IMPLEMENTING A RELIABLE UDP

Nabeel Hussain Syed (122937)

Contents

1. INTRODUCTION:

The main purpose of this project was to implement a reliable UDP in which are there are no packet losses or unordered packets. Our knowledge of Transport Layer Protocol was used in order to implement a UDP which performs kind of the same as that of TCP. To test whether our code works perfectly or not, we sent an audio file of 3MB from one sender to PC to another receiver PC. There were no packet losses or unordered packets and the file was received successfully. We could play the audio in the receiver PC which was sent from the sender PC.

So in order to make all this happen and implement a reliable UDP data transfer, following are the features that we implemented in UDP.

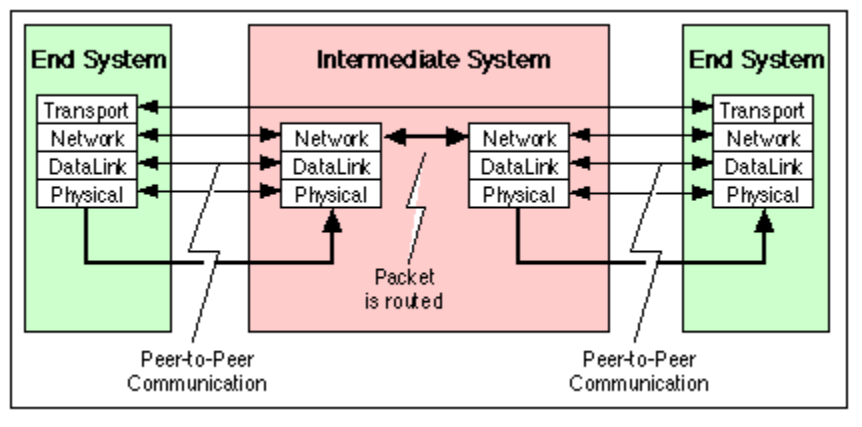
(a) Introduction of the sequence number for each packet that is being sent.
(b) Usage of Stop and Wait Protocol.
(c) Sending ACKS from the receiver side.
(d) Timer to detect packet loss.
(e) Sending pipelined 5 packets i.e the window size.
(f) Retransmitting the lost packets.
(g) Reordering the packets.

All the code was written in C in Linux and the file transferred from receiver PC to sender PC, both the PCs were running Linux.

2. TRANSPORT LAYER PROTOCOL:

The transport layer is the fourth layer of the OSI reference model. It provides transparent transfer of data between end systems using the services of the network layer (e.g. IP) below to move PDUs of data between the two communicating systems.

The transport service is said to perform "peer to peer" communication, with the remote (peer) transport entity. The data communicated by the transport layer is encapsulated in a transport layer PDU and sent in a network layer SDU. The network layer nodes (i.e. Intermediate Systems (IS)) transfer the transport PDU intact, without decoding or modifying the content of the PDU. In this way, only the peer transport entities actually communicate using the PDUs of the transport protocol.



2.1 User Datagram Protocol (UDP):

UDP is an unreliable and without connection protocol that is used to send only small amounts of data. When the data is larger, packets are lost in UDP.

2.2 Transmission Control Protocol (TCP):

TCP is connection oriented – once a connection is established, data can be sent bidirectional. TCP takes care of packet loss as it retransmits the data if the packets are lost and it is also responsible for re-ordering the data.

2.3 Differences between UDP and TCP:

TCP	UDP
Keeps track of lost packets. Makes sure that lost packets are re-sent	Doesn't keep track of lost packets
Adds sequence numbers to packets and reorders any packets that arrive in the wrong order	Doesn't care about packet arrival order
Slower, because of all added additional functionality	Faster, because it lacks any extra features
Requires more computer resources, because the OS needs to keep track of ongoing communication sessions and manage them on a much deeper level	Requires less computer resources
Examples of programs and services that use TCP: - HTTP - HTTPS - FTP - Many computer games	Examples of programs and services that use UDP: - DNS - IP telephony - DHCP - Many computer games

3. FUNCTIONALITY:

To modify the UDP in such a way that it also behaves like TCP or sends the data reliably, we had to change a lot of functionality in order to make sure that the UDP is sending data reliably just like TCP. But for this work, both the sender and the receiver codes are to be efficiently changed so that no packet loss or unordered packets are received.

3.1 Sequence Numbers:

Each packet was identified with a sequence number. While the packets are being sent to the receiver, each and every packet is given a sequence number from 0-9. So when the receiver receives the packet, it knows which packet it is and whether the receiver has received this packet already or not and in response it sends an acknowledgement to the sender that it has received by checking the sequence number. If the sender does not receive the acknowledgement number which was sent by the receiver, a packet loss may have occurred and sender can know which packet has been lost and sender can resend that packet.

```
//RUN THIS LOOP FOR TOTAL WINDOW SIZE
for (i = 0; i < Window_Size; i++){
    //SEQ # FOR NEXT PACKET TO BE SENT
    seq = L_Ack_R+i+1;
```

3.2 Acknowledgments (ACKS):

When the sender sends 5 packets in a pipeline that is at once, when the receiver receives the packet, the receiver can let the sender know that it has received the specific packet by sending the acknowledgment of that packet back to the sender, which lets the sender know that the receiver has received the packet successfully and the sender can then move on to the next packet.

```
//THIS RUNS FOR RECEIVING DATA PACKETS AND SENDING THE ACKS BACK TO SENDER
while(1){
    PACKET_BYTES = recvfrom(sockfd, buffer, PACKETSIZE, 0, (struct sockaddr *)&sender_addr, &their_addr_len);
    //IF BYTES RECIEVED ARE LESS THAN HEADER SIZE MEANS PACKET IS INVALID
    if (PACKET_BYTES < HEADER_SIZE)
        continue;
    //COPYING HEADER INFORMATION FROM RECEIVED BUFFER INTO PACKET HEADER
    memcpy(&Pkt_Header, buffer, HEADER_SIZE);
    //COPYING CODE OF PACKET INTO CODE
    CODE = Pkt_Header.CODE;

    //CHECK FOR FIN PACKET TO BREAK THE LOOP
    if (CODE == FIN){
        break;
    }

    //CHECK IF PACKET TYPE IS NOT DATA TOO THEN JUST SKIP ITERATION
    if (CODE != DATA){
        continue;
    }

    //COPYING SEQ# OF PACKET FROM PACKET HEADER
    seq = Pkt_Header.seq_no;
    //CALCULATING LENGTH OF DATA PACKET
    length = PACKET_BYTES - HEADER_SIZE;

    //CHECK IF PACKET IS RECEIVED FIRST TIME
    if(RECEIVED_PACKETS[seq] == 0){
        printf("Packet(%d) is received!\n", seq);
        printf("*****************************\n\n");
        RECEIVED_PACKETS[seq] = 1;
        //CHECKING FOR CORRECT POSITION OF CURSOR
        if (SEEK_CUR != seq * DATA_SIZE){
            //MOVING CURSOR TO CORRECT LOCATION
            //Here the receiver is reordering the packets
            fseek(RECEIVED_FILE, seq * DATA_SIZE, SEEK_SET);
        }
        //WRITING RECEIVED DATA INTO FILE FROM RECEIVED BUFFER (AFTER HEADER INFORMATION)
        fwrite(buffer + HEADER_SIZE, 1, length, RECEIVED_FILE);
        while(RECEIVED_PACKETS[NFE]){
            NFE++;
        }
```

3.3 Stop and Wait Protocol:

In stop and wait protocol, the sender first sends 5 pipelined packets to the receiver and then wait for all the acknowledgements of those packets from the receiver. For every packet there is a timer that is set, if the timer expires and the sender has not yet received the acknowledgement of that packet, then the sender retransmits the packet to the receiver. Then the sender waits again for the acknowledgements and once the sender receives all the acknowledgements of the 5 packets, then the sender moves on to the next 5 packets.

```
//RUNS FOR RECEIVING ACKS
for (i = 0; i < Window_Size; i++){
    //CHECK TO SKIP ITERATION IF ACKS ARE EQUAL TO TOTAL SEGMENTS
    //IF NOT THEN STOP AND WAIT FOR RECEIVING ACKS
    if(ACKS_COUNTER(ACKED_SEGMENTS,File_Segments) != File_Segments){
        bytes_in_packet = recvfrom(sockfd, buffer, PACKETSIZE, 0, &receiver_addr, &their_addr_len);
        if(bytes_in_packet == Header_size){
            //COPYING HEADER OF RECEIVED BUFFER INTO RECEIVER's HEADER
            memcpy(&receiver_Packet_Header, buffer, Header_size);
            //CHECKING IF RECEIVED PACKET HEADER HAS ACK IN ITS CODE.
            if (receiver_Packet_Header.CODE == ACK){
                printf("Acknowledgment ACK(%d) is received!\n", receiver_Packet_Header.seq_no);
                //SETTING THIS PACKET AS ACKED IN ACKED SEGMENTS ARRAY
                ACKED_SEGMENTS[receiver_Packet_Header.seq_no] = 1;
            }
        }
        else {
            break;
        }
    }
}
printf("*****************************\n");
```

3.4 Timer:

A timer is implemented to detect the packet loss. The timer starts when the sender sends a packet and then the sender waits for the timer to expire so that it can resend the packet. There are actually 2 types of things that can happen with timer which are as follows:

1) When the timer expires before the packet is received at the receiver side.

2) When the packet is received at the receiver side before the timer was expired.

In the first case, the sender then assumes that the packet is lost and retransmits the packet whereas in the second case, if the packet is received before time expires, then the client resets the timer and sends the next packet.

```c
//handler for Time_Out
void Handle_TimeOut(int signum){
    printf("Timeout has occured!\n"); //Printing on screen for time out
}

//setting Time_Out timer
void TIMER_STARTS(){
    printf("Timer has started!\n");
    struct itimerval timeout_timer;

    //expire timer after 3 seconds
    timeout_timer.it_value.tv_sec = 3;
    timeout_timer.it_value.tv_usec = 0;

    timeout_timer.it_interval.tv_sec = 0;
    timeout_timer.it_interval.tv_usec = 0;

    //STARTING THE TIMER
    setitimer (ITIMER_REAL, &timeout_timer, NULL);
}
```

3.5 Re-ordering of the packets at the receiver side:

We have used selective repeat protocol to make the reliable UDP and hence our receiving window size is equal to the sender window size. When the receiver is receiving the packets and if it receives an out of order packet then it doesn't discard it, rather it keeps the out of ordered packets in buffer that is the window which is the same size as that of the sender window and then it delivers the packets to the upper layer when it receives all the packets and all the packets are then reordered in the buffer.

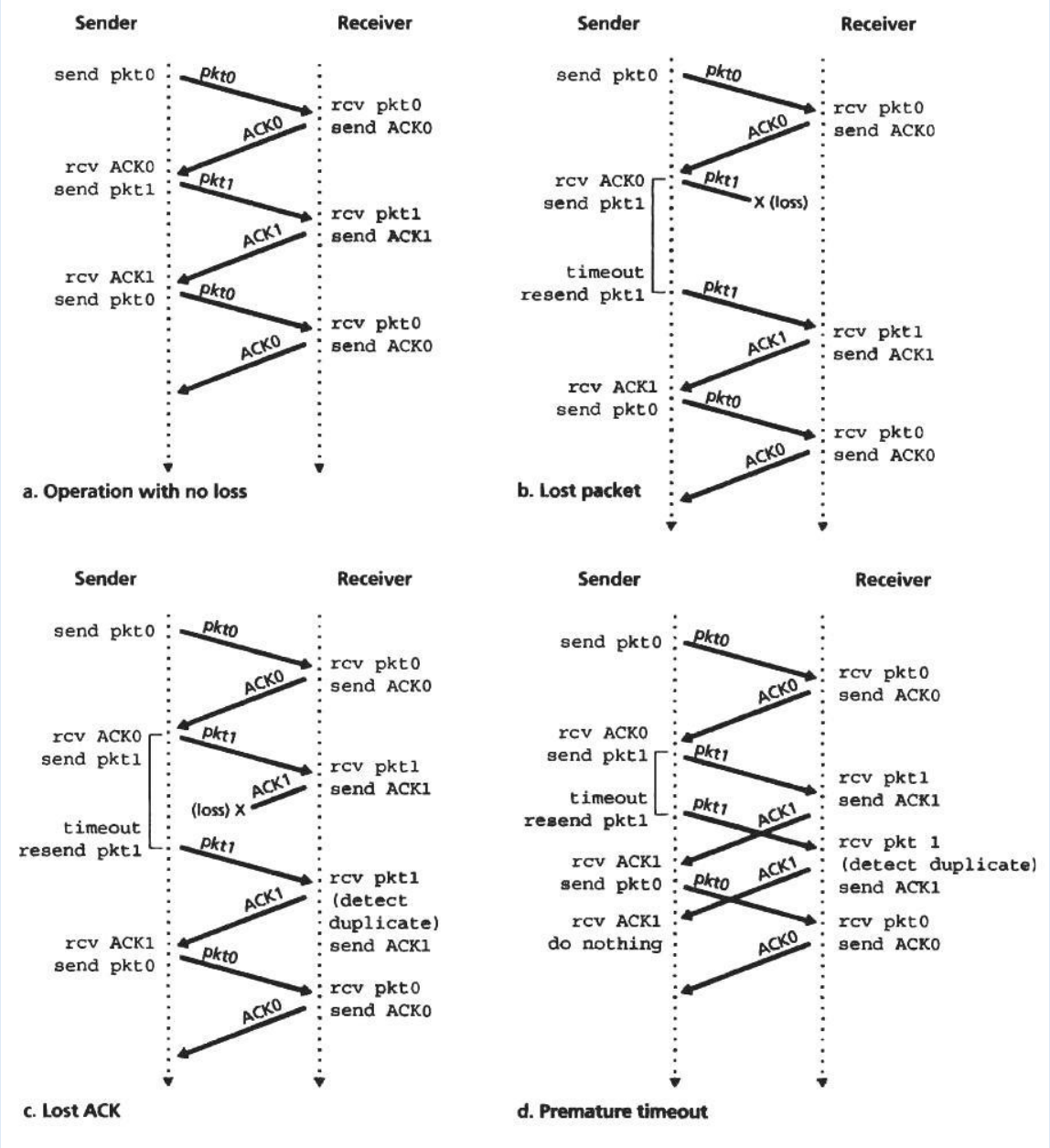
```
//CHECK IF PACKET IS RECEIVED FIRST TIME
if(RECEIVED_PACKETS[seq] == 0){
    printf("Packet(%d) is received!\n", seq);
    printf("*****************************\n\n");
    RECEIVED_PACKETS[seq] = 1;
    //CHECKING FOR CORRECT POSITION OF CURSOR
    if (SEEK_CUR != seq * DATA_SIZE){
        //MOVING CURSOR TO CORRECT LOCATION
        fseek(RECEIVED_FILE, seq * DATA_SIZE, SEEK_SET);
    }
    //WRITING RECEIVED DATA INTO FILE FROM RECEIVED BUFFER (AFTER HEADER INFORMATION)
    fwrite(buffer + HEADER_SIZE, 1, length, RECEIVED_FILE);
    while(RECEIVED_PACKETS[NFE]){
        NFE++;
    }
//ELSE IF PACKET IS RECEIVED NO NEED TO REWRITE IT
} else{
    printf("Duplicate Packet(%d) is received!\n", seq);
    printf("*****************************\n\n");
}
```

3.6 Re-transmission of the lost packets:

When the sender sends a packet and starts the timer, if the timer expires before the sender receives the acknowledgement of that packet then the sender is able to identify that a packet loss has occurred and the sender retransmits the same packet by checking the sequence number of the packet for which the ACK was not yet received.

```
memset(&NOT_RECEIVED_ACKS, 0, sizeof(NOT_RECEIVED_ACKS));
//VARIABLE TO COUNT NOT RECEIVED ACKS
int count = 0;
for (i=L_Ack_R+1; i < L_Ack_R+Window_Size; i++){
    //BREAKING LOOP IF ALL SEGMENTS ARE TRANSFERRED
    if(i >= File_Segments){
        break;
    } else{
        //CHECKING IF CURRENT SEGMENT IS ACKED
        if (!ACKED_SEGMENTS[i]){
            //STORING SEQ# OF NOT RECEIVED ACKS
            NOT_RECEIVED_ACKS[count] = i;
            count++;
            printf("Packet(%d) is lost!\n", i);

            //SETTING THAT SEQUENCE NUMBER IN SENDER's PACKET HEADER
            sender_Packet_Header.seq_no = i;
            //PUTTING THAT PACKET's HEADER INTO BUFFER TO SEND
            memcpy(buffer, &sender_Packet_Header, Header_size);
            //SETTING CURSOR TO LOCATION IN BFFER WHERE DATA NEEDS TO PUT
            if (SEEK_CUR != i * DATA_SIZE)
                //MOVING CURSOR TO THAT LOCATION
                fseek(out_file, DATA_SIZE * i, SEEK_SET);
            //CHECK FOR LAST SEHMENT OF REMAINING BYTES
            if (i == File_Segments-1 && final_seg_size > 0){
                //SETTING ITS SIZE TO PACKET LENGTH
                length = final_seg_size;
            }
            //ELSE SET THE LENGTH TO COMPLETE DATA LENGTH
            else{
                length = DATA_SIZE;
            }
            //PUTTING DATA FROM FILE INTO BUFFER AFTER THE HEADER
            fread(buffer + Header_size, 1, length, out_file);
            //SENDING PACKET
            if (bytes_in_packet = sendto(sockfd, buffer, length + Header_size, 0,
                (struct sockaddr*)&sendto_addr, sizeof(sendto_addr)) == -1)
            {
                perror("sender: sendto");
                exit(1);
            }
            //SETTING THAT PACKET AS SENT IN SENT SEGMENT ARRAY
```



4. DESIGN AND DEVELOPMENT:

Before designing this reliable data transfer in UDP, we had to go through all the functions that are implemented in TCP so that we should know how TCP implements reliability transfer and we can apply those functions in UDP.

4.1 Design:

Following are the features that are implemented in our design:

1) Sequence numbers are implemented from 0-9.

2) Acknowledgement numbers are implemented on the receiver side.

3) Window size is made which is of length 5.

4) Data is of 500 bytes for a single window.

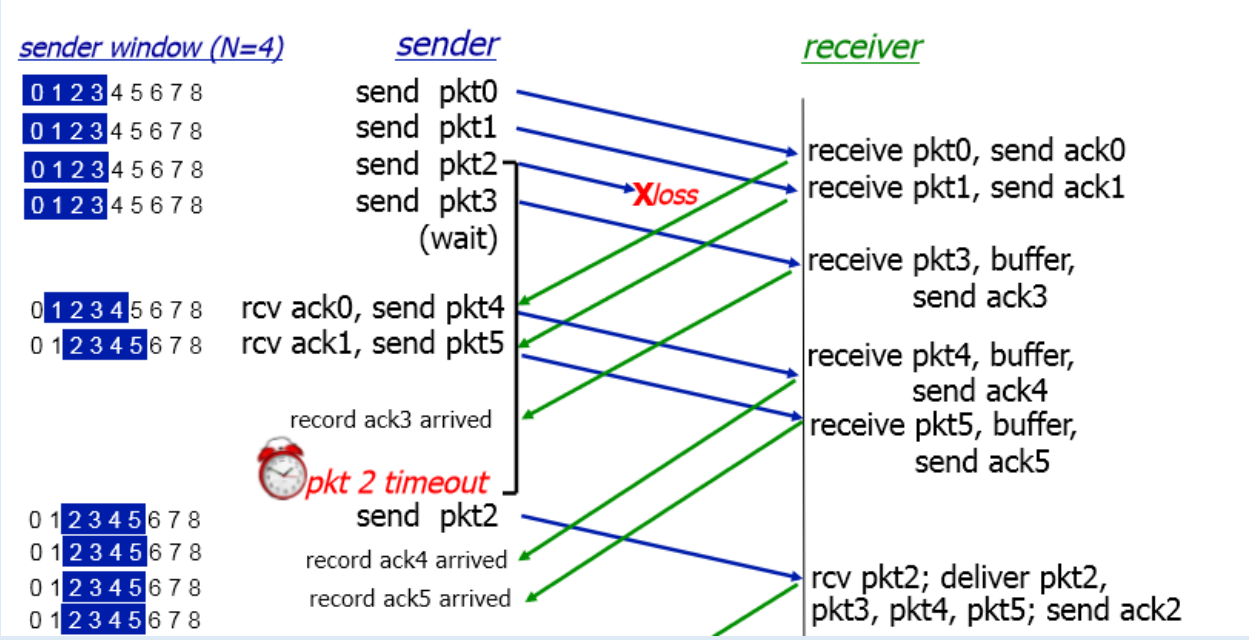
5) Selective repeat protocol is used.

6) Packets are re-ordered on the receiver side.

7) Timer expires after 3 seconds.

Selective Repeat Protocol:

```
//RUNNING LOOP FOR ALL NOT RECEIVED ACKS
for (i=0; NOT_RECEIVED_ACKS[i]!=0; i++){
    //WAITING FOR ACKS
    bytes_in_packet = recvfrom(sockfd, buffer, PACKETSIZE, 0, &receiver_addr, &their_addr_len);
    if(bytes_in_packet == Header_size){
        //COPYING HEADER INFO FROM RECEIVED BUFFER INTO RECEIVER'S PACKET HEADER
        memcpy(&receiver_Packet_Header, buffer, Header_size);
        //CHECKING IF ACK IS RECEIVED
        if (receiver_Packet_Header.CODE == ACK){
            printf("Acknowledgment ACK(%d) is received!\n", receiver_Packet_Header.seq_no);
            ACKED_SEGMENTS[receiver_Packet_Header.seq_no] = 1;
        }
    }
    //CHECKING FOR NOT RECEIVED ACKS
    else if (NOT_RECEIVED_ACKS[i] != 0){
        //SETTING SEQ# OF MISSSED ACKS
        sender_Packet_Header.seq_no = NOT_RECEIVED_ACKS[i];
        //PUTTING SENDER'S PACKET HEADER INTO BUFFER
        memcpy(buffer, &sender_Packet_Header, Header_size);
        //SETTING THE POSITION OF THE CURSOR
        if (SEEK_CUR != NOT_RECEIVED_ACKS[i] * DATA_SIZE)
            //MOVING CURSOR TO THAT LOCATION
            fseek(out_file, DATA_SIZE * NOT_RECEIVED_ACKS[i], SEEK_SET);
        //CHECK FOR LAST SEGMENT OF REMAINING BYTES
        if (NOT_RECEIVED_ACKS[i] == File_Segments-1 && final_seg_size > 0){
            //SETTING THE SIZE OF LAST PACKET
            length = final_seg_size;
        }
        //ELSE SET IT TO COMPLETE DATA LENGTH
        else{
            length = DATA_SIZE;
        }
        //PUTTING DATA INTO BUFFER FROM FILE AFTER THE HEADER INFORMATION
        fread(buffer + Header_size, 1, length, out_file);
        //SENDING PACKET
        if (bytes_in_packet = sendto(sockfd, buffer, length + Header_size, 0,
            (struct sockaddr*)&sendto_addr, sizeof(sendto_addr)) == -1)
        {
            perror("sender: error sending the packet");
            exit(1);
        }
        printf("Packet(%d) is retransmitted!\n", NOT_RECEIVED_ACKS[i]);
        //DECREASING COUNTER TO RECEIVE ACK OF THIS PACKET AGAIN
        i--;
        //RESETTING TIMER
        TIMER_STARTS();
        //CONTINUE THE LOOP
        continue;
    } else{
        break;
    }
```



4.2 Development and Behavior:

After our implementation of the code was complete, we were able to send an audio file which was of size 3MB and it was received at the receiver side without any packets loss. The behavior was significantly improved for UDP as it was able to transfer data reliably without any packets loss or unordered packets. Sender's behavior was changed by bringing in the timer to detect the packet loss and also the stop and wait protocol in which the sender sends a window of 5 pipelined packets and then wait for the acknowledgement numbers of those packets from the receiver but this makes our reliable UDP slower than the actual UDP but also we're able to detect packet loss and retransmit. Receiver's behavior was changed as the acknowledgement numbers were introduced with the buffer window so that the packets are re-ordered and the buffer window is of the same size as that of the sender window which helps the receiver to re-order any packets that were sent out of order from the sender.

5. Code:

5.1 Sender:

```
#include <signal.h>

#include <sys/time.h>

#include <netdb.h>

#include <arpa/inet.h>

#include <netinet/in.h>
```

```
#include <sys/socket.h>
#include <sys/types.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

#define PACKETSIZE 500  //Defining packet size
#define DATA 0
#define ACK 1
#define FIN 2
#define FIN_ACK 3
int TOTAL_ACKS = 0;
int TOTAL_SEGMENTS = 0;
int WINDOW_SIZE = 0;
int FILE_SIZE = 0;

//PROTOTYPES FOR ALL THE FUNCTIONS
//TIMEOUT HANDLER
void Handle_TimeOut(int);
//SETTER FUNCTION FOR TIMEOUT
void TIMER_STARTS();
//COUNTER FOR THE TOTAL ACKS RECEIVED
int ACKS_COUNTER(int[] , int);
//Fuction for transferring data reliablly using UDP
//FUNCTION FOR TRANSFERRING THE DATA RELIABLY OVER UDP.
int RELIABLE_UDP(char* , char * , char* );

int64_t Time_Out = 1;
int L_Ack_R = -1;
const int Window_Size = 5;

//Packet_Header
typedef struct {
        uint64_t sent_time;    //initializing sent time
```

```
        uint16_t seq_no;   //initializing sequence number
        uint16_t CODE;   //initializing content type
}Packet_Header;

//MAIN FUNCTION
int main(int argc, char** argv){
        //FOUR INPUTS ARE REQUIRED TO BE ENTERED BY USER IN COMMAND LINE
        if(argc != 4){
                fprintf(stderr, "Please type in terminal: %s hostname port filename_to_transfer\n\n", argv[0]);
                exit(1);
        }
        //CALLING FUNCTION FOR RELIABLE DATA TRANSFER
        RELIABLE_UDP(argv[1], argv[2], argv[3]);

        printf("\n\n********************SUMMARY OF THE RELIABLE UDP********************\n");
                        puts("\t\t\t      _____");
                        puts("\t\t\t     |TABLE|");
                puts("|----------------|----------------|-------------|------------------|");
                puts("|   Total ACKS   | Total Segments | Window Size |     File Size    |");
                puts("|----------------|----------------|-------------|------------------|");
        printf("| %8d      |     %2d      |     %3d     |     %2d      |\n"
       , (TOTAL_ACKS), TOTAL_SEGMENTS, WINDOW_SIZE, FILE_SIZE);
        puts("-------------------------------------------------------------------------");
}

//handler for Time_Out
void Handle_TimeOut(int signum){
        printf("Timeout has occured!\n"); //Printing on screen for time out
}

//setting Time_Out timer
void TIMER_STARTS(){
        printf("Timer has started!\n");
        struct itimerval timeout_timer;

        //expire timer after 3 seconds
```

```
		timeout_timer.it_value.tv_sec = 3;
		timeout_timer.it_value.tv_usec = 0;

		timeout_timer.it_interval.tv_sec = 0;
		timeout_timer.it_interval.tv_usec = 0;

		//STARTING THE TIMER
		setitimer (ITIMER_REAL, &timeout_timer, NULL);
}

//COUNTER FOR TOTAL ACKS THAT ARE RECEIVED
int ACKS_COUNTER(int ACKED_SEGMENTS[], int size){
		int acks_count = 0,a;
		for ( a = 0; a < size; a++){
			if(ACKED_SEGMENTS[a]==1){
				acks_count++;
			}
		}
		TOTAL_ACKS = acks_count;
		return acks_count;
}

//FUNCTION FOR TRANSFERRING DATA RELIABLY OVER UDP
int RELIABLE_UDP(char* hostname, char * udpPort, char* filename){
		//VARIABLE TO GIVE THE SIGNAL TO HANDLE THE TIMEOUT
		struct sigaction signal_action;
		memset (&signal_action, 0, sizeof (signal_action));
		//SETTING THE HANDLE_TIMEOUT FUNCTION AS SIGNAL ACTION HANDLER
		signal_action.sa_handler = &Handle_TimeOut;
		sigaction (SIGALRM, &signal_action, NULL);

		int sockfd, rv, bytes_in_packet, i;

		struct addrinfo myinfo;
		struct addrinfo *server_info;
		struct addrinfo *selected_info;
```

```
struct sockaddr_in bind_addr, sendto_addr;
struct sockaddr receiver_addr;
socklen_t their_addr_len = sizeof(receiver_addr);
char buffer[PACKETSIZE];

//SETTING UP THE SENDTO ADDRESS
uint16_t sendto_port = (uint16_t)atoi(udpPort);
memset(&sendto_addr, 0, sizeof(sendto_addr));

sendto_addr.sin_family = AF_INET;
sendto_addr.sin_port = htons(sendto_port);
inet_pton(AF_INET, hostname, &sendto_addr.sin_addr);

memset(&myinfo, 0, sizeof myinfo);
myinfo.ai_family = AF_INET;
myinfo.ai_socktype = SOCK_DGRAM;

if ((rv = getaddrinfo(hostname, udpPort, &myinfo, &server_info)) != 0) {
        printf("error: %s\n", gai_strerror(rv));
        return 1;
}

for(selected_info = server_info; selected_info != NULL; selected_info = selected_info->ai_next) {
        if ((sockfd = socket(selected_info->ai_family, selected_info->ai_socktype, selected_info->ai_protocol)) == -1)
{
                perror("sender: socket error");
                continue;
        }
        break;
}

if (selected_info == NULL) {
        fprintf(stderr, "sender: socket creation failed\n");
        return 2;
}
```

```
//PORT NUMBER EXTRACTION
uint16_t my_port = ((struct sockaddr_in *)selected_info->ai_addr )->sin_port;

freeaddrinfo(server_info);

memset(&bind_addr, 0, sizeof(bind_addr));
bind_addr.sin_family = AF_INET;
//PORT NUMBER SETTING
bind_addr.sin_port = htons(my_port);
bind_addr.sin_addr.s_addr = htonl(INADDR_ANY);

//BINDING THE SOCKET
if (bind(sockfd, (struct sockaddr *)&bind_addr, sizeof(struct sockaddr_in)) == -1)
{
        close(sockfd);
        perror("sender: binding failed");
        exit(1);
}

//INITIALIZING PACKET HEADERS
Packet_Header sender_Packet_Header, receiver_Packet_Header;
//STORE SENDER's PACKET HEADER SIZE IN Header_size
int Header_size = (int) sizeof(sender_Packet_Header);

FILE* out_file = fopen(filename, "rb");

fseek(out_file, 0, SEEK_END);
//FILE SIZE STORING IN file_size
int file_size = ftell(out_file);
fseek(out_file, 0, SEEK_SET);

//PACKETSIZE SUBTRACT HEADERSIZE GIVES DATASIZE
int DATA_SIZE = PACKETSIZE - Header_size;
//TOTAL SEGMENTS IN WHICH FILE CAN DEVIDED
int File_Segments = file_size/DATA_SIZE;
//SAVING REMAINING BYTES THAT ARE NOT DEVISIBLE IN SEGMENTS
```

```
int final_seg_size = file_size % DATA_SIZE;

//IF REAMAINIG BYTES EXIST MAKING A NEW SEGMENT FOR THEM
if (final_seg_size > 0)
        File_Segments++;

FILE_SIZE = file_size;
WINDOW_SIZE = Window_Size;

//FOR FIRST PACKET SEQ_NO IS 0
sender_Packet_Header.seq_no = 0;
//SETTING CODE TO DATA FOR SENDER PACKET HEADER
sender_Packet_Header.CODE = DATA;
//VARIABLE FOR COUNTING RETRANSMITTED PACKETS
        int double_sent = 0;

printf("Sending the file to the receiver...\n");
printf("-----------------------------------\n");

uint16_t seq, length;

//ARRAYS FOR ACKS AND SENT TO SAVE SEGMENT STATE
int *ACKED_SEGMENTS = (int *)malloc(sizeof(int)*File_Segments);
int *SENT_SEGMENTS = (int *)malloc(sizeof(int)*File_Segments);

//INITIALIZING ARRAYS WITH ZERO
for (i = 0; i < sizeof(ACKED_SEGMENTS); i++){
        ACKED_SEGMENTS[i] = 0;
        SENT_SEGMENTS[i] = 0;
}

//VARIABLE TO COUNT ACKS
int ACK_COUNT;

//LOOP FOR RECEIVING ALL ACKS
while (ACK_COUNT != File_Segments){
```

```
ACK_COUNT = 0;

//RUN THIS LOOP FOR TOTAL WINDOW SIZE
for (i = 0; i < Window_Size; i++){
    //SEQ # FOR NEXT PACKET TO BE SENT
    seq = L_Ack_R+i+1;
    //CHECKING IF ACK IS NOT RECEIVED
    if (ACKED_SEGMENTS[seq] == 0 && seq < File_Segments){
        //CHECKING FOR RETRANSMISSION
        if (SENT_SEGMENTS[seq] == 1){
            double_sent++;          //INCREMENT FOR RETRANSMITTED
PACKETS
        //IF PACKET IS TRANSMITTING FIRST TIME
        } else{
            //SET THAT PACKET AS 1 IN SENT SEGMENT ARRAY
            SENT_SEGMENTS[seq] = 1;
            //SETTING SEQ # FOR PACKET HEADER
            sender_Packet_Header.seq_no = seq;
            //PUTTING PACKET HEADER INTO BUFFER TO BE SENT
            memcpy(buffer, &sender_Packet_Header, Header_size);
            //SETTING CURSOR LOCATION TO PUT DATA INTO BUFFER
            if (SEEK_CUR != seq * DATA_SIZE)
                //MOVE CURSOR TO CORRECT LOCATION
                fseek(out_file, DATA_SIZE * seq, SEEK_SET);
            //CHECK FOR LAST SEGMENT OF REMAINING BYTES
            if (seq == File_Segments-1 && final_seg_size > 0){
                //SETTING LENGTH OF PACKET TO LAST SEGMENT SIZE
                length = final_seg_size;
            }
            //SETTING PACKKET LENGTH TO COMPLETE DATA LENGTH
            else{
                length = DATA_SIZE;
            }
            //PUTTING DATA FROM FILE INTO BUFFER AFTER HEADER
            fread(buffer + Header_size, 1, length, out_file);
            //SENDING PACKET TO RECEIVER
            if (bytes_in_packet = sendto(sockfd, buffer, length + Header_size, 0,
```

```
                                        (struct sockaddr*)&sendto_addr, sizeof(sendto_addr)) == -1)
                                    {
                                        perror("sender: error sending data packet");
                                        exit(1);
                                    }

                                    printf("Packet(%d) is sent!\n", seq);
                                }
                            }
                        }
                        printf("********************************\n");
                        //START TIMER FOR TIME OUT
                        TIMER_STARTS();

                        //RUNS FOR RECEIVING ACKS
                        for (i = 0; i < Window_Size; i++){
                            //CHECK TO SKIP ITERATION IF ACKS ARE EQUAL TO TOTAL SEGMENTS
                            //IF NOT THEN STOP AND WAIT FOR RECEIVING ACKS
                            if(ACKS_COUNTER(ACKED_SEGMENTS,File_Segments) != File_Segments){
                                bytes_in_packet = recvfrom(sockfd, buffer, PACKETSIZE, 0, &receiver_addr,
&their_addr_len);
                                if(bytes_in_packet == Header_size){
                                    //COPYING HEADER OF RECEIVED BUFFER INTO RECEIVER's HEADER
                                    memcpy(&receiver_Packet_Header, buffer, Header_size);
                                    //CHECKING IF RECEIVED PACKET HEADER HAS ACK IN ITS CODE.
                                    if (receiver_Packet_Header.CODE == ACK){
                                        printf("Acknowledgment ACK(%d) is received!\n",
receiver_Packet_Header.seq_no);
                                        //SETTING THIS PACKET AS ACKED IN ACKED SEGMENTS
ARRAY
                                        ACKED_SEGMENTS[receiver_Packet_Header.seq_no] = 1;
                                    }
                                }
                                else {
                                    break;
                                }
                            }
```

```
}
printf("*******************************\n");

//ARRAY FOR SAVING MISSED ACKS
int NOT_RECEIVED_ACKS[Window_Size];
//INITIALIZING ALL MISSED ACK WITH ZERO
memset(&NOT_RECEIVED_ACKS, 0, sizeof(NOT_RECEIVED_ACKS));
//VARIABLE TO COUNT NOT RECEIVED ACKS
int count = 0;
for (i=L_Ack_R+1; i < L_Ack_R+Window_Size; i++){
        //BREAKING LOOP IF ALL SEGMENTS ARE TRANSFERRED
        if(i >= File_Segments){
                break;
        } else{
                //CHECKING IF CURRENT SEGMENT IS ACKED
                if (!ACKED_SEGMENTS[i]){
                        //STORING SEQ# OF NOT RECEIVED ACKS
                        NOT_RECEIVED_ACKS[count] = i;
                        count++;
                        printf("Packet(%d) is lost!\n", i);

                        //SETTING THAT SEQUENCE NUMBER IN SENDER's PACKET HEADER
                        sender_Packet_Header.seq_no = i;
                        //PUTTING THAT PACKET's HEADER INTO BUFFER TO SEND
                        memcpy(buffer, &sender_Packet_Header, Header_size);
                        //SETTING CURSOR TO LOCATION IN BFFER WHERE DATA NEEDS TO
PUT
                        if (SEEK_CUR != i * DATA_SIZE)
                                //MOVING CURSOR TO THAT LOCATION
                                fseek(out_file, DATA_SIZE * i, SEEK_SET);
                        //CHECK FOR LAST SEHMENT OF REMAINING BYTES
                        if (i == File_Segments-1 && final_seg_size > 0){
                                //SETTING ITS SIZE TO PACKET LENGTH
                                length = final_seg_size;
                        }
                        //ELSE SET THE LENGTH TO COMPLETE DATA LENGTH
                        else{
```

```
                                        length = DATA_SIZE;
                                    }
                                    //PUTTING DATA FROM FILE INTO BUFFER AFTER THE HEADER
                                    fread(buffer + Header_size, 1, length, out_file);
                                    //SENDING PACKET
                                    if (bytes_in_packet = sendto(sockfd, buffer, length + Header_size, 0,
                                        (struct sockaddr*)&sendto_addr, sizeof(sendto_addr)) == -1)
                                    {
                                        perror("sender: sendto");
                                        exit(1);
                                    }
                                    //SETTING THAT PACKET AS SENT IN SENT SEGMENT ARRAY
                                    SENT_SEGMENTS[i] = 1;
                                    printf("Packet (%d) is retransmitted!\n", i);
                                }
                            }
                        }
                        printf("*******************************\n");

                        //CHECKING FOR NOT RECEIVED ACKS
                        if (count>0) {
                            //STARTING TIMER AGAIN
                            TIMER_STARTS();
                        }

                        //RUNNING LOOP FOR ALL NOT RECEIVED ACKS
                        for (i=0; NOT_RECEIVED_ACKS[i]!=0; i++){
                            //WAITING FOR ACKS
                            bytes_in_packet = recvfrom(sockfd, buffer, PACKETSIZE, 0, &receiver_addr, &their_addr_len);
                            if(bytes_in_packet == Header_size){
                                //COPYING HEADER INFO FROM RECEIVED BUFFER INTO RECEIVER'S PACKET
HEADER
                                memcpy(&receiver_Packet_Header, buffer, Header_size);
                                //CHECKING IF ACK IS RECEIVED
                                if (receiver_Packet_Header.CODE == ACK){
                                    printf("Acknowledgment ACK(%d) is received!\n",
receiver_Packet_Header.seq_no);
```

```
                    ACKED_SEGMENTS[receiver_Packet_Header.seq_no] = 1;
                }
            }
            //CHECKING FOR NOT RECEIVED ACKS
            else if (NOT_RECEIVED_ACKS[i] != 0){
                //SETTING SEQ# OF MISSSED ACKS
                sender_Packet_Header.seq_no = NOT_RECEIVED_ACKS[i];
                //PUTTING SENDER'S PACKET HEADER INTO BUFFER
                memcpy(buffer, &sender_Packet_Header, Header_size);
                //SETTING THE POSITION OF THE CURSOR
                if (SEEK_CUR != NOT_RECEIVED_ACKS[i] * DATA_SIZE)
                    //MOVING CURSOR TO THAT LOCATION
                    fseek(out_file, DATA_SIZE * NOT_RECEIVED_ACKS[i], SEEK_SET);
                //CHECK FOR LAST SEGMENT OF REMAINING BYTES
                if (NOT_RECEIVED_ACKS[i] == File_Segments-1 && final_seg_size > 0){
                    //SETTING THE SIZE OF LAST PACKET
                    length = final_seg_size;
                }
                //ELSE SET IT TO COMPLETE DATA LENGTH
                else{
                    length = DATA_SIZE;
                }
                //PUTTING DATA INTO BUFFER FROM FILE AFTER THE HEADER INFORMATION
                fread(buffer + Header_size, 1, length, out_file);
                //SENDING PACKET
                if (bytes_in_packet = sendto(sockfd, buffer, length + Header_size, 0,
                    (struct sockaddr*)&sendto_addr, sizeof(sendto_addr)) == -1)
                {
                    perror("sender: error sending the packet");
                    exit(1);
                }
                printf("Packet(%d) is retransmitted!\n", NOT_RECEIVED_ACKS[i]);
                //DECREASING COUNTER TO RECEIVE ACK OF THIS PACKET AGAIN
                i--;
                //RESETTING TIMER
                TIMER_STARTS();
```

```
                                        //CONTINUE THE LOOP
                                        continue;
                                } else{
                                        break;
                                }

                        }
                        printf("*******************************\n");
                        TOTAL_SEGMENTS = File_Segments;
                        //CEHCK FOR ALL SEGMENTS RECEIVED
                        ACK_COUNT = ACKS_COUNTER(ACKED_SEGMENTS, File_Segments);
                        L_Ack_R = ACK_COUNT-1;
                }

                //STARTING CONNECTION TEARDOWN
                // printf("----------------------\n");
                // printf("Closing connection...  \n");
                // printf("----------------------\n");
                //LOOP TO RECEIVE FIN ACK
                while(receiver_Packet_Header.CODE != FIN_ACK){
                        //SETTING PACKET HEADER's CODE TO FIN
                        sender_Packet_Header.CODE = FIN;
                        //PUTTING SENDER PACKET's HEADER INFO INTO BUFFER
                        memcpy(buffer, &sender_Packet_Header, Header_size);
                        //SEND THE FIN PACKET
                        if (bytes_in_packet = sendto(sockfd, buffer, Header_size, 0,
                                        (struct sockaddr*)&sendto_addr, sizeof(sendto_addr)) == -1)
                        {
                                perror("sender: error sending FIN packet");
                                exit(1);
                        }
                        //printf("[SND] FIN Sent. \n");
                        //WAIT FOR RECEIVING PACKET OF FIN ACK
                        bytes_in_packet = recvfrom(sockfd, buffer, PACKETSIZE, 0, &receiver_addr, &their_addr_len);
                        if(bytes_in_packet == Header_size){
                                memcpy(&receiver_Packet_Header, buffer, Header_size);
```

```
                                //CHECKING IF PACKET CONTAINS FIN ACK
                                if (receiver_Packet_Header.CODE == FIN_ACK){
                                        //printf("[RCV] FIN_ACK Received. \n");
                                }
                        }
                }

                //SETTING CODE OF PACKET HEADER TO ACK
                sender_Packet_Header.CODE = ACK;
                //PUTTING SENDER'S PACKET HEADER TO BUFFER
                memcpy(buffer, &sender_Packet_Header, Header_size);
                //SENDING LAST ACK PACKET
                if (bytes_in_packet = sendto(sockfd, buffer, Header_size, 0,
                        (struct sockaddr*)&sendto_addr, sizeof(sendto_addr)) == -1)
                {
                        perror("sender: error sending last ACK packet");
                        exit(1);
                }

                printf("Acknowledgment(ACK) Sent! \n");
                printf("*********************************\n");
                //CLOSING THE FILE
                fclose(out_file);
                //CLOSING SOCKET
                close(sockfd);
}
```

5.2 Receiver:

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <stdint.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netdb.h>
#include <errno.h>
#include <arpa/inet.h>

//DEFINING VARIABLES
#define PACKETSIZE 500
#define MAX_PACKETS_LIMIT 500000
#define DATA 0
#define ACK 1
#define FIN 2
#define FIN_ACK 3

//PROTOTYPES
//FUNCTION FOR RELIABLE DATA TRANSFER USING UDP
int RELIABLE_UDP(char * , char* );

int NFE = 0, LFA = -1;
//INITIALIZING ARRAY FOR RECEIVING PACKETS
uint8_t RECEIVED_PACKETS[MAX_PACKETS_LIMIT];

FILE * RECEIVED_FILE;

//PACKET HEADER
typedef struct {
        uint64_t sent_time;
        uint16_t seq_no;
        uint16_t CODE;
}Packet_Header;

//MAIN FUNCTION
int main(int argc, char** argv){
        //THREE ARGUMENTS ARE REQUIRED TO BE ENETERED BY USER IN COMMAND LINE
        if(argc != 3){
```

```
                fprintf(stderr, "usage: %s port filename_to_save\n\n", argv[0]);
                exit(1);
        }
        //CALLING FUNCTION FOR RELIABLE DATA TRANSFER
        RELIABLE_UDP(argv[1], argv[2]);
}

//FUNCTION FOR RELIABLE DATA TRANSFER USING UDP
int RELIABLE_UDP(char * udpPort, char* destinationFile) {
        int sockfd, rv, PACKET_BYTES, i;
        struct addrinfo myinfo;
        struct addrinfo *srvr_info;
        struct addrinfo *selected_info;
        struct sockaddr sender_addr;
        char buffer[PACKETSIZE];
        socklen_t their_addr_len = sizeof(sender_addr);

        memset(&myinfo, 0, sizeof myinfo);
        //set ai_family to AF_INET for IPv4 address
   myinfo.ai_family = AF_INET;
   myinfo.ai_socktype = SOCK_DGRAM;
   //USING OUR OWN IP ADDRESS
   myinfo.ai_flags = AI_PASSIVE;

   if ((rv = getaddrinfo(NULL, udpPort, &myinfo, &srvr_info)) != 0) {
        printf("error: %s\n", gai_strerror(rv));
        return 1;
   }

   for(selected_info = srvr_info; selected_info != NULL; selected_info = selected_info->ai_next) {
        if ((sockfd = socket(selected_info->ai_family, selected_info->ai_socktype, selected_info->ai_protocol)) == -1) {
                perror("receiver: socket creation failed");
                continue;
        }

        //BINDING THE SOCKET
```

```
            if (bind(sockfd, selected_info->ai_addr, selected_info->ai_addrlen) == -1) {
                        close(sockfd);
                        perror("receiver: binding failed");
                        continue;
            }
            break;
}

if (selected_info == NULL) {
            fprintf(stderr, "receiver: failed to bind socket\n");
            return 2;
}

freeaddrinfo(srvr_info);

            printf("Waiting to receive file from sender...\n");

Packet_Header Pkt_Header;
//STORING SIZE OF PACKET HEADER INTO HEADER_SIZE VARIABLE
int HEADER_SIZE = (int) sizeof(Pkt_Header);

            //PACKET SIZE SUBTRACT HEADER SIZE GIVES THE DATA SIZE
            int DATA_SIZE = PACKETSIZE - HEADER_SIZE;

            RECEIVED_FILE = fopen(destinationFile, "wb");

            uint16_t CODE, seq, length;

            //THIS RUNS FOR RECEIVING DATA PACKETS AND SENDING THE ACKS BACK TO SENDER
            while(1){
                        PACKET_BYTES = recvfrom(sockfd, buffer, PACKETSIZE, 0, (struct sockaddr *)&sender_addr,
            &their_addr_len);
                        //IF BYTES RECIEVED ARE LESS THAN HEADER SIZE MEANS PACKET IS INVALID
                        if (PACKET_BYTES < HEADER_SIZE)
                                    continue;
                        //COPYING HEADER INFORMATION FROM RECEIVED BUFFER INTO PACKET HEADER
                        memcpy(&Pkt_Header, buffer, HEADER_SIZE);
```

```
//COPYING CODE OF PACKET INTO CODE
CODE = Pkt_Header.CODE;

//CHECK FOR FIN PACKET TO BREAK THE LOOP
if (CODE == FIN){
        break;
}

//CHECK IF PACKET TYPE IS NOT DATA TOO THEN JUST SKIP ITERATION
if (CODE != DATA){
        continue;
}

//COPYING SEQ# OF PACKET FROM PACKET HEADER
seq = Pkt_Header.seq_no;
//CALCULATING LENGTH OF DATA PACKET
length = PACKET_BYTES - HEADER_SIZE;

//CHECK IF PACKET IS RECEIVED FIRST TIME
if(RECEIVED_PACKETS[seq] == 0){
        printf("Packet(%d) is received!\n", seq);
        printf("*******************************\n\n");
        RECEIVED_PACKETS[seq] = 1;
        //CHECKING FOR CORRECT POSITION OF CURSOR
        if (SEEK_CUR != seq * DATA_SIZE){
                //MOVING CURSOR TO CORRECT LOCATION
                //Here the receiver is reordering the packets
                fseek(RECEIVED_FILE, seq * DATA_SIZE, SEEK_SET);
        }
        //WRITING RECEIVED DATA INTO FILE FROM RECEIVED BUFFER (AFTER HEADER
INFORMATION)
        fwrite(buffer + HEADER_SIZE, 1, length, RECEIVED_FILE);
        while(RECEIVED_PACKETS[NFE]){
                NFE++;
        }
//ELSE IF PACKET IS RECEIVED NO NEED TO REWRITE IT
} else{
```

```
                printf("Duplicate Packet(%d) is received!\n", seq);
                printf("*******************************\n\n");
        }

        //SETTING CODE OF HEADER TO ACK TO SEND AS INDICATION FOR PACKET RECEIVED
        Pkt_Header.CODE = ACK;
        //PUT HEADER INFORMATION IN START OF BUFFER
        memcpy(buffer, &Pkt_Header, HEADER_SIZE);
        //SENDING THIS ACK PACKET
        if ((PACKET_BYTES = sendto(sockfd, buffer, HEADER_SIZE, 0, (struct sockaddr *)&sender_addr,
their_addr_len)) == -1){
                perror("receiver: error sending ACK packet");
                exit(2);
        }
        printf("Acknowledgement[ACK(%d)] is sent!\n", seq);
        printf("*******************************\n\n");
        //UPDATING LAST FRAME ACK VARIABLE
        if(seq > LFA){
                LFA = seq;
        }
}

//SETTING HEADER'S CODE TO FIN ACK TO REPLY FIN PACKET
Pkt_Header.CODE = FIN_ACK;
//PUT HEADER INFORMATION IN START OF BUFFER
memcpy(buffer, &Pkt_Header, HEADER_SIZE);

//WAIT FOR RECEIVING ACK FROM OTHER PART TO SEND FIN ACK AND KEEP SENDING FIN ACK
while(Pkt_Header.CODE != ACK){
        //SENDING PACKET CONTAINING FIN ACK
        if ((PACKET_BYTES = sendto(sockfd, buffer, HEADER_SIZE, 0, (struct sockaddr *)&sender_addr,
                their_addr_len)) == -1)
        {
                perror("receiver: error sending FIN_ACK packet");
                exit(2);
        }
        //WAITING FOR ACK PACKET IN RESPONSE OF SENT PACKET
```

```
                PACKET_BYTES = recvfrom(sockfd, buffer, PACKETSIZE, 0, (struct sockaddr *)&sender_addr, &their_addr_len);
                if (PACKET_BYTES == HEADER_SIZE){
                        //COPYING HEADER INFORMATION FROM BUFFER INTO PACKET HEADER
                        memcpy(&Pkt_Header, buffer, HEADER_SIZE);
                        //CHECK FOR ACK
                        if (Pkt_Header.CODE == ACK){
                        printf("Acknowledgement[ACK] is received!\n");
                        }
                }
        }
        printf("*****************************\n\n");
        //CLOSING FILE
        fclose(RECEIVED_FILE);
        //CLOSING SOCKET
        close(sockfd);
}
```