گزارش پروژه چهارم درس شبکه‌های کامپیوتری

دکتر یزدانی

بهار ۱۴۰۱

پرنیان فاضل ۸۱۰۱۹۸۵۱۶ - پریا خوش‌تاب ۸۱۰۱۹۸۳۸۷

The purpose of this exercise is to get acquainted with the function of **TCP** and to implement the mechanism of sending correct information along with congestion control using UDP sockets in the network.

Implementation

Our project consists of the following *.c and *.h files:

- packet.c / packet.h: These files are mainly used for creating tcp packets with header and space for data of a given size.

  Packet.h:

```
enum packet_type {
   DATA,
   ACK,
};

typedef struct {
   int dest;
   int src;
   int seqno;
   int ackno;
   int ctr_flags;
   int data_size;
}tcp_header;

#define MSS_SIZE     1500
#define UDP_HDR_SIZE     8
```

```
#define IP_HDR_SIZE    20
#define TCP_HDR_SIZE    sizeof(tcp_header)
#define DATA_SIZE   (MSS_SIZE - TCP_HDR_SIZE - UDP_HDR_SIZE -
IP_HDR_SIZE)
typedef struct {
   tcp_header  hdr;
   char    data[0];
}tcp_packet;

tcp_packet* make_packet(int seq);
int get_data_size(tcp_packet *pkt);
```

Note that the size of the packets that are sent is 1500 bytes(MSS_SIZE) , so if we subtract the size of the headers from it, the size of the pure data that is sent each time becomes 1448 bytes (DATA_SIZE).

1500 - 20 - 8 - (4*6) = 1448 byte

packet.c:

```
#include"packet.h"

static tcp_packet zero_packet = {.hdr={0}};
/*
 * create tcp packet with header and space for data of size len
 */
tcp_packet* make_packet(int len)
{
    tcp_packet *pkt;
    pkt = malloc(TCP_HDR_SIZE + len);

    *pkt = zero_packet;
    pkt->hdr.data_size = len;
    return pkt;
}

int get_data_size(tcp_packet *pkt)
{
    return pkt->hdr.data_size;
}
```

- queue.c / queue.h: In these files, we have implemented a FIFO queue data structure from scratch, because the C language does not support the queue library.

Queue.h:

```c
#define QUEUE_H

/* Represents a node in a queue */
typedef struct node
{
   void *item; // data contained in node
   struct node *next; // next item in the queue
} NODE;

/* Represents a queue */
typedef struct queue
{
   NODE *head; // head of queue
   NODE *tail; // tail of queue
   int size;   // size of queue
} QUEUE;

/*
* Creates an empty queue.
*
* Input:
* None
*
* Output:
* A pointer to the created QUEUE. NULL pointer
* if the operation fails.
*/
QUEUE *queueCreate(void);

/*
* Adds and item to the back of the queue.
*
* Input:
* Queue *: pointer to a QUEUE
* void *: item to be added
*
* Output:
```

```c
* 0 on success, -1 on failure
*/
int enqueue(QUEUE *, void *);

/*
* Removes an item from the front of the queue.
*
* Input:
* QUEUE *: pointer to a QUEUE
*
* Output:
* 0 on success, -1 on failure
*/
int dequeue(QUEUE *);

/*
* Returns the items at the front of the queue.
*
* Input:
* QUEUE *: pointer to a QUEUE
*
* Output:
* A pointer to the item at the front of the queue,
* NULL pointer if the queue is empty.
*/
void *peek(QUEUE *);

/*
* Returns the size of the queue
*/
int size(QUEUE *q);

#endif // QUEUE_H
```

queue.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "queue.h"
```

```
QUEUE *queueCreate()
{
    // allocate space for a queue struct
    QUEUE *queue = malloc(sizeof(QUEUE));

    if ( queue == NULL )
    {
        fprintf(stderr, "queueCreate(): malloc error\n");
        return NULL;
    }

    // set default values
    queue->head = NULL;
    queue->tail = NULL;
    queue->size = 0;

    return queue;
}

int enqueue(QUEUE *q, void *item)
{
    NODE *node;

    // set up new node
    node = malloc(sizeof(NODE));

    if ( node == NULL )
    {
        fprintf(stderr, "enqueue(): malloc error\n");
        return -1;
    }
    else
    {
        node->next = NULL;
        node->item = item;
    }

    // queue is empty
    if ( q->size == 0 )
    {
        q->head = node;
        q->tail = node;
```

```
        q->size = 1;
    }
    // queue is non-empty
    else
    {
        q->tail->next = node;
        q->tail = node;
        q->size++;
    }

    return 0;
}

int dequeue(QUEUE *q)
{
    // queue is empty
    if ( q->size == 0 )
    {
        fprintf(stderr, "dequeue(): cannot remove item from empty
queue\n");
        return -1;
    }

    // queue contains 1 item
    if ( q->size == 1 )
    {
        q->head = NULL;
        q->tail = NULL;
        q->size = 0;
    }
    // queue contains multiple items
    else
    {
        q->head = q->head->next;
        q->size--;
    }

    return 0;
}

void *peek(QUEUE *q)
{
    // queue is empty
```

```c
    if ( q->size == 0 )
    {
        fprintf(stderr, "peek(): queue is empty\n");
        return NULL;
    }
    // queue is non-empty
    return q->head->item;
}

int size(QUEUE *q)
{
    return q->size;
}
```

- common.c / common.h: In these files, we have implemented a wrapper for perror.

common.h:

```c
#define NONE    0x0
#define INFO    0x1
#define WARNING 0x10
#define INFO    0x1
#define DEBUG   0x100
#define ALL     0x111

#define VLOG(level, ... ) \
   if(level & verbose) { \
       fprintf(stderr, ##__VA_ARGS__ );\
       fprintf(stderr, "\n");\
   }\

void error(char *msg);

#endif
```

common.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include"common.h"

int verbose = ALL;

void error(char *msg) {
    perror(msg);
    exit(1);
}
```

- sender.c: For reliability, we implement a sliding window on the sender side. We keep a limit on the max number of packets that can be sent and not ACKed at any given time. If we receive back an ACK with sequence number higher than our current send base, then we slide the window forward and transmit new packets. Of course, we are also keeping a timer on the earliest unACKed packet. There can be a cumulative ACK which will indicate all the earlier packets have been successfully received. If there is a time-out, we retransmit the missing packet(s).

```c
#define STDIN_FD 0
#define RETRY 200 //milli second
#define PORT_ROUTER 8080
#define PORT_B 8081

int transmission_round = 0; //keep track of the round
int next_seqno; // next byte to send
int exp_seqno;  // expected byte to be acked
int send_base = 0;  // first byte in the window
float window_size = 50; // window size at the beginning of slow
start
int portno;
int timer_on = 0;
```

```
FILE *fp;

int sockfd, serverlen;
struct sockaddr_in serveraddr, senderAddr;
struct itimerval timer;
tcp_packet *recvpkt;
sigset_t sigmask;

void send_packet(char* buffer, int len, int seqno);
void resend_packets(int sig);
void init_timer(int delay, void (*sig_handler)(int));
void start_timer();
void stop_timer();

void send_packet(char* buffer, int len, int seqno) {
   tcp_packet *sndpkt = make_packet(len);
   memcpy(sndpkt->data, buffer, len);
   sndpkt->hdr.seqno = seqno;
   sndpkt->hdr.src = portno;
   sndpkt->hdr.dest = PORT_B;
   printf("Sending packet of sequence number %d of data size %d
to %s\n", seqno, len, inet_ntoa(serveraddr.sin_addr));
   if (sendto(sockfd, sndpkt, TCP_HDR_SIZE +
get_data_size(sndpkt), 0,
                       (const struct sockaddr *)&serveraddr,
serverlen) < 0)
   {
       error("sendto");
   }
   free(sndpkt);
}



void resend_packets(int sig)
{
   char buffer[DATA_SIZE];
   int len;

   if (sig == SIGALRM)
   {
       VLOG(INFO, "Timeout happened");
```

```
        //resend all packets range between send_base and
next_seqno

        for (int i = send_base; i < next_seqno; i += DATA_SIZE)
        {
            // locate the pointer to be read at next_seqno
            fseek(fp, i, SEEK_SET);
            // read bytes from fp to buffer
            len = fread(buffer, 1, DATA_SIZE, fp);
            // send pkt
            send_packet(buffer, len, i);
        }
    }
}

void start_timer()
{
    sigprocmask(SIG_UNBLOCK, &sigmask, NULL);
    setitimer(ITIMER_REAL, &timer, NULL);
    timer_on = 1;
    // printf("Timer on\n");
}

void stop_timer()
{
    sigprocmask(SIG_BLOCK, &sigmask, NULL);
    timer_on = 0;
    // printf("Timer off\n");
}

/*
* init_timer: Initialize timeer
* delay: delay in milli seconds
* sig_handler: signal handler function for resending unacknoledge
packets
*/
void init_timer(int delay, void (*sig_handler)(int))
{
    signal(SIGALRM, resend_packets);
    timer.it_interval.tv_sec = delay / 1000; // sets an interval
of the timer
    timer.it_interval.tv_usec = (delay % 1000) * 1000;
    timer.it_value.tv_sec = delay / 1000; // sets an initial value
```

```
    timer.it_value.tv_usec = (delay % 1000) * 1000;

    sigemptyset(&sigmask);
    sigaddset(&sigmask, SIGALRM);
}

int main(int argc, char **argv)
{
    char *hostname;
    char buffer[DATA_SIZE];
    int len;
    int dup_cnt; // count of continuous duplicate ACKs

    FILE* plot; // file that contains plot of window_size,
ssthresh and time
    // time_t now, start;
    struct timeval start, now;

    /* check command line arguments */
    if (argc != 4)
    {
        fprintf(stderr, "usage: %s <hostname> <port> <FILE>\n",
argv[0]);
        exit(0);
    }
    hostname = argv[1];
    portno = atoi(argv[2]);
    fp = fopen(argv[3], "r");
    if (fp == NULL)
    {
        error(argv[3]);
    }

    /* socket: create the socket */
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");


    memset(&senderAddr, 0, sizeof(senderAddr));
    senderAddr.sin_family = AF_INET; // IPv4
    senderAddr.sin_addr.s_addr = INADDR_ANY;
    senderAddr.sin_port = htons(portno);
```

```
    // Bind the socket with the server address
    if (bind(sockfd, (const struct sockaddr *)&senderAddr,
sizeof(senderAddr)) < 0){
        perror("bind failed");
        exit(EXIT_FAILURE);
    }


    /* initialize server server details */
    bzero((char *)&serveraddr, sizeof(serveraddr));
    serverlen = sizeof(serveraddr);

    /* covert host into network byte order */
    if (inet_aton(hostname, &serveraddr.sin_addr) == 0)
    {
        fprintf(stderr, "ERROR, invalid host %s\n", hostname);
        exit(0);
    }

    /* build the server's Internet address */
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(PORT_ROUTER);
    serveraddr.sin_addr.s_addr = INADDR_ANY;

    assert(MSS_SIZE - TCP_HDR_SIZE > 0);

    // init timer
    init_timer(RETRY, resend_packets);

    next_seqno = 0;
    exp_seqno = DATA_SIZE;

    dup_cnt = 1;

    plot = fopen("CWND.csv", "w+");
    gettimeofday(&start,NULL);
    clock_t start_time = clock();
    while (1)
    {
        gettimeofday(&now,NULL);

        // send all pkts in the effective window
```

```c
        while (next_seqno < send_base + (int)(window_size) *
DATA_SIZE)
        {
            // start the timer if not alr started
            if (timer_on == 0) start_timer();

            // printf("current send_base: %d \n", send_base);

            // locate the pointer to be read at next_seqno
            fseek(fp, next_seqno, SEEK_SET);
            // read bytes from fp to buffer
            len = fread(buffer, 1, DATA_SIZE, fp);
            // if end of file
            if (len <= 0)
            {
                VLOG(INFO, "End Of File read");
                send_packet(buffer, 0, 0);
                break;
            }

            // send pkt
            send_packet(buffer, len, next_seqno);

            // increment the next sequence number to be sent
            next_seqno += len;
        }

        // wait for ACK
        int n = recvfrom(sockfd, buffer, MSS_SIZE, MSG_WAITALL,
(struct sockaddr *)&serveraddr, (socklen_t *)&serverlen);

        // make recv pkt
        recvpkt = (tcp_packet *)buffer;
        assert(get_data_size(recvpkt) <= DATA_SIZE);

        // if receive ack for last pkt
        if (recvpkt->hdr.ackno % DATA_SIZE > 0) {
            printf("***All packets sent successfully\n");
            clock_t duration = clock() - start_time;
            double time_taken = ((double)duration)/CLOCKS_PER_SEC;
// calculate the elapsed time
```

```
printf("------------------------------------------->Time taken: %f secconds\n", time_taken);
            stop_timer();
            break;
        }

        if (recvpkt->hdr.ackno == send_base) {
            dup_cnt += 1;
        }

        if (exp_seqno <= recvpkt->hdr.ackno)
        {
            send_base = recvpkt->hdr.ackno;
            dup_cnt = 1; // reset count for dup ACKs
            exp_seqno = send_base + DATA_SIZE;
            stop_timer();
            // start timer for unacked pkts
            if (send_base < next_seqno)
                start_timer();
        }
    }
    fclose(plot);
    return 0;
}
```

- receiver.c: On the receiver side, we have an out-of-order buffer that contains all out-of-order packets (there is a limit for this buffer, if it's full then all other out-of-order packets will be dropped). After receiving an out-of-order packet, the receiver immediately sends back a duplicate ACK to indicate packet loss.

```
int sockfd;                         /* socket */
int portno;                         /* port to listen on */
int clientlen;                      /* byte size of client's address */
struct sockaddr_in serveraddr; /* server's addr */
struct sockaddr_in clientaddr; /* client addr */
```

```c
FILE *fp;
pthread_mutex_t m;
int dest;
int src;


void write_to_file(char* file_name, FILE* fp, int pos, char*
data, int len);
void send_ACK(int ackno);

void write_to_file(char* file_name, FILE* fp, int pos, char*
data, int len) {
   strcat(file_name, ".txt");
   fp = fopen(file_name, "a+");
   fseek(fp, pos, SEEK_SET);
   pthread_mutex_lock(&m);
   fwrite(data, 1, len, fp);
   pthread_mutex_unlock(&m);
   fclose(fp);

}

void send_ACK(int ackno) {
   tcp_packet *sndpkt = make_packet(0);
   sndpkt->hdr.ackno = ackno;
   sndpkt->hdr.dest = dest;
   sndpkt->hdr.src = src;
   sndpkt->hdr.ctr_flags = ACK;
   if (sendto(sockfd, sndpkt, TCP_HDR_SIZE, MSG_CONFIRM,
              (struct sockaddr *)&clientaddr, clientlen) < 0)
   {
       error("ERROR in sendto");
   }

}

int main(int argc, char **argv)
{
   int optval;                    /* flag value for setsockopt */
   tcp_packet *recvpkt;
   char* file_name;
   char buffer[MSS_SIZE];
   struct timeval tp;
```

```
    int cur_seqno;
    int ackno;

    int buffer_size = 4;
    tcp_packet* buffer_pkts[4]; // buffer to store out-of-order
pkts with size = 4 * DATA_SIZE
    int ind; // index of the last packet in the buffer
    /*
     * check command line arguments
     */
    if (argc != 3)
    {
        fprintf(stderr, "usage: %s <port> FILE_RECVD\n", argv[0]);
        exit(1);
    }
    portno = atoi(argv[1]);
    file_name = argv[2];

    fp = fopen(file_name, "w");
    if (fp == NULL)
    {
        error(argv[2]);
    }

    /*
     * socket: create the parent socket
     */
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");

    /* setsockopt: Handy debugging trick that lets
     * us rerun the server immediately after we kill it;
     * otherwise we have to wait about 20 secs.
     * Eliminates "ERROR on binding: Address already in use"
error.
     */
    optval = 1;
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR,
               (const void *)&optval, sizeof(int));

    /*
     * build the server's Internet address
```

```
    */
   bzero((char *)&serveraddr, sizeof(serveraddr));
   serveraddr.sin_family = AF_INET;
   serveraddr.sin_addr.s_addr = INADDR_ANY;
   serveraddr.sin_port = htons(PORT_B);

   clientaddr.sin_family = AF_INET;
   clientaddr.sin_addr.s_addr = INADDR_ANY;
   clientaddr.sin_port = htons(PORT_ROUTER);
   /*
    * bind: associate the parent socket with a port
    */
   if (bind(sockfd, (struct sockaddr *)&serveraddr,
            sizeof(serveraddr)) < 0)
       error("ERROR on binding");

   /*
    * main loop: wait for a datagram, then echo it
    */
   VLOG(DEBUG, "epoch time, bytes received, sequence number");

   clientlen = sizeof(clientaddr);
   cur_seqno = 0;
   ind = -1;

   // Need a buffer to store out-of-order pkts
   // Upon receiving out-of-order pkts (recvpkt->hdr.seqno -
cur_seqno > DATA_SIZE) we store these pkts in a buffer
   // Upon receiving an expected pkt (recvpkt->hdr.seqno -
cur_seqno <= DATA_SIZE), we look into the buffer
   // to find pkts that are in-order to this pkt and write all
these pkts to file

   while (1)
   {
       /*
        * recvfrom: receive a UDP datagram from a client
        */
       if (recvfrom(sockfd, buffer, MSS_SIZE, MSG_WAITALL,
(struct sockaddr *)&clientaddr, (socklen_t *)&clientlen) < 0)
       {
           error("ERROR in recvfrom");
       }
```

```
        recvpkt = (tcp_packet *)buffer;
        assert(get_data_size(recvpkt) <= DATA_SIZE);
        // if it is an empty pkt that signifies EOF
        if (recvpkt->hdr.data_size == 0)
        {
            continue;
        }

        dest = recvpkt->hdr.src;
        src = recvpkt->hdr.dest;

        gettimeofday(&tp, NULL);
        VLOG(DEBUG, "%lu, %d, %d", tp.tv_sec,
recvpkt->hdr.data_size, recvpkt->hdr.seqno);

        // handle in-order-packets
        if (recvpkt->hdr.seqno - cur_seqno <= DATA_SIZE)
        {
            cur_seqno = recvpkt->hdr.seqno; // update current
sequence number
            char name[20];
            sprintf(name, "%d", dest);
            write_to_file(name, fp, recvpkt->hdr.seqno,
recvpkt->data, recvpkt->hdr.data_size);

            ackno = recvpkt->hdr.seqno + recvpkt->hdr.data_size;

            int new_seqno = cur_seqno + recvpkt->hdr.data_size;
            int cnt = 0; // count of number of next expected pkts
found in the buffer

            //look into the buffer to find the next expected pkt
to write to file
            // IMPORTANT: seqno of received pkts might not always
be in an increasing order
            for (int i = 0; i <= ind; i++) {
                if (new_seqno == buffer_pkts[i]->hdr.seqno) {
                    cur_seqno = new_seqno;
                    char name[20];
                    sprintf(name, "%d", dest);
                    write_to_file(name, fp, cur_seqno,
buffer_pkts[i]->data, buffer_pkts[i]->hdr.data_size);
```

```
                new_seqno += get_data_size(buffer_pkts[i]);
                cnt += 1;
                ackno = new_seqno;
            }
        }
        // shift the packets to the left a number = cnt steps
        if (cnt >= 1){
            for (int i = cnt; i < buffer_size; i++) {
                buffer_pkts[i-cnt] = buffer_pkts[i];
            }
            // update index of the last pkt in the buffer
            ind -= cnt;
        }

        send_ACK(ackno);
    }

    // buffer out-of-order packets
    else {
        if (ind < buffer_size-1) {
            ind += 1;
            buffer_pkts[ind] = recvpkt;
        }
    }
  }
  return 0;
}
```

- router.c: To implement the router, we use two threads, one of which is used to send data and one of which is used to receive data from the hosts. In fact, when receiving data, we push the received packet to the queue, and when sending data, we pop the first element from the queue, and depending on the type of the packet (ACK or DATA), we send the data to the desired host(A or B). Note that in order to find the host (A or B) we want to send the data to, we have added src and dest fields to the tcp_header struct.

```
pthread_mutex_t bufferMutex;


#define PORT_ROUTER 8080
#define PORT_B   8081
#define MAXLINE 1500
#define DROP_PERCENTAGE 10

int sockfd, sockfd2;
double maxp = 0.02;
double tempP = 0;
double avg = 0;
int count = -1;
double weight = 0.003;
int minThreshold = 5, maxThreshold = 20;

struct sockaddr_in servaddr;
QUEUE* routerBuffer;
int random_drop(int percentage) {
   int randomP = (rand()%100);

   if (randomP < percentage) {
       return -1;
   }

   return 0;
}

int randomEarlyDetection(int s) {
   int res = 0;
   avg = ((1 - weight) * avg) + (weight * size(routerBuffer));

   if(minThreshold <= avg && avg < maxThreshold) {
       Count++;
```

```
        tempP = ((avg - minThreshold) * maxp)/(maxThreshold -
minThreshold);
        double P = tempP/(1 - (count * tempP));
        if(count == 50) {
            P = 1.0;
            res = -1;
        }
        double randomP = (rand()%100)/100.00;
        if(randomP <= P) {
            if(count != 50) {
                res = -1;
            }
            count = 0;
        } else {
            res = 0;
            count = -1;
        }
    } else if(maxThreshold <= avg) {
        res = -1;
        count = 0;
    } else {
        res = 0;
        count = -1;
    }
    return res;
}

void* recieve(void * x){
    while(1){
        char buffer[MAXLINE];
        int len, n;
        len = sizeof(servaddr);
        n = recvfrom(sockfd, buffer, MAXLINE, MSG_WAITALL, (
struct sockaddr *) &servaddr, (socklen_t*)&len);
        srand(time(NULL));
        // if(randomEarlyDetection(n) == 0){
            tcp_packet *recvpkt = (tcp_packet *)buffer;
            pthread_mutex_lock(&bufferMutex);
            enqueue(routerBuffer, recvpkt);
            pthread_mutex_unlock(&bufferMutex);
        // }
    }
}
```

```
void *sendHandler(void *x){
    while(1){
        if(size(routerBuffer) != 0){
            pthread_mutex_lock(&bufferMutex);
            tcp_packet *recvpkt = peek(routerBuffer);
            dequeue(routerBuffer);
            if(random_drop(DROP_PERCENTAGE) == 0 /*&&
size(routerBuffer) < 10*/){////
                int port = recvpkt->hdr.dest;
                struct sockaddr_in clientaddr;
                memset(&clientaddr, 0, sizeof(clientaddr));
                clientaddr.sin_family = AF_INET;
                clientaddr.sin_addr.s_addr = INADDR_ANY;
                clientaddr.sin_port = htons(port);
                int len = sizeof(clientaddr);
                if(recvpkt->hdr.ctr_flags == ACK){
                    sendto(sockfd, recvpkt, TCP_HDR_SIZE,
MSG_CONFIRM, (const struct sockaddr *) &clientaddr, len);
                }
                else{
                    sendto(sockfd, recvpkt, TCP_HDR_SIZE +
get_data_size(recvpkt), MSG_CONFIRM, (const struct sockaddr *)
&clientaddr, len);
                }
            }
            pthread_mutex_unlock(&bufferMutex);
        }
    }
}

int main() {
    routerBuffer = queueCreate();
    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }
    if ( (sockfd2 = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
        perror("socket2 creation failed");
        exit(EXIT_FAILURE);
    }
    memset(&servaddr, 0, sizeof(servaddr));
    // Filling server information
```

```
    servaddr.sin_family = AF_INET; // IPv4
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons(PORT_ROUTER);

    // Bind the socket with the server address
    if (bind(sockfd, (const struct sockaddr *)&servaddr,
sizeof(servaddr)) < 0){
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    pthread_t thread_receiver_A, thread_receiver_B, thread_sender;
    pthread_create(&thread_receiver_A, NULL, sendHandler, NULL);
    pthread_create(&thread_receiver_B, NULL, recieve, NULL);
    pthread_join(thread_receiver_A, NULL);
    pthread_join(thread_receiver_B, NULL);
    return 0;
}
```

Part 1

1)

Go Back N Protocol(GBN) :

- Go-Back-N ARQ is a specific instance of the automatic repeat request (ARQ) protocol, in which the sending process continues to send a number of frames specified by a window size even without receiving an acknowledgement (ACK) packet from the receiver.
- It uses the principle of protocol pipelining in which multiple frames can be sent before receiving the acknowledgment of the first frame. If we have five frames and the concept is Go-Back-3, which means that the three frames can be sent, i.e.,

frame no 1, frame no 2, frame no 3 can be sent before expecting the acknowledgment of frame no 1.

- In Go-Back-N ARQ, the frames are numbered sequentially as Go-Back-N ARQ sends the multiple frames at a time that requires the numbering approach to distinguish the frame from another frame, and these numbers are known as the sequential numbers.
- The number of frames that can be sent at a time totally depends on the size of the sender's window. So, we can say that 'N' is the number of frames that can be sent at a time before receiving the acknowledgment from the receiver.
- If the acknowledgment of a frame is not received within an agreed-upon time period, then all the frames available in the current window will be retransmitted. Suppose we have sent the frame no 5, but we didn't receive the acknowledgment of frame no 5, and the current window is holding three frames, then these three frames will be retransmitted.
- The sequence number of the outbound frames depends upon the size of the sender's window.

Selective Repeat Protocol(SR) :

- Selective repeat protocol, also called Selective Repeat ARQ (Automatic Repeat reQuest), is a data link layer protocol that uses a **sliding window** method for reliable delivery of data frames. Here, only the erroneous or lost frames are retransmitted, while the good frames are received and buffered.

- It provides for sending multiple frames depending upon the availability of frames in the sending window, even if it does not receive acknowledgement for any frame in the interim.
- If the receiver receives a corrupt frame, it does not directly discard it. It sends a negative acknowledgement to the sender. The sender sends that frame again as soon as on the receiving negative acknowledgment. There is no waiting for any time-out to send that frame.
- It is mainly used because we need the receiver to be able to accept packets out-of-order using buffer space, for a superior protocol to combine advantages of both Stop-Wait and GBN.Selective Repeat attempts to retransmit only those packets that are actually lost (due to errors).
- The maximum number of frames that can be sent depends upon the size of the sending window.
- Individual acknowledgements are used in Selective Repeat Protocol

When Buffer Space is of more concern than bandwidth then Go Back N Protocol is used, if Bandwidth is of more concern than buffer space then Selective Repeat Protocol is used.In Go-Back-N we have Less complexity, less CPU cycles while for Selective Repeat Protocol More processing power,cpu cycles at receiver.If error rate is low, use Go-back-N else if error rate is high use Selective Repeat Protocol.

GBN vs SR:

GBN	SR
In Go-Back-N Protocol, if the sent frame are find suspected then all the frames are re-transmitted from the lost packet to the last packet transmitted.	In selective Repeat protocol, only those frames are re-transmitted which are found suspected.
Receiver window size of Go-Back-N Protocol is 1.	Receiver window size of selective Repeat protocol is N.
In Go-Back-N Protocol, neither sender nor at receiver need sorting.	In selective Repeat protocol, receiver side needs sorting to sort the frames.
Go-Back-N Protocol is less complex.	selective Repeat protocol is more complex.
In Go-Back-N Protocol, Out-of-Order packets are NOT Accepted (discarded) and the entire window is re-transmitted.	In selective Repeat protocol, Out-of-Order packets are Accepted.
If error rate is high, it wastes a lot of bandwidth.	Comparatively less bandwidth is wasted in retransmitting.
Receiver do not store the frames received after the damaged frame until the damaged frame is retransmitted.	Receiver stores the frames received after the damaged frame in the buffer until the damaged frame is replaced.

2) In this project, we use GBN protocol for Sliding Window to provide reliable data transfer. The steps are shown below:

- For the first task, we set a fixed cwnd=10.
- For the logic, for every packet that the sender sends, it contains seqno which is the number of the first byte of the data from the data stream. The fixed MSS size is 1.5 KB.
- On the receiver side, we keep track of the expected seqno number of the packet to check if the next packet we receive is in-order to the data stream. Immediately after receiving the packet, we send back an ACK which tells the sender that we have received the packets. Even if our ACK gets lost, higher ACK seq will still confirm that earlier packets have been received. If we receive an out-of-order packet, we discard it and wait for retransmission.
- On the router side, We receive the data from the sender (A) and send it to the receiver (B). We also receive the ACK from the receiver (B) and send it to the sender (A).
- On the sender side, we check the ACK that we receive. If the ACK is larger than our sendbase (meaning former packets received), then we increase our sendbase to the next seqno number right after the last packet received. If we timeout before receiving ACK, we restransmit all the packets starting from the sendbase.

3)

- 200KB file:

```
Sending packet of sequence number 431504 of data size 1448 to 127.0.0.1
Sending packet of sequence number 432952 of data size 1448 to 127.0.0.1
Sending packet of sequence number 434400 of data size 1448 to 127.0.0.1
Sending packet of sequence number 435848 of data size 1448 to 127.0.0.1
Sending packet of sequence number 437296 of data size 1448 to 127.0.0.1
Sending packet of sequence number 438744 of data size 1448 to 127.0.0.1
Sending packet of sequence number 440192 of data size 1448 to 127.0.0.1
Sending packet of sequence number 441640 of data size 1448 to 127.0.0.1
Sending packet of sequence number 443088 of data size 1448 to 127.0.0.1
Sending packet of sequence number 444536 of data size 1448 to 127.0.0.1
Sending packet of sequence number 445984 of data size 1448 to 127.0.0.1
Sending packet of sequence number 447432 of data size 1448 to 127.0.0.1
Sending packet of sequence number 448880 of data size 1448 to 127.0.0.1
Sending packet of sequence number 450328 of data size 1448 to 127.0.0.1
Sending packet of sequence number 451776 of data size 1448 to 127.0.0.1
Sending packet of sequence number 453224 of data size 1448 to 127.0.0.1
Sending packet of sequence number 454672 of data size 1448 to 127.0.0.1
Sending packet of sequence number 456120 of data size 1448 to 127.0.0.1
Sending packet of sequence number 457568 of data size 1448 to 127.0.0.1
Sending packet of sequence number 459016 of data size 1448 to 127.0.0.1
Sending packet of sequence number 460464 of data size 1448 to 127.0.0.1
Sending packet of sequence number 461912 of data size 1448 to 127.0.0.1
Sending packet of sequence number 463360 of data size 1448 to 127.0.0.1
Sending packet of sequence number 464808 of data size 1448 to 127.0.0.1
Sending packet of sequence number 466256 of data size 1448 to 127.0.0.1
Sending packet of sequence number 467704 of data size 1448 to 127.0.0.1
Sending packet of sequence number 469152 of data size 1448 to 127.0.0.1
Sending packet of sequence number 470600 of data size 1448 to 127.0.0.1
Sending packet of sequence number 472048 of data size 1448 to 127.0.0.1
Sending packet of sequence number 473496 of data size 1448 to 127.0.0.1
Sending packet of sequence number 474944 of data size 1448 to 127.0.0.1
Sending packet of sequence number 476392 of data size 1448 to 127.0.0.1
Sending packet of sequence number 477840 of data size 1448 to 127.0.0.1
Sending packet of sequence number 479288 of data size 1448 to 127.0.0.1
Sending packet of sequence number 480736 of data size 1448 to 127.0.0.1
Sending packet of sequence number 482184 of data size 1448 to 127.0.0.1
Sending packet of sequence number 483632 of data size 1448 to 127.0.0.1
Sending packet of sequence number 485080 of data size 1448 to 127.0.0.1
Sending packet of sequence number 486528 of data size 1108 to 127.0.0.1
***All packets sent successfully
------------------------------------------->Time taken: 0.039811 secconds
```

- 1MB file:

```
End Of File read
Sending packet of sequence number 0 of data size 0 to 127.0.0.1
***All packets sent successfully
------------------------------------------->Time taken: 0.537036 secconds
```

4) Size of buffer = 10 & 1MB file

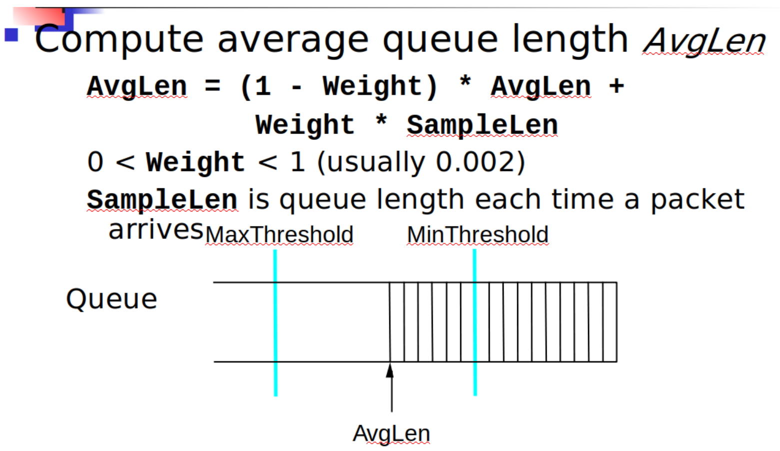
```
End Of File read
Sending packet of sequence number 0 of data size 0 to 127.0.0.1
***All packets sent successfully
--------------------------------------->Time taken: 0.776336 secconds
```

Part 2

1)

Random Early Drop

rather than wait for queue to become full, drop each arriving packet with some drop probability whenever the queue length exceeds some drop level.



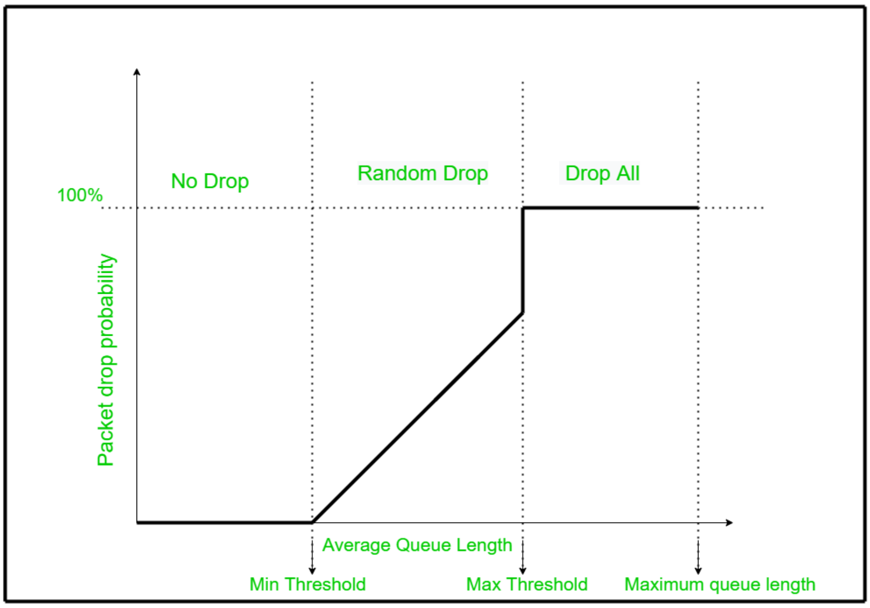
- P of a particular flow's packet(s) is roughly proportional to the share of the flow's bandwidth.
- MaxP is typically 0.02, meaning when is halfway between the two thresholds, router drops roughly one out of 50 packets.

- If traffic is bursty, then MinTh. should be sufficiently large to allow link utilization to be acceptably high.
- Difference between two thresholds should be larger than the typical increase in the calculated average queue length in one RTT; setting MaxThreshold to twice MinThreshold is reasonable.

Here is the algorithm of RED:

Two queue length thresholds

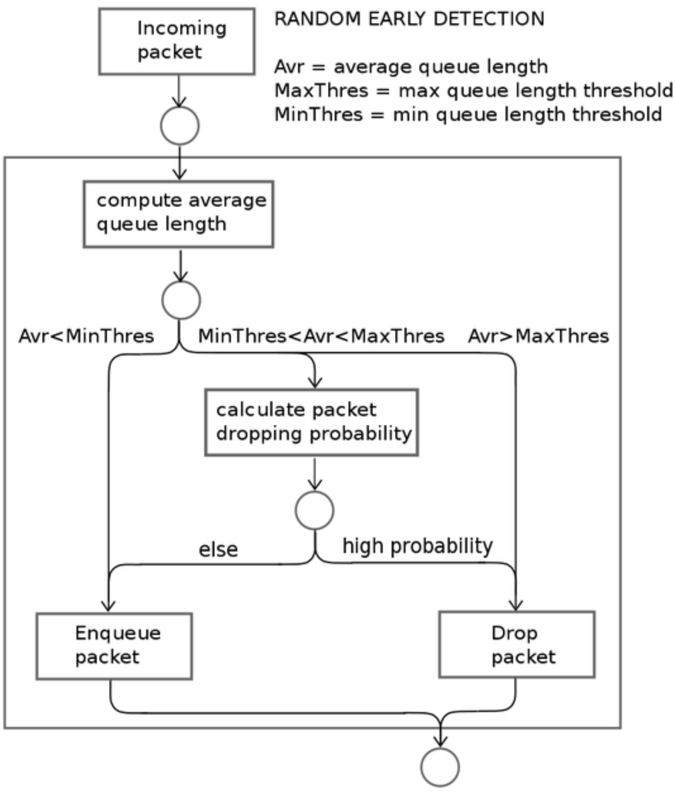
```
if AvgLen <= MinThreshold then
   enqueue the packet
if MinThreshold < AvgLen < MaxThreshold
then
   calculate probability P
   drop arriving packet with probability P
if ManThreshold <= AvgLen then
   drop arriving packet
```



RED monitors the average queue size and drops packets based on statistical probabilities. If the buffer is almost empty then all incoming packets are accepted. As the queue grows, the probability for dropping an incoming packet grows too. When the buffer is full, the probability has reached 1 and all incoming packets are dropped.

RED is more fair than tail drop, in the sense that it does not possess a bias against bursty traffic that uses only a small portion of the bandwidth. The more a host transmits, the more likely it is that its packets are dropped as the probability of a host's packet being dropped is proportional to the amount of data it has in a queue. Early detection helps avoid TCP global synchronization.

To sum up:




```
int randomEarlyDetection(int s) {
   int res = 0;
   avg = ((1 - weight) * avg) + (weight * size(routerBuffer));

   if(minThreshold <= avg && avg < maxThreshold) {
       count++;
```

```
        tempP = ((avg - minThreshold) * maxp)/(maxThreshold - 
minThreshold);
        double P = tempP/(1 - (count * tempP));
        if(count == 50) {
            P = 1.0;
            res = -1;
        }
        double randomP = (rand()%100)/100.00;
        if(randomP <= P) {
            if(count != 50) {
                res = -1;
            }
            count = 0;
        } else {
            res = 0;
            count = -1;
        }
    } else if(maxThreshold <= avg) {
        res = -1;
        count = 0;
    } else {
        res = 0;
        count = -1;
    }
    return res;
}
```

Assumptions:

```
double maxp = 0.02;
double tempP = 0;
double avg = 0;
int count = -1;
double weight = 0.003;
int minThreshold = 5, maxThreshold = 20;
```

2)

With 3 computers:

```
Sending packet of sequence number 480736 of data size 1448 to 127.0.0.1
Sending packet of sequence number 482184 of data size 1448 to 127.0.0.1
Sending packet of sequence number 483632 of data size 1448 to 127.0.0.1
Sending packet of sequence number 485080 of data size 1448 to 127.0.0.1
Sending packet of sequence number 486528 of data size 1108 to 127.0.0.1
***All packets sent successfully
------------------------------------------->Time taken: 0.066818 secconds
```

```
Sending packet of sequence number 483632 of data size 1448 to 127.0.0.1
Sending packet of sequence number 485080 of data size 1448 to 127.0.0.1
Sending packet of sequence number 486528 of data size 1108 to 127.0.0.1
***All packets sent successfully
------------------------------------------->Time taken: 0.041061 secconds
```

```
***All packets sent successfully
------------------------------------------->Time taken: 0.029933 secconds
```

Total time: 0.507361 s

References:

https://www.icir.org/floyd/papers/early.twocolumn.pdf

https://techdifferences.com/difference-between-go-back-n-and-selective-repeat-protocol.html

https://www.geeksforgeeks.org/difference-between-go-back-n-and-selective-repeat-protocol/