Project 3: Using Sockets for the

Simulation and Performance Evaluation of the

IEEE 802.3 MAC Protocol

CS158A-02

Project 3

Aishwarya Venketeswaran, Luke Sieben, and Chin Tsai

Submitted to Layla Pezeshkmehr

**Table of Contents**

# 

[Section A: Waiting time for a successful transmission](#h.eihekfl35gz1)

[Section B: N stations](#h.eihekfl35gz1)

[Section C: Poisson process of transmission attempts](#h.eihekfl35gz1)

[Section E: CSMA/CD Node Collision Simulation](#h.a65352e7mqr)

[References](#h.bnt1jxkxm16r)

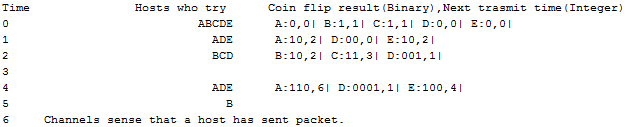
# 

# 

# Section A: Waiting time for a successful transmission

**Problem 2-49**:

1. We did the simulation using a coin toss. If the coin toss is tails, then the time is 1 and if the coin toss is heads, the time is 0. To simulate the backoff, at the nth collision, n flips are required to determine the backoff time. The table below summarizes the results:



1. In our simulation, when B transmits, all other packets detect immediately that B is transmitting and backoff. However, in reality, A and C (which maybe closer to B) detect this transmission right away. On the other hand, D and E that are further away, take time to detect this change.

**Simulation program to repeat (a) 100 times [Code]:**

|  |  |
| --- | --- |
| import java.util.ArrayList;  import java.util.Random;  /\*  \* Runs the simulation  \*/  public class main  {  private static ArrayList<host> hosts ;  private static Random randomGenerator;  private static int firstWaitTime;  private static int secondWaitTime;  private static int lastWaitTime;  private static int elementDone;  private static boolean justSent;    public static void main(String [] args)  {  int numOfHosts = 5;  //uncomment for b //for(numOfHosts=20;numOfHosts<=100;numOfHosts=numOfHosts+20){  firstWaitTime = 0;  secondWaitTime = 0;  lastWaitTime = 0;  for(int i=0;i<100;i++)  {  elementDone = 0;  simulateOneRun(numOfHosts);  }  System.out.println();  System.out.println("First Element Average delay:" + firstWaitTime/100.0);  System.out.println("Second Element Average delay:" + secondWaitTime/100.0);  System.out.println("Last Element Average delay:" + lastWaitTime/100.0);  //uncomment for b}  }    public static void simulateOneRun(int numOfElements)  {  System.out.print("Time ");  System.out.format("%32s","Hosts who try ");  System.out.print("Coin flip result(Binary),");  System.out.print("Next trasmit time(Integer)");  randomGenerator = new Random();  hosts = new ArrayList<host>(numOfElements);  //Initialize the array  for (int i=0;i<numOfElements;i++)  {  hosts.add(new host(i+65));    }  //While no one has sent data, continue this process  int time = 0;  ArrayList<host> runners = new ArrayList<host>();  while(!allSent())  {  System.out.println();  System.out.format("%s", time);  int sent = anyoneSent();  if(sent != -1) //if anyone was just sent, don't send anything in the current block.  {  switch(elementDone)  {  case 1:firstWaitTime+=time;  break;  case 2:secondWaitTime+=time;  break;  default:elementDone+=0;  }  hosts.remove(sent);  runners = findNumOfRunningHosts(time);  System.out.format("%32s", printAllRunners(runners));  System.out.print(" ");  for (host a:runners)  {  a.run(time,runners.size()+1,randomGenerator);  }  System.out.print("Channels recognize no one can be sent");  }  else  {  runners = findNumOfRunningHosts(time);  System.out.format("%32s", printAllRunners(runners));  System.out.print(" ");  for (host a:runners)  {  a.run(time,runners.size(),randomGenerator);  }  }  time++;  }  lastWaitTime+=time;  System.out.println();  System.out.print(time + " ");  //System.out.println("Channels sense that a host has sent packet.");  }    public static String printAllRunners(ArrayList<host> runners)  {  String temp = "";  for(int i=0;i<runners.size();i++)  {  temp = temp +(runners.get(i).getID());  }  return temp;  }    public static int anyoneSent()  {  for(int i=0;i<hosts.size();i++)  {  if(hosts.get(i).getPacketSentStatus())  {  elementDone++;  return i;  }  }  return -1;  }    public static boolean allSent()  {  for(int i=0;i<hosts.size();i++)  {  if(!hosts.get(i).getPacketSentStatus())  return false;  }  return true;  }    public static ArrayList<host> findNumOfRunningHosts(int time)  {  ArrayList<host> temp = new ArrayList<host>();  for(host a: hosts)  {  if(time == a.getNextTransTime())  temp.add(a);  }  return temp;  }  } | import java.util.Random;  /\*  \* Represents a host.  \* Each host stores its next transmission time and the number of collisions it has had.  \*  \*/  public class host  {  private int nextTransTime;  private int collisionsCompleted;  private boolean packetSent;  private final char id;    public host(int id)  {  nextTransTime=0;  collisionsCompleted = 0;  packetSent = false;  this.id = (char)(id);    }    public boolean getPacketSentStatus()  {  return packetSent;  }    public void run(int time, int numOfRunningHosts,Random randomGenerator)  {    //if I am the only host running, then I can send my packets  //Stop current  if(numOfRunningHosts == 1)  {  //I cam transmit packet  packetSent = true;  }  else //I collided  {  collisionsCompleted++;  nextTransTime = time+ 1 + findNextTransTime(randomGenerator);  packetSent = false;  }  }    public int findNextTransTime(Random randomGenerator)  {  printHost();  System.out.print(":");  int total = 0;  String totalS = "";  for(int i=0;i<getCollisionsCompleted();i++)  {  int coinFlipResult = 0;  //do a coin flip simulation [0 or 1]  int randomInt = randomGenerator.nextInt(2);  //append the value to total  //System.out.print(randomInt);  totalS = randomInt + totalS;  total = (int) (Math.pow(2,i)\*randomInt)+total;  }  //System.out.print(" ");  System.out.print(totalS+","+total+"| ");  return total;  }  /\*\*  \* @return the nextTransTime  \*/  public int getNextTransTime()  {  return nextTransTime;  }  /\*\*  \* @return the collisionsCompleted  \*/  public int getCollisionsCompleted()  {  return collisionsCompleted;  }    public void printHost()  {  System.out.print(this.id);  }    public char getID()  {  return id;  }    } |

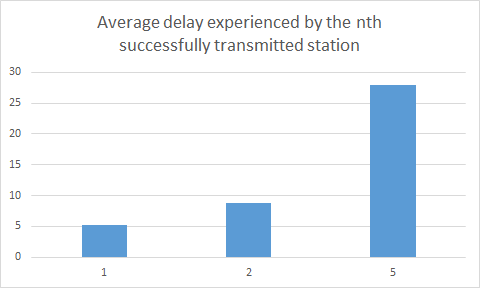
**Output[Summary]:**

First Element Average delay:5.16

Second Element Average delay:8.82

Last Element Average delay:27.97

**Histogram showing the average delay experienced by the nth [1,2,and 5] successfully transmitted station:**



# Section B: N stations

**N stations: We did Problem 2-50** (extended from step a). and repeated each N value 100 times.

Code is the same except that we called the program with different N values. There are two lines to uncomment from the code provided above for part a, so that we can run the program for part b.

Sample result:

Number of hosts = 5

First Element Average delay:5.03

Second Element Average delay:9.04

Last Element Average delay:28.8

Number of hosts = 20

First Element Average delay:12.25

Second Element Average delay:17.04

Last Element Average delay:193.51

Number of hosts = 40

First Element Average delay:19.61

Second Element Average delay:26.27

Last Element Average delay:474.04

Number of hosts = 60

First Element Average delay:26.18

Second Element Average delay:33.61

Last Element Average delay:772.77

Number of hosts = 80

First Element Average delay:32.31

Second Element Average delay:40.77

Last Element Average delay:1163.08

Number of hosts = 100

First Element Average delay:38.98

Second Element Average delay:49.22

Last Element Average delay:1505.86

**Line graph showing the average delay experienced by the nth [1,2,and 5] successfully transmitted station as the value of N increases:**

# 

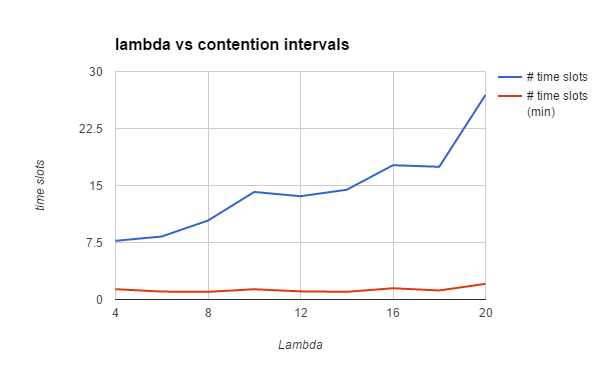
Between the first and second hosts that get transmit packets, there is not much difference. However, there is a drastic change, when the fifth host can transmit.This graph shows us that the relationship is linear. As the number of hosts increase, the time to transmit also increases (for every chance - 1,2 and 5).

# Section C: Poisson process of transmission attempts

Problem 2-52

In this problem we assume 20 stations. We have to find the time slots required before the first successful transmission attempt. The interval between consecutive attempts is determined by -Lambda\*log(u) with value u chosen randomly in the interval 0<u<1. Different values of Lambda is chosen to determine relationship between Lambda and time slots before a successful attempts. The result is presented below. The graph demonstrates that time slots before a successful transmission is proportional to Lambda. The minimum contention interval is 7.72. This means 7.72\*64=494.08 bytes will go wasted in 10Mbps link. Assuming this happens every 8 slot times, only 512 /494.08+512=51% percent of link will be utilized.

|  |  |  |
| --- | --- | --- |
| Lambda | # time slots | # time slots (min) |
| 4 | 7.72 | 1.36 |
| 6 | 8.29 | 1.04 |
| 8 | 10.39 | 1.01 |
| 10 | 14.15 | 1.35 |
| 12 | 13.59 | 1.07 |
| 14 | 14.44 | 1.01 |
| 16 | 17.67 | 1.48 |
| 18 | 17.46 | 1.19 |
| 20 | 26.92 | 2.06 |



PART C Code

import java.util.Arrays;

public class nodeTest {

public static void main(String[] args) {

test(4);

test(6);

test(8);

test(10);

test(12);

test(14);

test(16);

test(18);

test(20);

}

static void test(double lambda){

int N=50;

double sum=0;

double [] l=new double[N];

for (int i=0; i<N; i++){

double num= run(lambda);

l[i]=num;

sum = sum+ num;

}

Arrays.sort(l);

System.out.println(lambda + ": "+sum/N+ " "+l[0]);

}

static double run(double lambda){

node n = new node(lambda);

for(int i=0;i<20;i++){

if(n.check()){

break;

}

n=n.getNext(lambda);

}

return n.getNow();

}

}

public class node {

private double pre;

private double now;

private double next;

public node(double lam){

this.pre=0;

this.now=var(lam);

this.next=now+var(lam);

}

public node(double pre, double now, double next ){

this.pre=pre;

this.now=now;

this.next=next;

}

public boolean check(){

if(now - pre>1 && next - now>1){

return true;

}

return false;

}

public node getNext(double lam){

return new node(now,next,next+var(lam));

}

private double var(double lambda){

double val= -Math.log(Math.random())\*lambda;

return val;

}

public double getNow(){

return now;

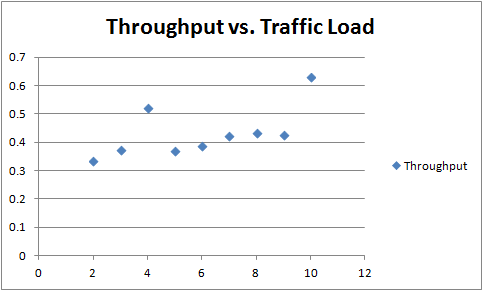
}

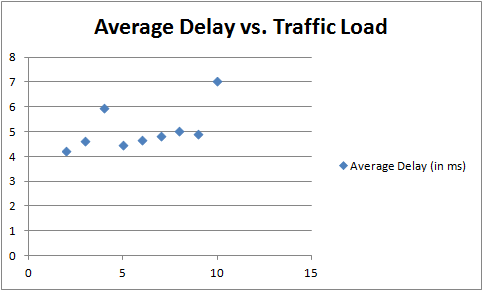
}

# Section E: CSMA/CD Node Collision Simulation

In this section, we simulated two CSMA/CD nodes that sent data at varying times to see if they would ever collide. If a node experienced a collision, we would run the exponential binary backoff algorithm on that node to hopefully change its retransmit time to one that would not result in a collision. As we decreased lambda we got less throughput because of more collisions, and shorter average delays as we were sending the data faster.

Sample Output in a Graph Format





Sample Output in a Table Format

|  |  |  |  |
| --- | --- | --- | --- |
| Lambda | Traffic Load | Throughput | Average Delay (in ms) |
| 20 | 10 | 0.6302 | 7.0416 |
| 18 | 9 | 0.4284 | 4.917 |
| 16 | 8 | 0.4362 | 5.0496 |
| 14 | 7 | 0.4228 | 4.8512 |
| 12 | 6 | 0.387 | 4.6638 |
| 10 | 5 | 0.3698 | 4.4924 |
| 8 | 4 | 0.521 | 5.979 |
| 6 | 3 | 0.3752 | 4.6598 |
| 4 | 2 | 0.3352 | 4.2546 |

**The code used to run the simulation:**

UDPServer.java

------------------------------------------------------------------------------------------------------

import java.io.IOException;

import java.net.\*;

/\*\*

\* The UDP server.

\*/

public class UDPServer {

public final static int PORT = 1201, BUFFER\_SIZE = 1024, TIME\_SLOTS = 5000;

public final static int SLOT\_TIME = 8;

public static void main(String[] args) {

runServer(20);

runServer(18);

runServer(16);

runServer(14);

runServer(12);

runServer(10);

runServer(8);

runServer(6);

runServer(4);

}

public static void runServer(int lambda) {

byte[] toSendBytes;

try {

DatagramSocket serverSocket = new DatagramSocket(PORT);

serverSocket.setReuseAddress(true);

long previousEndTime = 0;

int runs = 0;

int collisions = 0;

long totalDelay = 0;

boolean firstRun = true;

// stop the server after we get all of the client messages

System.out.println("Lambda: " + lambda);

System.out.println("Traffic Load: " + lambda / 2);

// start two threaded clients

(new Thread(new UDPClient("Client A", lambda))).start();

(new Thread(new UDPClient("Client B", lambda))).start();

// run for TIME\_SLOTS

while(runs < TIME\_SLOTS) {

byte[] received = new byte[BUFFER\_SIZE];

DatagramPacket receivedPacket = new DatagramPacket(received, received.length);

serverSocket.receive(receivedPacket);

String clientMessage = new String(receivedPacket.getData());

InetAddress clientAddress = receivedPacket.getAddress();

int port = receivedPacket.getPort();

long endTime = System.nanoTime();

long slotTimeDifference = (endTime - previousEndTime) / 100000;

if(!firstRun) {

totalDelay += slotTimeDifference;

}

// adjust previous end time to current end time

previousEndTime = endTime;

// add up times

totalDelay += (endTime - previousEndTime) / 100000;

// setup reply

String toSend = "Success";

if(slotTimeDifference <= SLOT\_TIME) {

toSend = "Collision";

collisions++;

}

toSendBytes = toSend.getBytes();

DatagramPacket sentPacket = new DatagramPacket(toSendBytes, toSendBytes.length, clientAddress, port);

serverSocket.send(sentPacket);

firstRun = false;

runs++;

}

serverSocket.close();

int successfulTransmissions = (TIME\_SLOTS - collisions);

double throughput = (successfulTransmissions + 0.0) / TIME\_SLOTS;

double averageDelay = (totalDelay + 0.0) / TIME\_SLOTS;

System.out.println("Collisions: " + collisions);

System.out.println("Successful transmissions: " + successfulTransmissions);

System.out.println("Average Delay: " + averageDelay);

System.out.println("Throughput: " + throughput);

System.out.println();

}

catch(IOException e) {

e.printStackTrace();

}

}

}

--------------------------------------------------------------------------------------------------------------

UDPClient.java

--------------------------------------------------------------------------------------------------------------

import java.io.IOException;

import java.net.\*;

import java.util.Random;

import java.util.concurrent.TimeUnit;

/\*\*

\* The UDP client.

\*/

public class UDPClient implements Runnable {

public final static int PORT = 1201, BUFFER\_SIZE = 1024;

public final static String ADDRESS = "127.0.0.1";

public static final int SLOT\_TIME = 8;

private Random random;

private String clientName;

private int lambda;

/\*\*

\* Creates a UDPClient with a clientName.

\* @param clientName the client name

\*/

public UDPClient(String clientName, int lambda) {

this.clientName = clientName;

this.lambda = lambda;

random = new Random();

}

/\*\*

\* The run method for threading purposes.

\*/

public void run() {

// setup string to send to server

String send = clientName;

for(int i = 0; i < BUFFER\_SIZE - clientName.length(); i++) {

send += ".";

}

byte[] sendBytes = send.getBytes();

try {

DatagramSocket clientSocket = new DatagramSocket();

InetAddress inetAddress = InetAddress.getByName(ADDRESS);

int n = 0;

while(true) {

// sleep for a bit before sending a message

TimeUnit.NANOSECONDS.sleep(getSlotTime(n));

DatagramPacket sendPacket = new DatagramPacket(sendBytes, sendBytes.length, inetAddress, PORT);

clientSocket.send(sendPacket);

byte[] received = new byte[BUFFER\_SIZE];

DatagramPacket receivePacket = new DatagramPacket(received, received.length);

clientSocket.receive(receivePacket);

// check out what type of message we received

String receiveMessage = new String(receivePacket.getData());

if(receiveMessage.startsWith("Collision")) {

n++;

}

else {

n /= 2; // halve it on success

}

}

}

catch(IOException e) {

e.printStackTrace();

}

catch(InterruptedException e) {

e.printStackTrace();

}

}

/\*\*

\* Gets the slot time.

\* @return the slot time

\*/

private int getSlotTime(int n) {

int exponentialBinaryBackoffAmount = 0;

if(n > 0) {

exponentialBinaryBackoffAmount = random.nextInt((int) (Math.pow(2, n) - 1));

}

return SLOT\_TIME \* lambda \* (exponentialBinaryBackoffAmount + 1);

}

}

-----------------------------------------------------------------------------------------------------------------

# 

# 

# References

a. Chapters 1 and 2 of the textbook by Peterson and Davie.

b. IEEE 802 standards or drafts.