Java file NetworkSimulator.Java

```java
import java.util.LinkedList;

/**
 * The network simulator class for testing packet congestion.
 */
public class NetworkSimulator {
    private LinkedList<Packet> packetQueue; // The packet queue of the server
    private double tickRatio;               // The time per tick
    private double ticks;                   // The total number of ticks to run the simulation
    private PacketGenerator generator;      // Monitors the packet arrival rate
    private PacketServer server;            // Monitors the packet service time

    /**
     * Create a network simulator
     * @param ratio - The ratio of time per tick
     */
    public NetworkSimulator(double ratio){
        packetQueue = new LinkedList<Packet>();
        generator = new PacketGenerator(packetQueue);
        server = new PacketServer(packetQueue);
        tickRatio = ratio;
    }

    /**
     * Start the network simulation.
     * @param simulationTime - the time to run the simulation for in seconds
     * @param lambda - average packets generation rate in packet per seconds
     * @param L - the size of the packet in bits
     * @param C - the link speed of the server in bit per second
     * @param K - the maximum limit to the queue (-1 = infinity)
     */
    public void discreteEventSimulator(double simulationTime, double lambda,
            double L, double C, int K){

        // Reset the environment and record the parameters
        ticks = simulationTime/tickRatio;
        Reporter.RecordParameters(tickRatio, ticks, lambda, L, C, K);
        generator.setup(lambda, K, tickRatio);
        server.setup(L, C, tickRatio);
        packetQueue.clear();

        // Run the simulation
        double i = 0;
        while (i <= ticks) {
            // Record queue size before the
            int queueSize =  packetQueue.size();

            // Check if a new packet arrived?
            Boolean lostPacket = generator.arrival(i);

            // Check if the server has sent a packet
            Packet sentPacket = server.service(i);
```

```
            // Update the reports the results of last tick
            Reporter.Update(i, queueSize, sentPacket, lostPacket);

            // Advance to the next event: packet arrival or packet serviced
            double nextArrival = generator.getNextArrivalTick();
            double nextService = server.getNextServiceTick();
            double nextTick = nextArrival;

            // If the server is not idle than check if the service is done before next arrival
            if (!server.isIdle() && nextTick > nextService) {
                nextTick = nextService;
            }

            // Update the reporter of the results of last tick for the metrics
            if (nextTick > ticks) {
                Reporter.Update(nextTick, packetQueue.size(), null, false);
            }

            // Advance to the next event time
            i = nextTick;
        }
    }
}
```

Java file Reporter.Java

```
import java.io.*;
import java.text.*;
import java.util.Date;

/**
 * The reporter records the events of the simulation a reports them to a CSV file
 */
public class Reporter {
    public static double tickRatio;         // The ratio of time per tick
    public static double totalTicks;        // The total number of ticks in the simulation
    public static double lastTick;          // The last tick that was recorded
    public static double sumQueueSize;      // The sum of queue size of each tick
    public static double sumSojournTime;    // The sum of sojourn time of each packet
    public static double sumIdle;           // The sum of queue of ticks when the queue was empty
    public static double sumLoss;           // The sum of loss packets
    public static double sumPacketsRx;      // The sum of total packets received
    public static double sumPacketsTx;      // The sum of total packets transmitted
    public static double rho;               // The network utilization ratio
    public static double lambda;            // The arrival rate of the packets
    public static double L;                 // The size of the packets in bits
    public static double C;                 // The service speed of the link in Mbps
    public static double K;                 // The limit of the queue (-1 = infinity)
    public static double EN;                // The average sojourn time
    public static double ET;                // The average sojourn time of a packet
    public static double P_IDLE;            // The percentage idle time of the queue
    public static double P_LOSS;            // The percentage of lost packets
    public static double lastSize;          // The last queue size

    /**
```

```
 * Reset the reporters records
 */
public static void reset() {
    lastTick = -1;
    sumQueueSize = sumSojournTime = sumIdle = sumLoss = sumPacketsRx = sumPacketsTx = 0;
    rho = lambda = L = C = K = EN = P_IDLE = ET = P_LOSS = lastSize = 0;
}

/**
 * Calculate the network utilization (rho) and record parameters it for reporting
 * @param ratio - The ratio of time per tick
 * @param ticks - The total number of ticks for simulation
 * @param arrivalRate - The arrival rate of the packets
 * @param length - The size of the packets in bits
 * @param serviceSpeed - The service speed of the link in Mbps
 * @param queueLimit - The limit of the queue (-1 = infinity)
 * @return rho - The network utilization ratio
 */
public static double RecordParameters(double ratio, double ticks, double arrivalRate,
        double length, double serviceSpeed, int queueLimit) {
    reset();
    tickRatio = ratio;
    totalTicks = ticks;
    lambda = arrivalRate;
    L = length;
    C = serviceSpeed;
    K = (double)queueLimit;
    rho = L * ( lambda / C);
    return rho;
}

/**
 * Update the report of what has occurred between events
 * @param tick - The number of ticks since the beginning of the simulation
 * @param queueSize - The number of packets in the queue
 * @param rxPacket - The last packet sent
 * @param lostPacket -  If true then a packet was lost last tick
 */
public static void Update(double tick, int queueSize, Packet rxPacket, Boolean lostPacket)
{
    // Calculate tick delta since last update
    // The time delta is used to multiply the data by skipped time
    // (assumes values do not change in between events)
    double tickDelta = tick - lastTick;

    // If queue empty record as idle otherwise record queue size
    if (queueSize == 0) {
        // Record that the queue has empty for delta ticks
        sumIdle += tickDelta;
    } else {
        // Record how big was the queue was since last update
        sumQueueSize += queueSize * tickDelta;
    }

    // Record the last packet's sojourn time if the packet was sent this tick
```

```
    if(rxPacket != null) {
        sumSojournTime += rxPacket.getSojournTime();
        sumPacketsRx++;
        sumPacketsTx++;
    }

    // If a packet was lost last tick then record it
    if(lostPacket) {
        sumLoss++;
        sumPacketsTx++;
    }

    // Remember the last tick value
    lastTick = tick;

    // Remember the last queue size
    lastSize = queueSize;
}

/**
 * Report on the simulation to the passed CSV file.
 * @param filename - The name of the file to write the report to.
 */
public static void Report(String filename) {
    FileWriter writer = null;

    try {
        // Check if file exists and create if not does not
        File file = new File(filename);
        Boolean fileExists = file.exists();
        if (!fileExists) {
            file.createNewFile();
        }

        // Create the file writer
        writer = new FileWriter(file, true);

        // If the file was created then write the header
        if (!fileExists) {
            writer.write("now,time,ticks,Tx,Rx,lost,lambda,K,rho,E[N],E[T],P_IDLE,P_LOSS\n");
        }

        // Add the last queue size to the transmitted queue size
        sumPacketsTx += lastSize;

        // Find the averages and ratios for simulations
        EN = sumQueueSize / totalTicks;
        P_IDLE = sumIdle / totalTicks;

        // Prevent divide by zero exception if no packets where sent
        if (sumPacketsRx != 0) ET = (sumSojournTime * tickRatio) / sumPacketsRx;
        if (sumPacketsTx != 0) P_LOSS = sumLoss / sumPacketsTx;

        // Get the current time of writing
        DateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
```

```
            String now = dateFormat.format(new Date());

            // Time of simulation
            double simTime = totalTicks * tickRatio;

            // Write each parameter and output to match header
            String f = "%1s,%2$f,%3$f,%4$f,%5$.3f,%6$f,%7$f,%8$f,%9$f,%10$e,%11$f,%12$f,%13$f\n" ;
            String CSVWrite = String.format(f, now, simTime, tickRatio, sumPacketsTx,
                sumPacketsRx, sumLoss, lambda, K, rho, EN, ET, P_IDLE, P_LOSS);
            writer.write(CSVWrite);
        } catch (IOException e) {
            System.out.print("Exception: failed to write to file " + filename);
            e.printStackTrace();
        }

        // try to close the file writer if opened
        try {
            if (writer != null)
            {
                writer.flush();
                writer.close();
            }
        } catch (IOException e) {
            System.out.print(" Exception: failed to close file " + filename);
        }
    }
}
```

Java file Packet.Java

```
/**
 * Simulate a Internet packet.
 */
public class Packet {
    private int id;                 // The packet id number
    private double arrivalTime;     // The tick when the packet arrived
    private double serviceTime;     // The tick when the service finished
    private double queueDelayTime;  // The delay of waiting in the queue

    /**
     * Create a new packet.
     * @param pid - The unique packet id
     * @param arrTime - The tick when the packet arrived
     */
    public Packet(int pid, double arrTime){
        id = pid;
        arrivalTime = arrTime;
    }

    /**
     * Get the tick when the packet arrived.
     * @return When the packet arrived.
     */
    public double getArrivalTime() {
        return arrivalTime;
```

```
}

/**
 * Set the tick when the packet arrived.
 * @param arrivalTime - The tick when the packet arrived.
 */
public void setArrivalTime(double arrivalTime) {
    this.arrivalTime = arrivalTime;
}

/**
 * Get the tick when the packet got serviced.
 * @return  When the packet serviced.
 */
public double getServiceTime() {
    return serviceTime;
}

/**
 * Set the tick when the packet got serviced.
 * @param serviceTime - The tick when the packet got serviced.
 */
public void setServiceTime(double serviceTime) {
    this.serviceTime = serviceTime;
    this.queueDelayTime = serviceTime - this.arrivalTime;
}

/**
 * The unique Id of the packet.
 * @return Returns the packet ID number
 */
public int getId(){
    return this.id;
}

/**
 * Calculate the sojourn time of the packet.
 * @return How long the packet took from the arrival to departure.
 */
public double getSojournTime(){
    return this.serviceTime - this.arrivalTime;
}

/**
 * Get the tick when the packet got out of the queue.
 * @return  When the packet got out of the queue.
 */
public double getQueueDelayTime() {
    return queueDelayTime;
}

/**
 * Set the tick packet got out of the queue.
 * @param queueTime - The tick when the packet got out of the queue.
 */
```

```
    public void setQueueDelayTime(double queueTime) {
        this.queueDelayTime = queueTime;
    }
}
```

Java file PacketGenerator.Java

```
import java.util.LinkedList;

/**
 * Generates packets and the exponential arrival rate
 */
public class PacketGenerator {

    private LinkedList<Packet> packetQueue; // Reference to the packet queue
    private double nextArrivalTick;         // Next tick when packet will arrival
    private int packetID;                   // Record the ID of the packet
    private double lambda;                  // The arrival rate in packets per second
    private int queueLimit;                 // The limit of the queue size
    private double tickRatio;               // The ratio of time to ticks

    /**
     * Create a new packet generator.
     * @param queue - Reference to the packet queue
     */
    public PacketGenerator(LinkedList<Packet> queue)
    {
        packetQueue = queue;
    }

    /**
     * Process if an arrival has occurred this tick.
     * @param ticks - The current tick in the simulation.
     * @return True if packet lost this tick otherwise false.
     */
    public Boolean arrival(double ticks){
        Boolean lostPacket = false;

        // Is it time to send an arrival tick?
        if(ticks >= nextArrivalTick) {
            Packet packet = new Packet(packetID++,ticks);

            // Add packet to queue unless the queue is full then drop it
            if (packetQueue.size() <= queueLimit || queueLimit == -1) {
                packetQueue.add(packet);
            } else {
                lostPacket = true;
            }

            // Calculate when the next packet should arrive
            double U = Math.random();
            double nextTime = (-1 / lambda) * Math.log(1 - U);
            double nextTick = nextTime / tickRatio;
            nextArrivalTick = ticks + nextTick;
        }
```

```
        return lostPacket;
    }

    /**
     * Gets the next tick that a packet is due to arrive
     * @return  The tick that a packet will arrive next
     */
    public double getNextArrivalTick() {
        return nextArrivalTick;
    }

    /**
     * Set up a new generator based on the simulation parameters.
     * @param arrivalRate - The packet arrival rate in packets per second
     * @param limit - The size limit to how many packets can be stored
     * @param ratio - The ratio of time to ticks
     */
    public void setup(double arrivalRate, int limit, double ratio)
    {
        nextArrivalTick = 0;
        queueLimit = limit;
        lambda = arrivalRate;
        tickRatio = ratio;
    }
}
```

Java file PacketServer.Java

```
import java.util.LinkedList;

/**
 * Services packets a constant rate.
 */
public class PacketServer {
    private LinkedList<Packet> packetQueue; // Reference to the packet queue
    private Packet packetBuffer;            // A spot to store the packet while it serviced
    private double nextServiceTick;         // Next tick when packet will be served
    private double serviceTicks;            // The amount of ticks to service a packet

    /**
     * Create a new packet server.
     * @param queue - Reference to the packet queue.
     */
    public PacketServer(LinkedList<Packet> queue)
    {
        packetQueue = queue;
    }

    /**
     * Is the queue idle?
     * @return True if both queue and buffer are empty, false otherwise.
     */
    public Boolean isIdle()
    {
```

```
        return packetQueue.size() == 0 && packetBuffer == null;
    }

    /**
     * Process the server to see if a new packet can be served
     * @param ticks - The current tick in the simulation.
     * @return The packet that is ready to be sent.
     */
    public Packet service(double ticks) {
        // Remember if a packet is sent
        Packet sendPacket = null;

        // If there is a packet in the buffer and service time is done then send packet
        if(packetBuffer != null && ticks >= nextServiceTick) {
            sendPacket = packetBuffer;
            sendPacket.setServiceTime(ticks);
            packetBuffer = null;
        }

        // If the queue is not empty and there is room in the buffer then service the packet
        if(!packetQueue.isEmpty() && packetBuffer == null) {
            packetBuffer = packetQueue.remove();
            packetBuffer.setQueueDelayTime(ticks);
            nextServiceTick = ticks + serviceTicks;
        }

        return sendPacket;
    }

    /**
     * Gets the next tick that a packet is served
     * @return The tick that a packet will be served
     */
    public double getNextServiceTick() {
        return nextServiceTick;
    }

    /**
     * Setup a new generator based on the simulation parameters.
     * @param length - The length of the packet in bits
     * @param linkSpeed - The link speed in bit per second
     * @param ratio - The ratio of time to ticks
     */
    public void setup(double length, double linkSpeed, double ratio)
    {
        // Reset server
        packetBuffer = null;
        nextServiceTick = 0;

        // Calculate how long it takes to service a packet
        double serviceTime = (length / linkSpeed);
        serviceTicks =  serviceTime / ratio;
    }
}
```

JUnit test Question2.Java

```java
import org.junit.Test;

/**
 * Unit tests for question 2 simulations
 */
public class Question2 {
    // The length time that simulation will run for in seconds
    public double simTime = 600;

    // Create a new simulator
    public NetworkSimulator sim = new NetworkSimulator(1);

    /**
     * Run simulations for no queue size
     */
    @Test
    public void Run()
    {
        double M;           // Number of times to run simulation
        double lambda;      // Packet arrival rate
        double L = 2000;    // Packet length is 2000 bits
        double C = 1e+6;    // Service speed is 1 Mbps
        double rho;         // Utilization of the queue

        // Simulation for different simulation
        for (rho = 0.2; rho <= 0.9; rho += 0.1)
        {
            // Calculate the arrival rate
            lambda = rho * ( C / L);

            System.out.print("M, simTime, lambda, L, C\n");

            // Run M simulations
            for(M = 0; M < 10; M++)
            {
                // Simulate for 10 minutes
                sim.discreteEventSimulator(simTime, lambda, L, C, -1);

                // Record the results of the test
                Reporter.Report("Q2.csv");

                String simFormat = "%1$f, %2$f, %3$f, %4$f,%5$f\n";
                String simResults = String.format(simFormat, M, simTime, lambda, L, C);
                System.out.print(simResults);
            }
        }
    }
}
```

JUnit test Question4.Java

```java
import org.junit.Test;
```

```java
/**
 * Unit tests for questions 4 simulations
 */
public class Question4 {
    // The length time that simulation will run for in seconds
    public double simTime = 600;

    // Create a new simulator
    public NetworkSimulator sim = new NetworkSimulator(1);

    /**
     * Run simulations for queue sizes K = [10, 25, 50]
     */
    @Test
    public void Run()
    {
        double M;           // Number of times to run simulation
        double lambda;      // Packet arrival rate
        double L = 2000;    // Packet length is 2000 bits
        double C = 1e+6;    // Service speed is 1 Mbps
        double rho;         // Utilization of the queue

        // Run for different queue sizes
        int[] queueSize = {10, 25, 50};
        for (int K : queueSize)
        {
            // Simulation for different simulation
            for (rho = 0.5; rho < 1.55; rho += 0.1)
            {
                // Calculate the arrival rate
                lambda = rho * ( C / L);

                System.out.print("M, simTime, lambda, L, C, K\n");

                // Run N simulations
                for(M = 0; M <10; M++)
                {
                    // Simulate for 20 minutes
                    sim.discreteEventSimulator(simTime, lambda, L, C, K);

                    // Record the results of the test
                    Reporter.Report("Q4.csv");

                    String simFormat = "%1$f, %2$f, %3$f, %4$f, %5$f, %6$d\n";
                    String simResults = String.format(simFormat, M, simTime, lambda, L, C, K);
                    System.out.print(simResults);
                }
            }
        }
    }
}
```