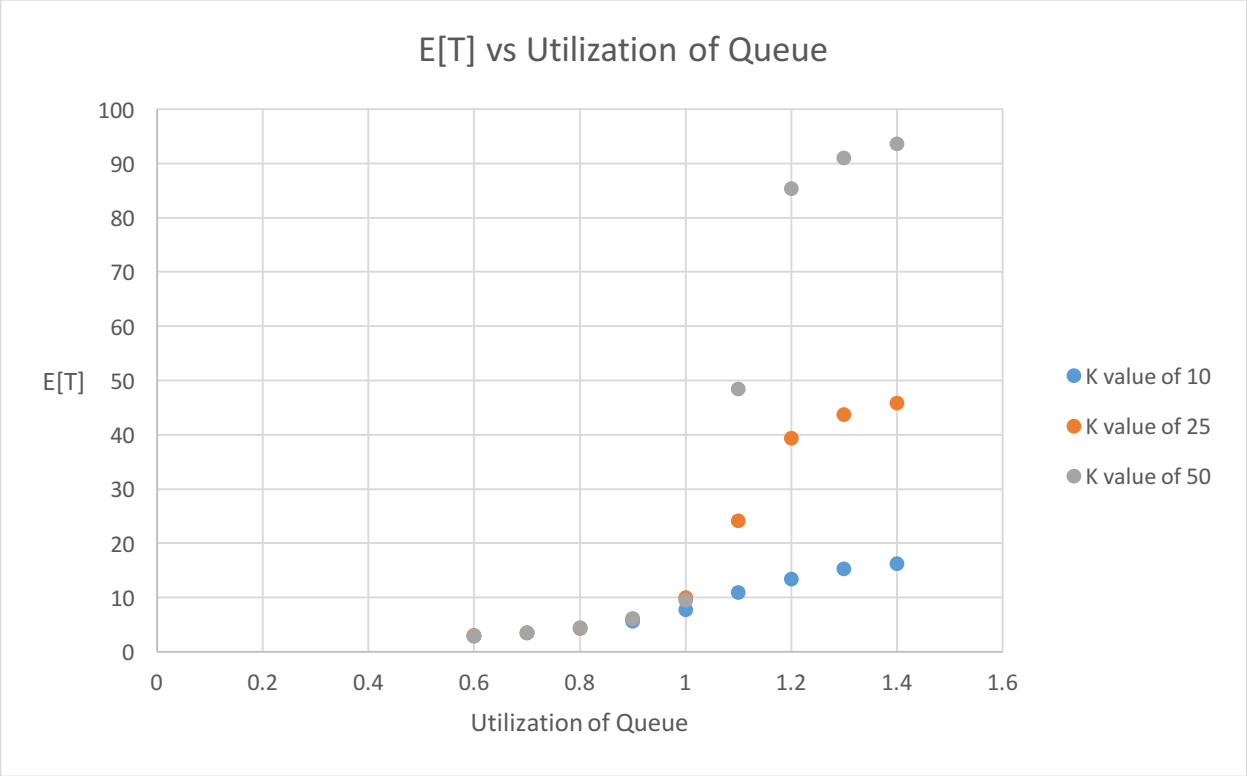
To further examine the runtime of the `MD1KQueueSession`, there are approximately 9 lambda values generated for the simulation. In addition, there are 3 K values/buffer sizes that need to be simulated, along with 5 runs for averaging. As a result, the `MD1KQueueSession` would be simulating approximately 135 simulations that need to run for the `timeLength`. Hence, getting simulation numbers for the MD1K queue takes a lot longer than the MD1 queue.

In essence, the code for MD1KQueue ***simulation*** is the exact same as it is for MD1Queue simulations. However, the `LinkBuffer` used in the MD1KQueue simulation has a finite size now, as required. Also, the lambdas are calculated differently.

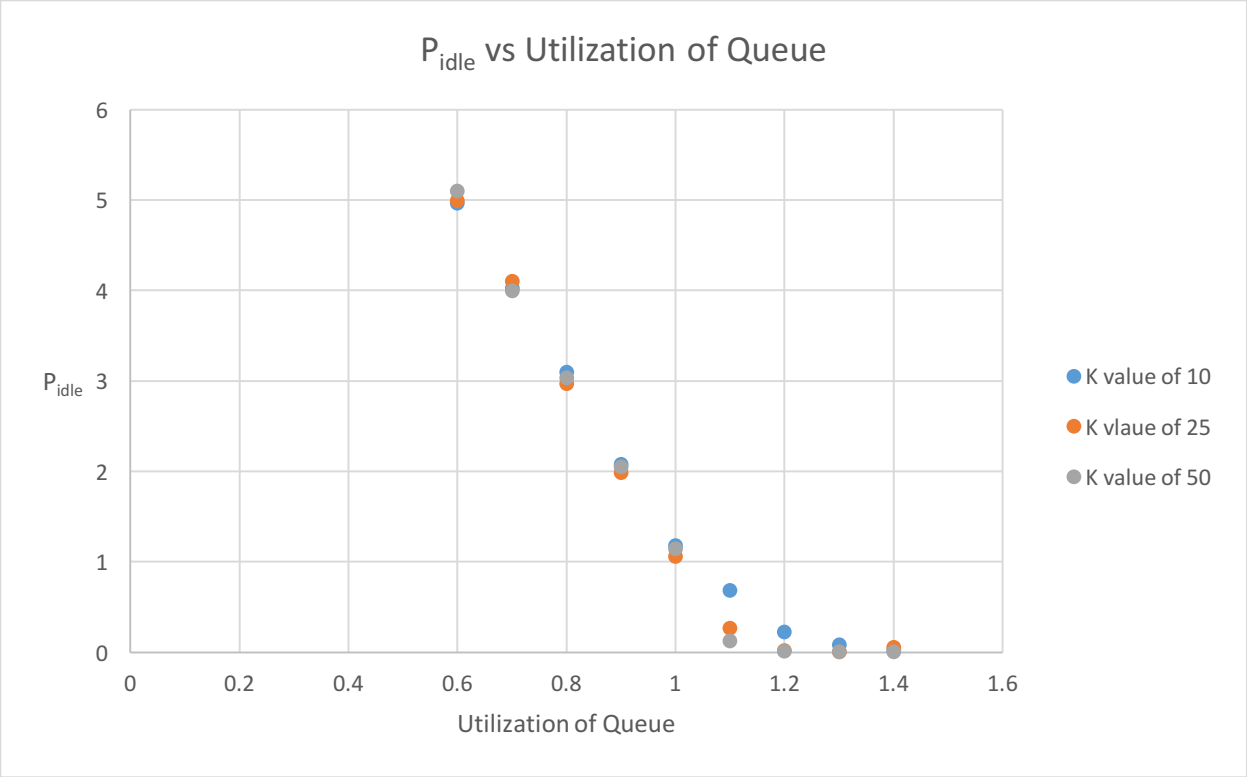
MD1K Queue Simulations



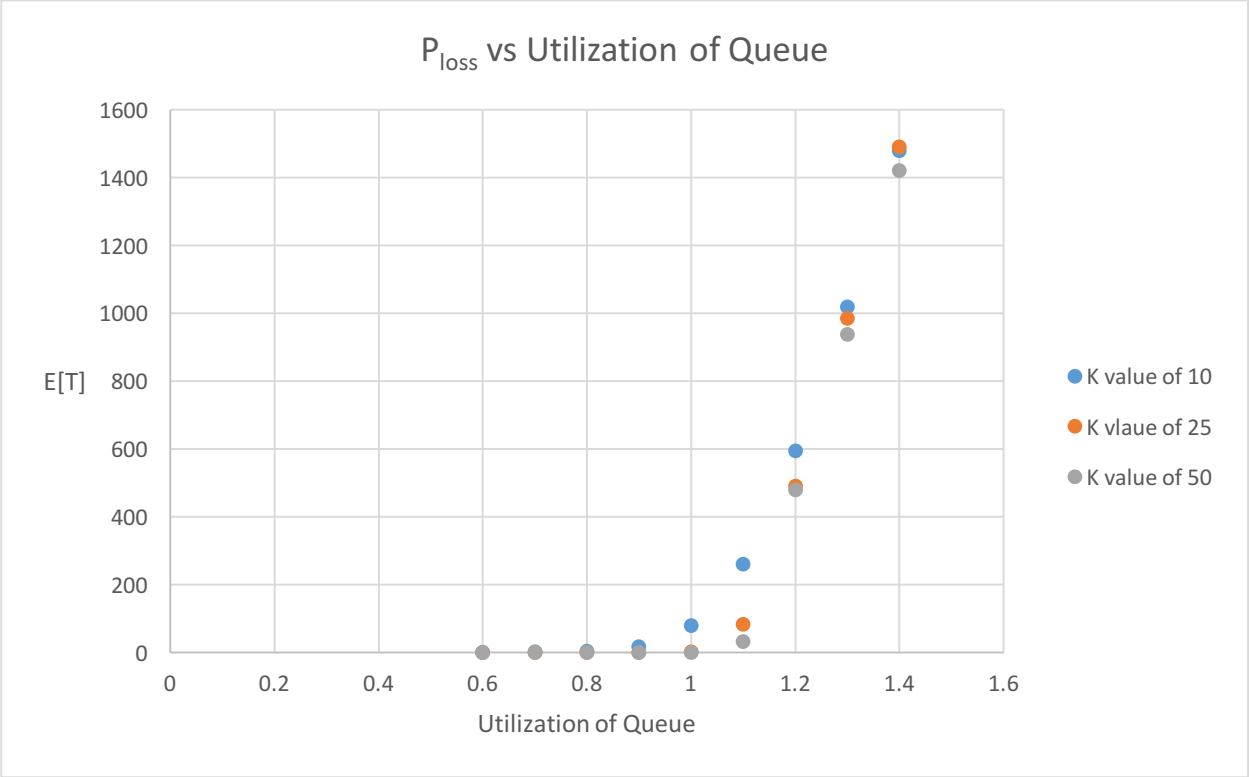
A logarithmic relationship exists between the utilization of the queue and the average number of packets in the queue. This occurs due to their being a limitation on the queue size. Even as you try to increase the amount of packets generated per second, once you max out the queue, the majority of the time it is impossible to increase the average amount of packets in the queue past capacity. As the buffer size increases the amount of packets available for the queue also increases but the relationship between average number of packets and utilization of the queue remains the same.



The behavior between the average processing time and the utilization of the queue is also on a logarithmic scale. This is similar to the previous graph, the amount of time needed to process a packet will increase with the more items in the queue. As more items are generated per minute the queue will be full more constantly increasing the amount of time needed. However, as the queue reaches capacity more frequently the amount of time needed to process a packet will begin to stabilize. Also, as the value of K increases the relationship remains the same with higher limits.



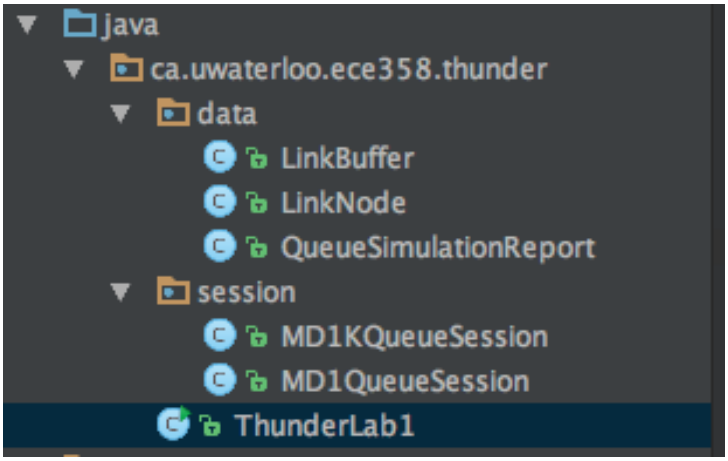
This graph demonstrates that as queue utilization is increased, the amount of time the server spends idle decreases. This occurs since as the number of packets sent increases, there are less instances when the server is not servicing a packet which directly correlates with idle time. However, is slopes towards 0 because idle time cannot be less than 0 and increasing the amount of packets sent past a threshold has diminishing returns.



Our final graph shows that as the utilization of the queue increases, the amount of packets lost also increases. This occurs since the limitation of a buffer size will be more frequent as you are trying to send out more packets. As you fill the queue faster, there are more instances where a packet will be lost. Our graph also shows that past a certain threshold the queue cannot keep up with the amount of packets being sent at all and starts to lose packets in an increasing exponential way. It is also noted that as you increase your buffer size the threshold mentioned gets pushed back and you can send more packets safely. However, after you hit that threshold, regardless of queue size, you will start to lose many packets.

Source Code for Simulations

Class Structure



ThunderLab1.java

```
package ca.uwaterloo.ece358.thunder;


import ca.uwaterloo.ece358.thunder.data.QueueSimulationReport;
import ca.uwaterloo.ece358.thunder.session.MD1KQueueSession;
import ca.uwaterloo.ece358.thunder.session.MD1QueueSession;


import java.io.PrintWriter;
import java.util.List;


public class ThunderLab1 {
    public static final int MILLION = 1000000;


    //Length of packets in bits
    private static int packetSize = 2000;


    //Service time received by a packet (bits/second) ---
    private static double service = 1 * MILLION;


    //How long to keep queueing for
    private static int timeLength = 10;


    //Run the same simulation 5 times with the same parameters and average the results
```

```
    private static int simulationSampleSize = 5;

    public static void main(String[] args) {
        runMD1QueueSimulation();
        runMD1KQueueSimulation();
    }

    private static void runMD1QueueSimulation() {
        List<QueueSimulationReport>[] reports = new List[simulationSampleSize];

        MD1QueueSession md1QueueSession = new MD1QueueSession(timeLength, packetSize, service);
        for (int i = 0; i < reports.length; i++) {
            System.out.println("Running MD1 simulation: " + i);
            reports[i] = md1QueueSession.runSimulations();
        }

        outputReports(reports, "MD1");
    }

    private static void runMD1KQueueSimulation() {
        List<QueueSimulationReport>[] reports = new List[simulationSampleSize];

        int[] bufferSizes = {10, 25, 50};

        for (int i = 0; i < reports.length; i++) {
            for (int j = 0; j < bufferSizes.length; j++) {
                System.out.println("Running MD1K simulation for buffer size " + bufferSizes[j]);
                MD1KQueueSession md1KQueueSession = new MD1KQueueSession(bufferSizes[j], timeLength,
packetSize, service);
                if (reports[i] == null) {
                    reports[i] = md1KQueueSession.runSimulations();
                } else {
                    reports[i].addAll(md1KQueueSession.runSimulations());
                }
            }
        }
```

```
        outputReports(reports, "MD1K");
    }

    private static void outputReports(List<QueueSimulationReport>[] reports, String fileName) {
        StringBuilder sb = new StringBuilder();
        boolean header = false;
        for (int i = 0; i < reports.length; i++) {
            for (QueueSimulationReport q : reports[i]) {
                if (!header) {
                    sb.append(QueueSimulationReport.getCSVHeader());
                    sb.append("\n");
                    System.out.println(QueueSimulationReport.getCSVHeader());
                    header = true;
                }
                sb.append(q.toCSV());
                sb.append("\n");
                System.out.println(q.toCSV());
            }
        }

        try {
            PrintWriter writer = new PrintWriter(fileName + ".csv", "UTF-8");
            writer.println(sb.toString());
            writer.close();
        }catch (Exception e){
            System.out.println(sb.toString());
        }
    }
}
```

MD1QueueSession.java

```
package ca.uwaterloo.ece358.thunder.session;

import ca.uwaterloo.ece358.thunder.ThunderLab1;
import ca.uwaterloo.ece358.thunder.data.LinkBuffer;
import ca.uwaterloo.ece358.thunder.data.QueueSimulationReport;

import java.util.ArrayList;
import java.util.List;

public class MD1QueueSession {
    protected int timeLength;
    protected int processTime;
    protected int packetSize;
    protected int bufferSize;
    protected double serviceTime;

    public MD1QueueSession(int bufferSize, int ticks, int packetSize, double service) {
        this.bufferSize = bufferSize;
        this.timeLength = ticks;
        this.packetSize = packetSize;
        this.serviceTime = service;
        this.processTime = (int) (packetSize / service * ThunderLab1.MILLION);
    }

    public MD1QueueSession(int ticks, int packetSize, double service) {
        this(-1, ticks, packetSize, service);
    }

    public List<QueueSimulationReport> runSimulations() {
        List<QueueSimulationReport> rtnReports =
                new ArrayList<QueueSimulationReport>();

        double[] lambdas = extractLambdas();
        for (int i = 0; i < lambdas.length; i++) {
            QueueSimulationReport report = simulate(lambdas[i]);
```

```
            if (report != null) {
                rtnReports.add(report);
            } else {
                i--;
            }
        }

        return rtnReports;
    }

    protected double[] extractLambdas() {
        List<Double> lambdas = new ArrayList<Double>();
        for (double i = 0.3; i < 0.75; i += 0.1) {
            lambdas.add((i * serviceTime) / packetSize);
        }

        double[] rtnLambdas = new double[lambdas.size()];
        for (int i = 0; i < lambdas.size(); i++) {
            rtnLambdas[i] = lambdas.get(i);
        }

        return rtnLambdas;
    }

    private QueueSimulationReport simulate(double lambda) {
        QueueSimulationReport qReport =
                new QueueSimulationReport(bufferSize,
                                          packetSize, serviceTime,
                                          lambda, timeLength);

        LinkBuffer queue = new LinkBuffer(bufferSize, lambda);

        int nextPacket = (int) (queue.getRandom() * ThunderLab1.MILLION);
        int last = 0;

        for (int time = 0; time < timeLength * ThunderLab1.MILLION; time++) {
```

```
            if (queue.length == 0) {
                qReport.idle++;
            }

            //Make sure current packet is greater than previous packet
            if (time == last + nextPacket) {
                qReport.sentPackets++;
                if (!queue.append(time)) {
                    qReport.packetLoss++;
                }

                nextPacket = (int) (queue.getRandom() * ThunderLab1.MILLION);
                last = time;
            }

            qReport.cumulativeTime += queue.check(time, processTime);
            qReport.packetBuffer += queue.length;
        }

        qReport.cumulativeTime += queue.length * processTime;

        if (qReport.sentPackets > lambda * timeLength - 200 &&
                qReport.sentPackets < lambda * timeLength + 200) {
            return qReport;
        }

        return null;
    }
}
```

MD1KQueueSession.java

```
package ca.uwaterloo.ece358.thunder.session;

import java.util.ArrayList;
import java.util.List;

public class MD1KQueueSession extends MD1QueueSession {
    public MD1KQueueSession(int bufferSize, int ticks, int packetSize, double service) {
        super(bufferSize, ticks, packetSize, service);
    }

    protected double[] extractLambdas() {
        List<Double> lambdas = new ArrayList<Double>();
        for (double i = 0.5; i < 1.5; i += 0.1) {
            lambdas.add((i * serviceTime) / packetSize);
        }

        double[] rtnLambdas = new double[lambdas.size()];
        for (int i = 0; i < lambdas.size(); i++) {
            rtnLambdas[i] = lambdas.get(i);
        }

        return rtnLambdas;
    }
}
```

QueueSimulationReport.java

```
package ca.uwaterloo.ece358.thunder.data;


public class QueueSimulationReport {

    public int bufferSize;

    public int packetSize;

    public double serviceTime;

    public double lambda;

    public int timeLength;


    public int idle = 0;

    public int packetLoss = 0;

    public int cumulativeTime = 0;

    public int sentPackets = 0;

    public int packetBuffer = 0;


    public QueueSimulationReport(int bufferSize, int packetSize, double serviceTime, double lambda, int timeLength) {

        this.bufferSize = bufferSize;

        this.packetSize = packetSize;

        this.serviceTime = serviceTime;

        this.lambda = lambda;

        this.timeLength = timeLength;

    }


    public String toCSV() {

        double averagePacket = (double) sentPackets / timeLength;

        double idleTime = ((double) idle / 1000000);

        double sojournTime = ((double) cumulativeTime / ((sentPackets - packetLoss) * 1000));

        double averagePacketsInBuffer = ((double) packetBuffer / (timeLength * 1000000));


        StringBuilder sb = new StringBuilder();

        sb.append(bufferSize + ",");

        sb.append(packetSize + ",");

        sb.append(serviceTime + ",");

        sb.append(lambda + ",");

        sb.append(timeLength + ",");
```

```
        sb.append(averagePacket + ",");
        sb.append(packetLoss + ",");
        sb.append(idleTime + ",");
        sb.append(sojournTime + ",");
        sb.append(averagePacketsInBuffer);
        return sb.toString();
    }

    public static String getCSVHeader() {
        return "buffersize,packetsize,servicetime,lambda,timelength,averagepacket,packetloss,idletime,sojourn,packetsinbuffer";
    }
}
```

LinkNode.java

```
package ca.uwaterloo.ece358.thunder.data;

public class LinkNode {
    public int time;
    public int data;
    public LinkNode next;

    public LinkNode(int data) {
        this.time = -1;
        this.data = data;
    }
}
```

LinkBuffer.java

```
package ca.uwaterloo.ece358.thunder.data;

import java.util.Random;

public class LinkBuffer {
    public int length;
    private LinkNode head;
    private LinkNode tail;
    private int max;
    private int last;

    private double lambda;
    private Random random;

    public LinkBuffer(int size, double lambda) {
        this.last = 0;
        this.length = 0;
        this.max = size;
        this.lambda = lambda;
        this.random = new Random();
    }

    public boolean append(int data) {
        //Packet loss can never happen in infinite queue
        if (max == -1 || length + 1 <= max) {
            LinkNode n = new LinkNode(data);

            //If empty, assign to all tracers
            if (length == 0) {
                head = n;
                tail = n;
                head.time = data;
                last = data;
            } else {
                //Append to LinkedBuffer
```

```
            tail.next = n;
            tail = n;
            last = tail.data;
        }

        length++;
        return true;
    }

    last = data;
    return false;
}

public int check(int referenceTime, int processTime) {
    if (length == 0) {
        return 0;
    }

    // Add head time to process time and see if packet x can be processed
    if (head.time + processTime == referenceTime) {
        LinkNode n;
        if (length == 1) {
            n = head;
            head = null;
        } else {
            n = head;
            head = head.next;

            //Update the current head time
            head.time = referenceTime + 1;
        }

        length--;
        return referenceTime - n.data;
    }
```

```
        return 0;
    }

    public double getRandom() {
        //To get more distribution of randoms
        try {
            Thread.sleep(5);
        } catch (InterruptedException ex) {
            //Do nothing
        }

        return (-1 / lambda) * Math.log(1 - random.nextDouble());
    }

}
```