Chris Bird, Blake Tacklind

997337048, 997051049

ECS 152A/EEC 173A – Computer Networks

Due: 12th March, 2015

Ad Hoc Network – Project Phase 2

Taken directly from the project description, the above graphs have throughput as the number of bytes sent per millisecond, and the average network delay as the number of total delay divided by throughput. However, these numbers did not seem to be useful, so we got a few others to include with the report.

In these graphs, average network delay is instead calculated by total delay divided by bytes sent total. This results in more realistic units of measure, rather than (time^2)/bytes, this is time/bytes. This is *not* what the project asked for, but it seems more logical to include it.

Since our other two simulations did not have much collision, we decided to make the lambda values large enough to show collisions happening. Our lambda values were the values given in the project specifications multiplied by one thousand. We used the same average network delay calculations as the second set, which is total delay divided by bytes sent.

Phase I Write up – Description and Logic

Discrete Event Simulation

Discrete event simulation is the simulating based on distinct events. Those events and only those events are what affect the state of the simulation. Other properties such as time and stacks only exist as a byproduct of the events occurring.

Description of Phase I

In Phase I, we simulated a single server as it receives and distributes response packets. Packets are received and handled in a negative exponentially distributed time (a decent approximation of real networks). Our simulated server therefore had to deal with any number of incoming packets and occasionally had to drop the packets due to insufficient incoming buffer size.

Logic of Phase I

Our phase I simulation ran until an arbitrary time was passed. Each cycle handled one event advancing time to the time of the next events occurrence. The next event would always be either an incoming event or a departure event (if the buffer was not empty). On incoming event we would check if we had room in our buffer and if so add a departure event to our buffer. Then decide when the next incoming event would occur. If a departure was the next event we would pop an event off the departure queue and decide when the next departure event is to occur. We would repeat this process with different lambda and buffer sizes for all conditions in the simulation.

Phase II Write up – Description and Logic

Description of Phase II

In phase II, the goal was to use our phase I code along with some modifications to make an Ad Hoc network. This means that there is no central hub or router for the network, but that each host communicates directly with all other hosts. In order to make sure that this is successful, there can only be one packet being broadcast at any given time. This is to ensure that the data doesn’t get incorrectly read by the destination host. To secure this, CSMA/CA protocol is used with DCF to detect possible collisions and stop them from happening. If at any given moment in time, the network has been idle for more than DFIS time, and an ACK has been sent acknowledging the most recently sent packet (also greater than SIFS time), the network opens up again to have another packet sent. If more than one host wants to send a packet at that exact time, they both get told to ‘back off’ and have to defer their packet by a randomly generated amount of time. This back off time is increased by a factor of 100%, then 200% etcetera for each time it is told to back-off. If at a given time the network is idle and only one wants to send, it gets to send and all others must wait until the network is idle again.

As far as our implementation goes, a few assumptions have been made, the most notable of which is that all packets that get sent are guaranteed to be received by their intended host. This is important to note as once a packet is sent, it will certainly get an ACK back after SIFS plus time to send 64 bytes has expired. Other assumptions include that there are no hidden hosts, hosts that are out of range of any other host, and that the max packet size is 1544 bytes long.

Description of Phase II Logic

Unlike our implementation of phase I, we used a for loop that counted through time in single steps at a time. In phase I we ‘jumped’ the clock to the next time value when an arrival or departure happened and ignored all the ones in between. However, since so many different things needed to be done on each clock cycle, we found it fitting to step through the clock on step at a time rather than skipping to the next appropriate time. Before this clock loop, some initializations needed to be done to prepare for smooth operation. First, using the number of networks given by the simulation, a list of hosts called ‘network’ was made that contained pointers to each of the hosts created. Using this list, each host could be accessed and worked with without needing to give it its own name. When each host is created, it generates a time until it will attempt to send its first packet, which will also be assigned a size based on a negative exponential distribution. From here, the clock can proceed as normal until a packet wants to be sent.

On each clock cycle, which increments in hundredths of a millisecond, if the channel is not busy, each the DIFS counter is decremented, letting the hosts know how long until they can attempt to send again. Also, if the channel isn’t busy but a packet was recently received by a host, a SIFS amount of time is waited, and then the receiver of the packet sends an ACK back to the sender. Due to SIFS being less than DIFS, the ACK will always be sent *before* the channel opens up to new traffic, ensuring proper acknowledgment of packets.

Once SIFS and DIFS are zero, and the channel is idle, each host is then polled for whether it wants to send a packet at this very moment or not. If it does, it is added onto a list called ‘sending’. This list is used to determine if there is a collision or not, and if so, to handle that collision. If the size of sending is exactly one, that packet is removed from the outgoing queue on that host and sent to the received after the amount of time it takes to send a packet of that size. If the number of items in sending is greater than one, a collision notice is sent to both, and they both set backoff counters before retrying to send that packet. Since these backoff counters are negatively exponentially generated, there is a low chance that they will try to send again at the exact same time. However, each time a packet collides with any other packet, it will wait a longer period of time before trying again. This makes it harder and harder for the backoff counters to be the same, allowing for traffic to flow once again.

After the clock has run its course, and is now equal to the total amount of time set to run the network, throughput and average delay is calculated using basic equations. A description of how we got our numbers can be found in the next section.

Phase II – Data Analysis



There are three data sets listed above, these data sets were used to create the graphs you can see on pages one through three of this report. Data set one is the exact data as requested by the project specification sheet. Using the given lambda values and number of networks, we determined throughput by calculating the total number of bytes sent through the network, and then dividing by the total amount of time that the network was running. Next, the average delay was calculated by taking the total amount of delay both in time the packet was spent sending and the time from creation to sending of the packet, and then dividing by the throughput. This yielded some somewhat unexpected results.

In data set two, we modified the calculation of average delay to use number of bytes sent as the denominator rather than the throughput. This put our units of measurement at milliseconds per byte rather than the milliseconds squared over bytes that was calculated in the first data set. We created this data set, along with data set three, to expand on the project and determine some new information. In data set three, we multiplied lambda by one thousand in order to force collision to happen. We also used our new average network delay calculation rather than the project defined one in this section as well. As can be seen from the data, the through put has multiplied exponentially as the packets are being sent much faster, but the delay has also increased dramatically as the number of collisions and need for back-off counters is much greater than in previous data sets.

Final Thoughts and Conclusions

Looking back over our code, we are very proud of how it turned out. We have made many tweaks, many changes, and many deletions of code, and as a result have a very short and optimized run time. However, there are still some things that could be made more efficient, which we may work on after finals. Running our code with the number of networks set at two thousand, regardless of lambda value, produced very predictable results. There were a ton of collisions, throughput was still relatively high, but the average wait time for a packet was very high as well. Since we have a counter for number of packets sent from each host, it was interesting to see how long it would take before a second packet was sent from a server.

In closing, our main function is very neat, and laid out in a very easy to understand fashion, and the packet and host classes both are very easy to understand their function and practicality. There are very few things that could be made more optimized and the code itself works beautifully. Running many simulations on this code would take very little time and almost no edits. It should also be noted that upon completion of each run, all the data is written to a .csv file that can be easily opened and edited in Microsoft Excel. This allows for easy creation of graphs, data sheets, and trend lines.

//Networking Project Phase 2.cpp

//Chris Bird and Blake Tacklind

//ECS 152A/EEC 173 Computer Networks

//Main code that controls the creation, running, and cleaning of hosts.

#include "stdafx.h"

#include "packets.h"

#include "host.h"

#include "time.h"

#include "stdio.h"

#include <iostream>

#include <fstream>

#define totalTime 100000

//100 seconds, time in milliseconds

#define timeStep .01

//Each clock cycle is one hundreth of a millisecond, 1/5th of SIFS

#define SIFS .05

//Time between packet receive and ACK sent

#define DIFS .1 //Time after channel is marked idle that new packets cannot be sent

#define maxSpeed 13.75

//If 100% traffic, no lost packets, this is max speed

double lambda = 0.9;

double mu = 2685;

//Mu value such that average packet size is 512

int numNetworks = 20;

//N value from prompt

double simClock;

double lamdArr[7] = { .01, .05, .1, .3, .6, .8, .9 };

//Lambda and #Networks values for simulations

int numNArr[2] = { 10, 20 };

using namespace std;

int main(int argc, char\* argv[])

{

ofstream FILE; //Creating and filling a file with the simulation results for easy graphing

FILE.open("output.csv");

for (int j = 0; j < 14; j++)

{

lambda = lamdArr[j % 7];

numNetworks = numNArr[j / 7];

cout << "Test " << j << " - Lambda: " << lambda << " - Number of Networks: " << numNetworks << endl;

while (!network.empty())

//Clearing list of networks between simulations

network.pop\_front();

srand((unsigned int)time(NULL));

//New Seed for Rand

for (int i = 0; i < numNetworks; i++)

{

network.push\_back(host(i));

//Once done, network.begin = host-0 etc

}

int DIFSCounter = (int)(DIFS / timeStep);

int SIFSCounter = 0;

double busyTill = 0.0;

bool ACKSent = false;

bool isSending = false;

int bytesSent = 0;

double totalDelay = 0;

packet sent;

for (simClock = 0; simClock < totalTime; simClock += timeStep)

//Time is in msec, but increment by hundreths of a msec

{

if (isSending && (busyTill <= simClock))

//If still marked as sending, but done sending, unmark

{

//cout << "Current simClock: " << simClock << endl;

//cout << "Done Sending" << endl;

isSending = false;

}

if ((busyTill <= simClock) && DIFSCounter)

//If not sending, and DIFS counter > 0, count down

{

DIFSCounter--;

}

if (SIFSCounter && (busyTill <= simClock))

//If SIFS counter is > 0 and not busy, send ACK

{

if (!ACKSent && !(--SIFSCounter))

{

isSending = true;

ACKSent = true;

//cout << "Current simClock: " << simClock << endl;

(sent.returnDest())->receivedPacket(ACK(sent.returnNumber(), sent.returnSend(), NULL));

//Send ACK here

bytesSent += 64;

totalDelay += ACKTime;

DIFSCounter = (int)(DIFS / timeStep);

busyTill = simClock + ACKTime;

}

}

if (!SIFSCounter && !DIFSCounter && (busyTill <= simClock))

//If done sending and both DIFS and SIFS are done, can check for next packet

{

list<host\*> sending;

for (list<host>::iterator it = network.begin(); it != network.end(); ++it)

//Iterate through list of hosts everytime

{

(\*it).decSIFSTime();

(\*it).runNormalOp(simClock);

if (!(\*it).returnBackOff() && (\*it).wantsToSend())

//If a packet is to be sent from given host THIS MOMENT, add to sending list

sending.push\_back(&\*it);

else

(\*it).decBackOff();

}

if (sending.size() == 1)

//If only ONE person wants to send, send their packet

{

isSending = true;

sending.front()->resetBOCounter();

//Reset BO Counter to 0

SIFSCounter = (int)(SIFS / timeStep);

//Reset DIFS because something has sent

sent = sending.front()->popOutgoing();

sent.returnDest()->receivedPacket(sent); //Sending packet to host

bytesSent += sent.returnSize();

totalDelay += (simClock - sent.returnTimeOC() + sent.returnTime());

//cout << "Sending From: " << (\*(sent.returnSend())).returnHostNumber() << " Sending To: " << (\*(sent.returnDest())).returnHostNumber() << " Packet Number: " << sent.returnNumber() << "Packet Time: " << sent.returnTime() << endl;

busyTill = sent.returnTime() + simClock;

//Set next busyTill to end of next packet transmission

ACKSent = false;

}

else if (sending.size() > 1)

//If more than one person wants to send, tell them all to backoff

{

for (list<host\*>::iterator it = sending.begin(); it != sending.end(); ++it)

(\*it)->setBackOff();

}

}

}

double through = ((double)bytesSent / totalTime) \* timeStep;

//This block displays proper information based on simulations

//cout << "Always Busy Speed: " << maxSpeed << " - Bytes per hundreths of a msec" << endl;

//cout << "Bytes Sent: " << bytesSent << " - Bytes" << endl;

cout << "Throughput: " << through << " - Bytes per hundreth of a msec" << endl;

//cout << "Total Delay: " << totalDelay << " - msec" << endl;

double avgDelay;

if (through != 0)

{

avgDelay = totalDelay / bytesSent; //(through \* 100);

cout << "Average Network Delay: " << avgDelay << " - msec" << endl;

}

else

cout << "Average Network Delay cannot be determined as throughput is 0" << endl;

cout << endl << endl;

FILE << lambda << "," << numNetworks << "," << through << "," << avgDelay << endl; //Write to and close file written

}

cout << "Simulations complete" << endl;

FILE.close();

char input; //Wait for user input to keep item open

cin >> input;

return 0;

}

//host.h

//Chris Bird and Blake Tacklind

//ECS 152A/EEC 173 Computer Networks

//Class Host defines, and allows for processes that involve specific packets, and the sending queue

#ifndef HOST\_H

#define HOST\_H

#include "packets.h"

#include "math.h"

#include <queue>

#include <list>

extern list<host> network;

using namespace std;

class host

{

public:

host(int hNum);

~host();

double generateTimeToNext();

void receivedPacket(packet Input); //Manages how to handle received packets

void setBackOff();

//Randomly generate time until next on queue can be sent

inline int returnBackOff() { return backOffCounter; } //Gives value of backoff to network

void decBackOff();

//Reduces Back Off counter by one each time loop

packet popOutgoing();

//Pops front of queue to send

host\* returnRandHost();

//Decide on a host for the packet to go to

inline packet\* peekOutgoing() { return &(outgoingBuffer.front()); }

//Lets us look at front of queue from main

void decSIFSTime();

//Decreases all unproccessed ACKs by one cycle

inline void resetBOCounter() { failedBOCounter = 0; } //Once packet sends, set failed BO counter back to 0

void runNormalOp(double sClock);

//Determine if TTNSend is <= simClock, if so, add to outgoing buffer

inline int returnHostNumber() { return hostNumber; }

//Identify yourself fool!

inline bool wantsToSend() { return !outgoingBuffer.empty(); }

//Returns if it has something to send

private:

protected:

queue<packet> outgoingBuffer;

//Buffer for outgoing packets, ACK and data

list<ACK> unprocACK;

//All ACKs that have not waited their SIFS time go here

//list<packet> unACK;

//List of unacknowlged packets

int packetNumber;

//Unique number given toa ll sent data packets, ACKs get number pased on sentData

int hostNumber;

//Unique number for host

int backOffCounter;

//Counter until it is able to try and send again

int failedBOCounter;

//Every time we fail to send after BO reaches 0, increase time of next BO by failedBOCounter

double TTNSend;

//Time To Next Send, used for determining time between current simClock and packet send attempt

};

#endif

//host.cpp

//Chris Bird and Blake Tacklind

//ECS 152A/EEC 173 Computer Networks

#include "stdafx.h"

#include "host.h"

#include "packets.h"

#include "stdio.h"

#include "iostream"

#include "time.h"

#include <queue>

#include <list>

#include <stack>

#define MRT 5

using namespace std;

extern double lambda;

list<host> network;

host::host(int hNum)

{

packetNumber = 0;

hostNumber = hNum;

TTNSend = generateTimeToNext();

backOffCounter = 0;

failedBOCounter = 0;

};

host::~host()

{

//Destructor

};

double host::generateTimeToNext()

{

double u; //w

u = (rand() / (RAND\_MAX + 1.0));

double temp = (((-1 / lambda) \* log(1 - u)) \* 1000); //Make sure that we convert from double to int before returning

return temp;

};

void host::receivedPacket(packet Input)

{

//Wait receive time

if (Input.getAckValue()) //If packet received is an ACK

{

//ACK Received!

/\*if (!unACK.empty())

{

list<packet>::iterator it = unACK.begin();

while (it != unACK.end() || (\*it).returnNumber() != Input.returnNumber()) //After this, we have \*it pointing to sentPacket in unACK list

++it;

unACK.erase(it);

//Erases packet from unACK'd list, saying it was successful

}\*/

}

else

//If packet received is data

{

//unprocACK.push\_back(ACK(Input.returnNumber(), Input.returnSend(), this)); //Put new ACK onto unproccessed ACK queue to wait their time

}

};

void host::setBackOff()

{

double u = (rand() / (RAND\_MAX + 1.0));

u \*= MRT;

u \*= ++failedBOCounter;

backOffCounter = (int)u;

}

void host::decBackOff()

{

if (backOffCounter)

backOffCounter--;

}

packet host::popOutgoing()

{

packet temp = outgoingBuffer.front();

outgoingBuffer.pop();

return temp;

}

host\* host::returnRandHost()

{

int dest;

do{ (dest = rand() % network.size()); } while (dest == this->hostNumber); //Sets dest to a random non-this host

list<host>::iterator it = network.begin();

for (int i = 0; i < dest; i++) //Gets iterator to host chosen by dest

++it;

return &\*it; //Returns pointer to host chosen

}

void host::decSIFSTime() //Decrements all SIFS times by 1 step (of 5)

{

if (!unprocACK.empty()) //If list is empty, do not use

for (list<ACK>::iterator it = unprocACK.begin(); it != unprocACK.end(); ++it)

if ((\*it).decSIFS()) //Decrements, and if it's 0, puts onto outgoing queue

outgoingBuffer.push(\*it);

}

void host::runNormalOp(double sClock)

{

if (TTNSend <= sClock)

{

outgoingBuffer.push(sentData(packetNumber++, returnRandHost(), this)); //Puts new packet into outgoingBuffer after TTN has arrived

TTNSend = sClock + generateTimeToNext(); //Creates next timestamp when a new packet will be sent

}

}

//packets.h

//Chris Bird and Blake Tacklind

//ECS 152A/EEC 173 Computer Networks

//Defines and sets up the packet class, which has two child classes, ACK and sentData

//sentData is a data packet, where as ACK is purely an acknowledgment of 64 bytes in size

#ifndef PACKETS\_H

#define PACKETS\_H

class host;

//#include "host.h"

#include <string>

#define sizeOfACK 64

extern double speedOfMedium;

extern double ACKTime;

using namespace std;

//Event class, parent to arrivals and ACK packet

class packet

{

public:

inline packet() {;}

inline ~packet() {;}

virtual string toString() { return "Packet"; }

inline int returnSize(double sTime) { return (int)round((sTime \* speedOfMedium) / 8); }

//Returns size of packet and sets it

inline int returnSize() { return size; } //Returns size of current packet

inline double returnTime() { return 1000 \* ((8.0 \* size) / speedOfMedium); }

//Returns time to process packet

inline bool getAckValue() { return ack; } //Tells if packet is ACK or not

inline int returnNumber() { return packetNumber; } //Unique number given to packet to differentiate from others from same host

inline host\* returnDest() { return destination; } //Tells destination of packet

inline host\* returnSend() { return sender; } //Gives access to name of host sending packets

inline double returnTimeOC() { return timeOC; } //Returns the time that the packet was created

private:

protected:

int size; //Packet size

double serviceTime;

//Time to process

bool ack; //Determines if packet is an ACK or not (can be ambiguous based on size if random-generated size is 64)

int packetNumber; //Identifier for packet

host\* destination;

//Is destination for packet

host\* sender; //Points to sender of packet (us)

double timeOC;

};

class sentData : public packet //Class of sentData packet

{

public:

sentData(int pNum, host\* destination, host\* send);

~sentData();

double generatePacketServiceTime();

inline string toString()

{ return "Data"; }

private:

protected:

};

class ACK : public packet //Class of ACK packet

{

public:

ACK(int pNum, host\* destination, host\* send);

~ACK();

inline string toString() { return "ACK"; }

inline bool decSIFS() { return !(--SIFStime); }

private:

protected:

double timeQueued;

int SIFStime;

};

#endif

//packets.cpp

//Chris Bird and Blake Tacklind

//ECS 152A/EEC 173 Computer Networks

#include "stdafx.h"

#include "stdio.h"

#include "iostream"

#include "stdlib.h"

#include "math.h"

#include <queue>

#include <list>

#include "packets.h"

#define maxTime 1.12

#define maxSize 1544

using namespace std;

double speedOfMedium = (11 \* (1000000));

//This is the speed of the network (11 Mbps)

double ACKTime = ((8.0 \* sizeOfACK) / speedOfMedium);

//Time it takes to send an ACK

float lambda;

static int timeToNext; //Time until next packet is to be sent

extern double simClock;

extern double mu;

sentData::sentData(int pNum, host\* dest, host\* send)

{

size = returnSize(serviceTime = generatePacketServiceTime());

//Creates time and size of packet

if (size > maxSize) //If randomly generated size is greater than max size, set to max size

size = maxSize;

ack = false;

packetNumber = pNum;

destination = dest;

sender = send;

timeOC = simClock;

};

sentData::~sentData()

{

//Destructor

};

double sentData::generatePacketServiceTime()

//Generates the time to the next arriving packet

{

double u; //w

u = (rand() / (RAND\_MAX + 1.0));

double temp = ((-1 / mu) \* log(1 - u));

//Make sure that we convert from double to int before returning

return temp;

};

ACK::ACK(int pNum, host\* dest, host\* send)

{

size = 64;

serviceTime = returnTime();

ack = true;

packetNumber = pNum;

destination = dest;

sender = send;

timeQueued = simClock;

SIFStime = 5;

}

ACK::~ACK()

{

//Destructor

};