Alexandra Lomotan, David Delgado, Matthew Chi

Project 3 Final Draft

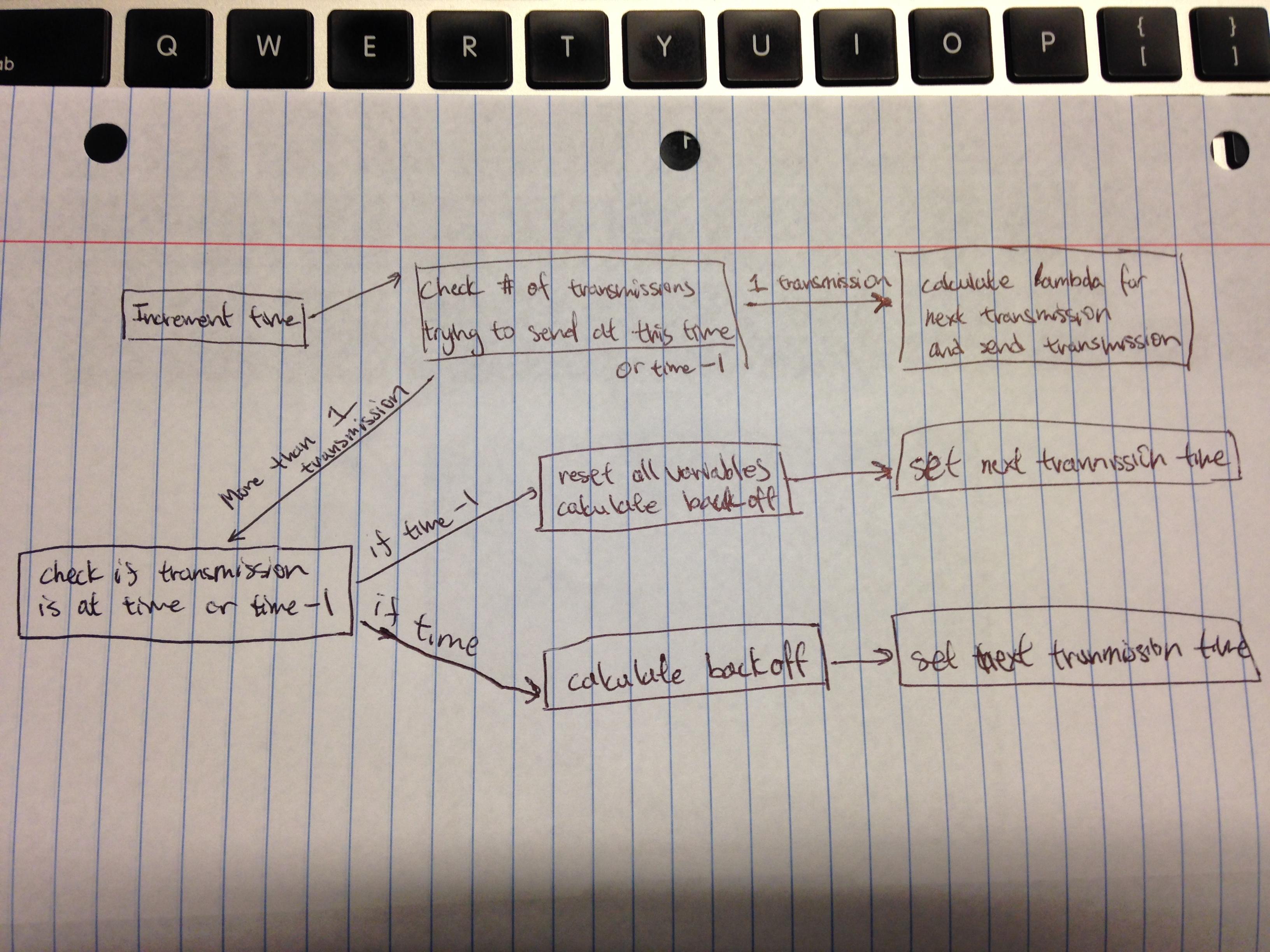
May 12, 2015

Part D

How to calculate:

* Throughput = (number of transmission\*transmission size)/ (total time slots\*time per slot)
* Average delay = sum of transmission delays/number of transmissions
* Average delay jitter = sqrt(((sum of squares of individual delays - avg delay)/number of transmissions)/number of stations)

Flow chart:



Part E

|  |  |  |  |
| --- | --- | --- | --- |
| Lambda | Throughput (Mb) | Traffic Load | Average Delay |
| 20 | 308 | 0.1 | 26 |
| 18 | 347 | 0.111111111 | 23 |
| 16 | 398 | 0.125 | 20 |
| 14 | 391 | 0.142857143 | 20 |
| 12 | 465 | 0.166666667 | 17 |
| 10 | 557 | 0.2 | 14 |
| 8 | 712 | 0.25 | 11 |
| 6 | 1012 | 0.333333333 | 8 |
| 4 | 1282 | 0.5 | 6 |

Code for Part E:

Client:

#include <stdio.h>

#include <stdlib.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <sys/time.h>

#include <pthread.h>

#include <math.h>

#include <string.h>

#include <fcntl.h>

struct timeval slotTime;

int sockfd, port, length;

struct sockaddr\_in serv\_addr, my\_addr;

socklen\_t servlen;

int min(int n, int m) {

if (n < m)

return n;

return m;

}

int computeBackoff(int n) {

int k = min(n, 10);

k = (int)pow(2, k);

return rand() % k;

}

int computeSend(int lambda) {

return (0 - lambda) \* (log((double)rand()/(double)RAND\_MAX) / log(2));

}

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

{

if (argc < 3) {

fprintf(stderr,"usage %s hostname port\n", argv[0]);

exit(0);

}

port = atoi(argv[2]);

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

if (sockfd < 0) {

fprintf(stderr,"ERROR opening socket\n");

exit(0);

}

my\_addr.sin\_family = AF\_INET;

my\_addr.sin\_addr.s\_addr = INADDR\_ANY;

my\_addr.sin\_port = htons(port);

serv\_addr.sin\_addr.s\_addr = inet\_addr(argv[1]);

serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_port = htons(port);

servlen = sizeof(serv\_addr);

if (connect(sockfd, (struct sockaddr \*) &serv\_addr, servlen) < 0) {

printf("Error on connect\n");

exit(1);

}

fd\_set set, readset;

FD\_ZERO(&set);

FD\_SET(sockfd, &set);

// done = 0;

// timeslot = 0;

int collisions = 0;

int timeToNext = 1;

setsockopt(sockfd, SOL\_SOCKET, SO\_RCVTIMEO, &slotTime, sizeof(slotTime));

int successes[21] = {0};

int selected = 0;

int i;

char message[1024];

// for (j = 0; j < 1024; j++)

// message[j] = 'a';

int lambda;

for (lambda = 20; lambda > 3; lambda -= 2) {

// printf("Lambda %d:\n", lambda);

for (i = 0; i < 5000; i++) {

readset = set;

timeToNext--;

// printf("timeslot %d, timetonext %d\n", i, timeToNext);

// if supposed to send

if (timeToNext == 0) {

memset(message, 'a', sizeof(message));

length = write(sockfd, message, sizeof(message));

// printf("Sending in timeslot %d\n", i);

if (length < 0) {

fprintf(stderr,"ERROR writing to socket\n");

exit(0);

}

}

memset(message, 0, sizeof(message));

length = 0;

// if sent this timeslot, wait 2x timeslots to receive response

// if didn't send this timeslot, select will take up timeslot (timeout)

slotTime.tv\_sec = 0;

slotTime.tv\_usec = 800;

selected = select(FD\_SETSIZE, &readset, (fd\_set \*)0, (fd\_set \*)0, &slotTime);

if (timeToNext < 1) {

if (selected < 0)

printf("Error on select\n");

// else if (selected == 0)

// printf("Timeout\n");

else if (selected != 0) {

// printf("Message is %d bytes\n", strlen(message));

length = read(sockfd, &message, sizeof(message));

// printf("Read %d bytes\n", length);

}

}

if (length > 0) {

// printf("Message: %s\n", message);

// if collision, compute backoff until attempt retransmission

if (length > 8) {

// printf("Message: %s\n", message);

// printf("Collision sending in timeslot %d\n", i);

collisions++;

timeToNext = computeBackoff(collisions);

}

// if success, compute next transmission

else {

// printf("Success sending in timeslot %d\n", i);

timeToNext = computeSend(lambda);

successes[lambda]++;

collisions = 0;

}

}

// dunno what else happens

else {

if (timeToNext < 0)

timeToNext = 1;

//printf("Got %d from recvfrom function in timeslot %d\n", length, i);

}

}

}

close(sockfd);

printf("\n");

// print final output

for (lambda = 20; lambda > 3; lambda -= 2) {

printf("Lambda %2d, packets: %4d, throughput: %4d, avg delay: %3d, load: %01.2f\n", lambda, successes[lambda], successes[lambda]\*1024\*8/5000, 5000/successes[lambda], 2.0/(double)lambda);

}

return 0;

}

Server:

#include <stdio.h>

#include <stdlib.h>

#include <sys/socket.h>

#include <sys/time.h>

#include <sys/types.h>

#include <unistd.h>

#include <netinet/in.h>

#include <string.h>

#include <errno.h>

#include <arpa/inet.h>

#include <fcntl.h>

#define MAX\_CONNECTIONS 2

void readmsg(int sockfd, char \*message, char \*buffer, int size) {

int n;

while (strlen(message) < size) {

n = read(sockfd, buffer, 1024);

if (n < 0)

error("ERROR reading from socket");

// if (n == 0)

// printf("read == 0\n");

// if there are more bytes than fit for the current message

if (strlen(message) + strlen(buffer) > size) {

int amount = size - strlen(message); // calculate bytes for current message

int i;

// copy remaining byes for current message

for (i = 0; i < amount; i++) {

message[strlen(message)] = buffer[i];

}

// zero the rest of the buffer, leaving the excess belonging to next message

memset(buffer + amount, 0, 1024 - amount);

}

else {

strcat(message, buffer);

memset(buffer, 0, 1024);

}

}

// printf("length = %d strlen(message) = %d size = %d\n", n, strlen(message), size);

}

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

{

if (argc < 2) {

fprintf(stderr,"ERROR, no port provided\n");

exit(1);

}

int sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

if (sockfd < 0) {

fprintf(stderr,"ERROR opening socket\n");

exit(1);

}

// allow multiple connections to the server address

int opt = 1;

if (setsockopt(sockfd, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(opt)) < 0) {

fprintf(stderr, "ERROR setsockopt reuseaddr\n");

exit(1);

}

// setup address

int port = atoi(argv[1]);

struct sockaddr\_in address;

socklen\_t addr\_len = sizeof(address);

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_port = htons(port);

// client address

struct sockaddr\_in cli\_addr;

socklen\_t clilen;

if (bind(sockfd, (struct sockaddr \*) &address, addr\_len) < 0) {

fprintf(stderr,"ERROR on binding\n");

exit(1);

}

// listen for MAX\_CONNECTIONS connections

if (listen(sockfd, MAX\_CONNECTIONS) < 0) {

fprintf(stderr, "ERROR listen\n");

exit(1);

}

// socket data

fd\_set readfds;

int client\_socket[MAX\_CONNECTIONS];

int i;

for (i = 0; i < MAX\_CONNECTIONS; i++)

client\_socket[i] = 0;

int max\_sd, sd, new\_sd;

// message buffers

char \*message = malloc(1024);

char \*message2 = malloc(1024);

char \*buffer = malloc(1024);

memset(message, 0, 1024);

memset(message2, 0, 1024);

memset(buffer, 0, 1024);

// other variables used in infinite loop

int length = 0;

int value;

struct timeval slotTime;

// printf("Entering infinite loop\n");

while (1) {

// clear and add main sockfd to set

FD\_ZERO(&readfds);

FD\_SET(sockfd, &readfds);

// printf("Added sockfd to FD\_SET\n");

max\_sd = sockfd;

// add client sockets to set

for ( i = 0 ; i < MAX\_CONNECTIONS ; i++)

{

sd = client\_socket[i];

if(sd > 0) {

FD\_SET(sd , &readfds);

// printf("Added socket %d to FD\_SET\n", sd);

}

if(sd > max\_sd)

max\_sd = sd;

}

slotTime.tv\_sec = 0;

slotTime.tv\_usec = 800;

// for a slot time, wait to check if something happened on a socket

value = select(FD\_SETSIZE, &readfds, NULL, NULL, &slotTime);

// printf("Selected %d\n", value);

if (value < 0 && errno != EINTR)

printf("ERROR select\n");

// if something happened on sockfd, it is a new connection

if (FD\_ISSET(sockfd, &readfds)) {

new\_sd = accept(sockfd, (struct sockaddr \*)&cli\_addr, (socklen\_t \*)&clilen);

if (new\_sd < 0) {

fprintf(stderr, "ERROR accept\n");

exit(1);

}

// printf("New connection, descriptor is %d, ip is %s, port is %d\n", new\_sd, inet\_ntoa(cli\_addr.sin\_addr), ntohs(cli\_addr.sin\_port));

int flag = 0;

// add new socket to array

for (i = 0; i < MAX\_CONNECTIONS; i++)

{

//if position is empty

if( client\_socket[i] == 0 && flag == 0 )

{

client\_socket[i] = new\_sd;

// printf("Adding socket %d to list of sockets as %d\n", new\_sd ,i);

flag = 1;

}

}

}

// if a client socket has data

if (FD\_ISSET(client\_socket[0], &readfds)) {

// printf("Data from client\_socket[0]\n");

memset(buffer, 0, 1024);

readmsg(client\_socket[0], message, buffer, 1024);

// printf("Read %d bytes from client 0\n", strlen(message));

}

if (FD\_ISSET(client\_socket[1], &readfds)) {

// printf("Data from client\_socket[1]\n");

memset(buffer, 0, 1024);

readmsg(client\_socket[1], message2, buffer, 1024);

// printf("Read %d bytes from client 1\n", strlen(message2));

}

// if received from both clients

if (strlen(message) > 0 && strlen(message2) > 0) {

memset(message, 0, 1024);

strncpy(message, "Collision", 9);

length = write(client\_socket[0], message, strlen(message));

if (length < 0) {

fprintf(stderr, "ERROR write client 0\n");

}

length = write(client\_socket[1], message, strlen(message));

if (length < 0) {

fprintf(stderr, "ERROR write client 1\n");

}

}

// if received from one client

else if (strlen(message) || strlen(message2)) {

if (strlen(message) > 0) {

memset(message, 0, 1024);

strncpy(message, "Success", 7);

length = write(client\_socket[0], message, strlen(message));

if (length < 0) {

fprintf(stderr, "ERROR write client 0\n");

}

}

else {

memset(message2, 0, 1024);

strncpy(message2, "Success", 7);

length = write(client\_socket[1], message2, strlen(message2));

if (length < 0) {

fprintf(stderr, "ERROR write client 1\n");

}

}

// printf("Wrote to client\n");

}

// clear message buffers

memset(message, 0, 1024);

memset(message2, 0, 1024);

}

close(sockfd);

return 0;

}

Part D code:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

#define ITERATIONS 10

#define MAX\_STATIONS 200

int min(int n, int m) {

if (n < m)

return n;

return m;

}

int max(int n, int m) {

if (n < m)

return m;

return n;

}

int max\_array(int a[], int num\_elements){

int i, max = -1;

for (i = 0; i < num\_elements; i++)

{

if (a[i] > max)

{

max = a[i];

}

}

return(max);

}

int add\_array(int a[], int num\_elements){

int i, sum = 0;

for(i = 0; i < num\_elements; i++)

sum += a[i];

return sum;

}

int nextTransmission(int lambda) {

int temp = 0;

while(temp == 0)

temp = (0 - lambda) \* (log((double)rand()/(double)RAND\_MAX) / log(2));

return temp;

}

int computeBackoff(int n) {

int k = min(n, 10);

k = (int)pow(2, k);

return rand() % k;

}

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

int lambda = 200;

int stations\_ = 20;

int p = 512;

int defrun = 0;

if(argc != 1 && argc != 4){

printf("Invalid Input\n");

exit(1);

}

if(argc == 4){

lambda = atoi(argv[1]);

defrun = atoi(argv[1]);

stations\_ = atoi(argv[2]);

p = atoi(argv[3]);

}

srand(time(NULL));

for (; lambda > 10; lambda -= 20) {

int contentionInterval[ITERATIONS] = {0};

int iteration;

for (iteration = 0; iteration < ITERATIONS; iteration++) {

// init data for iteration

int timeSent[MAX\_STATIONS] = {0};

int lastTimeSent[MAX\_STATIONS] = {0};

int nextTimeToSend[MAX\_STATIONS] = {0};

int lastCollisionTime[MAX\_STATIONS] = {0};

int transmissionsSent[MAX\_STATIONS] = {0};

int currentCollisionCount[MAX\_STATIONS] = {1};

int lastCollisiondCount[MAX\_STATIONS] = {0};

int counterdelay[MAX\_STATIONS] = {0};

int delayjitter[MAX\_STATIONS][4000] = {{0}};

int T = 0;

// loop for all STATIONS to transmit

while (T < 200000) {

// set sending counter to 0, sendingIndex to invalid

int sending = 0;

int sendingIndex = -1;

int i;

// check how many STATIONS want to send in this interval

for (i = 0; i < stations\_; i++) {

if (nextTimeToSend[i] == T-1 || nextTimeToSend[i] == T) {

sending++;

if (nextTimeToSend[i] == T)

sendingIndex = i;

}

}

// if only one STATIONS wants to send in the interval, and it wants to send

// at time T, then send.

if (sending == 1 && sendingIndex != -1) {

if (nextTimeToSend[sendingIndex] == T) {

int dif = T - lastTimeSent[sendingIndex];

lastTimeSent[sendingIndex] = timeSent[sendingIndex];

timeSent[sendingIndex] = T;

lastCollisiondCount[sendingIndex] = currentCollisionCount[sendingIndex];

currentCollisionCount[sendingIndex] = 0;

nextTimeToSend[sendingIndex] += nextTransmission(lambda);

transmissionsSent[sendingIndex]++;

delayjitter[sendingIndex][counterdelay[sendingIndex]++] = dif;

}

}

// if more than one STATION wants to send in the interval, check if something tried

// to send at time T-1. If so, reset it. Compute nextTimeToSend for all colliding STATION

else if (sending > 1) {

for (i = 0; i < stations\_; i++) {

if (timeSent[i] == T-1) {

timeSent[i] = lastTimeSent[i];

lastCollisionTime[i] = nextTimeToSend[i];

currentCollisionCount[i] = lastCollisiondCount[i];

currentCollisionCount[i]++;

transmissionsSent[i]--;

delayjitter[i][counterdelay[i]--] = 0;

while (nextTimeToSend[i] <= T)

nextTimeToSend[i] += computeBackoff(currentCollisionCount[i]);

}

if (nextTimeToSend[i] == T-1 || nextTimeToSend[i] == T) {

lastCollisionTime[i] = nextTimeToSend[i];

currentCollisionCount[i]++;

while (nextTimeToSend[i] <= T)

nextTimeToSend[i] += computeBackoff(currentCollisionCount[i]);

}

}

}

// increment time

T++;

}

int numStations;

float avgstandev = 0, avgmean = 0;

for(numStations = 0; numStations < stations\_; numStations++){

float sumdev = 0;

float mean = add\_array(delayjitter[numStations], transmissionsSent[numStations]);

mean /= transmissionsSent[numStations];

avgmean += mean;

int numTransmissions;

for(numTransmissions = 0; numTransmissions < transmissionsSent[numStations]; numTransmissions++){

sumdev += (delayjitter[numStations][numTransmissions] - mean) \* (delayjitter[numStations][numTransmissions] - mean);

}

float standev = sqrt(sumdev/numTransmissions);

avgstandev += standev;

printf("Standard Deviation Lambda %2d, Station %2d: %6f\n", lambda, numStations, standev);

}

avgstandev /= numStations;

avgmean /= numStations;

// find the max timeSent, divide by number of STATIONS to get contention interval

int maximum = 0;

int i;

for (i = 0; i < stations\_; i++) {

maximum = max(maximum, timeSent[i]);

}

contentionInterval[iteration] = maximum / stations\_;

int z;

printf("\nTime Sent: ");

for(z = 0; z < stations\_; z++){

printf("%6d ", timeSent[z]);

}

printf("\nNext Time to Send: ");

for(z = 0; z < stations\_; z++){

printf("%6d ", nextTimeToSend[z]);

}

printf("\nLast Colision Time: ");

for(z = 0; z < stations\_; z++){

printf("%6d ", lastCollisionTime[z]);

}

printf("\nCollision Count: ");

for(z = 0; z < stations\_; z++){

printf("%6d ", currentCollisionCount[z]);

}

printf("\nTransmissions Sent: ");

for(z = 0; z < stations\_; z++){

printf("%6d ", transmissionsSent[z]);

}

printf("\n\n");

printf("Throughput: ");

int ans = add\_array(transmissionsSent, stations\_);

ans \*= p;

ans /= (200000\*.0000512);

printf("%6dbps\n", ans);

printf("Average Delay: ");

printf("%6fs\n", avgmean);

printf("Average Delay Jitter: ");

printf("%6f\n\n", avgstandev);

/\*printf("Time Sent: ");

printf("%6d ", max\_array(timeSent, stations\_));

printf("\nNext Time to Send: ");

printf("%6d ", max\_array(nextTimeToSend, stations\_));

printf("\nLast Colision Time: ");

printf("%6d ", max\_array(lastCollisionTime, stations\_));

printf("\nCollision Count: ");

printf("%6d ", max\_array(currentCollisionCount, stations\_));

printf("\nTransmissions Sent:");

printf("%6d ", max\_array(transmissionsSent, stations\_));

printf("\n\n");\*/

//printf("Contention interval is %3d for lambda %2d iteration %3d\n", contentionInterval[iteration], lambda, iteration);

}

//not default run

if(defrun != 0)

break;

}

}